

WebSphere Process Server for z/OS



Developing and Deploying Modules

Version 7.0.0

30 April 2010

This edition applies to version 7, release 0, modification 0 of WebSphere Process Server for z/OS (product number 5655-N53) and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, send an e-mail message to doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright IBM Corporation 2006, 2010.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables	v
-------------------------	----------

Part 1. Developing applications **1**

Developing business process management solutions **3**

Business integration architecture and patterns	5
Business integration scenarios	5
Roles, products, and technical challenges	6

Bindings **9**

Export and import binding overview	11
Export and import binding configuration	14
Data format transformation in imports and exports	15
Function selectors in export bindings	19
Fault handling	21
Interoperability between SCA modules and Open SCA services	26
Binding types	29
Selecting appropriate bindings	29
SCA bindings	31
Web service bindings	31
HTTP bindings	49
EJB bindings	57
EIS bindings	64
JMS bindings	70
Generic JMS bindings	79
WebSphere MQ JMS bindings	86
WebSphere MQ bindings	93
Limitations of bindings	102

Programming guides and techniques **105**

Service Component Architecture programming	105
Service Component Definition Language	105
SCA programming model fundamentals	114
SCA programming techniques	142
Business objects programming	147
Programming model	147
Programming using business object services	171
Programming techniques	174
Business rule management programming	197
Programming model	198
Examples	225
Common operations classes	291
Widget programming	300

Developing client applications for business processes and tasks **301**

Comparison of the programming interfaces for interacting with business processes and human tasks	301
Queries on business process and task data	303

Comparison of the programming interfaces for retrieving process and task data	303
Query tables in Business Process Choreographer	305
Business Process Choreographer EJB query API	358
Developing EJB client applications for business processes and human tasks	374
Accessing the EJB APIs	375
Developing applications for business processes	381
Developing applications for human tasks	404
Developing applications for business processes and human tasks	421
Handling exceptions and faults	426
Developing Web services API client applications for business processes and human tasks	429
Web service components and sequence of control	430
Web service API requirements for business processes and human tasks	431
JAX-WS-based Business Process Choreographer Web services APIs	431
Business Process Choreographer Web services API: Standards	432
Publishing and exporting artifacts from the server environment for Web services client applications	432
Developing client applications in the Java Web services environment	436
Adding security	440
Adding transaction support	440
Developing client applications using the Business Process Choreographer JMS API	441
Requirements for business processes	441
Authorization for JMS renderings	441
Accessing the JMS interface	442
Copying artifacts for JMS client applications	446
Checking the response message for business exceptions	446
Example: executing a long running process using the Business Process Choreographer JMS API	447
Developing Web applications for business processes and human tasks, using JSF components	448
Business Process Choreographer Explorer components	450
Error handling in JSF components	452
Default converters and labels for client model objects	453
Adding the List component to a JSF application	453
Adding the Details component to a JSF application	460
Adding the CommandBar component to a JSF application	462
Adding the Message component to a JSF application	466
Developing JSP pages for task and process messages	469

User-defined JSP fragments	470
Creating plug-ins to customize human task functionality	471
Creating API event handlers for Business Process Choreographer	471
Creating notification event handlers for Business Process Choreographer	474
Installing API event handler and notification event handler plug-ins for human tasks	475
Registering API event handler and notification event handler plug-ins with task templates, task models, and tasks	476
Using a plug-in to post-process people query results	477

Part 2. Deploying applications. 479

Overview of preparing and installing modules 481

Libraries and JAR files overview	481
EAR file overview	483
Preparing to deploy to a server	483
Considerations for installing service applications on clusters	485

Deploying a module 487

Installing versioned SCA modules in a production environment.	488
--	-----

Installing an SCA module with the console	489
Creating an installable EAR file using serviceDeploy	490
Deploying applications using Apache Ant tasks	491

Installing business process and human task applications 493

How business process and human task applications are installed in a network deployment environment	493
Deployment of business processes and human tasks	494
Installing business process and human task applications interactively	494
Configuring process application data source and set reference settings	495
Uninstalling business process and human task applications, using the administrative console	496
Uninstalling business process and human task applications, using an administrative command	497

Adapters and their installation 501

Troubleshooting a failed deployment 503

Deleting JCA activation specifications	504
Deleting SIBus destinations.	505

Part 3. Appendixes 507

Tables

1. Predefined data handlers	16	44. Methods for queries run on query tables	338
2. Predefined data bindings for JMS bindings	17	45. Parameters of the query table API	340
3. Predefined data bindings for WebSphere MQ bindings	18	46. Query table API parameters: Filter options	341
4. Predefined data bindings for HTTP bindings	19	47. Query table API parameters: Authorization option defaults for instance-based authorization	343
5. Predefined function selectors for JMS bindings	20	48. Query table API parameters: AdminAuthorizationOptions	344
6. Predefined function selectors for WebSphere MQ bindings	21	49. User parameters for the query table API	345
7. Predefined function selectors for HTTP bindings	21	50. Entity result set properties of a query table API entity.	346
8. Prepackaged fault selectors	25	51. Entity properties of a query table API entity	346
9. How security headers are passed	34	52. Row result set properties of a query table API row	347
10. How the attachment is generated	44	53. Methods for meta data retrieval on query tables	348
11. How the attachment is generated	44	54. Meta data related to query table structure	349
12. Supplied HTTP header information	52	55. Meta data related to query table internationalization	349
13. Return values	63	56. Query performance impact of composite query table options	353
14. Primary artifacts that make up an SCA service module	105	57. Query performance impact of query table API options	354
15. Summary of key methods and interfaces for dynamic client invocation	129	58. Query table performance: Other considerations	355
16. Summary of qualifiers	138	59. Query syntax for different object types	359
17. WSDL type to Java class conversion	144	60. API methods for process templates	402
18. Data abstractions and the corresponding implementations	147	61. API methods that are related to starting process instances	402
19. XSD artifact support	156	62. API methods for controlling the life cycle of process instances	402
20. WSDL artifact support	157	63. API methods for controlling the life cycle of activity instances	403
21. Runtime artifact support	157	64. API methods for variables and custom properties.	404
22. Business object services	171	65. API methods for task templates	419
23. Business Rule Group problems.	223	66. API methods for task instances	420
24. Rule set and Decision Table problems	224	67. API methods for working with escalations	420
25. Properties of predefined query tables	306	68. API methods for variables and custom properties.	421
26. Predefined query tables containing instance data.	307	69. File artifacts and XML definition namespaces for the JAX-WS-based Web services	431
27. Predefined query tables containing template data.	308	70. Mapping of the reference bindings to JNDI names	450
28. Properties of supplemental query tables	310	71. How Business Process Choreographer interfaces are mapped to client model objects .	453
29. Valid contents of a composite query table	315	72. bpe:list attributes	459
30. Invalid contents of a composite query table	315	73. bpe:column attributes.	459
31. Properties of composite query tables.	315	74. bpe:details attributes	461
32. Query table development steps	319	75. bpe:property attributes	462
33. Attributes for query table expressions	323	76. bpe:commandbar attributes	465
34. Types of authorization for query tables	328	77. bpe:command attributes	466
35. Work item types	330	78. bpe:form attributes.	468
36. Work items and people assignment criteria	331		
37. Attribute types	332		
38. Database type to attribute type mapping	333		
39. Database types to attribute types mapping example	334		
40. Attribute type to literal values mapping	334		
41. Attribute type to user parameter values mapping	335		
42. Attribute type to Java object type mapping	336		
43. Attribute type compatibility.	337		

Part 1. Developing applications

Developing business process management solutions

This section discusses the fundamentals of the business process management (BPM) programming model. It introduces the Service Component Architecture (SCA) and discusses patterns related to business integration.

BPM is the discipline that enables companies to identify, consolidate, and optimize business processes. The objective is to improve productivity and maximize organizational effectiveness. Interest in BPM has become more acute as companies merge and consolidate, and as they grow a library of disparate information assets. These assets often lack consistency and coordination, thus giving rise to “islands of information.”

BPM has strong links to Service-Oriented Architecture (SOA). Depending on the nature of the company and the extent of the integration needs, BPM poses different requirements for IT departments. Some projects deal with only a few aspects, whereas some larger projects encompass many of these requirements. Here are some of the most common aspects of BPM projects:

- **Application integration** is a common requirement. The complexity of application integration projects varies from simple cases, in which you need to ensure that few applications can share information, to more complex situations, in which transactions and data exchanges need to be reflected simultaneously on multiple back-end applications. Complex application integration often requires complex unit-of-work management as well as transformation and mapping.
- **Process automation** is another key aspect that ensures that activities performed by an individual or organization systematically trigger consequential activities elsewhere. This ensures the successful completion of the overall business process. For example, when a company hires an employee, payroll information has to be updated, appropriate actions need to be taken by the security department, the necessary tools need to be given to the employee, and so on. Some activities in a process might capture human input and interaction, whereas others might invoke scripts on back-end systems and other services in the environment.
- **Connectivity** is an abstract, yet critical, aspect both in a company and in terms of business partners. Connectivity refers to both the flow of information between organizations or companies and the ability to access distributed IT services.

Some of the technical challenges of business integration implementations can be summarized as follows:

- Dealing with different data formats and therefore not being able to perform efficient data transformation
- Dealing with different protocols and mechanisms for accessing IT services that have been developed using different technologies
- Orchestrating different IT services that may be geographically distributed or offered by different organizations
- Providing rules and mechanisms to classify and manage the services that are available (governance)

As such, BPM encompasses many of the themes and elements that are also common to SOA. IBM®'s vision of BPM builds on many of the same foundational concepts that are found in SOA. One of the immediate consequences of this vision

is that BPM solutions require various products for their realization. IBM provides a portfolio of tools and runtime platforms to support all the various stages and operational aspects.

To paraphrase IBM's vision of BPM, it enables companies to define, create, merge, consolidate, and streamline business processes using applications that run on a SOA IT infrastructure. BPM work is truly role-based. At the macro level, it involves modeling, developing, governance, managing, and monitoring business process applications. With the help of proper tools and procedures, it enables you to automate business processes involving people and heterogeneous systems, both inside and outside the enterprise. One of the key aspects of BPM is the ability to optimize your business operations so that they are efficient, scalable, reliable, and flexible enough to handle change.

BPM requires development tools, runtime servers, monitoring tools, a service repository, toolkits, and process templates. Because there are so many aspects to BPM, you will find that you have to use more than one development tool to develop a solution. These tools enable integration developers to assemble complex business solutions. A server is a high-performance business engine or service container that runs complex applications. Management always wants to know who is doing what in the organization, and that is where monitoring tools come into play. As enterprises create these business processes or services, governance, classification, and storage of these services becomes critical. That function is served by a service repository. Specific toolkits to create specialized parts of the solution, such as connectors or adapters to existing systems, are often required.

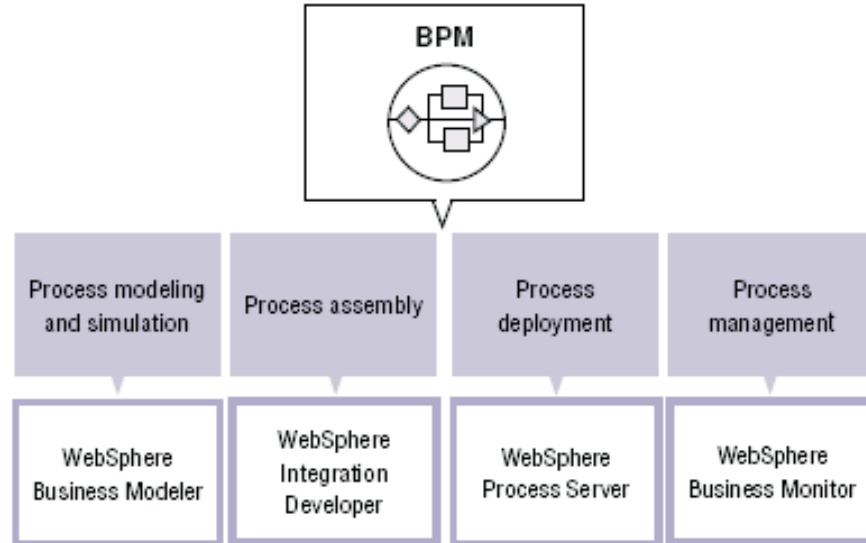


Figure 1. IBM tools span the entire BPM life cycle, enabling you to model, assemble, deploy, and manage your processes.

BPM is not based on a single product. It involves almost everybody and all business aspects within an organization and across organizations. BPM encompasses many of the services and elements in the SOA reference architecture.

For more details about these concepts, along with programming examples, see:

- *WebSphere® Business Integration Primer: Process Server, BPEL, SCA, and SOA*, IBM Press, 2008.

- *Getting Started with IBM WebSphere Process Server and IBM WebSphere Enterprise Service Bus Part 1: Development*, IBM Redbooks®, SG24-7608-00, June 2008.
- *IBM Business Process Management Reviewer's Guide*, IBM Redpapers, REDP-4433-01, April 2009.

Business integration architecture and patterns

A typical business management project involves coordinating several different IT assets, potentially running on different platforms, and having been developed at different times using different technologies. Being able to easily manipulate and exchange information with a diverse set of components is a major technical challenge. It is best addressed by the programming model used to develop business integration solutions.

This section introduces the Service Component Architecture (SCA) and discusses patterns related to business integration. Patterns seem to permeate our lives. Sewing patterns, think-and-learn patterns for children, home construction patterns, wood-carving patterns, flight patterns, wind patterns, practice patterns in medicine, customer buying patterns, workflow patterns, design patterns in computer science, and many more exist.

Patterns have proven successful in helping solution designers and developers. Therefore, it is not surprising that we now have business integration patterns and enterprise integration patterns. There is a wide array of patterns that are applicable to business integration, including patterns for request and response routing, channel patterns (such as publish/subscribe), and many more. Abstract patterns provide a template for resolving a certain category of problems, whereas concrete patterns provide more specific indications of how to implement a specific solution. This section focuses on patterns that deal with data and service invocation, which are at the foundation of the programming model of the IBM software strategy for WebSphere business integration.

Business integration scenarios

Enterprises have many different software systems that they use to run their business. In addition, they have their own ways of integrating these business components.

The two most prevalent business integration scenarios are as follows:

- **Integration broker:** In this scenario, the business integration solution acts as an intermediary located among various back-end applications. For example, you might have to ensure that when a customer places an order using the online order management application, the transaction updates relevant information in your Customer Relationship Management (CRM) back end. In this scenario, the integration solution must be able to capture and possibly transform the necessary information from the order management application and invoke the appropriate services in the CRM application.
- **Process automation:** In this scenario, the integration solution acts as the glue among different IT services that would otherwise be unrelated. For example, when a company hires an employee, the following sequence of actions must occur:
 - The information for the employee is added to the payroll system.
 - The employee must be granted physical access to the facilities, and a badge must be provided.

- The company might have to assign a set of physical assets to the employee (office space, a computer, and so on).
- The IT department must create a user profile for the employee and grant access to a series of applications.

Automating this process is also a common use case in a business integration scenario. In this scenario, the solution implements an automated flow that is triggered by the addition of the employee to the payroll system. After, the flow triggers the other steps by creating work items for the people who are responsible for taking action or by calling the appropriate services.

In both scenarios, the integration solution must:

1. Work with disparate sources of information and different data formats, and be able to convert information between different formats
2. Be able to invoke various services, potentially using different invocation mechanisms and protocols.

Roles, products, and technical challenges

Successful business integration projects depend on the blending of specialized development roles, programming techniques, and tool suites.

Business integration projects require a few basic ingredients:

- A clear separation of roles in the development organization to promote specialization, which typically improves the quality of the individual components that are developed
- A common business object (BO) model that enables business information to be represented in a common logical model
- A programming model that strongly separates interfaces from implementations and that supports a generic service invocation mechanism that is totally independent of the implementation and that only involves dealing with interfaces
- An integrated set of tools and products that supports development roles and preserves their separation

The following sections elaborate on each of these ingredients.

Clear separation of roles

A business integration project requires people in four collaborative, but distinctly separate, roles:

- **Business analyst:** Business analysts are domain experts responsible for capturing the business aspects of a process and for creating a process model that adequately represents the process itself. Their focus is to optimize the financial performance of a process. Business analysts are not concerned with the technical aspects of implementing processes.
- **Component developer:** Component developers are responsible for implementing individual services and components. Their focus is the specific technology used for the implementation. This role requires a strong programming background.
- **Integration specialist:** This relatively new role describes the person who is responsible for assembling a set of existing components into a larger business integration solution. Integration developers do not need to know the technical details of each of the components and services they reuse and wire together. Ideally, integration developers are concerned only with understanding the

interfaces of the services that they are assembling. Integration developers should rely on integration tools for the assembly process.

- **Solution deployer:** Solution deployers and administrators are concerned with making business integration solutions available to end users. Ideally, a solution deployer is primarily concerned with binding a solution to the physical resources ready for it to function (databases, queue managers, and so on) and not with having a deep understanding of the internals of a solution. The solution deployer's focus is quality of service (QoS).

A common business object model

As we discussed, the key aspects of a business integration project include the ability to coordinate the invocation of several components and the ability to handle the data exchange among those. In particular, different components can use different techniques to represent business items such as the data in an order, a customer's information, and so on. For example, you might have to integrate a Java™ application that uses entity Enterprise Java Beans (EJBs) to represent business items and a legacy application that organizes information in COBOL copybooks. Therefore, a platform that aims to simplify the creation of integration solutions should also provide a generic way to represent business items, irrespective of the techniques used by the back-end systems for data handling. This goal is achieved in WebSphere Process Server and WebSphere Enterprise Service Bus thanks to the *business object framework*.

The business object framework enables developers to use XML Schemas to define the structure of business data and access and manipulate instances of these data structures (business objects) via XPath or Java code. The business object framework is based on the Service Data Object (SDO) standard.

The Service Component Architecture (SCA) programming model

The SCA programming model represents the foundation for any solution to be developed on WebSphere Process Server and WebSphere Enterprise Service Bus. SCA provides a way for developers to encapsulate service implementations in reusable components. It enables you to define interfaces, implementations, and references in a way that is independent of which technology you use. This approach gives you the opportunity to bind the elements to whichever technology you choose. There is also an SCA client programming model that enables the invocation of those components. In particular, it enables runtime infrastructures based on Java to interact with non-Java runtimes. SCA uses business objects as the data items for service invocation.

Tools and products

IBM WebSphere Integration Developer is the integrated development environment that has all the necessary tools to create and compose business integration solutions based on the technologies just mentioned. These solutions typically are deployed to the WebSphere Process Server or, in some cases, to the WebSphere Enterprise Service Bus.

Bindings

At the core of a service-oriented architecture is the concept of a *service*, a unit of functionality accomplished by an interaction between computing devices. An *export* defines the external interface (or access point) of a module, so that Service Component Architecture (SCA) components within the module can provide their services to external clients. An *import* defines an interface to services outside a module, so the services can be called from within the module. You use protocol-specific *bindings* with imports and exports to specify the means of transporting the data into or out of the module.

Exports

External clients can invoke SCA components in an integration module over a variety of protocols (such as HTTP, JMS, MQ, and RMI/IIOP) with data in a variety of formats (such as XML, CSV, COBOL, and JavaBean). Exports are components that receive these requests from external sources and then invoke WebSphere Process Server components using the SCA programming model.

For example, in the following figure, an export receives a request over the HTTP protocol from a client application. The data is transformed into a business object, the format used by the SCA component. The component is then invoked with that data object.

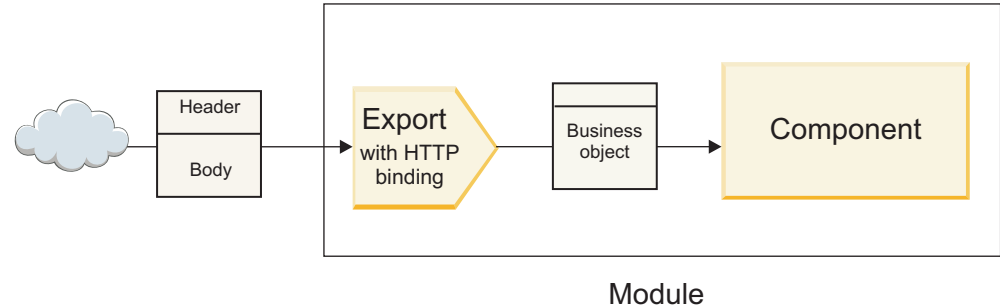


Figure 2. An export with HTTP binding

Imports

An SCA component might want to invoke a non-SCA external service that expects data in a different format. An import is used by the SCA component to invoke the external service using the SCA programming model. The import then invokes the target service in the way that the service expects.

For example, in the following figure, a request from an SCA component is sent, by the import, to an external service. The business object, which is the format used by the SCA component, is transformed to the format expected by the service, and the service is invoked.

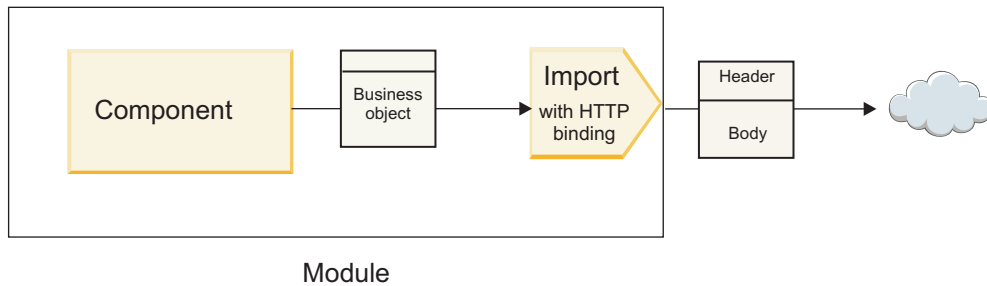


Figure 3. An import with HTTP binding

List of bindings

You use WebSphere Integration Developer to generate a binding for an import or export and to configure the binding. The types of bindings that are available are described in the following list.

- SCA

The SCA binding, which is the default, lets your service communicate with services in other SCA modules. You use an import with an SCA binding to access a service in another SCA module. You use an export with an SCA binding to offer a service to other SCA modules.

- Web service

A Web service binding lets you access an external service using interoperable SOAP messages and qualities of service. You can also use Web service bindings to include attachments as part of the SOAP message.

The Web service binding can use a transport protocol of either SOAP/HTTP (SOAP over HTTP) or SOAP/JMS (SOAP over JMS). Regardless of the transport (HTTP or JMS) used to convey the SOAP messages, Web service bindings always handle request/response interactions synchronously.

- HTTP

The HTTP binding lets you access an external service using the HTTP protocol, where non-SOAP messages are used, or where direct HTTP access is required. This binding is used when you are working with Web services that are based on the HTTP model (that is, services that use well-known HTTP interface operations such as GET, PUT, DELETE, and so on).

- Enterprise JavaBeans™ (EJB)

The EJB bindings let SCA components interact with services provided by Java EE business logic running on a Java EE server.

- EIS

The EIS (enterprise information system) binding, when used with a JCA resource adapter, lets you access services on an enterprise information system or make your services available to the EIS.

- JMS bindings

Java Message Service (JMS), generic JMS, and WebSphere MQ JMS (MQ JMS) bindings are used for interactions with messaging systems, where asynchronous communication through message queues is critical for reliability.

An export with one of the JMS bindings watches a queue for the arrival of a message and asynchronously sends the response, if any, to the reply queue. An import with one of the JMS bindings builds and sends a message to a JMS queue and watches a queue for the arrival of the response, if any.

- JMS

- The JMS binding lets you access the WebSphere-embedded JMS provider.
 - Generic JMS
 - The generic JMS binding lets you access a non-IBM vendor messaging system.
 - MQ JMS
 - The MQ JMS binding lets you access the JMS subset of a WebSphere MQ messaging system. You would use this binding when the JMS subset of functions is sufficient for your application.
 - MQ
 - The WebSphere MQ binding lets you communicate with MQ native applications, bringing them into the service oriented architecture framework and providing access to MQ-specific header information. You would use this binding when you need to use MQ native functions.

Export and import binding overview

An export lets you make services in an integration module available to external clients, and an import makes it possible for your SCA components in an integration module to call external services. The binding associated with the export or import specifies the relationship between protocol messages and business objects. It also specifies the way that operations and faults are selected.

Flow of information through an export

An export receives a request, which is intended for the component to which the export is wired, over a specific transport determined by the associated binding (for example, HTTP).

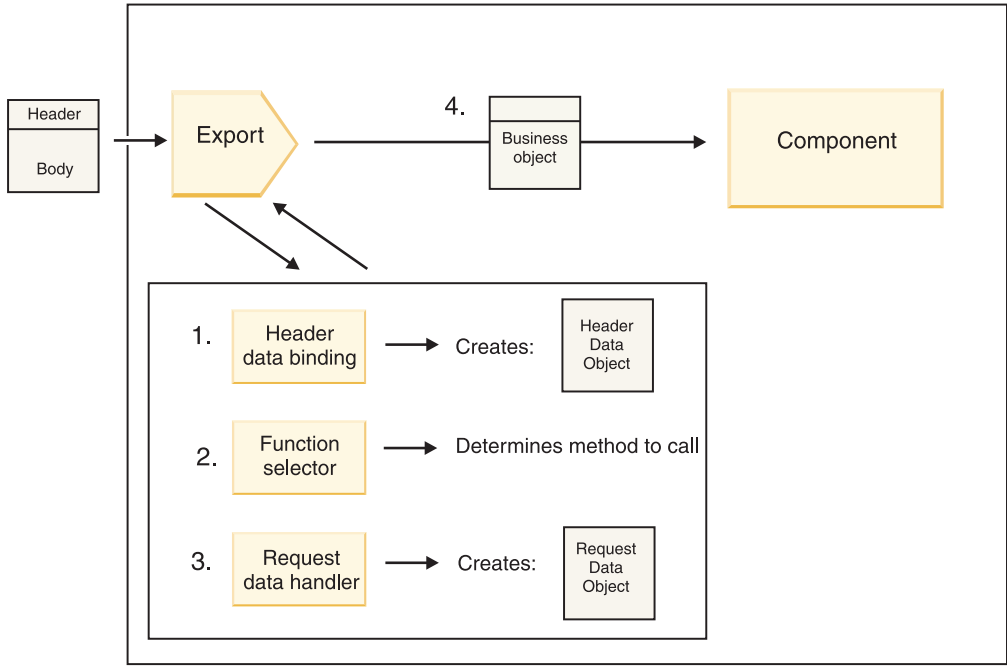


Figure 4. Flow of a request through the export to a component

When the export receives the request, the following sequence of events occurs:

1. For WebSphere MQ bindings only, the header data binding transforms the protocol header into a header data object.

2. The function selector determines the native method name from the protocol message. The native method name is mapped by the export configuration to the name of an operation on the interface of the export.
3. The request data handler or data binding on the method transforms the request to a request business object.
4. The export invokes the component method with the request business object.
 - The HTTP export binding, the Web service export binding, and the EJB export binding invoke the SCA component synchronously.
 - The JMS, Generic JMS, MQ JMS, and WebSphere MQ export bindings invoke the SCA component asynchronously.

Note that an export can propagate the headers and user properties it receives over the protocol, if context propagation is enabled. Components that are wired to the export can then access these headers and user properties. See the “Propagation” topic in the WebSphere Integration Developer information center for more information.

If this is a two-way operation, the component returns a response.

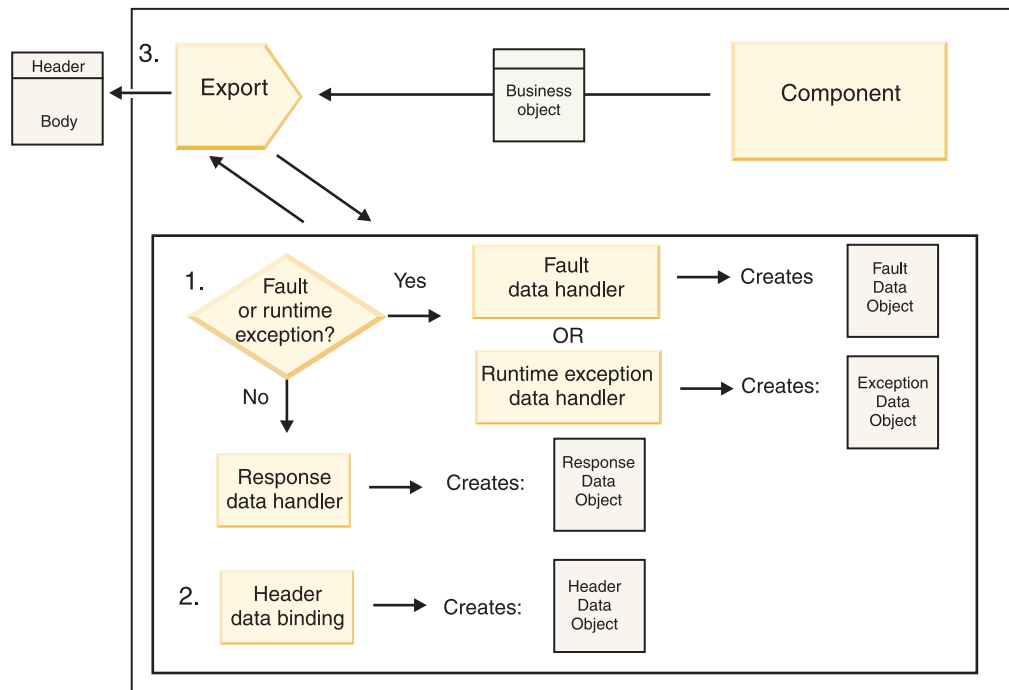


Figure 5. Flow of a response back through the export

The following sequence of steps occurs:

1. If a normal response message is received by the export binding, the response data handler or data binding on the method transforms the business object to a response.
 - If the response is a fault, the fault data handler or data binding on the method transforms the fault to a fault response.
 - For HTTP export bindings only, if the response is a runtime exception, the runtime exception data handler, if configured, is called.
2. For WebSphere MQ bindings only, the header data binding transforms the header data objects into protocol headers.

3. The export sends the service response over the transport.

Flow of information through an import

Components send requests to services outside the module using an import. The request is sent, over a specific transport determined by the associated binding.

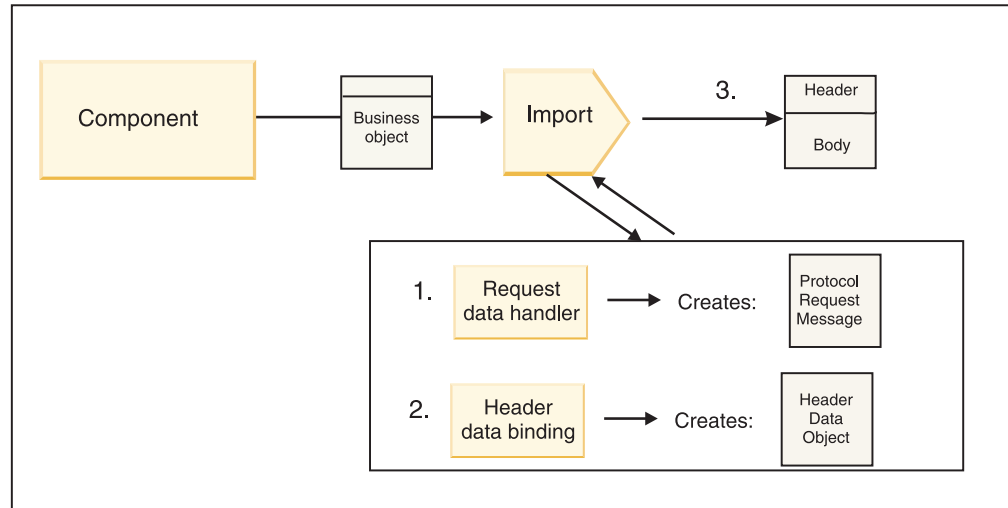


Figure 6. Flow from a component through the import to a service

The component invokes the import with a request business object.

Note:

- The HTTP import binding, the Web service import binding, and the EJB import binding should be invoked synchronously by the calling component.
- The JMS, Generic JMS, MQ JMS, and WebSphere MQ import binding should be invoked asynchronously.

After the component invokes the import, the following sequence of events occurs:

1. The request data handler or data binding on the method transforms the request business object into a protocol request message.
2. For WebSphere MQ bindings only, the header data binding on the method sets the header business object in the protocol header.
3. The import invokes the service with the service request over the transport.

If this is a two-way operation, the service returns a response, and the following sequence of steps occurs:

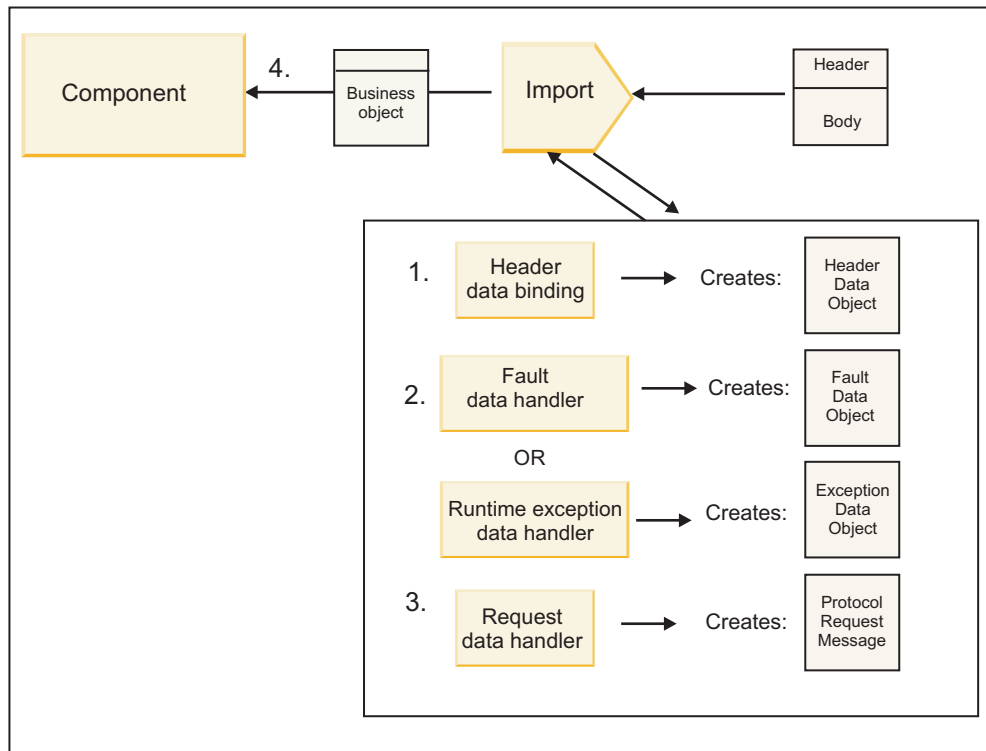


Figure 7. Flow of a response back through the import

1. For WebSphere MQ bindings only, the header data binding transforms the protocol header into a header data object.
2. A determination is made about whether the response is a fault.
 - If the response is a fault, the fault selector inspects the fault to determine which WSDL fault it maps to. The fault data handler on the method then transforms the fault to a fault response.
 - If the response is a runtime exception, the runtime exception data handler, if configured, is called.
3. The response data handler or binding on the method transforms the response to a response business object.
4. The import returns the response business object to the component.

Export and import binding configuration

One of the key aspects of export and import bindings is data format transformation, which indicates how data is mapped (deserialized) from a native wire format to a business object or how it is mapped (serialized) from a business object to a native wire format. For bindings associated with exports, you can also specify a function selector to indicate which operation should be performed on the data. For bindings associated with exports or imports, you can indicate how faults that occur during processing should be handled.

In addition, you specify transport-specific information on bindings. For example, for an HTTP binding, you specify the endpoint URL. You can find more information in the WebSphere Integration Developer information center. For example, for the HTTP binding, the transport-specific information is described in the “Generating an HTTP import binding” and “Generating an HTTP export binding” topics.

Data format transformation in imports and exports

When an export or import binding is configured in WebSphere Integration Developer, one of the configuration properties that you specify is the data format used by the binding.

- For export bindings, in which a client application sends requests to and receives responses from an SCA component, you indicate the format of the native data. Depending on the format, the system selects the appropriate data handler or data binding to transform the native data to a business object (which is used by the SCA component) and conversely to transform the business object to native data (which is the response to the client application).
- For import bindings, in which an SCA component sends requests to and receives responses from a service outside the module, you indicate the data format of the native data. Depending on the format, the system selects the appropriate data handler or data binding to transform the business object to native data and vice versa.

WebSphere Process Server provides a set of predefined data formats and corresponding data handlers or data bindings that support the formats. You can also create your own custom data handlers and register the data format for those data handlers. For more information, see the “Developing data handlers” topic in the WebSphere Integration Developer information center.

- *Data handlers* are protocol-neutral and transform data from one format to another. In WebSphere Process Server, data handlers typically transform native data (such as XML, CSV, and COBOL) to a business object and a business object to native data. Because they are protocol-neutral, you can reuse the same data handler with a variety of export and import bindings. For example, you can use the same XML data handler with an HTTP export or import binding or with a JMS export or import binding.
- *Data bindings* also transform native data to a business object (and vice versa), but they are protocol-specific. For example, an HTTP data binding can be used with an HTTP export or import binding only. Unlike data handlers, an HTTP data binding cannot be reused with an MQ export or import binding.

Note: Three HTTP data bindings (HTTPStreamDataBindingSOAP, HTTPStreamDataBindingXML, and HTTPServiceGatewayDataBinding) are deprecated as of WebSphere Process Server Version 7.0. Use data handlers whenever possible.

As noted earlier, you can create custom data handlers, if necessary. You can also create custom data bindings; however, it is recommended that you create custom data handlers because they can be used across multiple bindings.

Data handlers

Data handlers are configured against export and import bindings to transform data from one format to another in a protocol-neutral fashion. Several data handlers are provided as part of the product, but you can also create your own data handler, if necessary. You can associate a data handler with an export or import binding at one of two levels: you can associate it with all operations in the interface of the export or import, or you can associate it with a specific operation for the request or response.

Predefined data handlers

You use WebSphere Integration Developer to specify the data handler that you want to use.

The data handlers that are predefined for your use are listed in the following table, which also describes how each data handler transforms inbound and outbound data.

Note: Except where noted, these data handlers can be used with JMS, Generic JMS, MQ JMS, WebSphere MQ, and HTTP bindings. See the “Data handlers” topic in the WebSphere Integration Developer information center for more detailed information.

Table 1. Predefined data handlers

Data handler	Native data to business object	Business object to native data
ATOM	Parses ATOM feeds into an ATOM feed business object.	Serializes an ATOM feed business object to ATOM feeds.
Delimited	Parses delimited data into a business object.	Serializes a business object to delimited data, including CSV.
Fixed Width	Parses fixed-width data into a business object.	Serializes a business object to fixed-width data.
Handled by WTX	Delegates data format transformation to the WebSphere Transformation Extender (WTX). The WTX map name is derived by the data handler.	Delegates data format transformation to the WebSphere Transformation Extender (WTX). The WTX map name is derived by the data handler.
Handled by WTX Invoker	Delegates the data format transformation to the WebSphere Transformation Extender (WTX). The WTX map name is supplied by the user.	Delegates the data format transformation to the WebSphere Transformation Extender (WTX). The WTX map name is supplied by the user.
JAXB	Serializes Java beans to a business object using the mapping rules defined by the Java Architecture for XML Binding (JAXB) specification.	Deserializes a business object to Java beans using the mapping rules defined by the JAXB specification.
JAXWS Note: The JAXWS data handler can be used only with the EJB binding.	Used by an EJB binding to transform a response Java object or exception Java object to a response business object using the mapping rules defined by the Java API for XML Web Services (JAX-WS) specification.	Used by an EJB binding to transform a business object to the outgoing Java method parameters using the mapping rules defined by the JAX-WS specification.
JSON	Parses JSON data into a business object.	Serializes a business object to JSON data.
Native body	Parses the native bytes, text, map, stream, or object into one of five base business objects (text, bytes, map, stream, or object).	Transforms the five base business objects into byte, text, map, stream, or object.

Table 1. Predefined data handlers (continued)

Data handler	Native data to business object	Business object to native data
SOAP	Parses the SOAP message (and the header) into a business object.	Serializes a business object to a SOAP message.
XML	Parses XML data into a business object.	Serializes a business object to XML data.
UTF8XMLDataHandler	Parses UTF-8 encoded XML data into a business object.	Serializes a business object into UTF-8 encoded XML data when sending a message.

Creating a data handler

Detailed information about creating a data handler can be found in the “Developing data handlers” topic in the WebSphere Integration Developer information center.

Data bindings

Data bindings are configured against export and import bindings to transform data from one format to another. Data bindings are specific to a protocol. Several data bindings are provided as part of the product, but you can also create your own data binding, if necessary. You can associate a data binding with an export or import binding at one of two levels—you can associate it with all operations in the interface of the export or import, or you can associate it with a specific operation for the request or response.

You use WebSphere Integration Developer to specify which data binding you want to use or to create your own data binding. A discussion of creating data bindings can be found in the “Overview of JMS, MQ JMS and generic JMS bindings” section of the WebSphere Integration Developer information center.

JMS bindings

The following table lists the data bindings that can be used with:

- JMS bindings
- Generic JMS bindings
- WebSphere MQ JMS bindings

The table also includes a description of the tasks that the data bindings perform.

Table 2. Predefined data bindings for JMS bindings

Data binding	Native data to business object	Business object to native data
Serialized Java object	Transforms the Java serialized object into a business object (which is mapped as the input or output type in the WSDL).	Serializes a business object to the Java serialized object in the JMS object message.

Table 2. Predefined data bindings for JMS bindings (continued)

Data binding	Native data to business object	Business object to native data
Wrapped bytes	Extracts the bytes from the incoming JMS bytes message and wraps them into the JMSBytesBody business object.	Extracts the bytes from the JMSBytesBody business object and wraps them into the outgoing JMS bytes message
Wrapped map entry	Extracts the name, value, and type information for every entry in the incoming JMS map message and creates a list of MapEntry business objects. It then wraps the list into the JMSMapBody business object	Extracts the name, value, and type information from the MapEntry list in the JMSMapBody business object and creates the corresponding entries in the outgoing JMS map message.
Wrapped object	Extracts the object from the incoming JMS object message and wraps it into the JMSObjectBody business object.	Extracts the object from the JMSObjectBody business object and wraps it into the outgoing JMS object message.
Wrapped text	Extracts the text from the incoming JMS text message and wraps it into the JMSTextBody business object.	Extracts the text from the JMSTextBody business object and wraps it into the outgoing JMS text message.

WebSphere MQ bindings

The following table lists the data bindings that can be used with WebSphere MQ and describes the tasks that the data bindings perform.

Table 3. Predefined data bindings for WebSphere MQ bindings

Data binding	Native data to business object	Business object to native data
Serialized Java object	Transforms the Java serialized object from the incoming message into a business object (which is mapped as the input or output type in the WSDL).	Transforms a business object to the Java serialized object in the outgoing message
Wrapped bytes	Extracts the bytes from the unstructured MQ bytes message and wraps them into the JMSBytesBody business object.	Extracts the bytes from a JMSBytesBody business object and wraps the bytes into the outgoing unstructured MQ bytes message.
Wrapped text	Extracts the text from an unstructured MQ text message and wraps it into a JMSTextBody business object.	Extracts text from a JMSTextBody business object and wraps it in an unstructured MQ text message.

Table 3. Predefined data bindings for WebSphere MQ bindings (continued)

Data binding	Native data to business object	Business object to native data
Wrapped stream entry	Extracts the name and type information for every entry in the incoming JMS stream message and creates a list of the StreamEntry business objects. It then wraps the list into the JMSStreamBody business object.	Extracts the name and type information from the StreamEntry list in the JMSStreamBody business object and creates corresponding entries in the outgoing JMSStreamMessage.

In addition to the data bindings listed in Table 3 on page 18, WebSphere MQ also uses header data bindings. See the WebSphere Integration Developer information center for details.

HTTP bindings

The following table lists the data bindings that can be used with HTTP and describes the tasks that the data bindings perform.

Table 4. Predefined data bindings for HTTP bindings

Data binding	Native data to business object	Business object to native data
Wrapped bytes	Extracts the bytes from the body of the incoming HTTP message and wraps them into the HTTPBytes business object.	Extracts the bytes from the HTTPBytes business object and adds them to the body of the outgoing HTTP message.
Wrapped text	Extracts the text from the body of the incoming HTTP message and wraps it into the HTTPText business object.	Extracts the text from the HTTPText business object and adds it to the body of the outgoing HTTP message.

Function selectors in export bindings

A function selector is used to indicate which operation should be performed on the data for a request message. Function selectors are configured as part of an export binding.

Consider an SCA export that exposes an interface. The interface contains two operations—Create and Update. The export has a JMS binding that reads from a queue.

When a message arrives on the queue, the export is passed the associated data, but which operation from the export's interface should be invoked on the wired component? The operation is determined by the function selector and the export binding configuration.

The function selector returns the native function name (the function name in the client system that sent the message). The native function name is then mapped to the operation or function name on the interface associated with the export. For example, in the following figure, the function selector returns the native function name (CRT) from the incoming message, the native function name is mapped to

the Create operation, and the business object is sent to the SCA component with the Create operation.

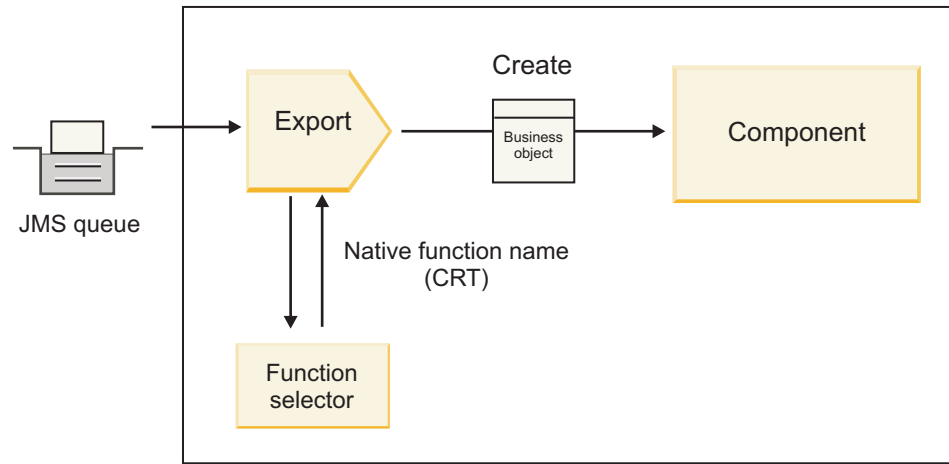


Figure 8. The function selector

If the interface has only one operation, there is no need to specify a function selector.

Several prepackaged function selectors are available and are listed in the sections that follow.

JMS bindings

The following table lists the function selectors that can be used with:

- JMS bindings
- Generic JMS bindings
- WebSphere MQ JMS bindings

Table 5. Predefined function selectors for JMS bindings

Function selector	Description
JMS function selector for simple JMS data bindings	Uses the JMSType property of the message to select the operation name.
JMS header property function selector	Returns the value of the JMS String Property, TargetFunctionName, from the header.
JMS service gateway function selector	Determines if the request is a one-way or two-way operation by examining the JMSReplyTo property set by the client.

WebSphere MQ bindings

The following table lists the function selectors that can be used with WebSphere MQ bindings.

Table 6. Predefined function selectors for WebSphere MQ bindings

Function selector	Description
MQ handleMessage function selector	Returns handleMessage as a value, which is mapped using the export method bindings to the name of an operation on the interface.
MQ uses JMS default function selector	Reads the native operation from the TargetFunctionName property of the folder of an MQRFH2 header.
MQ uses message body format as native function	Finds the Format field of the last header and returns that field as a String.
MQ type function selector	Creates a method in your export binding by retrieving a URL containing the Msd, Set, Type and Format properties found in the MQRFH2 header.
MQ service gateway function selector	Uses the MsgType property in the MQMD header to determine the operation name.

HTTP bindings

The following table lists the function selectors that can be used with HTTP bindings.

Table 7. Predefined function selectors for HTTP bindings

Function selector	Description
HTTP function selector based on the TargetFunctionName header	Uses the TargetFunctionName HTTP header property from the client to determine which operation to invoke at runtime from the export.
HTTP function selector based on the URL and HTTP method	Uses the relative path from the URL appended with the HTTP method from the client to determine the native operation defined on the export.
HTTP service gateway function selector based on URL with an operation name	Determines the method to invoke based on the URL if "operationMode = oneWay" has been appended to the request URL.

Note: You can also create your own function selector, using WebSphere Integration Developer. Information about creating a function selector is provided in the WebSphere Integration Developer information center. For example, a description of creating a function selector for WebSphere MQ bindings can be found in "Overview of the MQ function selectors".

Fault handling

You can configure your import and export bindings to handle faults (for example, business exceptions) that occur during processing by specifying fault data handlers. You can set up a fault data handler at three levels—you can associate a fault data handler with a fault, with an operation, or for all operations with a binding.

A fault data handler processes fault data and transforms it into the correct format to be sent by the export or import binding.

- For an export binding, the fault data handler transforms the exception business object sent from the component to a response message that can be used by the client application.
- For an import binding, the fault data handler transforms the fault data or response message sent from a service into an exception business object that can be used by the SCA component.

For import bindings, the binding calls the fault selector, which determines whether the response message is a normal response, a business fault, or a runtime exception.

You can specify a fault data handler for a particular fault, for an operation, and for all operations with a binding.

- If the fault data handler is set at all three levels, the data handler associated with a particular fault is called.
- If fault data handlers are set at the operation and binding levels, the data handler associated with the operation is called.

Two editors are used in WebSphere Integration Developer to specify fault handling. The interface editor is used to indicate whether there will be a fault on an operation. After a binding is generated with this interface, the editor in the properties view lets you configure how the fault will be handled. For more information, see the “Fault selectors” topic in the WebSphere Integration Developer information center.

How faults are handled in export bindings

When a fault occurs during the processing of the request from a client application, the export binding can return the fault information to the client. You configure the export binding to specify how the fault should be processed and returned to the client.

You configure the export binding using WebSphere Integration Developer.

During request processing, a client invokes an export with a request, and the export invokes the SCA component. During the processing of the request, the SCA component can either return a business response or can throw a service business exception or a service runtime exception. When this occurs, the export binding transforms the exception into a fault message and sends it to the client, as shown in the following figure and described in the sections that follow.

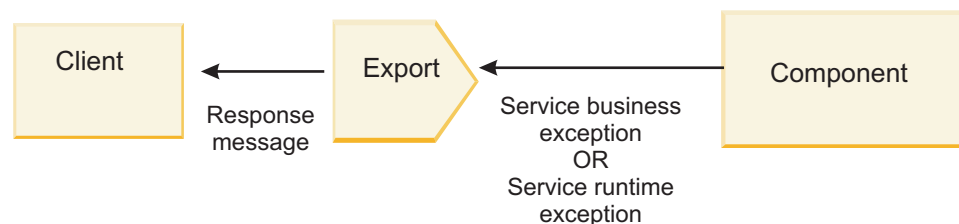


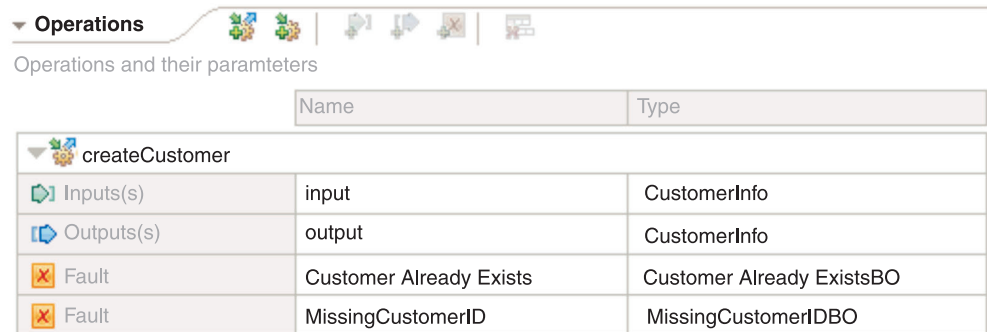
Figure 9. How fault information is sent from the component through the export binding to the client

You can create a custom data handler or data binding to handle faults.

Business faults

Business faults are business errors or exceptions that occur during processing.

Consider the following interface, which has a createCustomer operation on it. This operation has two business faults defined: CustomerAlreadyExists and MissingCustomerId.



The screenshot shows a software interface for defining an operation. At the top, there is a tab labeled "Operations" and a toolbar with various icons. Below the toolbar, the text "Operations and their parameters" is displayed. A table lists the details for the "createCustomer" operation. The table has two columns: "Name" and "Type".

	Name	Type
▼ createCustomer		
Inputs(s)	input	CustomerInfo
Outputs(s)	output	CustomerInfo
Fault	Customer Already Exists	Customer Already ExistsBO
Fault	MissingCustomerId	MissingCustomerIdBO

Figure 10. Interface with two faults

In this example, if a client sends a request to create a customer (to this SCA component) and that customer already exists, the component throws a CustomerAlreadyExists fault to the export. The export needs to propagate this business fault back to the calling client. To do so, it uses the fault data handler that is set up on the export binding.

When a business fault is received by the export binding, the following processing occurs:

1. The binding determines which fault data handler to invoke for handling the fault. If the service business exception contains the fault name, the data handler that is set up on the fault is called. If the service business exception does not contain the name of the fault, the fault name is derived by matching the fault types.
2. The binding calls the fault data handler with the data object from the service business exception.
3. The fault data handler transforms the fault data object to a response message and returns it to the export binding.
4. The export returns the response message to the client.

If the service business exception contains the fault name, the data handler that is set up on the fault is called. If the service business exception does not contain the name of the fault, the fault name is derived by matching the fault types.

Runtime exceptions

A runtime exception is an exception that occurs in the SCA application during the processing of a request that does not correspond to a business fault. Unlike business faults, runtime exceptions are not defined on the interface.

In certain scenarios, you might want to propagate these runtime exceptions to the client application so that the client application can take the appropriate action.

For example, if a client sends a request (to the SCA component) to create a customer and an authorization error occurs during processing of the request, the

component throws a runtime exception. This runtime exception has to be propagated back to the calling client so it can take the appropriate action regarding the authorization. This is achieved by the runtime exception data handler configured on the export binding.

Note: You can configure a runtime exception data handler only on HTTP bindings.

The processing of a runtime exception is similar to the processing of a business fault. If a runtime exception data handler was set up, the following processing occurs:

1. The export binding calls the appropriate data handler with the service runtime exception.
2. The data handler transforms the fault data object to a response message and returns it to the export binding.
3. The export returns the response message to the client.

Fault handling and runtime exception handling are optional. If you do not want to propagate faults or runtime exceptions to the calling client, do not configure the fault data handler or runtime exception data handler.

How faults are handled in import bindings

A component uses an import to send a request to a service outside the module. When a fault occurs during the processing of the request, the service returns the fault to the import binding. You can configure the import binding to specify how the fault should be processed and returned to the component.

You configure the import binding using WebSphere Integration Developer. You can specify a fault data handler (or data binding), and you also specify a fault selector.

Fault data handlers

The service that processes the request sends, to the import binding, fault information in the form of an exception or a response message that contains the fault data.

The import binding transforms the service exception or response message into a service business exception or service runtime exception, as shown in the following figure and described in the sections that follow.

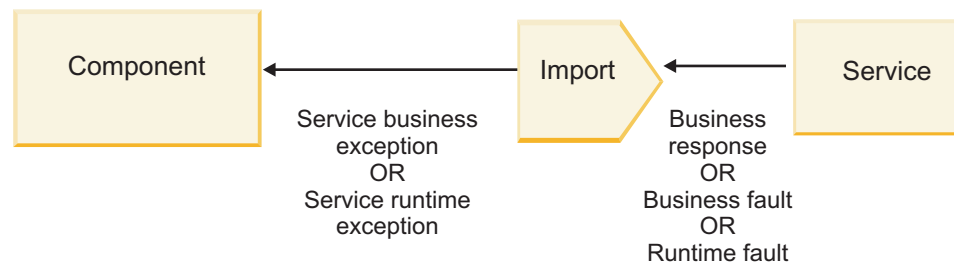


Figure 11. How fault information is sent from the service through the import to the component

You can create a custom data handler or data binding to handle faults.

Fault selectors

When you configure an import binding, you can specify a fault selector. The fault selector determines whether the import response is an actual response, a business exception, or a runtime fault. It also determines, from the response body or header, the native fault name, which is mapped by the binding configuration to the name of a fault in the associated interface.

Two types of prepackaged fault selectors are available for use with JMS, MQ JMS, Generic JMS, WebSphere MQ, and HTTP imports:

Table 8. Prepackaged fault selectors

Fault selector type	Description
Header-based	Determines whether a response message is a business fault, a runtime exception, or a normal message based on the headers in the incoming response message.
SOAP	Determines whether the response SOAP message is a normal response, business fault, or runtime exception.

The following shows examples of header-based fault selectors and the SOAP fault selector.

- Header-based fault selector

If an application wants to indicate that the incoming message is a business fault, there must be two headers in the incoming message for business faults, which is shown as follows:

```
Header name = FaultType, Header value = Business
Header name = FaultName, Header value = <user defined native fault name>
```

If an application wants to indicate that the incoming response message is a runtime exception, then there must be one header in the incoming message, which is shown as follows:

```
Header name = FaultType, Header value = Runtime
```

- SOAP fault selector

A business fault can be sent as part of the SOAP message with the following custom SOAP header. "CustomerAlreadyExists" is the name of the fault in this case.

```
<ibmSoap:BusinessFaultName
xmlns:ibmSoap="http://www.ibm.com/soap">CustomerAlreadyExists
</ibmSoap:BusinessFaultName>
```

The fault selector is optional. If you do not specify a fault selector, the import binding cannot determine the type of response. The binding therefore treats it as a business response and calls the response data handler or data binding.

You can create a custom fault selector. The steps for creating a custom fault selector are provided in the "Developing a custom fault selector" topic of the WebSphere Integration Developer information center.

Business faults

A business fault can occur when there is an error in the processing of a request. For example, if you send a request to create a customer and that customer already exists, the service sends a business exception to the import binding.

When a business exception is received by the binding, the processing steps depend on whether a fault selector has been set up for the binding.

- If no fault selector was set up, the binding calls the response data handler or data binding.
- If a fault selector was set up, the following processing occurs:
 1. The import binding calls the fault selector to determine whether the response is business fault, business response, or runtime fault.
 2. If the response is a business fault, the import binding calls the fault selector to provide the native fault name.
 3. The import binding determines the WSDL fault corresponding to the native fault name returned by the fault selector.
 4. The import binding determines the fault data handler that is configured for this WSDL fault.
 5. The import binding calls this fault data handler with the fault data.
 6. The fault data handler transforms the fault data to a data object and returns it to the import binding.
 7. The import binding constructs a service business exception object with the data object and the fault name.
 8. The import returns the service business exception object to the component.

Runtime exceptions

A runtime exception can occur when there is a problem in communicating with the service. The processing of a runtime exception is similar to the processing of a business exception. If a fault selector was set up, the following processing occurs:

1. The import binding calls the appropriate runtime exception data handler with the exception data.
2. The runtime exception data handler transforms the exception data to a service runtime exception object and returns it to the import binding.
3. The import returns the service runtime exception object to the component.

Interoperability between SCA modules and Open SCA services

The IBM WebSphere Application Server V7.0 Feature Pack for Service Component Architecture (SCA) provides a simple, yet powerful programming model for constructing applications based on the Open SCA specifications. The SCA modules of WebSphere Process Server use import and export bindings to interoperate with Open SCA services developed in a Rational® Application Developer environment and hosted by the WebSphere Application Server Feature Pack for Service Component Architecture.

An SCA application invokes an Open SCA application by way of an import binding. An SCA application receives a call from an Open SCA application by way of an export binding. A list of supported bindings is shown in “Invoking services over interoperable bindings” on page 28.

Invoking Open SCA services from SCA modules

SCA applications developed with WebSphere Integration Developer can invoke Open SCA applications developed in a Rational Application Developer environment. This section provides an example of invoking an Open SCA service from an SCA module using an SCA import binding.

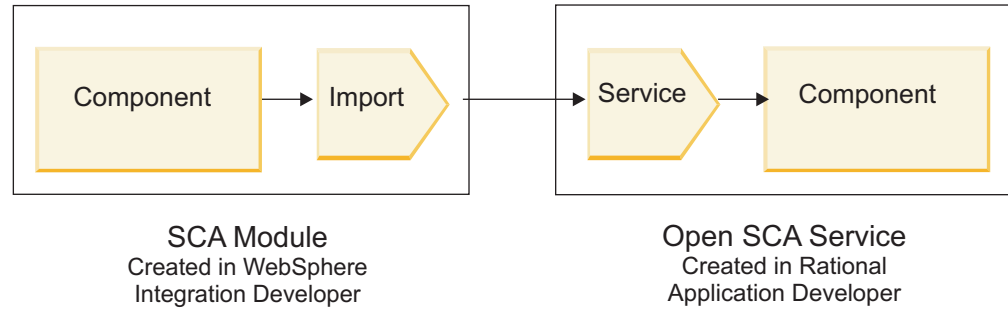


Figure 12. Component in SCA module invoking Open SCA service

No special configuration is required to invoke an Open SCA service.

To connect to an Open SCA service by way of an SCA import binding, you provide the component name and service name of the Open SCA service in the import binding.

1. To obtain the name of the target component and service from the Open SCA composite, perform the following steps:
 - a. Ensure that the **Properties** tab is open by clicking **Window** → **Show View** → **Properties**.
 - b. Open the composite editor by double-clicking the composite diagram that contains the component and service. For example, for a component named **customer**, the composite diagram is **customer.composite_diagram**.
 - c. Click the target component.
 - d. In the **Name** field of the **Properties** tab, note the name of the target component.
 - e. Click the service icon associated with the component.
 - f. In the **Name** field of the **Properties** tab, note the name of the service.
2. To configure the WebSphere Process Server import to connect it to the Open SCA service, perform the following steps:
 - a. In WebSphere Integration Developer, navigate to the **Properties** tab of the SCA import that you want to connect to the Open SCA service.
 - b. In the **Module name** field, enter the component name from step 1d.
 - c. In the **Export name** field, enter the service name from step 1f.
 - d. Save your work by pressing Ctrl+S.

Invoking SCA modules from Open SCA services

Open SCA applications developed in a Rational Application Developer environment can invoke SCA applications developed with WebSphere Integration Developer. This section provides an example of invoking an SCA module (by way of an SCA export binding) from an Open SCA service.

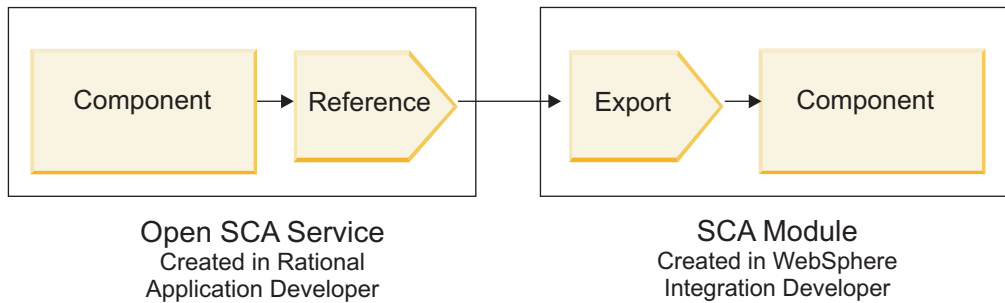


Figure 13. Open SCA service invoking component in SCA module

To connect to an SCA component by way of an Open SCA reference binding, you provide the module name and export name.

1. To obtain the name of the target module and export, perform the following steps:
 - a. In WebSphere Integration Developer, open the module in the assembly editor by double-clicking the module.
 - b. Click the export.
 - c. In the **Name** field of the **Properties** tab, note the name of the export.
2. Configure the Open SCA reference that you want to connect to the WebSphere Process Server module and export:
 - a. In Rational Application Developer, open the composite editor by double-clicking the composite diagram that contains the component and service.
 - b. Click the reference icon of the component reference to display the reference properties in the **Properties** tab.
 - c. Click the **Binding** tab on the left side of the page.
 - d. Click **Bindings** and then click **Add**.
 - e. Select the **SCA** binding.
 - f. In the **Uri** field, enter the WebSphere Process Server module name, followed by a slash ("/"), followed by the export name (which you determined in step 1c).
 - g. Click **OK**.
 - h. Save your work by pressing Ctrl+S.

Invoking services over interoperable bindings

The following bindings are supported for interoperability with an Open SCA service.

- SCA binding

When a WebSphere Process Server SCA module invokes an Open SCA service by way of an SCA import binding, the following invocation styles are supported:

- Asynchronous (one-way)
- Synchronous (request/response)

The SCA import interface and the Open SCA service interface must use a Web services interoperability (WS-I) compliant WSDL interface.

Note that the SCA binding supports transaction and security context propagation.

- Web service (JAX-WS) binding with either the SOAP1.1/HTTP or SOAP1.2/HTTP protocol
The SCA import interface and the Open SCA service interface must use a Web services interoperability (WS-I) compliant WSDL interface.
In addition, the following qualities of service are supported:
 - Web Services Atomic Transaction
 - Web Services Security
- EJB binding
A Java interface is used to define the interaction between an SCA module and an Open SCA service when the EJB binding is used.
Note that the EJB binding supports transaction and security context propagation.
- JMS bindings
The SCA import interface and the Open SCA service interface must use a Web services interoperability (WS-I) compliant WSDL interface.
The following JMS providers are supported:
 - WebSphere Platform Messaging (JMS Binding)
 - WebSphere MQ (MQ JMS Binding)

Note: Business Graphs are not interoperable across any SCA bindings and, therefore, are not supported in interfaces used to interoperate with the WebSphere Application Server Feature Pack for Service Component Architecture.

Binding types

You use protocol-specific *bindings* with imports and exports to specify the means of transporting data into or out of a module.

Selecting appropriate bindings

There are various bindings that are available to suit the needs of your application.

The bindings available in WebSphere Integration Developer provide a range of choices. This list helps you identify when one type of binding could be more suitable for the needs of your application.

Consider an *SCA* binding when these factors are applicable:

- All services are contained in WebSphere Integration Developer modules; that is, there are no external services.
- You want to separate function into different SCA modules that interact directly with each other.
- The modules are tightly coupled.

Consider a *Web Service* binding when these factors are applicable:

- You need to access an external service over the Internet or provide a service over the Internet.
- The services are loosely coupled.
- Synchronous communication is preferred; that is, a request from one service can wait for a response from another.
- The protocol of the external service you are accessing or the service you want to provide is SOAP/HTTP or SOAP/JMS.

Consider an *HTTP* binding when these factors are applicable:

- You need to access an external service over the Internet or provide a service over the Internet and you are working with other Web services based on the HTTP model (that is, using well-known HTTP interface operations such as GET, PUT, DELETE, and so on).
- The services are loosely coupled.
- Synchronous communication is preferred; that is, a request from one service can wait for a response from another.

Consider an *EJB* binding when these factors are applicable:

- The binding is for an imported service that is itself an EJB or that needs to be accessed by EJB clients.
- The imported service is loosely coupled.
- Stateful EJB interactions are not required.
- Synchronous communication is preferred; that is, a request from one service can wait for a response from another.

Consider an *EIS* binding when these factors are applicable:

- You need to access a service on an EIS system using a resource adapter.
- Synchronous data transmission is preferred over asynchronous.

Consider a *JMS* binding when these factors are applicable:

Note: There are several types of JMS bindings. If you expect to exchange SOAP messages using JMS, consider the Web service binding with the SOAP/JMS protocol. See “Web service bindings” on page 31.

- You need to access a messaging system.
- The services are loosely coupled.
- Asynchronous data transmission is preferred over synchronous.

Consider a *Generic JMS* binding when these factors are applicable:

- You need to access a non-IBM vendor messaging system.
- The services are loosely coupled.
- Reliability is more important than performance; that is, asynchronous data transmission is preferred over synchronous.

Consider an *MQ* binding when these factors are applicable:

- You need to access a WebSphere MQ messaging system and need to use the MQ native functions.
- The services are loosely coupled.
- Reliability is more important than performance; that is, asynchronous data transmission is preferred over synchronous.

Consider an *MQ JMS* binding when these factors are applicable:

- You need to access a WebSphere MQ messaging system but can do so within a JMS context; that is, the JMS subset of functions is sufficient for your application.
- The services are loosely coupled.
- Reliability is more important than performance; that is, asynchronous data transmission is preferred over synchronous.

SCA bindings

A Service Component Architecture (SCA) binding lets a service communicate with other services in other modules. An import with an SCA binding lets you access a service in another SCA module. An export with an SCA binding lets you offer a service to other modules.

You use WebSphere Integration Developer to generate and configure SCA bindings on imports and exports in SCA modules.

If modules are running on the same server or are deployed in the same cluster, an SCA binding is the easiest and fastest binding to use.

After the module that contains the SCA binding is deployed to the server, you can use the administrative console to view information about the binding or, in the case of an import binding, to change selected properties of the binding.

Web service bindings

A Web service binding is the means of transmitting messages from a Service Component Architecture (SCA) component to a Web service (and vice versa).

Web service bindings overview

A Web service import binding allows you to call an external Web service from your Service Component Architecture (SCA) components. A Web service export binding allows you to expose your SCA components to clients as Web services.

With a Web service binding, you access external services using interoperable SOAP messages and qualities of service (QoS).

You use WebSphere Integration Developer to generate and configure Web service bindings on imports and exports in SCA modules. The following types of Web service bindings are available:

- SOAP1.2/HTTP and SOAP1.1/HTTP

These bindings are based on Java API for XML Web Services (JAX-WS), a Java programming API for creating Web services.

- Use SOAP1.2/HTTP if your Web service conforms to the SOAP 1.2 specification.
- Use SOAP1.1/HTTP if your Web service conforms to the SOAP 1.1 specification.

When you select one of these bindings, you can send attachments with your SOAP messages.

The Web service bindings work with standard SOAP messages. Using one of the Web service JAX-WS bindings, however, you can customize the way that SOAP messages are parsed or written. For example, you can handle nonstandard elements in SOAP messages or apply additional processing to the SOAP message. When you configure the binding, you specify a custom data handler that performs this processing on the SOAP message.

- SOAP1.1/HTTP

Use this binding if you want to create Web services that use a SOAP-encoded message based on Java API for XML-based RPC (JAX-RPC).

- SOAP1.1/JMS

Use this binding to send or receive SOAP messages using a Java Message Service (JMS) destination.

Regardless of the transport (HTTP or JMS) that is used to convey the SOAP message, Web service bindings always handle request/response interactions synchronously. The thread making the invocation on the service provider is blocked until a response is received from the provider. See “Synchronous invocation” for more information about this invocation style.

Important: The following combinations of Web service bindings cannot be used on exports in the same module. If you need to expose components using more than one of these export bindings, you need to have each in a separate module and then connect those modules to your components using the SCA binding:

- SOAP 1.1/JMS and SOAP 1.1/HTTP using JAX-RPC
- SOAP 1.1/HTTP using JAX-RPC and SOAP 1.1/HTTP using JAX-WS
- SOAP 1.1/HTTP using JAX-RPC and SOAP 1.2/HTTP using JAX-WS

After the SCA module that contains the Web service binding is deployed to the server, you can use the administrative console to view information about the binding or to change selected properties of the binding.

Note: Web services allow applications to interoperate by using standard descriptions of services and standard formats for the messages they exchange. For example, the Web service import and export bindings can interoperate with services that are implemented using Web Services Enhancements (WSE) Version 3.5 and Windows® Communication Foundation (WCF) Version 3.5 for Microsoft® .NET. When interoperating with such services, you must ensure that:

- The Web Services Description Language (WSDL) file that is used to access a Web service export includes a non-empty SOAP action value for each operation in the interface.
- The Web service client sets either the SOAPAction header or the wsa:Action header when sending messages to a Web service export.

SOAP header propagation

When handling SOAP messages, you might need to access information from certain SOAP headers in messages that are received, ensure that messages with SOAP headers are sent with specific values, or allow SOAP headers to pass across a module.

When you configure a Web service binding in WebSphere Integration Developer, you can indicate that you want SOAP headers to be propagated.

- When requests are received at an export or responses received at an import, the SOAP header information can be accessed, allowing logic in the module to be based on header values and allowing those headers to be modified.
- When requests are sent from an export or responses sent from an import, SOAP headers can be included in those messages.

The form and presence of the propagated SOAP headers might be affected by policy sets configured on the import or export, as explained in Table 9 on page 34.

To configure the propagation of SOAP headers for an import or export, you select (from the Properties view of WebSphere Integration Developer) the **Propagate Protocol Header** tab and select the options you require.

WS-Addressing header

The WS-Addressing header can be propagated by the Web service (JAX-WS) binding.

When you propagate the WS-Addressing header, be aware of the following information:

- If you enable propagation for the WS-Addressing header, the header will be propagated into the module in the following circumstances:
 - When requests are received at an export
 - When responses are received at an import
- The WS-Addressing header is not propagated into outbound messages from WebSphere Process Server (that is, the header is not propagated when requests are sent from an import or when responses are sent from the export).

WS-Security header

The WS-Security header can be propagated by both the Web service (JAX-WS) binding and the Web service (JAX-RPC) binding.

The Web services WS-Security specification describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies.

When you propagate the WS-Security header, be aware of the following information:

- If you enable propagation for the WS-Security header, the header will be propagated across the module in the following circumstances:
 - When requests are received at an export
 - When requests are sent from an import
 - When responses are received at an import
- The header will *not*, by default, be propagated when responses are sent from the export. However, if you set the JVM property `WSSECURITY.ECHO.ENABLED` to `true`, the header will be propagated when responses are sent from the export. In this case, if the WS-Security header on the request path is not modified, WS-Security headers might be automatically echoed from requests into responses.
- The exact form of the SOAP message sent from an import for a request or from an export for a response might not exactly match the SOAP message that was originally received. For this reason, any digital signature should be assumed to become invalid. If a digital signature is required in messages that are sent, it must be established using the appropriate security policy set, and WS-Security headers relating to digital signature in received messages should be removed within the module.

To propagate the WS-Security header, you must include the WS-Security schema with the application module. See “Including the WS-Security schema in an application module” on page 34 for the procedure to include the schema.

How headers are propagated

The way that headers are propagated depends on the security policy setting on the import or export binding, as shown in the Table 9 on page 34:

Table 9. How security headers are passed

	Export binding with no security policy	Export binding with security policy
Import binding with no security policy	<p>Security headers are passed as-is through the module. They are not decrypted.</p> <p>The headers are sent outbound in the same form in which they were received.</p> <p>The digital signature might become invalid.</p>	<p>Security headers are decrypted and passed through the module with signature verification and authentication.</p> <p>The decrypted headers are sent outbound.</p> <p>The digital signature might become invalid.</p>
Import binding with security policy	<p>Security headers are passed as-is through the module. They are not decrypted.</p> <p>The headers should not be propagated to the import. Otherwise, an error occurs because of duplication.</p>	<p>Security headers are decrypted and passed through the module with signature verification and authentication.</p> <p>The headers should not be propagated to the import. Otherwise, an error occurs because of duplication.</p>

Configure appropriate policy sets on the export and import bindings, because this isolates the service requester from changes to the configuration or QoS requirements of the service provider. Having standard SOAP headers visible in a module can then be used to influence the processing (for example, logging and tracing) in the module. Propagating SOAP headers across a module from a received message to a sent message does mean that the isolation benefits of the module are reduced.

Standard headers, such as WS-Security headers, should not be propagated on a request to an import or response to an export when the import or export has an associated policy set that would normally result in the generation of those headers. Otherwise, an error will occur because of a duplication of the headers. Instead, the headers should be explicitly removed or the import or export binding should be configured to prevent propagation of protocol headers.

Accessing SOAP headers

When a message that contains SOAP headers is received from a Web service import or export, the headers are placed in the headers section of the service message object (SMO). You can access the header information, as described in “Accessing SOAP header information in the SMO”.

Including the WS-Security schema in an application module

The following procedure outlines the steps for including the schema in the application module:

- If the computer on which WebSphere Integration Developer is running has access to the Internet, perform the following steps:
 1. In the Business Integration perspective, select **Dependencies** for your project.

2. Expand **Predefined Resources** and select either **WS-Security 1.0 schema files** or **WS-Security 1.1 schema files** to import the schema into your module.
 3. Clean and rebuild the project.
- If a computer on which WebSphere Integration Developer is running does not have Internet access, you can download the schema to a second computer that does have Internet access. You can then copy it to the computer on which WebSphere Integration Developer is running.
 1. From the computer that has Internet access, download the remote schema:
 - a. Click **File** → **Import** → **Business Integration** → **WSDL and XSD**.
 - b. Select **Remote WSDL** or **XSD file**.
 - c. Import the following schemas:
 - `http://www.w3.org/2003/05/soap-envelope/`
 - `http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd`
 - `http://www.w3.org/TR/xmlsig-core/xmlsig-core-schema.xsd`
 2. Copy the schemas to the computer that does not have Internet access.
 3. From the computer that has no Internet access, import the schema:
 - a. Click **File** → **Import** → **Business Integration** → **WSDL and XSD**.
 - b. Select **Local WSDL** or **XSD file**.
 4. Change the schema locations for `oasis-wss-wssecurity-secext-1.1.xsd`:
 - a. Open the schema at `workplace_location/module_name/StandardImportFilesGen/oasis-wss-wssecurity-secext-1.1.xsd`.
 - b. Change:


```
<xs:import namespace='http://www.w3.org/2003/05/soap-envelope'
  schemaLocation='http://www.w3.org/2003/05/soap-envelope/' />
to:
<xs:import namespace='http://www.w3.org/2003/05/soap-envelope'
  schemaLocation='../w3/_2003/_05/soap_envelope.xsd' />
```
 - c. Change:


```
<xs:import namespace='http://www.w3.org/2001/04/xmlenc#'
  schemaLocation='http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd' />
to:
<xs:import namespace='http://www.w3.org/2001/04/xmlenc#'
  schemaLocation='../w3/tr/_2002/rec_xmlenc_core_20021210/xenc-schema.xsd' />
```
 5. Change the schema location for `oasis-200401-wss-wssecurity-secext-1.0.xsd`:
 - a. Open the schema at `workplace_location/module_name/StandardImportFilesGen/oasis-200401-wss-wssecurity-secext-1.0.xsd`.
 - b. Change:


```
<xsd:import namespace="http://www.w3.org/2000/09/xmlsig#"
  schemaLocation="http://www.w3.org/TR/xmlsig-core/xmlsig-core-schema.xsd" />
to:
<xsd:import namespace="http://www.w3.org/2000/09/xmlsig#"
  schemaLocation="../w3/tr/_2002/rec_xmlsig_core_20020212/xmlsig-core-schema.xsd" />
```
 6. Clean and rebuild the project.

Attachments in SOAP messages

You can send and receive SOAP messages that include binary data (such as PDF files or JPEG images) as attachments. Attachments can be *referenced* (that is, represented explicitly as message parts in the service interface) or *unreferenced* (in which arbitrary numbers and types of attachments can be included).

A referenced attachment can be represented in one of the following ways:

- As a wsi:swaRef-typed element in the message schema
Attachments defined using the wsi:swaRef type conform to the Web Services Interoperability Organization (WS-I) *Attachments Profile Version 1.0* (<http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html>), which defines how message elements are related to MIME parts.
- As a top-level message part, using a binary schema type
Attachments represented as top-level message parts conform to the *SOAP Messages with Attachments* (<http://www.w3.org/TR/SOAP-attachments>) specification.

An unreferenced attachment is carried in a SOAP message without any representation in the message schema.

In all cases, the WSDL SOAP binding should include a MIME binding for attachments to be used, and the maximum size of the attachments should not exceed 20 MB.

Note: To send or receive SOAP messages with attachments, you must use one of the Web service bindings based on the Java API for XML Web Services (JAX-WS).

Referenced attachments: swaRef-typed elements:

You can send and receive SOAP messages that include attachments represented in the service interface as swaRef-typed elements.

An swaRef-typed element is defined in the Web Services Interoperability Organization (WS-I) *Attachments Profile Version 1.0* (<http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html>), which defines how message elements are related to MIME parts.

Note: The SOAP messages produced or consumed are not guaranteed to conform to the WS-I Attachments Profile. In particular, the “content-id part encoding,” as described in section 3.8 of the WS-I Attachments Profile 1.0, is not supported.

In the SOAP message, the SOAP body contains an swaRef-typed element that identifies the content ID of the attachment.

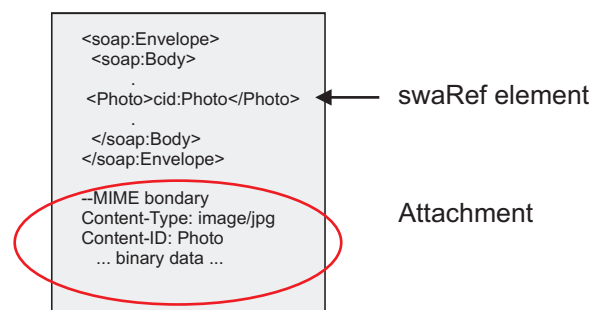


Figure 14. A SOAP message with an swaRef element

The WSDL for this SOAP message contains an swaRef-typed element within a message part that identifies the attachment.

```
<element name="sendPhoto">
  <complexType>
    <sequence>
```

```

        <element name="Photo" type="wsi:swaRef"/>
    </sequence>
</complexType>
</element>

```

The WSDL should also contain a MIME binding that indicates MIME multipart messages are to be used.

Note: The WSDL does *not* include a MIME binding for the specific swaRef-typed message element, because MIME bindings apply only to top-level message parts.

Attachments represented as swaRef-typed elements can be propagated only across mediation flow components. If an attachment must be accessed by or propagated to another component type, use a mediation flow component to move the attachment to a location that is accessible by that component.

Inbound processing of attachments

You use WebSphere Integration Developer to configure an export binding to receive the attachment. You create a module and its associated interface and operations, including an element of type swaRef. You then create a Web service (JAX-WS) binding.

Note: See the “Working with attachments” topic in the WebSphere Integration Developer information center for more detailed information.

When a client passes a SOAP message with an swaRef attachment to a Service Component Architecture (SCA) component, the Web service (JAX-WS) export binding first removes the attachment. It then parses the SOAP part of the message and creates a business object. Finally, the binding sets the content ID of the attachment in the business object.

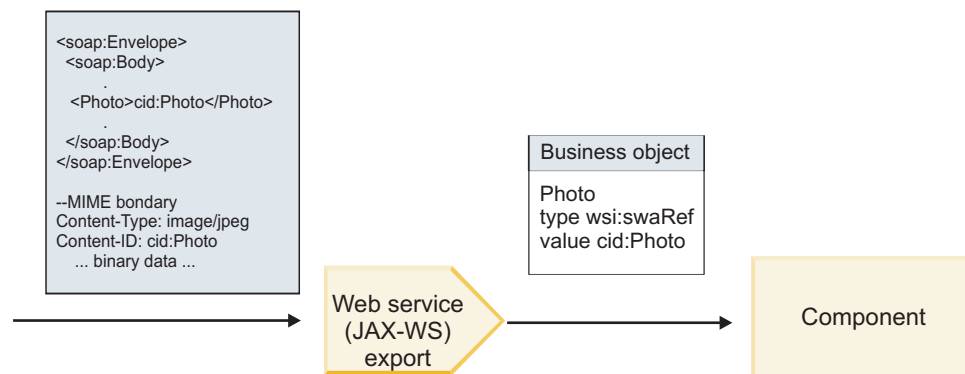


Figure 15. How the Web service (JAX-WS) export binding processes a SOAP message with an swaRef attachment

Accessing attachment metadata in a mediation flow component

As shown in Figure 16 on page 38, when swaRef attachments are accessed by components, the attachment content identifier appears as an element of type swaRef.

Each attachment of a SOAP message also has a corresponding attachments element in the SMO. When using the WS-I swaRef type, the attachments element includes the attachment content type and content ID as well as the actual binary data of the attachment.

To obtain the value of an swaRef attachment, it is therefore necessary to obtain the value of the swaRef-typed element, and then locate the attachments element with the corresponding contentID value. Note that the contentID value typically has the cid: prefix removed from the swaRef value.

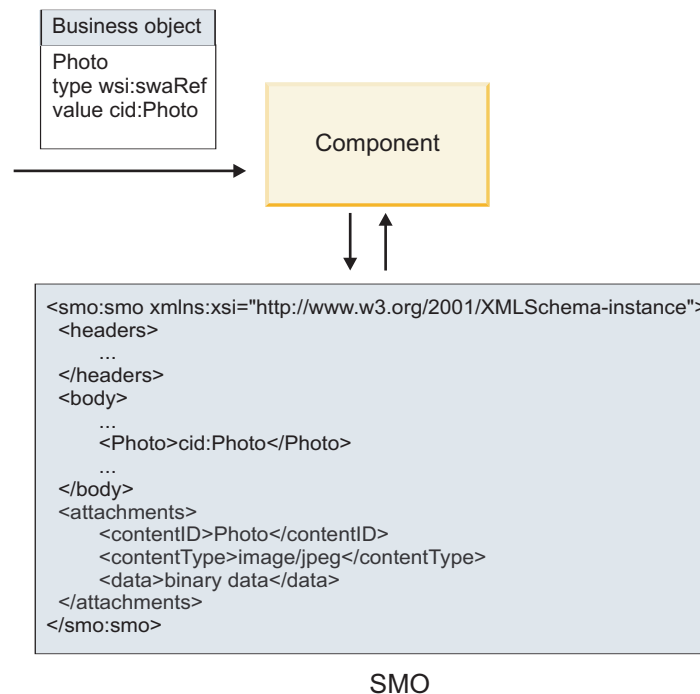


Figure 16. How swaRef attachments appear in the SMO

Outbound processing

You use WebSphere Integration Developer to configure a Web service (JAX-WS) import binding to invoke an external Web service. The import binding is configured with a WSDL document that describes the Web service to be invoked and defines the attachment that will be passed to the Web service.

When an SCA message is received by a Web service (JAX-WS) import binding, swaRef-typed elements are sent as attachments if the import is wired to a mediation flow component and the swaRef-typed element has a corresponding attachments element.

For outbound processing, swaRef-typed elements are always sent with their content ID values; however the mediation module must ensure that there is a corresponding attachments element with a matching contentID value.

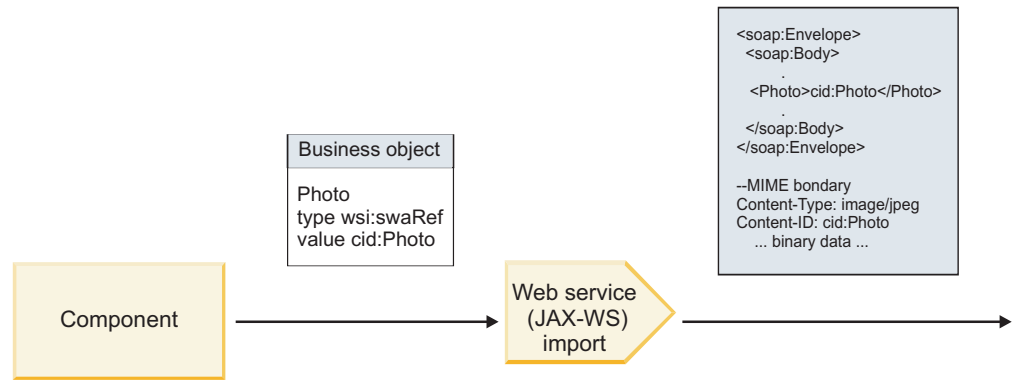


Figure 17. How the Web service (JAX-WS) import binding generates a SOAP message with an swaRef attachment

Setting attachment metadata in a mediation flow component

If, in the SMO, there is an swaRef-typed element value and an attachments element, the binding prepares the SOAP message (with the attachment) and sends it to a recipient.

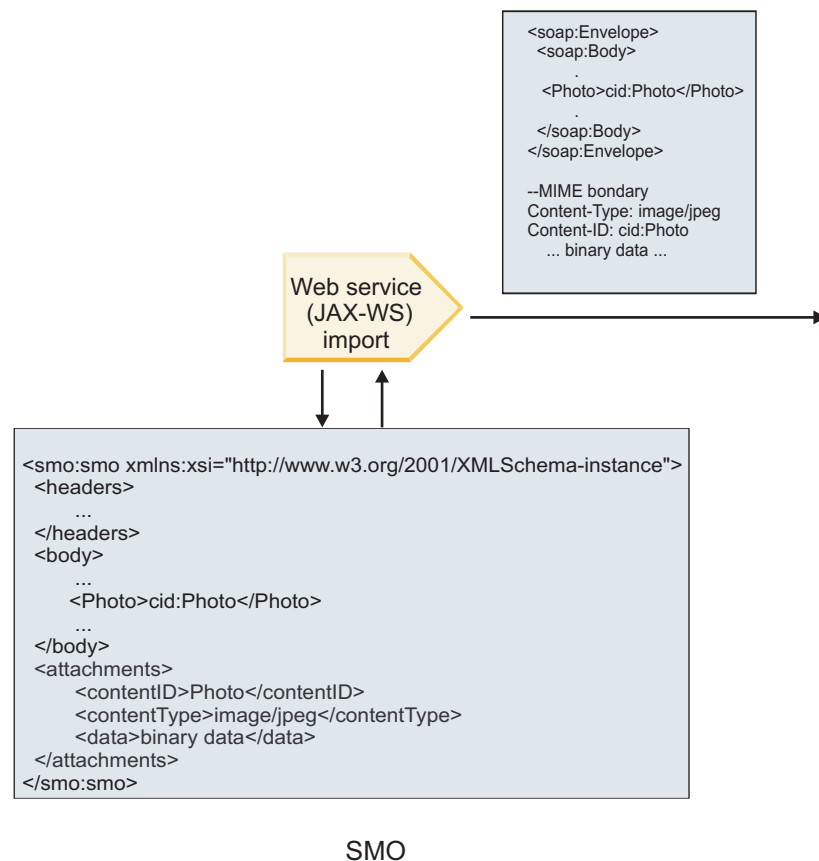


Figure 18. How an swaRef attachment in the SMO is accessed to create the SOAP message

The attachments element is present in the SMO only if a mediation flow component is connected directly to the import or export; it does not get passed across other component types. If the values are needed in a module that contains other component types, a mediation flow component should be used to copy the

values into a location where they can then be accessed in the module, and another mediation flow component used to set the correct values before an outbound invocation by way of a Web service import.

Important: As described in “XML representation of SMO,” the XSL Transformation mediation primitive transforms messages using an XSLT 1.0 transformation. The transformation operates on an XML serialization of the SMO. The XSL Transformation mediation primitive allows the root of the serialization to be specified, and the root element of the XML document reflects this root.

When you are sending SOAP messages with attachments, the root element you choose determines how attachments are propagated.

- If you use “/body” as the root of the XML map, all attachments are propagated across the map by default.
- If you use “/” as the root of the map, you can control the propagation of attachments.

Referenced attachments: top-level message parts:

You can send and receive SOAP messages that include binary attachments that are declared as parts in your service interface.

In a MIME multipart SOAP message, the SOAP body is the first part of the message, and the attachments are in subsequent parts. References to the attachments are included in the SOAP body.

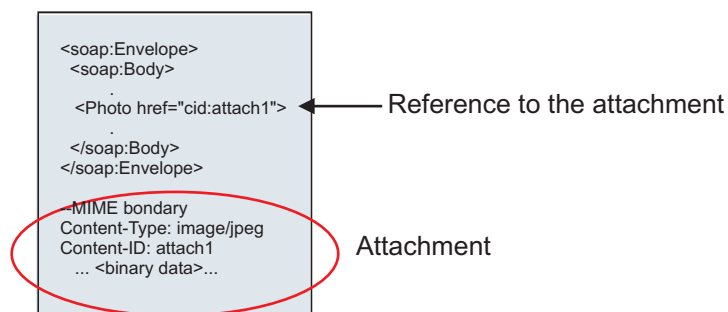


Figure 19. A SOAP message with a referenced attachment

What is the advantage of sending or receiving a referenced attachment in a SOAP message? The binary data that makes up the attachment (which is often quite large) is held separately from the SOAP message body so that it does not need to be parsed as XML. This results in more efficient processing than if the binary data were held within an XML element.

Inbound processing of referenced attachments

You use WebSphere Integration Developer to configure the export binding. You create a module and its associated interface and operations. You then create a Web service (JAX-WS) binding. The Referenced attachments page displays all the binary parts from the created operation, and you select which parts will be attachments.

Note: Only top-level message parts (that is, elements defined in the WSDL portType as parts within the input or output message) that have a binary type (either base64Binary or hexBinary) or can be sent or received as referenced attachments.

See the “Working with attachments” topic in the WebSphere Integration Developer information center for more detailed information.

When a client passes a SOAP message with an attachment to a Service Component Architecture (SCA) component, the Web service (JAX-WS) export binding first removes the attachment. It then parses the SOAP part of the message and creates a business object. Finally, the binding sets the attachment binary in the business object.

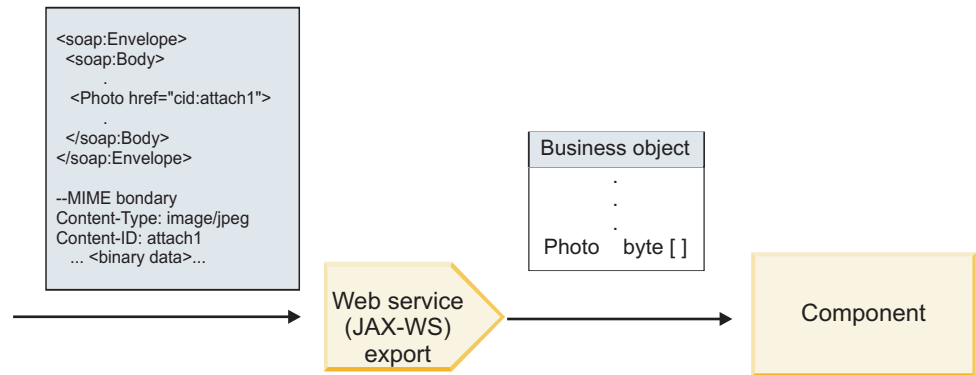


Figure 20. How the Web service (JAX-WS) export binding processes a SOAP message with a referenced attachment

Accessing attachment metadata in a mediation flow component

As shown in Figure 20, when referenced attachments are accessed by components, the attachment data appears as a byte array.

Each referenced attachment of a SOAP message also has a corresponding attachments element in the SMO. The attachments element includes the attachment content type and the path to the message body element where the attachment is held.

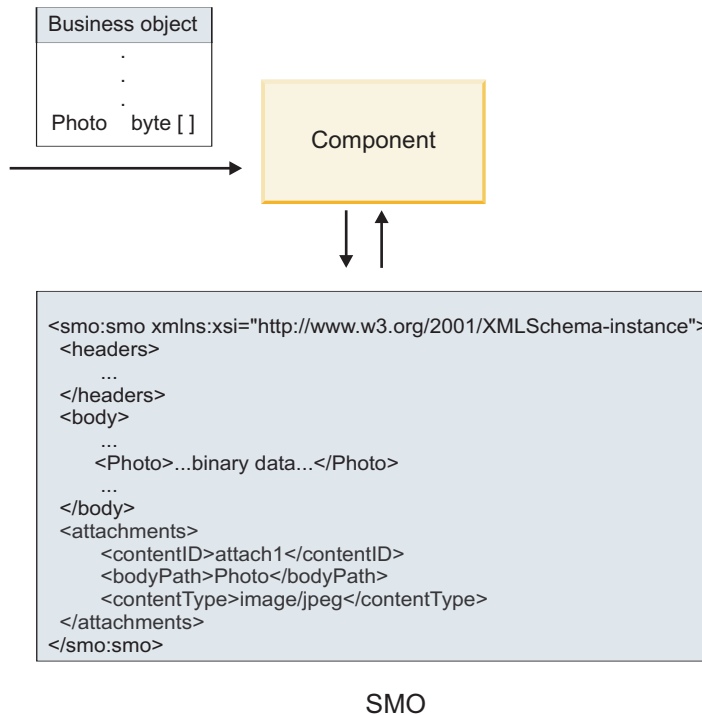


Figure 21. How referenced attachments appear in the SMO

Important: The path to the message body element is not automatically updated if the message is transformed and the attachment moved. You can use a mediation flow to update the attachments element with the new path (for example, as part of the transform or using a separate message element setter).

Outbound processing of referenced attachments

You use WebSphere Integration Developer to configure a Web service (JAX-WS) import binding to invoke an external Web service. The import binding is configured with a WSDL document that describes the Web service to be invoked and defines which message parts should be passed as attachments.

Note: The part that represents an attachment, as defined in the WSDL, must be a simple type (either base64Binary or hexBinary). If a part is defined by a complexType, that part is not treated as an attachment.

The import binding uses information in the SMO to determine how the binary top-level message parts are sent as attachments.

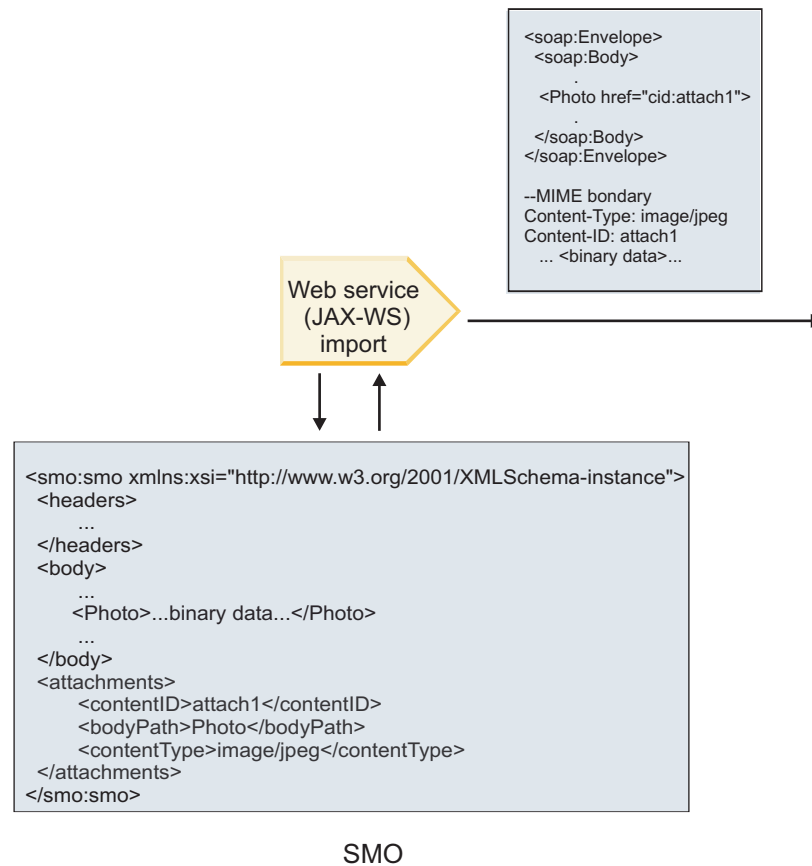


Figure 22. How the referenced attachment in the SMO is accessed to create the SOAP message

The attachments element is present in the SMO only if a mediation flow component is connected directly to the import or export; it does not get passed across other component types. If the values are needed in a module that contains other component types, a mediation flow component should be used to copy the values into a location where they can then be accessed in the module, and another mediation flow component used to set the correct values before an outbound invocation by way of a Web service import.

The binding uses a combination of the following conditions to determine how (or whether) the message is sent:

- Whether there is a WSDL MIME binding for the top-level binary message part and, if so, how the content type is defined
- Whether there is an attachments element in the SMO whose bodyPath value references a top-level binary part

How attachments are created when an attachment element exists in the SMO

The following table shows how an attachment is created and sent if the SMO contains an attachment element with a bodyPath that matches a message name part:

Table 10. How the attachment is generated

Status of WSDL MIME binding for top-level binary message part	How message is created and sent
Present with one of the following: <ul style="list-style-type: none"> • No defined content type for the message part • Multiple content types defined • Wildcard content type defined 	Message part is sent as an attachment. Content-Id is set to the value in the attachments element if present; otherwise, one is generated. Content-Type is set to the value in the attachments element if present; otherwise, it is set to application/octet-stream.
Present with single, non-wildcard content for the message part	Message part is sent as an attachment. Content-Id is set to the value in the attachments element if present; otherwise, one is generated. Content-Type is set to the value in the attachments element if present; otherwise, it is set to the type defined in the WSDL MIME content element.
Not present	Message part is sent as an attachment. Content-Id is set to the value in the attachments element if present; otherwise, one is generated. Content-Type is set to the value in the attachments element if present; otherwise, it is set to application/octet-stream. Note: Sending message parts as attachments when not defined as such in the WSDL may break compliance with the WS-I Attachments Profile 1.0 and so should be avoided if possible.

How attachments are created when no attachment element exists in the SMO

The following table shows how an attachment is created and sent if the SMO does not contain an attachment element with a bodyPath that matches a message name part:

Table 11. How the attachment is generated

Status of WSDL MIME binding for top-level binary message part	How message is created and sent
Present with one of the following: <ul style="list-style-type: none"> • No defined content type for the message part • Multiple content types defined • Wildcard content type defined 	Message part is sent as an attachment. Content-Id is generated. Content-Type is set to application/octet-stream.
Present with single, non-wildcard content for the message part	Message part is sent as an attachment. Content-Id is generated. Content-Type is set to the type defined in the WSDL MIME content element.

Table 11. How the attachment is generated (continued)

Status of WSDL MIME binding for top-level binary message part	How message is created and sent
Not present	Message part is not sent as an attachment.

Important: As described in “XML representation of SMO,” the XSL Transformation mediation primitive transforms messages using an XSLT 1.0 transformation. The transformation operates on an XML serialization of the SMO. The XSL Transformation mediation primitive allows the root of the serialization to be specified, and the root element of the XML document reflects this root.

When you are sending SOAP messages with attachments, the root element you choose determines how attachments are propagated.

- If you use “/body” as the root of the XML map, all attachments are propagated across the map by default.
- If you use “/” as the root of the map, you can control the propagation of attachments.

Unreferenced attachments:

You can send and receive *unreferenced* attachments that are not declared in the service interface.

In a MIME multipart SOAP message, the SOAP body is the first part of the message, and the attachments are in subsequent parts. No reference to the attachment is included in the SOAP body.

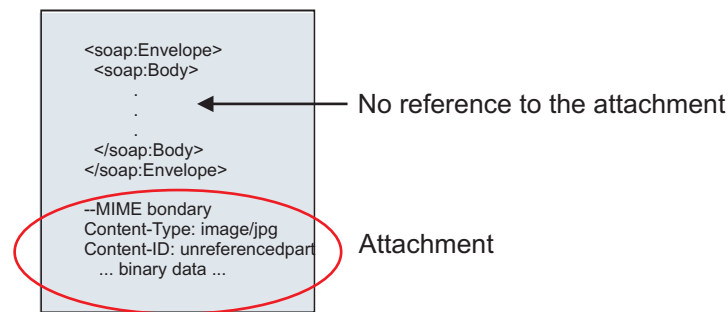


Figure 23. A SOAP message with an unreferenced attachment

You can send a SOAP message with an unreferenced attachment through a Web service export to a Web service import. The output message, which is sent to the target Web service, contains the attachment.

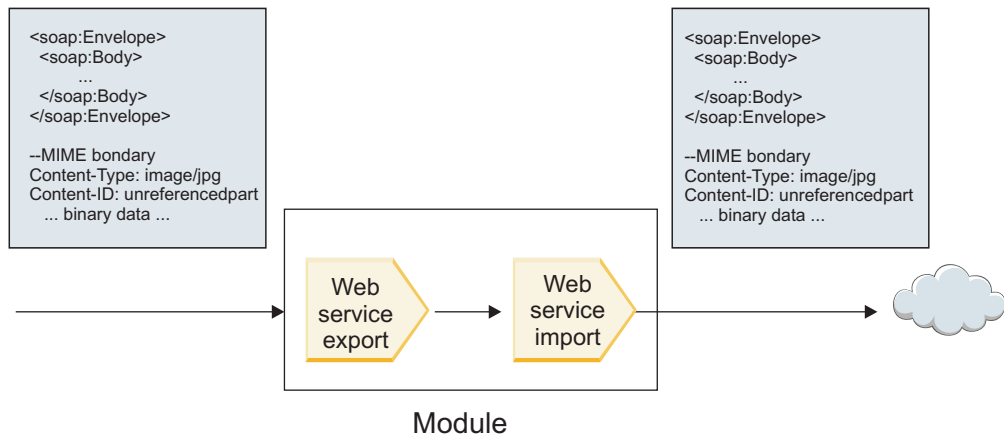


Figure 24. An attachment passing through an SCA module

In Figure 24, the SOAP message, with the attachment, passes through without modification.

You can also modify the SOAP message by using a mediation flow component. For example, you can use the mediation flow component to extract data from the SOAP message (binary data in the body of the message, in this case) and create a SOAP with attachments message. The data is processed as part of the attachments element of a service message object (SMO).

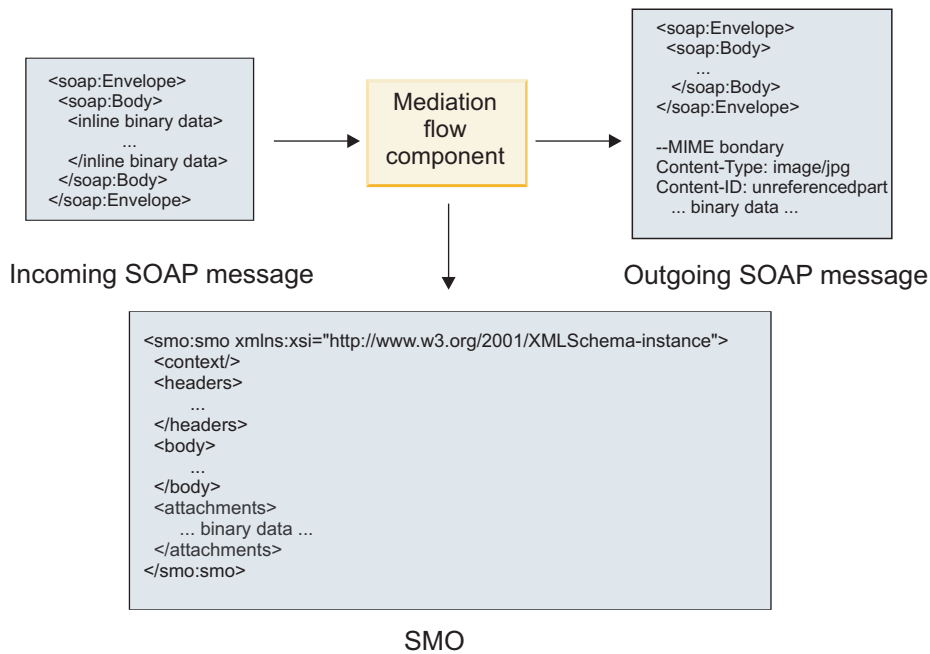


Figure 25. A message processed by a mediation flow component

Conversely, the mediation flow component can transform the incoming message by extracting and encoding the attachment and then transmitting the message with no attachments.

Instead of extracting data from an incoming SOAP message to form a SOAP with attachments message, you can obtain the attachment data from an external source,

such as a database.

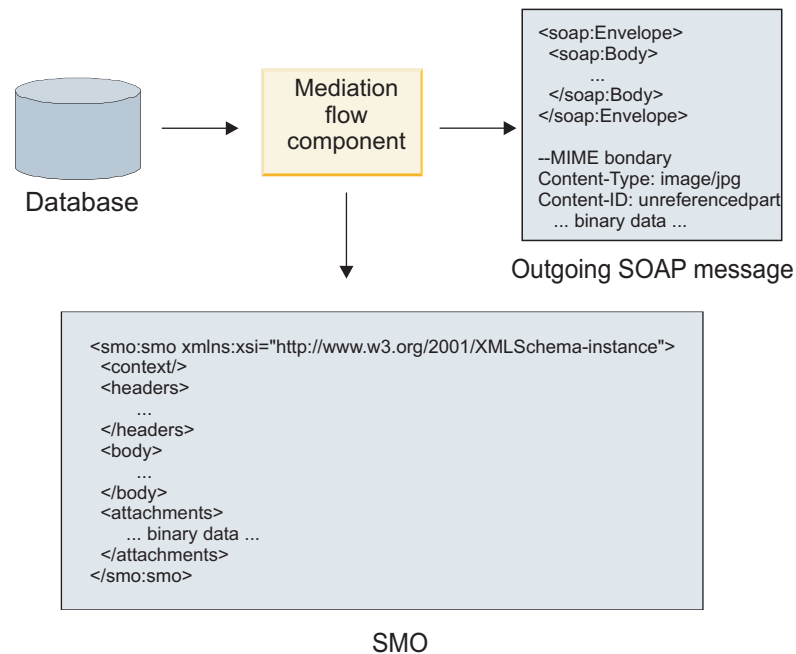


Figure 26. An attachment obtained from a database and added to the SOAP message

Conversely, the mediation flow component can extract the attachment from an incoming SOAP message and process the message (for example, store the attachment in a database).

Unreferenced attachments can be propagated only across mediation flow components. If an attachment must be accessed by or propagated to another component type, use a mediation flow component to move the attachment to a location that is accessible by that component.

Important: As described in “XML representation of SMO,” the XSL Transformation mediation primitive transforms messages using an XSLT 1.0 transformation. The transformation operates on an XML serialization of the SMO. The XSL Transformation mediation primitive allows the root of the serialization to be specified, and the root element of the XML document reflects this root.

When you are sending SOAP messages with attachments, the root element you choose determines how attachments are propagated.

- If you use “/body” as the root of the XML map, all attachments are propagated across the map by default.
- If you use “/” as the root of the map, you can control the propagation of attachments.

Use of WSDL document style binding with multipart messages

The Web Services Interoperability Organization (WS-I) organization has defined a set of rules regarding how Web services should be described by way of a WSDL and how the corresponding SOAP messages should be formed, in order to ensure interoperability.

These rules are specified in the *WS-I Basic Profile Version 1.1* (<http://www.ws-i.org/Profiles/BasicProfile-1.1.html>).

In particular, for a document style SOAP binding, WS-I profile conformance requires that, in a WSDL document, for an operation that uses the document style, only a single message part is ever bound to the SOAP body, and that the SOAP message corresponding to this contains a single child element matching the part that was so bound.

This means that, when using a document style SOAP binding for an operation with messages (input, output, or fault) are defined with multiple parts, only one of those parts should be bound to the SOAP body in order to be compliant with the WS-I Basic Profile 1.1.

The following approach is used when WSDL descriptions are generated for exports with Web service (JAX-WS and JAX-RPC) bindings in this case:

- The first message part is bound to the SOAP body.
- For the JAX-WS binding, all other message parts of type "hexBinary" or "base64Binary" are bound as referenced attachments. See "Referenced attachments: top-level message parts" on page 40.
- All other message parts are bound as SOAP headers.

The JAX-RPC and JAX-WS import bindings honor the SOAP binding in an existing WSDL document with multipart document style messages even if it does bind multiple parts to the SOAP body; however, you are not able to generate Web service clients for such WSDL documents in Rational Application Developer.

Note: The JAX-RPC binding does not support attachments.

The recommended pattern when using multipart messages with an operation that has document style SOAP binding is therefore:

1. Use the document/literal wrapped style. In this case, messages always have a single part; however, attachments can be unreferenced (as described in "Unreferenced attachments" on page 45) or swaRef-typed (as described in "Referenced attachments: swaRef-typed elements" on page 36) in this case.
2. Use the RPC/literal style. In this case, there are no restrictions on the WSDL binding in terms of number of parts bound to the SOAP body; the SOAP message that results always has a single child that represents the operation being invoked, with the message parts being children of that element.
3. For the JAX-WS binding, have the first message part be one that is not of type "hexBinary" or "base64Binary" and all other parts of one of those two types. This is then all bound as attachments.
4. Any other cases are subject to the behavior described.

Note: When receiving multipart document-style SOAP messages with referenced attachments, the JAX-WS binding expects each referenced attachment to be represented by a SOAP body child element with an href attribute value which identifies the attachment by its content ID. The JAX-WS binding sends referenced attachments for such messages in the same way. This behavior is not compliant with the WS-I Basic Profile. The WS-I Attachments Profile defines a "content-id part encoding," which allows the child element with the href attribute to be omitted and therefore makes such messages compliant with the Basic Profile. The JAX-WS binding does not support sending or receiving messages that use the content-id part encoding. To ensure that your messages do comply, follow approach 1 or 2 in the previous list or avoid the use of referenced attachments for such messages and use unreferenced or swaRef-typed attachments instead.

HTTP bindings

The HTTP binding is designed to provide Service Component Architecture (SCA) connectivity to HTTP. Consequently, existing or newly-developed HTTP applications can participate in Service Oriented Architecture (SOA) environments.

Hypertext Transfer Protocol (HTTP) is a widely-used protocol for transferring information on the Web. When you are working with an external application that uses the HTTP protocol, an HTTP binding is necessary. The HTTP binding handles the transformation of data passed in as a message in a native format to a business object in an SCA application. The HTTP binding also can transform data passed out as a business object to the native format expected by the external application. for an incoming messaging.

Note: If you want to interact with clients and services that use the Web services SOAP/HTTP protocol, consider using one of the Web service bindings, which provide additional functionality with respect to handling Web services standard qualities of service.

Some common scenarios for using the HTTP binding are described in the following list:

- SCA-hosted services can invoke HTTP applications using an HTTP import.
- SCA-hosted services can expose themselves as HTTP-enabled applications, so they can be used by HTTP clients, using an HTTP export.
- WebSphere Process Server and WebSphere Enterprise Service Bus can communicate between themselves across an HTTP infrastructure, consequently users can manage their communications according to corporate standards.
- WebSphere Process Server and WebSphere Enterprise Service Bus can act as mediators of HTTP communications, transforming and routing messages, which improves the integration of applications using a HTTP network.
- WebSphere Process Server and WebSphere Enterprise Service Bus can be used to bridge between HTTP and other protocols, such as SOAP/HTTP Web services, Java Connector Architecture (JCA)-based resource adapters, JMS, and so on.

Detailed information about creating HTTP import and export bindings can be found in the WebSphere Integration Developer information center. See the **Developing integration applications** → **Accessing external services with HTTP**> topics.

HTTP bindings overview

The HTTP binding provides connectivity to HTTP-hosted applications. It mediates communication between HTTP applications and allows existing HTTP-based applications to be called from a module.

HTTP import bindings

The HTTP import binding provides outbound connectivity from Service Component Architecture (SCA) applications to an HTTP server or applications.

The import invokes an HTTP endpoint URL. The URL can be specified in one of three ways:

- The URL can be set dynamically in the HTTP headers by way of the dynamic override URL.
- The URL can be set dynamically in the SMO target address element.
- The URL can be specified as a configuration property on the import.

This invocation is always synchronous in nature.

Although HTTP invocations are always request-reply, the HTTP import supports both one-way and two-way operations and ignores the response in the case of a one-way operation.

HTTP export bindings

The HTTP export binding provides inbound connectivity from HTTP applications to an SCA application.

A URL is defined on the HTTP export. HTTP applications that want to send request messages to the export use this URL to invoke the export.

The HTTP export also supports pings.

HTTP bindings at runtime

An import with an HTTP binding at runtime sends a request with or without data in the body of the message from the SCA application to the external Web service. The request is made from the SCA application to the external Web service, as shown in Figure 27.



Figure 27. Flow of a request from the SCA application to the Web application

Optionally, the import with the HTTP binding can receive data back from the Web application in a response to the request.

With an export, the request is made by a client application to a Web service, as shown in Figure 28.

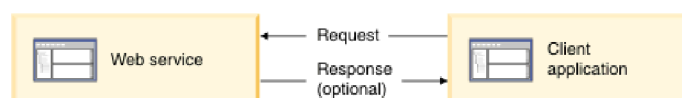


Figure 28. Flow of a request from the Web service to the client application.

The Web service is a Web application running on the server. The export is implemented in that Web application as a servlet so the client sends its request to a URL address. The servlet passes the request to the SCA application in the runtime.

Optionally, the export may send data to the client application in response to the request.

HTTP headers

HTTP import and export bindings allow configuration of HTTP headers and their values to be used for outbound messages. The HTTP import uses these headers for requests, and the HTTP export uses them for responses.

Statically configured headers and control information take precedence over values dynamically set at runtime. However, the dynamic override URL, Version, and Method control values override the static values, which are otherwise considered defaults.

The binding supports the dynamic nature of the HTTP import URL by determining the value of HTTP target URL, Version, and Method at run time. These values are determined by extracting the value of Endpoint Reference, Dynamic Override URL, Version, and Method.

- For Endpoint Reference, use `com.ibm.websphere.sca.addressing.EndpointReference` APIs or set the `/headers/SMOHeader/Target/address` field in the SMO header.
- For Dynamic Override URL, Version, and Method, use the HTTP control parameters section of the Service Component Architecture (SCA) message. Note that the Dynamic Override URL takes precedence over the target Endpoint Reference; however, the Endpoint Reference applies across bindings, so it is the preferred approach and should be used where possible.

Note: See Related Concepts for information about dynamic invocation and for specific information about the URL format, syntax, and usage.

The control and header information for outbound messages under HTTP export and import bindings is processed in the following order:

1. Header and control information excluding HTTP dynamic override URL, Version, and Method from the SCA Message (lowest priority)
2. Changes from the administrative console on the export/import level
3. Changes from the administrative console on the method level of the export or import
4. Target address specified by way of the Endpoint Reference or the SMO header
5. Dynamic Override URL, Version, and Method from the SCA message
6. Headers and control information from the data handler or data binding (highest priority)

The HTTP export and import will populate inbound direction headers and control parameters with data from the incoming message (`HTTPExportRequest` and `HTTPImportResponse`) only if protocol header propagation is set to `True`. Inversely, the HTTP export and import will read and process outbound headers and control parameters (`HTTPExportResponse` and `HTTPImportRequest`) only if protocol header propagation is set to `True`.

Note: Data handler or data binding changes to headers or control parameters in the import response or export request will not alter the processing instructions of the message inside the import or export binding and should be used only to propagate modified values to downstream SCA components.

The context service is responsible for propagating the context (including the protocol headers, such as the HTTP header, and the user context, such as account ID) along an SCA invocation path. During development in WebSphere Integration Developer, you can control the propagation of context by way of import and export properties. For more details, see the import and export bindings information in the WebSphere Integration Developer information center.

Supplied HTTP header structures and support

Table 12 itemizes the request/response parameters for HTTP Import and HTTP Export requests and responses.

Table 12. Supplied HTTP header information

Control name	HTTP Import request	HTTP Import response	HTTP Export request	HTTP Export response
URL	Ignored	Not set	Read from the request message. Note: Query string is also part of the URL control parameter.	Ignored
Version (possible values: 1.0, 1.1; default is 1.1)	Ignored	Not set	Read from the request message	Ignored
Method	Ignored	Not set	Read from the request message	Ignored
Dynamic Override URL	If set in the data handler or data binding, overrides the HTTP Import URL. Written to the message in the request line. Note: Query string is also part of the URL control parameter.	Not set	Not set	Ignored
Dynamic Override Version	If set, overrides the HTTP Import Version. Written to the message in the request line.	Not set	Not set	Ignored
Dynamic Override Method	If set, overrides the HTTP Import Method. Written to the message in the request line.	Not set	Not set	Ignored
Media Type (This control parameter carries part of the value of the Content-Type HTTP header.)	If present, written to the message as part of the Content-Type header. Note: This control element value should be provided by the data handler or data binding.	Read from the response message, Content-Type header	Read from the request message, Content-Type header	If present, written to the message as part of Content-Type header. Note: This control element value should be provided by the data handler or data binding.

Table 12. Supplied HTTP header information (continued)

Control name	HTTP Import request	HTTP Import response	HTTP Export request	HTTP Export response
Character set (default: UTF-8)	If present, written to the message as part of the Content-Type header. Note: This control element value should be provided by the data binding.	Read from the response message, Content-Type header	Read from the request message, Content-Type header	Supported; written to the message as part of the Content-Type header. Note: This control element value should be provided by the data binding.
Transfer Encoding (Possible values: chunked, identity; default is identity)	If present, written to the message as a header and controls how the message transformation is encoded.	Read from the response message	Read from the request message	If present, written to the message as a header and controls how the message transformation is encoded.
Content Encoding (Possible values: gzip, x-gzip, deflate, identity; default is identity)	If present, written to the message as a header and controls how the payload is encoded.	Read from the response message	Read from the request message	If present, written to the message as a header and controls how the payload is encoded.
Content-Length	Ignored	Read from the response message	Read from the request message	Ignored
StatusCode (default: 200)	Not supported	Read from the response message	Not supported	If present, written to the message in the response line
ReasonPhrase (default: OK)	Not supported	Read from the response message	Not supported	Control value ignored. The message response line value is generated from the StatusCode.

Table 12. Supplied HTTP header information (continued)

Control name	HTTP Import request	HTTP Import response	HTTP Export request	HTTP Export response
Authentication (contains multiple properties)	If present, used to construct the Basic Authentication header. Note: The value for this header will be encoded only on the HTTP protocol. In the SCA, it will be decoded and passed as clear text.	Not applicable	Read from the request message Basic Authentication header. The presence of this header does not indicate the user has been authenticated. Authentication should be controlled in the servlet configuration. Note: The value for this header will be encoded only on the HTTP protocol. In the SCA, it will be decoded and passed as clear text.	Not applicable
Proxy (contains multiple properties: Host, Port, Authentication)	If present, used to establish connection through proxy.	Not applicable	Not applicable	Not applicable
SSL (contains multiple properties: Keystore, Keystore Password, Trustore, Trustore Password, ClientAuth)	If populated and the destination url is HTTPS, it is used to establish a connection through SSL.	Not applicable	Not applicable	Not applicable

HTTP data bindings

For each different mapping of data between a Service Component Architecture (SCA) message and an HTTP protocol message, a data handler or an HTTP data binding must be configured. Data handlers provide a binding-neutral interface that allows reuse across transport bindings and represent the recommended approach; data bindings are specific to a particular transport binding. HTTP-specific data binding classes are supplied; you can also write custom data handlers or data bindings.

Note: The three HTTP data binding classes described in this topic (HTTPStreamDataBindingSOAP, HTTPStreamDataBindingXML, and HTTPServiceGatewayDataBinding) are deprecated as of WebSphere Process Server Version 7.0. Instead of using the data bindings described in this topic, consider the following data handlers:

- Use SOAPDataHandler instead of HTTPStreamDataBindingSOAP.
- Use UTF8XMLDataHandler instead of HTTPStreamDataBindingXML
- Use GatewayTextDataHandler instead of HTTPServiceGatewayDataBinding

Data bindings are provided for use with HTTP imports and HTTP exports: binary data binding, XML data binding, and SOAP data binding. A response data binding is not required for one-way operations. A data binding is represented by the name of a Java class whose instances can convert both from HTTP to ServiceDataObject and vice-versa. A function selector must be used on an export which, in conjunction with method bindings, can determine which data binding is used and which operation is invoked. The supplied data bindings are:

- Binary data bindings, which treat the body as unstructured binary data. The implementation of the binary data binding XSD schema is as follows:

```
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://com.ibm.websphere.http.data.bindings/schema"
  xmlns:tns="http://com.ibm.websphere.http.data.bindings/schema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="HTTPBaseBody">
    <xsd:sequence/>
  </xsd:complexType>

  <xsd:complexType name="HTTPBytesBody">
    <xsd:complexContent>
      <xsd:extension base="tns:HTTPBaseBody">
        <xsd:sequence>
          <xsd:element name="value" type="xsd:hexBinary"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

- XML data bindings, which support the body as XML data. The implementation of the XML data binding is similar to the JMS XML data binding and has no restrictions on the interface schema.
- SOAP data bindings, which support the body as SOAP data. The implementation of the SOAP data binding has no restrictions on the interface schema.

Implementing custom HTTP data bindings

This section describes how to implement a custom HTTP data binding.

Note: The recommended approach is to implement a custom data handler because it can be reused across transport bindings.

HTTPStreamDataBinding is the principal interface for handling custom HTTP messages. The interface is designed to allow handling of large payloads. However, in order for such implementation to work, this data binding must return the control information and headers before writing the message into the stream.

The methods and their order of execution, listed below, must be implemented by the custom data binding.

To customize a data binding, write a class that implements HTTPStreamDataBinding. The data binding should have four private properties:

- private DataObject pDataObject
- private HTTPControl pCtrl

- private HTTPHeaders pHeaders
- private yourNativeDataType nativeData

The HTTP binding will invoke the customized data binding in the following order:

- Outbound processing (DataObject to Native format):
 1. setDataObject(...)
 2. setHeaders(...)
 3. setControlParameters(...)
 4. setBusinessException(...)
 5. convertToNativeData()
 6. getControlParameters()
 7. getHeaders()
 8. write(...)
- Inbound processing (Native format to DataObject):
 1. setControlParameters(...)
 2. setHeaders(...)
 3. convertFromNativeData(...)
 4. isBusinessException()
 5. getDataObject()
 6. getControlParameters()
 7. getHeaders()

You need to invoke setDataObject(...) in convertFromNativeData(...) to set the value of dataObject, which is converted from native data to the private property "pDataObject".

```
public void setDataObject(DataObject dataObject)
    throws DataBindingException {
    pDataObject = dataObject;
}

public void setControlParameters(HTTPControl arg0) {
    this.pCtrl = arg0;
}

public void setHeaders(HTTPHeaders arg0) {
    this.pHeaders = arg0;
}
/*
 * Add http header "IsBusinessException" in pHeaders.
 * Two steps:
 * 1.Remove all the header with name IsBusinessException (case-insensitive) first.
 * This is to make sure only one header is present.
 * 2.Add the new header "IsBusinessException"
 */
public void setBusinessException(boolean isBusinessException) {
    //remove all the header with name IsBusinessException (case-insensitive) first.
    //This is to make sure only one header is present.
    //add the new header "IsBusinessException", code example:
    HTTPHeader header=HeadersFactory.eINSTANCE.createHTTPHeader();
    header.setName("IsBusinessException");
    header.setValue(Boolean.toString(isBusinessException));
    this.pHeaders.getHeader().add(header);
}

public HTTPControl getControlParameters() {
    return pCtrl;
}
```

```

}
public HTTPHeaders getHeaders() {
    return pHeaders;
}
public DataObject getDataObject() throws DataBindingException {
    return pDataObject;
}
}
/*
 * Get header "IsBusinessException" from pHeaders, return its boolean value
 */
public boolean isBusinessException() {
    String headerValue = getHeaderValue(pHeaders,"IsBusinessException");
    boolean result=Boolean.parseBoolean(headerValue);
    return result;
}
public void convertToNativeData() throws DataBindingException {
    DataObject dataObject = getDataObject();
    this.nativeData=realConvertWorkFromSDOToNativeData(dataObject);
}
public void convertFromNativeData(HTTPInputStream arg0){
    //Customer-developed method to
    //Read data from HTTPInputStream
    //Convert it to DataObject
    DataObject dataobject=realConvertWorkFromNativeDataToSDO(arg0);
    setDataObject(dataobject);
}
public void write(HTTPOutputStream output) throws IOException {
    if (nativeData != null)
        output.write(nativeData);
}
}

```

EJB bindings

Enterprise JavaBeans (EJB) import bindings enable Service Component Architecture (SCA) components to invoke services provided by Java EE business logic running on a Java EE server. EJB export bindings allow SCA components to be exposed as Enterprise JavaBeans so that Java EE business logic can invoke SCA components otherwise unavailable to them.

EJB import bindings

EJB import bindings allow an SCA module to call EJB implementations by specifying the way that the consuming module is bound to the external EJB. Importing services from an external EJB implementation allows users to plug their business logic into the WebSphere Process Server environment and participate in a business process.

You use WebSphere Integration Developer to create EJB import bindings. You can use either of the following procedures to generate the bindings:

- Creating EJB import using the external service wizard

You can use the external service wizard in WebSphere Integration Developer to build an EJB import based on an existing implementation. The external service wizard creates services based on criteria that you provide. It then generates business objects, interfaces, and import files based on the services discovered.
- Creating EJB import using the assembly editor

You can create an EJB import within an assembly diagram using the WebSphere Integration Developer assembly editor. From the palette, you can use either an Import or use a Java class to create the EJB binding.

The generated import has data bindings that make the Java-WSDL connection instead of requiring a Java bridge component. You can directly wire a component

with a Web Services Description Language (WSDL) reference to the EJB import that communicates to an EJB-based service using a Java interface.

The EJB import can interact with Java EE business logic using either the EJB 2.1 programming model or the EJB 3.0 programming model.

The invocation to the Java EE business logic can be local (for EJB 3.0 only) or remote.

- Local invocation is used when you want to call Java EE business logic that resides on the same server as the import.

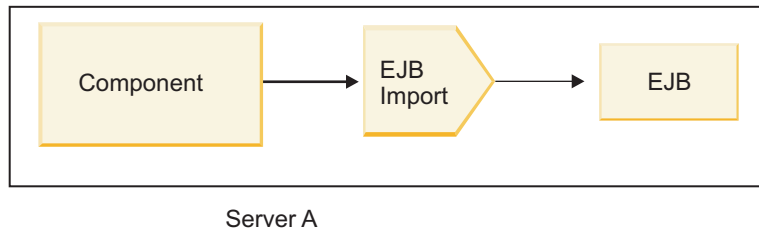


Figure 29. Local invocation of an EJB (EJB 3.0 only)

- Remote invocation is used when you want to call Java EE business logic that does not reside on the same server as the import.
For example, in the following figure, an EJB import uses the Remote Method Invocation over Internet InterORB Protocol (RMI/IIOP) to invoke an EJB method on another server.

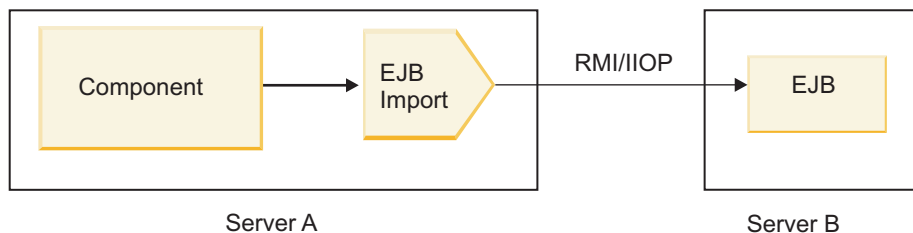


Figure 30. Remote invocation of an EJB

When it configures the EJB binding, WebSphere Integration Developer uses the JNDI name to determine the EJB programming model level and the type of invocation (local or remote).

EJB import bindings contain the following major components:

- JAX-WS data handler
- EJB fault selector
- EJB import function selector

If your user scenario is not based on the JAX-WS mapping, you might need a custom data handler, function selector, and fault selector to perform the tasks otherwise completed by the components that are part of the EJB import bindings. This includes the mapping normally completed by the custom mapping algorithm.

EJB export bindings

External Java EE applications can invoke an SCA component by way of an EJB export binding. Using an EJB export lets you expose SCA components so that external Java EE applications can invoke those components using the EJB programming model.

Note: The EJB export is a stateless bean.

You use WebSphere Integration Developer to create EJB bindings. You can use either of the following procedures to generate the bindings:

- Creating EJB export bindings using the external service wizard
You can use the external service wizard in WebSphere Integration Developer to build an EJB export service based on an existing implementation. The external service wizard creates services based on criteria that you provide. It then generates business objects, interfaces, and export files based on the services discovered.
- Creating EJB export bindings using the assembly editor
You can create an EJB export using the WebSphere Integration Developer assembly editor.

You can generate the binding from an existing SCA component, or you can generate an export with an EJB binding for a Java interface.

- When you generate an export for an existing SCA component that has an existing WSDL interface, the export is assigned a Java interface.
- When you generate an export for a Java interface, you can select either a WSDL or a Java interface for the export.

Note: A Java interface used to create an EJB export has the following limitations with regard to the objects (input and output parameters and exceptions) passed as parameters on a remote call:

- They must be of concrete type (instead of an interface or abstract type).
- They must conform to the Enterprise JavaBean specification. They must be serializable and have the default no-argument constructor, and all properties must be accessible through getter and setter methods.

Refer to the Sun Microsystems, Inc., Web site at <http://java.sun.com> for information about the Enterprise JavaBean specification.

In addition, the exception must be a checked exception, inherited from `java.lang.Exception`, and it must be singular (that is, it does not support throwing multiple checked exception types).

Note also that the business interface of a Java EnterpriseBean is a plain Java interface and must not extend `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject`. The methods of the business interface should not throw `java.rmi.RemoteException`.

The EJB export bindings can interact with Java EE business logic using either the EJB 2.1 programming model or the EJB 3.0 programming model.

The invocation can be local (for EJB 3.0 only) or remote.

- Local invocation is used when the Java EE business logic calls an SCA component that resides on the same server as the export.

- Remote invocation is used when the Java EE business logic does not reside on the same server as the export.

For example, in the following figure, an EJB uses RMI/IIOP to call an SCA component on a different server.

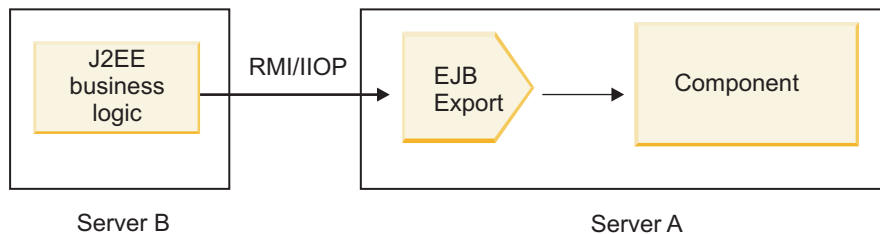


Figure 31. Remote call from a client to an SCA component by way of an EJB export

When it configures the EJB binding, WebSphere Integration Developer uses the JNDI name to determine the EJB programming model level and the type of invocation (local or remote).

EJB export bindings contain the following major components:

- JAX-WS data handler
- EJB export function selector

If your user scenario is not based on the JAX-WS mapping, you might need a custom data handler and function selector to perform the tasks otherwise completed by the components that are part of the EJB export bindings. This includes the mapping normally completed by the custom mapping algorithm.

EJB binding properties

EJB import bindings use their configured JNDI names to determine the EJB programming model level and type of invocation (local or remote). EJB import and export bindings use the JAX-WS data handler for data transformation. The EJB import binding uses an EJB import function selector and an EJB fault selector, and the EJB export binding uses an EJB export function selector.

JNDI names and EJB import bindings:

When it configures the EJB binding on an import, WebSphere Integration Developer uses the JNDI name to determine the EJB programming model level and type of invocation (local or remote).

If no JNDI name is specified, the default EJB interface binding is used. The default names that are created depend on whether you are invoking an EJB 2.1 JavaBean or an EJB 3.0 JavaBean.

Note: Refer to the WebSphere Application Server information center for more detailed information about naming conventions: EJB 3.0 application bindings overview.

- EJB 2.1 JavaBean

The default JNDI name preselected by WebSphere Integration Developer is the default EJB 2.1 binding, which takes the form `ejb/` plus the home interface, separated by slashes.

For example, for the home interface of an EJB 2.1 JavaBean for `com.mycompany.myremotebusinesshome`, the default binding is:

ejb/com/mycompany/myremotebusinesshome

For EJB 2.1, only remote EJB invocation is supported.

- EJB 3.0 JavaBean

The default JNDI name preselected by WebSphere Integration Developer for the local JNDI is the fully qualified class name of the local interface preceded by `ejblocal:`. For example, for the fully qualified interface of the local interface `com.mycompany.mylocalbusiness`, the preselected EJB 3.0 JNDI is:

`ejblocal:com.mycompany.mylocalbusiness`

For the remote interface `com.mycompany.myremotebusiness`, the preselected EJB 3.0 JNDI is the fully qualified interface:

`com.mycompany.myremotebusiness`

The EJB 3.0 default application bindings are described at the following location: [EJB 3.0 application bindings overview](#).

WebSphere Integration Developer will use the "short" name as the default JNDI location for EJBs using the version 3.0 programming model.

Note: If the deployed JNDI reference of the target EJB is different from the default JNDI binding location because a custom mapping was used or configured, the target JNDI name must be properly specified. You can specify the name in WebSphere Integration Developer before deployment, or, for the import binding, you can change the name in the administrative console (after deployment) to match the JNDI name of the target EJB.

For more information on creating EJB bindings, see the section devoted to [Working with EJB bindings](#) in the WebSphere Integration Developer information center.

JAX-WS data handler:

The Enterprise JavaBeans (EJB) import binding uses the JAX-WS data handler to turn request business objects into Java object parameters and to turn the Java object return value into the response business object. The EJB export binding uses the JAX-WS data handler to turn request EJBs into request business objects and to turn the response business object into a return value.

This data handler maps data from the SCA-specified WSDL interface to the target EJB Java interface (and vice versa) using the Java API for XML Web Services (JAX-WS) specification and the Java Architecture for XML Binding (JAXB) specification.

Note: Current support is restricted to the JAX-WS 2.1.1 and JAXB 2.1.3 specifications.

The data handler specified at the EJB binding level is used to perform request, response, fault, and runtime exception processing.

Note: For faults, a specific data handler can be specified for each fault by specifying the `faultBindingType` configuration property. This overrides the value specified at the EJB binding level.

The JAX-WS data handler is used by default when the EJB binding has a WSDL interface. This data handler cannot be used to transform a SOAP message representing a JAX-WS invocation to a data object.

The EJB import binding uses a data handler to transform a data object into a Java Object array (Object[]). During outbound communications, the following processing takes place:

1. The EJB binding sets the expected type, expected element, and targeted method name in the BindingContext to match those specified in the WSDL.
2. The EJB binding invokes the transform method for the data object requiring data transformation.
3. The data handler returns an Object[] representing the parameters of the method (in the order of their definition within the method).
4. The EJB binding uses the Object[] to invoke the method on the target EJB interface.

The binding also prepares an Object[] to process the response from the EJB invocation.

- The first element in the Object[] is the return value from the Java method invocation.
- The subsequent values represent the input parameters for the method.

This is required to support the In/Out and Out types of parameters.

For parameters of type Out, the values must be returned in the response data object.

The data handler processes and transforms values found in the Object[] and then returns a response to the data object.

The data handler supports xs:AnyType, xs:AnySimpleType, and xs:Any along with other XSD data types. To enable support for xs:Any, use the @XmlAnyElement (lax=true) for the JavaBean property in the Java code, as shown in the following example:

```
public class TestType {
    private Object[] object;

    @XmlAnyElement (lax=true)
    public Object[] getObject() {
        return object;
    }

    public void setObject (Object[] object) {
        this.object=object;
    }
}
```

This makes the property object in TestType an xs:any field. The Java class value used in the xs:any field should have the @XmlAnyElement annotation. For example, if Address is the Java class being used to populate the object array, the Address class should have the annotation @XmlRootElement.

Note: To customize the mapping from the XSD type to Java types defined by the JAX-WS specification, change the JAXB annotations to fit your business need. The JAX-WS data handler supports xs:any, xs:anyType, and xs:anySimpleType.

The following restrictions are applicable for the JAX-WS data handler:

- The data handler does not include support for the header attribute @WebParam annotation.

- The namespace for business object schema files (XSD files) does not include default mapping from the Java package name. The annotation `@XMLSchema` in `package-info.java` also does not work. The only way to create an XSD with a namespace is to use the `@XmlType` and `@XmlElement` annotations. `@XmlElement` defines the target namespace for the global element in `JavaBean` types.
- The EJB import wizard does not create XSD files for unrelated classes. Version 2.0 does not support the `@XmlSeeAlso` annotation, so if the child class is not referenced directly from the parent class, an XSD is not created. The solution to this problem is to run SchemaGen for such child classes.
SchemaGen is a command line utility (located in the `WPS_Install_Home/bin` directory) provided to create XSD files for a given `JavaBean`. These XSDs must be manually copied to the module for the solution to work.

EJB fault selector:

The EJB fault selector determines if an EJB invocation has resulted in a fault, a runtime exception, or a successful response.

If a fault is detected, the EJB fault selector returns the native fault name to the binding runtime so the JAX-WS data handler can convert the exception object into a fault business object.

On a successful (non-fault) response, the EJB import binding assembles a Java object array (`Object[]`) to return the values.

- The first element in the `Object[]` is the return value from the Java method invocation.
- The subsequent values represent the input parameters for the method.

This is required to support the In/Out and Out types of parameters.

For exception scenarios, the binding assembles an `Object[]` and the first element represents the exception thrown by the method.

The fault selector can return any of the following values:

Table 13. Return values

Type	Return value	Description
Fault	<code>ResponseType.FAULT</code>	Returned when the passed <code>Object[]</code> contains an exception object.
Runtime exception	<code>ResponseType.RUNTIME</code>	Returned if the exception object does not match any of the declared exception types on the method.
Normal response	<code>ResponseType.RESPONSE</code>	Returned in all other cases.

If the fault selector returns a value of `ResponseType.FAULT`, the native fault name is returned. This native fault name is used by the binding to determine the corresponding WSDL fault name from the model and to invoke the correct fault data handler.

EJB function selector:

The EJB bindings use an import function selector (for outbound processing) or an export function selector (for inbound processing) to determine the EJB method to call.

Import function selector

For outbound processing, the import function selector derives the EJB method type based on the name of the operation invoked by the SCA component that is wired to the EJB import. The function selector looks for the `@WebMethod` annotation on the WebSphere Integration Developer-generated JAX-WS annotated Java class to determine the associated target operation name.

- If the `@WebMethod` annotation is present, the function selector uses the `@WebMethod` annotation to determine the correct Java method mapping for the WSDL method.
- If the `@WebMethod` annotation is missing, the function selector assumes that the Java method name is the same as the invoked operation name.

Note: This function selector is valid only for a WSDL-typed interface on an EJB import, not for a Java-typed interface on an EJB import.

The function selector returns a `java.lang.reflect.Method` object that represents the method of the EJB interface.

The function selector uses a Java Object array (`Object[]`) to contain the response from the target method. The first element in the `Object[]` is a Java method with the name of the WSDL, and the second element in the `Object[]` is the input business object.

Export function selector

For inbound processing, the export function selector derives the target method to be invoked from the Java method.

The export function selector maps the Java operation name invoked by the EJB client into the name of the operation in the interface of the target component. The method name is returned as a `String` and is resolved by the SCA runtime depending on the interface type of the target component.

EIS bindings

Enterprise information system (EIS) bindings provide connectivity between SCA components and an external EIS. This communication is achieved using EIS exports and EIS imports that support JCA 1.5 resource adapters and Websphere Adapters.

Your SCA components might require that data be transferred to or from an external EIS. When you create an SCA module requiring such connectivity, you will include (in addition to your SCA component) an import or export with an EIS binding for communication with a specific external EIS.

Resource adapters in WebSphere Integration Developer are used within the context of an import or an export. You develop an import or an export with the external service wizard and, in developing it, include the resource adapter. An EIS import, which lets your application invoke a service on an EIS system, or an EIS export, which lets an application on an EIS system invoke a service developed in

WebSphere Integration Developer, are created with a particular resource adapter. For example, you would create an import with the JD Edwards adapter to invoke a service on the JD Edwards system.

When you use the external service wizard, the EIS binding information is created for you. You can also use another tool, the assembly editor, to add or modify binding information. See WebSphere Integration Developer Information Center for more information.

After the SCA module containing the EIS binding is deployed to the server, you can use the administrative console to view information about the binding or to configure the binding.

EIS bindings overview

The EIS (enterprise information system) binding, when used with a JCA resource adapter, lets you access services on an enterprise information system or make your services available to the EIS.

The following example shows how an SCA module named ContactSyncModule synchronizes contact information between a Siebel system and an SAP system.

1. The SCA component named ContactSync listens (by way of an EIS application export named Siebel Contact) for changes to Siebel contacts.
2. The ContactSync SCA component itself makes use of an SAP application (through an EIS application import) in order to update the SAP contact information accordingly.

Because the data structures used for storing contacts are different in Siebel and SAP systems, the ContactSync SCA component must provide mapping.

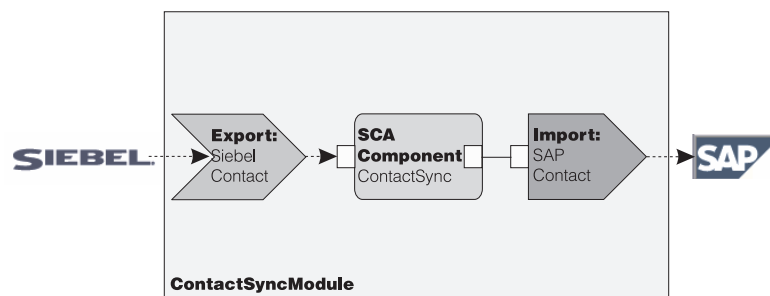


Figure 32. Flow from a Siebel system to an SAP system

The Siebel Contact export and the SAP Contact import have the appropriate resource adapters configured.

Key features of EIS bindings

An EIS import is a Service Component Architecture (SCA) import that allows components in the SCA module to use EIS applications defined outside the SCA module. An EIS import is used to transfer data from the SCA component to an external EIS; an EIS export is used to transfer data from an external EIS into the SCA module.

Imports

The role of the EIS import is to bridge the gap between SCA components and external EIS systems. External applications can be treated as an EIS import. In this case, the EIS import sends data to the external EIS and optionally receives data in response.

The EIS import provides SCA components with a uniform view of the applications external to the module. This allows components to communicate with an external EIS, such as SAP, Siebel, or PeopleSoft, using a consistent SCA model.

On the client side of the import, there is an interface, exposed by the EIS import application, with one or more methods, each taking data objects as arguments and return values. On the implementation side, there is a Common Client Interface (CCI) implemented by a resource adapter.

The runtime implementation of an EIS import connects the client-side interface and the CCI. The import maps the invocation of the method on the interface to the invocation on the CCI.

Bindings are created at three levels: the interface binding, which then uses the contained method bindings, which in turn use data bindings.

The interface binding relates the interface of the import to the connection to the EIS system providing the application. This reflects the fact that the set of applications, represented by the interface, is provided by the specific instance of the EIS and the connection provides access to this instance. The binding element contains properties with enough information to create the connection (these properties are part of the `javax.resource.spi.ManagedConnectionFactory` instance).

The method binding associates the method with the specific interaction with the EIS system. For JCA, the interaction is characterized by the set of properties of the `javax.resource.cci.InteractionSpec` interface implementation. The interaction element of the method binding contains these properties, along with the name of the class, thus providing enough information to perform the interaction. The method binding uses data bindings describing the mapping of the argument and result of the interface method to EIS representation.

The runtime scenario for an EIS import is as follows:

1. The method on the import interface is invoked using the SCA programming model.
2. The request, reaching the EIS import, contains the name of the method and its arguments.
3. The import first creates an interface binding implementation; then, using data from the import binding, it creates a `ConnectionFactory` and associates the two. That is, the import calls `setConnectionFactory` on the interface binding.
4. The method binding implementation matching the invoked method is created.
5. The `javax.resource.cci.InteractionSpec` instance is created and populated; then, data bindings are used to bind the method arguments to a format understood by the resource adapter.
6. The CCI interface is used to perform the interaction.
7. When the call returns, the data binding is used to create the result of the invocation, and the result is returned to the caller.

Exports

The role of the EIS export is to bridge the gap between an SCA component and an external EIS. External applications can be treated as an EIS export. In this case, the external application sends its data in the form of periodic notifications. An EIS export can be thought of as a subscription application listening to an external request from an EIS. The SCA component that uses the EIS export views it as a local application.

The EIS export provides SCA components with a uniform view of the applications external to the module. This allows components to communicate with an EIS, such as SAP, Siebel, or PeopleSoft, using a consistent SCA model.

The export features a listener implementation receiving requests from the EIS. The listener implements a resource adapter-specific listener interface. The export also contains a component implementing interface, exposed to the EIS through the export.

The runtime implementation of an EIS export connects the listener with the component implementing interface. The export maps the EIS request to the invocation of the appropriate operation on the component. Bindings are created at three levels: a listener binding, which then uses a contained native method binding, which in turn uses a data binding .

The listener binding relates the listener receiving requests with the component exposed through the export. The export definition contains the name of the component; the runtime locates it and forwards requests to it.

The native method binding associates the native method or the event type received by the listener to the operation implemented by the component exposed by way of the export. There is no relationship between the method invoked on the listener and the event type; all the events arrive through one or more methods of the listener. The native method binding uses the function selector defined in the export to extract the native method name from the inbound data and data bindings to bind the data format of the EIS to a format understood by the component.

The runtime scenario for an EIS export is as follows:

1. The EIS request triggers invocation of the method on the listener implementation.
2. The listener locates and invokes the export, passing to it all the invocation arguments.
3. The export creates the listener binding implementation.
4. The export instantiates the function selector and sets it on the listener binding.
5. The export initializes native method bindings and adds them to the listener binding. For each native method binding, the data bindings are also initialized.
6. The export invokes the listener binding.
7. The listener binding locates exported components and uses the function selector to retrieve the native method name.
8. This name is used to locate the native method binding, which then invokes the target component.

The adapter interaction style allows for the EIS export binding to invoke the target component either asynchronously (the default) or synchronously.

Resource adapters

You develop an import or an export with the external service wizard and, in developing it, include a resource adapter. The adapters that come with WebSphere Integration Developer used to access CICS, IMS, JD Edwards, PeopleSoft, SAP and Siebel systems are intended for development and test purposes only. This means you use them to develop and test your applications.

Once you deploy your application, you will need licensed runtime adapters to run your application. However, when you build your service you can embed the adapter with your service. Your adapter licensing might allow you to use the embedded adapter as the licensed runtime adapter. These adapters are compliant with the Java EE Connector Architecture (JCA 1.5). JCA, an open standard, is the Java EE standard for EIS connectivity. JCA provides a managed framework; that is, Quality of Service (QoS) is provided by the application server, which offers life-cycle management and security to transactions. They are also compliant with the Enterprise Metadata Discovery specification with the exception of IBM CICS ECI Resource Adapter and IBM IMS Connector for Java.

The WebSphere Business Integration Adapters, an older set of adapters, are also supported by the wizard.

Java EE resources

The EIS module, an SCA module that follows the EIS module pattern, can be deployed to the Java EE platform.

The deployment of the EIS module to the Java EE platform results in an application that is ready to execute, packaged as an EAR file and deployed to the server. All Java EE artifacts and resources are created; the application is configured and ready to be run.

JCA Interaction Spec and Connection Spec dynamic properties

The EIS binding can accept input for the InteractionSpec and ConnectionSpec specified by using a well-defined child data object that accompanies the payload. This allows for dynamic request-response interactions with a resource adapter through the InteractionSpec and component authentication through the ConnectionSpec.

The `javax.cci.InteractionSpec` carries information on how the interaction request with the resource adapter should be handled. It can also carry information on how the interaction was achieved after the request. These two-way communications through the interactions are sometimes referred to as *conversations*.

The EIS binding expects the payload that will be the argument to the resource adapter to contain a child data object called `properties`. This property data object will contain name/value pairs, with the name of the Interaction Spec properties in a specific format. The formatting rules are:

- Names must begin with the prefix `IS`, followed by the property name. For example, an interaction spec with a JavaBeans property called `InteractionId` would specify the property name as `ISInteractionId`.
- The name/value pair represents the name and the value of the simple type of the Interaction Spec property.

In this example, an interface specifies that the input of an operation is an Account data object. This interface invokes an EIS import binding application with the intention to send and receive a dynamic InteractionSpec property called workingSet with the value xyz.

The business graph or business objects in the server contain an underlying properties business object that permits the sending of protocol-specific data with the payload. This properties business object is built-in and does not need to be specified in the XML schema when constructing a business object. It only needs to be created and used. If you have your own data types defined based on an XML schema, you need to specify a properties element that contains your expected name/value pairs.

```
BOfactory dataFactory = (BOfactory) \
serviceManager.locateService("com/ibm/websphere/bo/BOfactory");
//Wrapper for doc-lit wrapped style interfaces,
//skip to payload for non doc-lit
DataObject docLitWrapper = dataFactory.createByElement /
("http://mytest/eis/Account", "AccountWrapper");
```

Create the payload.

```
DataObject account = docLitWrapper.createDataObject(0);
DataObject accountInfo = account.createDataObject("AccountInfo");
//Perform your setting up of payload
```

```
//Construct properties data for dynamic interaction
```

```
DataObject properties = account.createDataObject("properties");
```

For name workingSet, set the value expected (xyz).

```
properties.setString("ISworkingSet", "xyz");
```

```
//Invoke the service with argument
```

```
Service accountImport = (Service) \
serviceManager.locateService("AccountOutbound");
DataObject result = accountImport.invoke("createAccount", docLitWrapper);
```

```
//Get returned property
DataObject retProperties = result.getDataObject("properties");
```

```
String workingset = retProperties.getString("ISworkingSet");
```

You can use ConnectionSpec properties for dynamic component authentication. The same rules apply as above, except that the property name prefix needs to be CS (instead of IS). ConnectionSpec properties are not two-way. The same properties data object can contain both IS and CS properties.

To use ConnectionSpec properties, set the resAuth specified on the import binding to Application. Also, make sure the resource adapter supports component authorization. See chapter 8 of the J2EE Connector Architecture Specification for more details.

External clients with EIS bindings

The server can send messages to, or receive messages from, external clients using EIS bindings.

An external client, for example a Web portal or an EIS, needs to send a message to an SCA module in the server or needs to be invoked by a component from within the server.

The client invokes the EIS import as with any other application, using either the Dynamic Invocation Interface (DII) or Java interface.

1. The external client creates an instance of the ServiceManager and looks up the EIS import using its reference name. The result of the lookup is a service interface implementation.
2. The client creates an input argument, a generic data object, created dynamically using the data object schema. This step is done using the Service Data Object DataFactory interface implementation.
3. The external client invokes the EIS and obtains the required results.

Alternatively, the client can invoke the EIS import using the Java interface.

1. The client creates an instance of the ServiceManager and looks up the EIS import using its reference name. The result of the lookup is a Java interface of the EIS import.
2. The client creates an input argument and a typed data object.
3. The client invokes EIS and obtains the required results.

The EIS export interface defines the interface of the exported SCA component that is available to the external EIS applications. This interface can be thought of as the interface that an external application (such as SAP or PeopleSoft) will invoke through the implementation of the EIS export application runtime.

The export uses EISExportBinding to bind exported services to the external EIS application. It allows you to subscribe an application contained in your SCA module to listen for EIS service requests. The EIS export binding specifies the mapping between the definition of inbound events as it is understood by the resource adapter (using Java EE Connector Architecture interfaces) and the invocation of SCA operations.

The EISExportBinding requires external EIS services to be based on Java EE Connector Architecture 1.5 inbound contracts. The EISExportBinding requires that a data handler or data binding be specified either at the binding level or the method level.

JMS bindings

A Java Message Service (JMS) provider enables messaging based on the Java Messaging Service API and programming model. It provides Java EE connection factories to create connections for JMS destinations and to send and receive messages.

Three JMS bindings are provided:

- Service integration bus (SIB) provider binding compliant with JMS JCA 1.5 (*JMS binding*)
- Generic, non-JCA JMS binding, compliant with JMS 1.1 (*Generic JMS binding*)
- WebSphere MQ JMS binding, providing JMS provider support for WebSphere MQ and allowing Java EE application interoperability (*WebSphere MQ JMS binding*)

The JMS export and import bindings allow a Service Component Architecture (SCA) module to make calls to, and receive messages from, external JMS systems.

Also supported are WebSphere MQ bindings (*WebSphere MQ binding*) which allow native MQ users to handle arbitrary incoming and outgoing message formats (WebSphere MQ required).

The JMS import and export bindings provide integration with JMS applications using the JCA 1.5-based SIB JMS provider that is part of WebSphere Application Server. Other JCA 1.5-based JMS resource adapters are not supported

JMS bindings overview

JMS bindings provide connectivity between the Service Component Architecture (SCA) environment and JMS systems.

JMS bindings

The major components of both JMS import and JMS export bindings are:

- Resource adapter: enables managed, bidirectional connectivity between an SCA module and external JMS systems
- Connections: encapsulate a virtual connection between a client and a provider application
- Destinations: used by a client to specify the target of messages it produces or the source of messages it consumes
- Authentication data: used to secure access to the binding

JMS import bindings

JMS import bindings allow you to import an external JMS application to be used inside your SCA module. JMS import bindings allow components within your SCA module to communicate with services provided by external JMS applications.

Connections to the associated JMS provider of JMS destinations are created by using a JMS connection factory. Use connection factory administrative objects to manage JMS connection factories for the default messaging provider.

Interaction with external JMS systems includes the use of destinations for sending requests and receiving replies.

Two types of usage scenarios for the JMS import binding are supported, depending on the type of operation being invoked:

- One-way: The JMS import puts a message on the send destination configured in the import binding. Nothing is set in the replyTo field of the JMS header.
- Two-way (request-response): The JMS import puts a message on the send destination and then persists the reply it receives from the SCA component.

The import binding can be configured (using the **Response correlation scheme** field in WebSphere Integration Developer) to expect the response message correlation ID to have been copied from the request message ID (the default), or from the request message correlation ID. The import binding can also be configured to use a temporary dynamic response destination to correlate responses with requests. A temporary destination is created for each request and the import uses this destination to receive the response.

The receive destination is set in the replyTo header property of the outbound message. A message listener is deployed to listen on the receive destination, and when a reply is received, the message listener passes the reply back to the component.

For both one-way and two-way usage scenarios, dynamic and static header properties can be specified. Static properties can be set from the JMS import method binding. Some of these properties have special meanings to the SCA JMS runtime.

It is important to note that JMS is an asynchronous binding. If a calling component invokes a JMS import synchronously (for a two-way operation), the calling component is blocked until the response is returned by the JMS service.

Figure 33 illustrates how the import is linked to the external service.

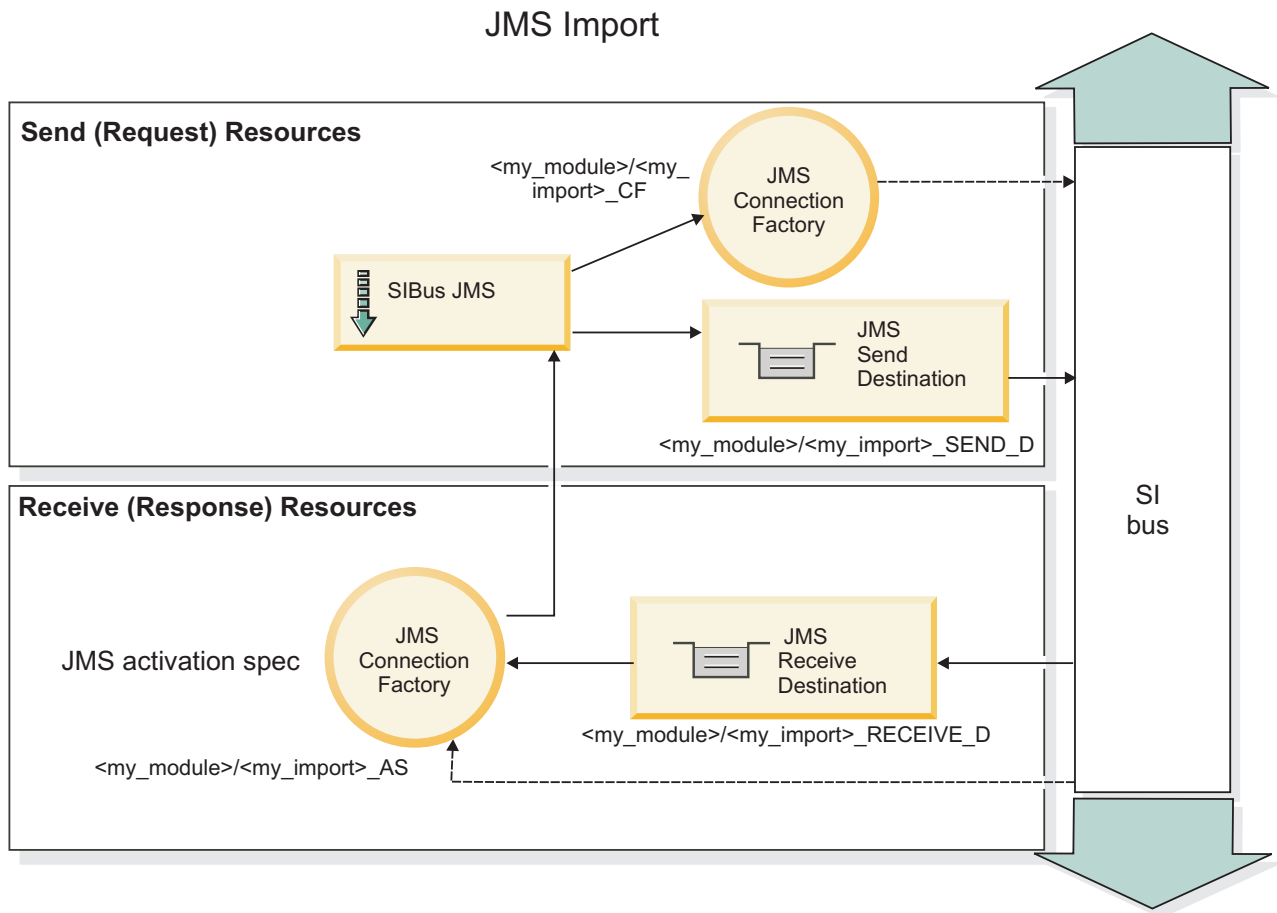


Figure 33. JMS import binding resources

JMS export bindings

JMS export bindings provide the means for SCA modules to provide services to external JMS applications.

The connection that is part of a JMS export is a configurable activation specification.

A JMS export has send and receive destinations.

- The receive destination is where the incoming message for the target component should be placed.
- The send destination is where the reply will be sent, unless the incoming message has overridden this using the replyTo header property.

A message listener is deployed to listen to requests incoming to the receive destination specified in the export binding. The destination specified in the send field is used to send the reply to the inbound request if the invoked component provides a reply. The destination specified in the replyTo field of the incoming message overrides the destination specified in the send.

Figure 34 illustrates how the external requestor is linked to the export.

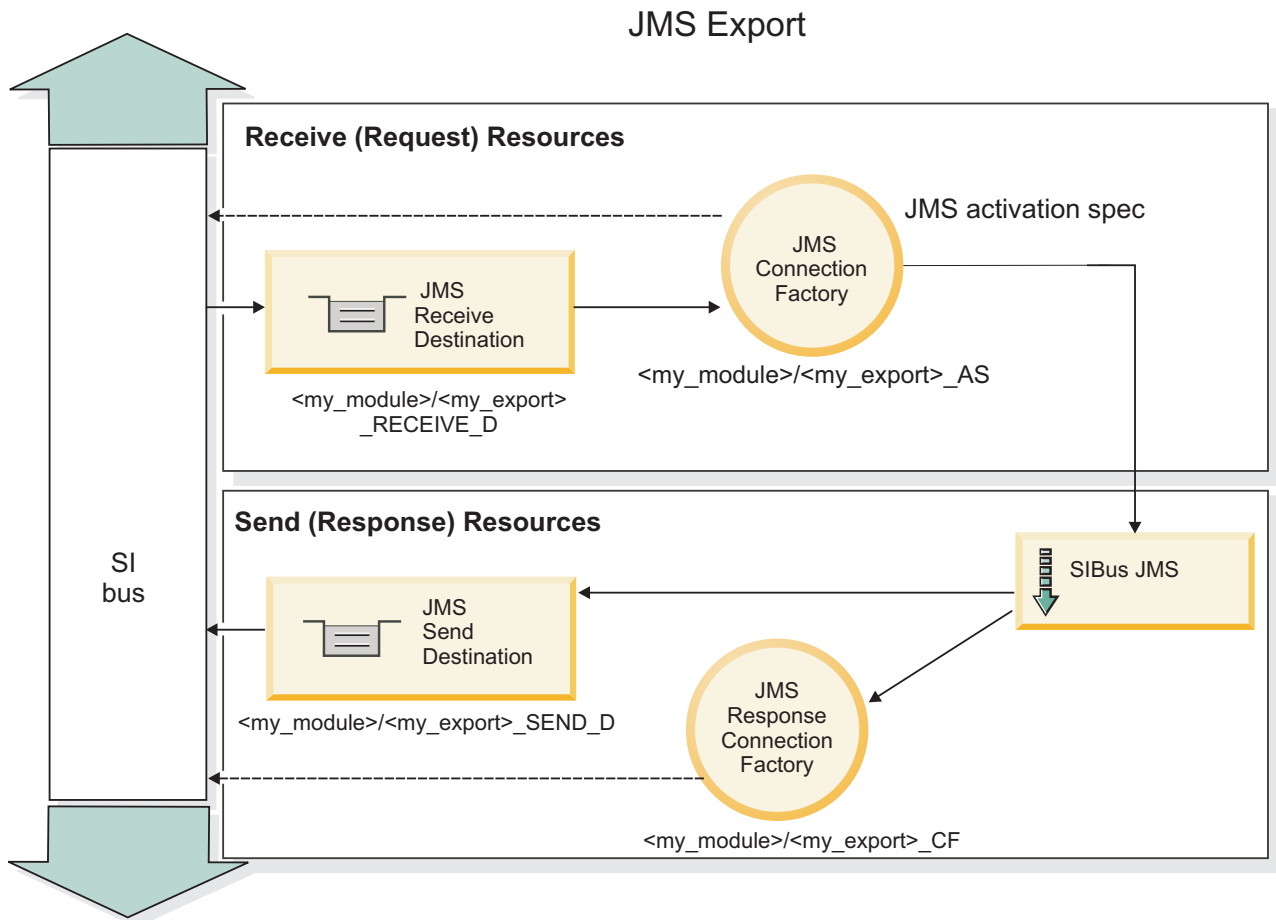


Figure 34. JMS export binding resources

JMS integration and resource adapters

The Java Message Service (JMS) provides integration through an available JMS JCA 1.5-based resource adapter. Complete support for JMS integration is provided for the Service Integration Bus (SIB) JMS resource adapter.

Use a JMS provider for JCA 1.5 resource adapter when you want to integrate with an external JCA 1.5-compliant JMS system. External services compliant with JCA 1.5 can receive messages and send messages to integrate with your service component architecture (SCA) components using the SIB JMS resource adapter.

The use of other provider-specific JCA 1.5 resource adapters is not supported.

JMS modules cannot be deployed to a Java SE environment. Such modules are only deployable to a Java EE environment.

Key features of JMS bindings

Key features of JMS import and export bindings include headers and created Java EE resources.

Special headers

Special header properties are used in JMS imports and exports to tell the target how to handle the message.

For example, `TargetFunctionName` maps from the native method to the operation method.

Java EE resources

A number of Java EE resources are created when JMS imports and exports are deployed to a Java EE environment.

ConnectionFactory

Used by clients to create a connection to the JMS provider.

ActivationSpec

Imports use this for receiving the response to a request; exports use it when configuring the message endpoints that represent message listeners in their interactions with the messaging system.

Destinations

- Send destination: on an import, this is where the request or outgoing message is sent; on an export, this is the destination where the response message will be sent, if not superseded by the `JMSReplyTo` header field in the incoming message.
- Receive destination: where the incoming message should be placed; with imports, this is a response; with exports, this is a request.
- Callback destination: SCA JMS system destination used to store correlation information. Do not read or write to this destination.

The installation task creates the `ConnectionFactory` and three destinations. It also creates the `ActivationSpec` to enable the runtime message listener to listen for replies on the receive destination. The properties of these resources are specified in the import or export file.

JMS headers

A JMS message contains two types of headers—the JMS system header and multiple JMS properties. Both types of headers can be accessed either in a mediation module in the Service Message Object (SMO) or by using the `ContextService` API.

JMS system header

The JMS system header is represented in the SMO by the `JMSHeader` element, which contains all the fields typically found in a JMS header. Although these can be modified in the mediation (or `ContextService`), some JMS system header fields set in the SMO will not be propagated in the outbound JMS message as they are overridden by system or static values.

The key fields in the JMS system header that can be updated in a mediation (or `ContextService`) are:

- **JMS**Type and **JMSCorrelationID** – values of the specific predefined message header properties
- **JMSDeliveryMode** – values for delivery mode (persistent or nonpersistent; default is persistent)
- **JMSPriority** – priority value (0 to 9; default is JMS_Default_Priority)

JMS properties

JMS properties are represented in the SMO as entries in the Properties list. The properties can be added, updated, or deleted in a mediation or by using the ContextService API.

Properties can also be set statically in the JMS binding. Properties that are set statically override settings (with the same name) that are set dynamically.

User properties propagated from other bindings (for example, an HTTP binding) will be output in the JMS binding as JMS properties.

Header propagation settings

The propagation of the JMS system header and properties either from the inbound JMS message to downstream components or from upstream components to the outbound JMS message can be controlled by the Propagate Protocol Header flag on the binding.

When Propagate Protocol Header is set, header information is allowed to flow to the message or to the target component, as described in the following list:

- JMS export request
The JMS header received in the message will be propagated to target components by way of the context service. JMS properties received in the message will be propagated to target components by way of the context service.
- JMS export response
Any JMS header fields set in the context service will be used in the outbound message, if not overridden by static properties set on the JMS export binding. Any properties set in the context service will be used in the outbound message if not overridden by static properties set on the JMS export binding.
- JMS import request
Any JMS header fields set in the context service will be used in the outbound message, if not overridden by static properties set on the JMS import binding. Any properties set in the context service will be used in the outbound message if not overridden by static properties set on the JMS import binding.
- JMS import response
The JMS header received in the message will be propagated to target components by way of the context service. JMS properties received in the message will be propagated to target components by way of the context service.

JMS temporary dynamic response destination correlation scheme

The temporary dynamic response destination correlation scheme causes a unique dynamic queue or topic to be created for each request sent.

The static response destination specified in the import is used to derive the nature of the temporary dynamic destination queue or topic. This is set in the **ReplyTo** field of the request, and the JMS import listens for responses on that destination.

When the response is received it is requeued to the static response destination for asynchronous processing. The **CorrelationID** field of the response is not used, and does not need to be set.

Transactional issues

When a temporary dynamic destination is being used, the response must be consumed in the same thread as the sent response. The request must be sent outside the global transaction, and must be committed before it is received by the backend service, and a response returned.

Persistence

Temporary dynamic queues are short-lived entities and do not guarantee the same level of persistence associated with a static queue or topic. A temporary dynamic queue or topic will not survive a server restart and neither will messages. After the message has been requeued to the static response destination it retains the persistence defined in the message.

Timeout

The import waits to receive the response on the temporary dynamic response destination for a fixed amount of time. This time interval will be taken from the SCA Response Expiration time qualifier, if it is set, otherwise the time defaults to 60 seconds. If the wait time is exceeded the import throws a `ServiceTimeoutRuntimeException`.

External clients

The server can send messages to, or receive messages from, external clients using JMS bindings.

An external client (such as a Web portal or an enterprise information system) can send a message to an SCA module in the server, or it can be invoked by a component from within the server.

The JMS export components deploy message listeners to listen to requests incoming to the receive destination specified in the export binding. The destination specified in the send field is used to send the reply to the inbound request if the invoked application provides a reply. Thus, an external client is able to invoke applications with the export binding.

JMS imports bind to, and can deliver messages to, external clients. This message might or might not demand a response from the external client.

Working with external clients:

An external client (that is, outside the server) might need to interact with an application installed in the server.

About this task

Consider a very simple scenario in which an external client wants to interact with an application on the server. The figure depicts a typical simple scenario.

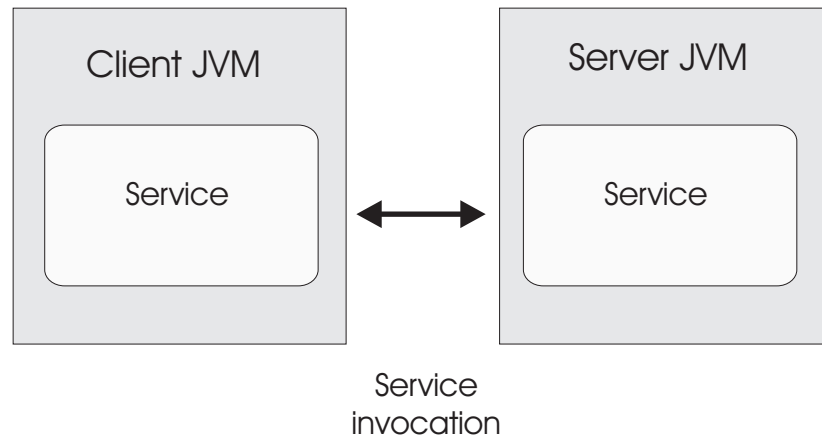


Figure 35. Simple use-case scenario: external client interacts with server application

The SCA application includes an export with a JMS binding; this makes the application available to external clients.

When you have an external client in a Java virtual machine (JVM) separate from your server, there are several steps you must take in order to make a connection and interact with a JMS export. The client obtains an InitialContext with the correct values and then looks up the resources through JNDI. The client then uses the JMS 1.1 specification client to access the destinations and the send and receive messages on the destinations.

The default JNDI names of the resources created automatically by the runtime are listed in the configuration topic of this section. However, if you have pre-created resources, use those JNDI names.

Procedure

1. Configure JMS destinations and the connection factory to send the message.
2. Make sure that the JNDI context, the port for the SIB resource adapter, and the messaging bootstrapping port are correct.

The server uses some default ports, but if there are more servers installed on that system, alternate ports are created at installation time to avoid conflicts with other server instances. You can use the administrative console to determine which ports your server is employing. Go to **Servers** → **Application Servers** → *your_server_name* → **Configuration** and click **Ports** under **Communication**. You can then edit the port being used.

3. The client obtains an initial context with the correct values and then looks up the resources through JNDI.
4. Using JMS 1.1 specifications, the client accesses the destinations and the send and receive messages on the destinations.

Troubleshooting JMS bindings

You can diagnose and fix problems with JMS bindings.

Implementation exceptions

In response to various error conditions, the JMS import and export implementation can return one of two types of exceptions:

- **Service Business Exception:** this exception is returned if the fault specified on the service business interface (WSDL port type) occurred.
- **Service Runtime Exception:** raised in all other cases. In most cases, the cause exception will contain the original exception (JMSEException).

For example, an import expects only one response message for each request message. If more than one response arrives, or if a late response (one for which the SCA response expiration has expired) arrives, a Service Runtime Exception is thrown. The transaction is rolled back, and the response message is backed out of the queue or handled by the failed event manager.

Primary failure conditions

The primary failure conditions of JMS bindings are determined by transactional semantics, by JMS provider configuration, or by reference to existing behavior in other components. The primary failure conditions include:

- **Failure to connect to the JMS provider or destination.**
A failure to connect to the JMS provider to receive messages will result in the message listener failing to start. This condition will be logged in the WebSphere Application Server log. Persistent messages will remain on the destination until they are successfully retrieved (or expired).
A failure to connect to the JMS provider to send outbound messages will cause rollback of the transaction controlling the send.
- **Failure to parse an inbound message or to construct an outbound message.**
A failure in the data binding or data handler causes rollback of the transaction controlling the work.
- **Failure to send the outbound message.**
A failure to send a message causes rollback of the relevant transaction.
- **Multiple or unexpected late response messages.**
The import expects only one response message for each request message. Also the valid time period in which a response can be received is determined by the SCA Response Expiration qualifier on the request. When a response arrives or the expiration time is exceeded, the correlation record is deleted. If response messages arrive unexpectedly or arrive late, a Service Runtime Exception is thrown.
- **Service timeout runtime exception caused by late response when using the temporary dynamic response destination correlation scheme.**
The JMS import will timeout after a period of time determined by the SCA response expiration qualifier, or if this is not set it will default to 60 seconds.

JMS-based SCA messages not appearing in the failed event manager

If SCA messages originated through a JMS interaction fail, you would expect to find these messages in the failed event manager. If such messages are not appearing in the failed event manager, ensure that the underlying SIB destination of the JMS destination has a maximum failed deliveries value greater than 1. Setting this value to 2 or more enables interaction with the failed event manager during SCA invocations for the JMS bindings.

Handling exceptions

The way in which the binding is configured determines how exceptions that are raised by data handlers or data bindings are handled. Additionally, the nature of the mediation flow dictates the behavior of the system when such an exception is thrown.

A variety of problems can occur when a data handler or data binding is called by your binding. For example, a data handler might receive a message that has a corrupt payload, or it might try to read a message that has an incorrect format.

The way your binding handles such an exception is determined by how you implement the data handler or data binding. The recommended behavior is that you design your data binding to throw a `DataBindingException`.

When any runtime exception, including a `DataBindingException`, is thrown:

- If the mediation flow is configured to be transactional, the JMS message, by default, is stored in the Failed Event Manager for manual replay or deletion.

Note: You can change the recovery mode on the binding so that the message is rolled back instead of being stored in the Failed Event Manager.

- If the mediation flow is not transactional, the exception is logged and the message is lost.

The situation is similar for a data handler. Since the data handler is invoked by the data binding, any data handler exception is wrapped into a data binding exception. Therefore a `DataHandlerException` is reported to you as a `DataBindingException`.

Generic JMS bindings

The Generic JMS binding provides connectivity to third-party JMS 1.1 compliant providers. The operation of the Generic JMS bindings is similar to that of JMS bindings.

The service provided through a JMS binding allows a Service Component Architecture (SCA) module to make calls or receive messages from external systems. The system can be an external JMS system.

The Generic JMS binding provides integration with non-JCA 1.5-compliant JMS providers that support JMS 1.1 and implement the optional JMS Application Server Facility. The Generic JMS binding supports those JMS providers (including Oracle AQ, TIBCO, SonicMQ, WebMethods, BEA WebLogic, and WebSphere MQ) that do not support JCA 1.5 but do support the Application Server Facility of the JMS 1.1 specification. The WebSphere embedded JMS provider (SIBJMS), which is a JCA 1.5 JMS provider, is not supported by this binding; when using that provider, use the “JMS bindings” on page 70.

Use this Generic binding when integrating with a non-JCA 1.5-compliant JMS-based system within an SCA environment. The target external applications can then receive messages and send messages to integrate with an SCA component.

Generic JMS bindings overview

Generic JMS bindings are non-JCA JMS bindings that provide connectivity between the Service Component Architecture (SCA) environment and JMS systems that are compliant with JMS 1.1 and that implement the optional JMS Application Server Facility.

Generic JMS bindings

The major aspects of Generic JMS import and export bindings are:

- Listener port: enables non-JCA-based JMS providers to receive messages and dispatch them to a Message Driven Bean (MDB)

- **Connections:** encapsulate a virtual connection between a client and a provider application
- **Destinations:** used by a client to specify the target of messages it produces or the source of messages it consumes
- **Authentication data:** used to secure access to the binding

Generic JMS import bindings

Generic JMS import bindings allow components within your SCA module to communicate with services provided by external non-JCA 1.5-compliant JMS providers.

The connection part of a JMS import is a connection factory. A connection factory, the object a client uses to create a connection to a provider, encapsulates a set of connection configuration parameters defined by an administrator. Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface.

Interaction with external JMS systems includes the use of destinations for sending requests and receiving replies.

Two types of usage scenarios for the Generic JMS import binding are supported, depending on the type of operation being invoked:

- **One-way:** The Generic JMS import puts a message on the send destination configured in the import binding. Nothing is sent to the `replyTo` field of the JMS header.
- **Two-way (request-response):** The Generic JMS import puts a message on the send destination and then persists the reply it receives from the SCA component. The receive destination is set in the `replyTo` header property of the outbound message. A message driven bean (MDB) is deployed to listen on the receive destination, and when a reply is received, the MDB passes the reply back to the component.

The import binding can be configured (using the **Response correlation scheme** field in WebSphere Integration Developer) to expect the response message correlation ID to have been copied from the request message ID (the default) or from the request message correlation ID.

For both one-way and two-way usage scenarios, dynamic and static header properties can be specified. Static properties can be set from the Generic JMS import method binding. Some of these properties have special meanings to the SCA JMS runtime.

It is important to note that Generic JMS is an asynchronous binding. If a calling component invokes a Generic JMS import synchronously (for a two-way operation), the calling component is blocked until the response is returned by the JMS service.

Figure 36 on page 81 illustrates how the import is linked to the external service.

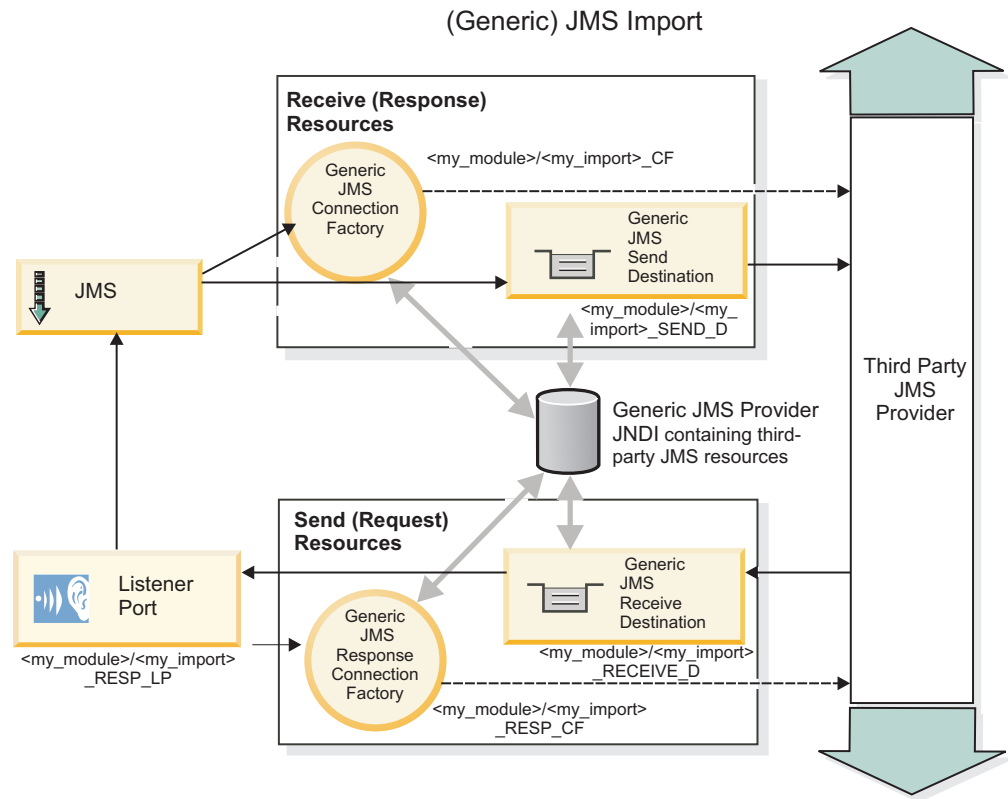


Figure 36. Generic JMS import binding resources

Generic JMS export bindings

Generic JMS export bindings provide the means for SCA modules to provide services to external JMS applications.

The connection part of a JMS export is composed of a ConnectionFactory and a ListenerPort.

A Generic JMS export has send and receive destinations.

- The receive destination is where the incoming message for the target component should be placed.
- The send destination is where the reply will be sent, unless the incoming message has overridden this using the replyTo header property.

An MDB is deployed to listen to requests incoming to the receive destination specified in the export binding.

- The destination specified in the send field is used to send the reply to the inbound request if the invoked component provides a reply.
- The destination specified in the replyTo field of the incoming message overrides the destination specified in the send field.
- For request/response scenarios, the import binding can be configured (using the **Response correlation scheme** field in WebSphere Integration Developer) to expect the response to copy the request message ID to the correlation ID field of the response message (default), or the response can copy the request correlation ID to the correlation ID field of the response message.

Figure 37 illustrates how the external requestor is linked to the export.

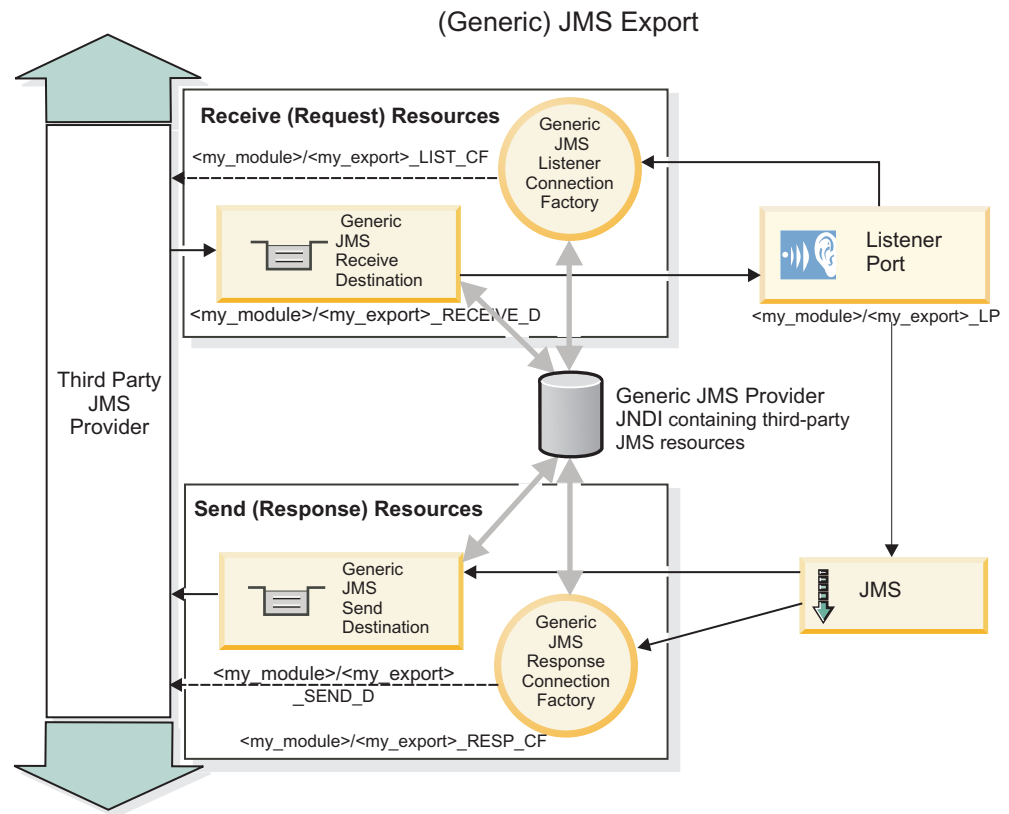


Figure 37. Generic JMS export binding resources

Key features of Generic JMS bindings

The features of the Generic JMS import and export binding are consistent with those of the WebSphere embedded JMS and MQ JMS import bindings. Key features include header definitions and access to existing Java EE resources. However, because of its generic nature, there are no JMS provider-specific connectivity options, and this binding has limited capability to generate resources at deployment and installation.

Generic imports

Like the MQ JMS import application, the Generic JMS implementation is asynchronous and supports three invocations: one-way, two-way (also known as request-response), and callback.

When the JMS import is deployed, a message driven bean (MDB) provided by the runtime environment is deployed. The MDB listens for replies to the request message. The MDB is associated with (listens on) the destination sent with the request in the `replyTo` header field of the JMS message.

Generic exports

Generic JMS export bindings differ from EIS export bindings in their handling of the return of the result. A Generic JMS export explicitly sends the response to the `replyTo` destination specified on the incoming message. If none is specified, the send destination is used.

When the Generic JMS export is deployed, a message driven bean (a different MDB than the one used for Generic JMS imports) is deployed. It listens for the incoming requests on the receive destination and then dispatches the requests to be processed by the SCA runtime.

Special headers

Special header properties are used in Generic JMS imports and exports to tell the target binding how to handle the message.

For example, the `TargetFunctionName` property is used by the default function selector to identify the name of the operation in the export interface that is being invoked.

Note: The import binding can be configured to set the `TargetFunctionName` header to the operation name for each operation.

Java EE resources

A number of Java EE resources are created when a JMS binding is deployed to a Java EE environment.

- Listener port for listening on the receive (response) destination (two-way only) for imports and on the receive (request) destination for exports
- Generic JMS connection factory for the `outboundConnection` (import) and `inboundConnection` (export)
- Generic JMS destination for the send (import) and receive (export; two-way only) destinations
- Generic JMS connection factory for the `responseConnection` (two-way only and optional; otherwise, `outboundConnection` is used for imports, and `inboundConnection` is used for exports)
- Generic JMS destination for the receive (import) and send (export) destination (two-way only)
- Default messaging provider callback JMS destination used to access the SIB callback queue destination (two-way only)
- Default messaging provider callback JMS connection factory used to access the callback JMS destination (two-way only)
- SIB callback queue destination used to store information about the request message for use during response processing (two-way only)

The installation task creates the `ConnectionFactory`, the three destinations, and the `ActivationSpec` from the information in the import and export files.

Generic JMS headers

Generic JMS headers are Service Data Objects (SDO) that contain all the properties of the Generic JMS message properties. These properties can be from the inbound message or they can be the properties that will be applied to the outbound message.

A JMS message contains two types of headers—the JMS system header and multiple JMS properties. Both types of headers can be accessed either in a mediation module in the Service Message Object (SMO) or by using the `ContextService` API.

The following properties are set statically on the `methodBinding`:

- `JMSType`

- JMSCorrelationID
- JMSDeliveryMode
- JMSPriority

The Generic JMS binding also supports dynamic modification of JMS headers and properties in the same manner as the JMS and MQ JMS bindings.

Some Generic JMS providers place restrictions on which properties can be set by the application and in what combinations. You must consult your third-party product documentation for more information. However, an additional property has been added to the methodBinding, ignoreInvalidOutboundJMSProperties, which allows any exceptions to be propagated.

The Generic JMS headers and message properties are used only when the base service component architecture SCDL binding switch is turned on. When the switch is turned on, context information is propagated. By default, this switch is on. To prevent context information propagation, change the value to false.

When context propagation is enabled, header information is allowed to flow to the message or to the target component. To turn on and off context propagation, specify true or false for the contextPropagationEnabled attribute of the import and export bindings. For example:

```
<esbBinding xsi:type="eis:JMSImportBinding" contextProagationEnabled="true">
```

The default is true.

Troubleshooting Generic JMS bindings

You can diagnose and fix problems with Generic JMS binding.

Implementation exceptions

In response to various error conditions, the Generic JMS import and export implementation can return one of two types of exceptions:

- Service Business Exception: this exception is returned if the fault specified on the service business interface (WSDL port type) occurred.
- Service Runtime Exception: raised in all other cases. In most cases, the cause exception will contain the original exception (JMSException).

Troubleshooting Generic JMS message expiry

A request message by the JMS provider is subject to expiration.

Request expiry refers to the expiration of a request message by the JMS provider when the JMSExpiration time on the request message is reached. As with other JMS bindings, the Generic JMS binding handles the request expiry by setting expiration on the callback message placed by the import to be the same as for the outgoing request. Notification of the expiration of the callback message will indicate that the request message has expired and the client should be notified by means of a business exception.

If the callback destination is moved to the third-party provider, however, this type of request expiry is not supported.

Response expiry refers to the expiration of a response message by the JMS provider when the JMSExpiration time on the response message is reached.

Response expiry for the generic JMS binding is not supported, because the exact expiry behavior of a third-party JMS provider is not defined. You can, however, check that the response is not expired if and when it is received.

For outbound request messages, the `JMSExpiration` value will be calculated from the time waited and from the `requestExpiration` values carried in the `asyncHeader`, if set.

Troubleshooting Generic JMS connection factory errors

When you define certain types of connection factories in your Generic JMS provider, you might receive an error message when you try to start an application. You can modify the external connection factory to avoid this problem.

When launching an application, you might receive the following error message:

```
MDB Listener Port JMSConnectionFactory type does not match
JMSDestination type
```

This problem can arise when you are defining external connection factories. Specifically, the exception can be thrown when you create a JMS 1.0.2 Topic Connection Factory, instead of a JMS 1.1 (unified) Connection Factory (that is, one that is able to support both point-to-point and publish/subscribe communication).

To resolve this issue, take the following steps:

1. Access the Generic JMS provider that you are using.
2. Replace the JMS 1.0.2 Topic Connection Factory that you defined with a JMS 1.1 (unified) Connection Factory.

When you launch the application with the newly defined JMS 1.1 Connection Factory, you should no longer receive an error message.

Generic JMS-based SCA messages not appearing in the failed event manager

If SCA messages originated through a generic JMS interaction fail, you would expect to find these messages in the failed event manager. If such messages are not appearing in the failed event manager, ensure that the value of the `maximumRetries` property on the underlying listener port is equal to or greater than 1. Setting this value to 1 or more enables interaction with the failed event manager during SCA invocations for the generic JMS bindings.

Handling exceptions

The way in which the binding is configured determines how exceptions that are raised by data handlers or data bindings are handled. Additionally, the nature of the mediation flow dictates the behavior of the system when such an exception is thrown.

A variety of problems can occur when a data handler or data binding is called by your binding. For example, a data handler might receive a message that has a corrupt payload, or it might try to read a message that has an incorrect format.

The way your binding handles such an exception is determined by how you implement the data handler or data binding. The recommended behavior is that you design your data binding to throw a `DataBindingException`.

The situation is similar for a data handler. Since the data handler is invoked by the data binding, any data handler exception is wrapped into a data binding exception. Therefore a `DataHandlerException` is reported to you as a `DataBindingException`.

When any runtime exception, including a `DataBindingException` exception, is thrown:

- If the mediation flow is configured to be transactional, the JMS message is stored in the Failed Event Manager by default for manual replay or deletion.

Note: You can change the recovery mode on the binding so that the message is rolled back instead of being stored in the failed event manager.

- If the mediation flow is not transactional, the exception is logged and the message is lost.

The situation is similar for a data handler. Because the data handler is called by the data binding, a data handler exception is produced inside a data binding exception. Therefore, a `DataHandlerException` is reported to you as a `DataBindingException`.

WebSphere MQ JMS bindings

The WebSphere MQ JMS binding provides integration with external applications that use a WebSphere MQ JMS-based provider.

Use the WebSphere MQ JMS export and import bindings to integrate directly with external JMS or MQ JMS systems from your server environment. This eliminates the need to use MQ Link or Client Link features of the Service Integration Bus.

When a component interacts with a WebSphere MQ JMS-based service by way of an import, the WebSphere MQ JMS import binding utilizes a destination to which data will be sent and a destination where the reply can be received. Conversion of the data to and from a JMS message is accomplished through the JMS data handler or data binding edge component.

When an SCA module provides a service to WebSphere MQ JMS clients, the WebSphere MQ JMS export binding utilizes a destination where the request can be received and the response can be sent. The conversion of the data to and from a JMS message is done through the JMS data handler or data binding.

The function selector provides a mapping to the operation on the target component to be invoked.

WebSphere MQ JMS bindings overview

The WebSphere MQ JMS binding provides integration with external applications that use the WebSphere MQ JMS provider.

WebSphere MQ administrative tasks

The WebSphere MQ system administrator is expected to create the underlying WebSphere MQ Queue Manager, which the WebSphere MQ JMS bindings will use, before running an application containing these bindings.

WebSphere MQ JMS import bindings

The WebSphere MQ JMS import allows components within your SCA module to communicate with services provided by WebSphere MQ JMS-based providers. You

must be using a supported version of WebSphere MQ. Detailed hardware and software requirements can be found on the IBM support pages.

Two types of usage scenarios for WebSphere MQ JMS import bindings are supported, depending on the type of operation being invoked:

- One-way: The WebSphere MQ JMS import puts a message on the send destination configured in the import binding. Nothing is sent to the replyTo field of the JMS header.
- Two-way (request-response): The WebSphere MQ JMS import puts a message on the send destination.

The receive destination is set in the replyTo header field. A message-driven bean (MDB) is deployed to listen on the receive destination, and when a reply is received, the MDB passes the reply back to the component.

The import binding can be configured (using the **Response correlation scheme** field in WebSphere Integration Developer) to expect the response message correlation ID to have been copied from the request message ID (the default) or from the request message correlation ID.

For both one-way and two-way usage scenarios, dynamic and static header properties can be specified. Static properties can be set from the JMS import method binding. Some of these properties have special meanings to the SCA JMS runtime.

It is important to note that WebSphere MQ JMS is an asynchronous binding. If a calling component invokes a WebSphere MQ JMS import synchronously (for a two-way operation), the calling component is blocked until the response is returned by the JMS service.

Figure 38 illustrates how the import is linked to the external service.

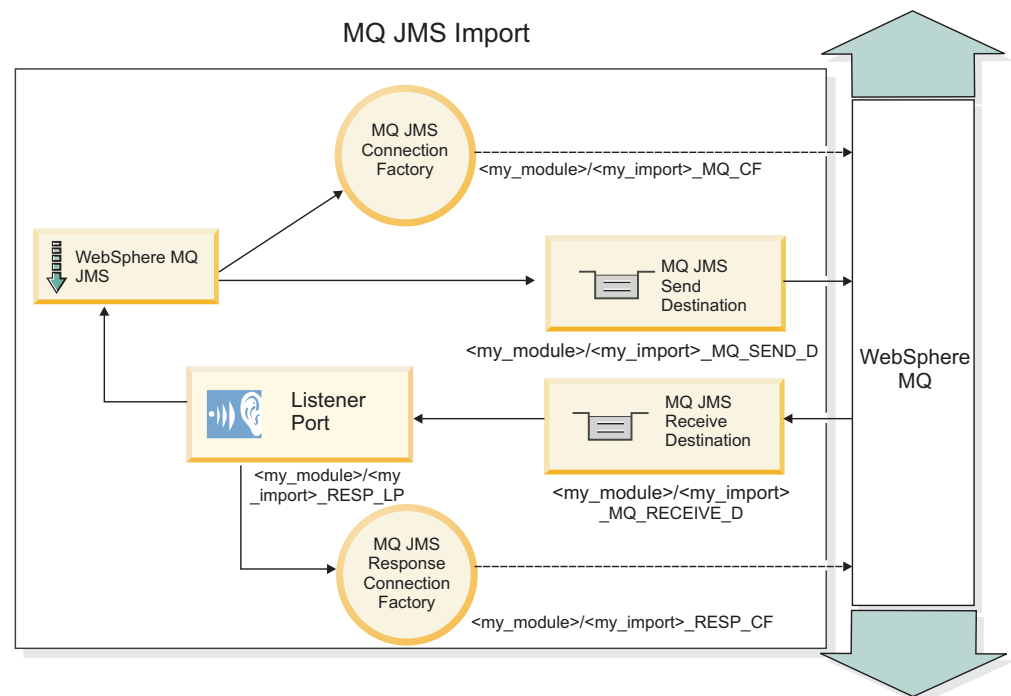


Figure 38. WebSphere MQ JMS import binding resources

WebSphere MQ JMS export bindings

The WebSphere MQ JMS export binding provides the means for SCA modules to provide services to external JMS applications on the WebSphere MQ-based JMS provider.

An MDB is deployed to listen to requests incoming to the receive destination specified in the export binding. The destination specified in the send field is used to send the reply to the inbound request if the invoked component provides a reply. The destination specified in the replyTo field of the response message overrides the destination specified in the send field.

Figure 39 illustrates how the external requestor is linked to the export.

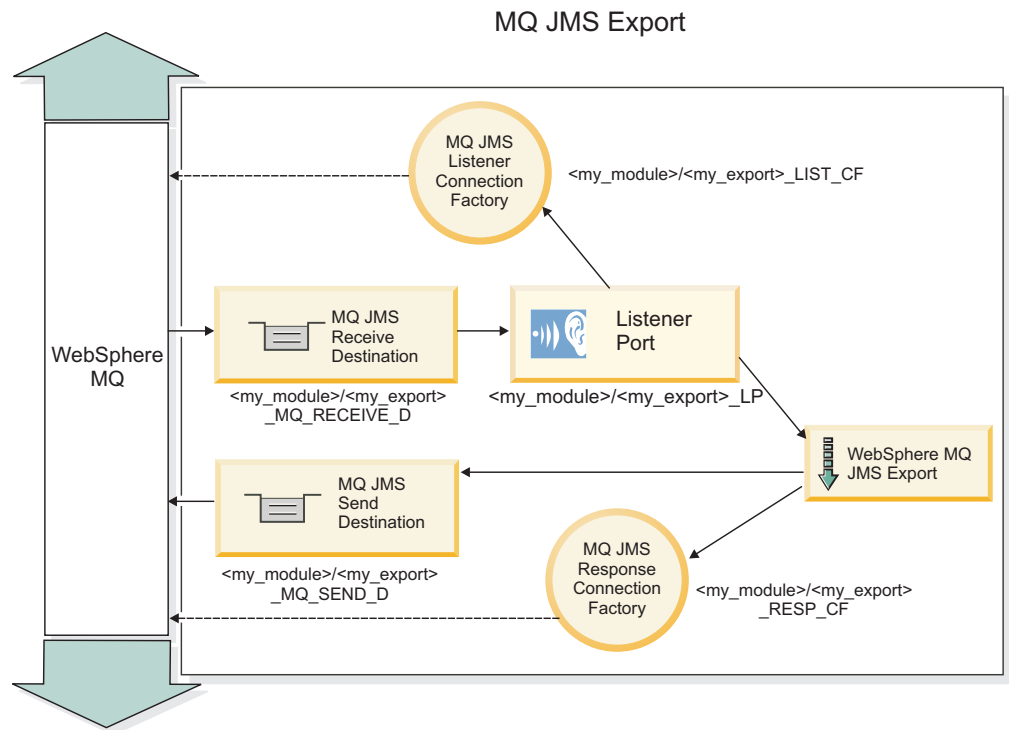


Figure 39. WebSphere MQ JMS export binding resources

Note: Figure 38 on page 87 and Figure 39 illustrate how an application from a previous version of WebSphere Process Server is linked to an external service. For applications developed for WebSphere Process Server Version 7.0, the Activation specification is used instead of the Listener Port and Connection Factory.

Key features of WebSphere MQ JMS bindings

Key features of WebSphere MQ JMS bindings include headers, Java EE artifacts, and created Java EE resources.

Headers

A JMS message header contains a number of predefined fields containing values used by both clients and providers to identify and to route messages. You can use binding properties to configure these headers with fixed values, or the headers can be specified dynamically at runtime.

JMSCorrelationID

Links to a related message. Typically, this field is set to the message identifier string of the message that is being replied to.

TargetFunctionName

This header is used by one of the supplied function selectors to identify the operation being invoked. Setting the TargetFunctionName JMS header property in messages sent to a JMS export allows this function selector to be used. The property can be set directly in JMS client applications or when connecting an import with a JMS binding to such an export. In this case, the JMS import binding should be configured to set the TargetFunctionName header for each operation in the interface to the name of the operation.

Correlation schemes

The WebSphere MQ JMS bindings provide various correlation schemes that are used to determine how to correlate request messages with response messages.

RequestMsgIDToCorrelID

The JMSMessageID is copied to the JMSCorrelationID field. This is the default setting.

RequestCorrelIDToCorrelID

The JMSCorrelationID is copied to the JMSCorrelationID field.

Java EE resources

A number of Java EE resources are created when an MQ JMS import is deployed to a Java EE environment.

Parameters

MQ Connection Factory

Used by clients to create a connection to the MQ JMS provider.

Response Connection Factory

Used by the SCA MQ JMS runtime when the send destination is on a different Queue Manager than the receive destination.

Activation specification

An MQ JMS activation specification is associated with one or more message-driven beans and provides the configuration necessary for them to receive messages.

Destinations

- Send destination:
 - Imports: Where the request or outgoing message is sent.
 - Exports: Where the response message will be sent if it is not superseded by the JMSReplyTo header field of the incoming message.
- Receive destination:
 - Imports: Where the response or incoming message should be placed.
 - Exports: Where the incoming or request message should be placed.

JMS headers

A JMS message contains two types of headers—the JMS system header and multiple JMS properties. Both types of headers can be accessed either in a mediation module in the Service Message Object (SMO) or by using the ContextService API.

JMS system header

The JMS system header is represented in the SMO by the `JMSHeader` element, which contains all the fields typically found in a JMS header. Although these can be modified in the mediation (or `ContextService`), some JMS system header fields set in the SMO will not be propagated in the outbound JMS message as they are overridden by system or static values.

The key fields in the JMS system header that can be updated in a mediation (or `ContextService`) are:

- **JMSType** and **JMSCorrelationID** – values of the specific predefined message header properties
- **JMSDeliveryMode** – values for delivery mode (persistent or nonpersistent; default is persistent)
- **JMSPriority** – priority value (0 to 9; default is `JMS_Default_Priority`)

JMS properties

JMS properties are represented in the SMO as entries in the Properties list. The properties can be added, updated, or deleted in a mediation or by using the `ContextService` API.

Properties can also be set statically in the JMS binding. Properties that are set statically override settings (with the same name) that are set dynamically.

User properties propagated from other bindings (for example, an HTTP binding) will be output in the JMS binding as JMS properties.

Header propagation settings

The propagation of the JMS system header and properties either from the inbound JMS message to downstream components or from upstream components to the outbound JMS message can be controlled by the Propagate Protocol Header flag on the binding.

When Propagate Protocol Header is set, header information is allowed to flow to the message or to the target component, as described in the following list:

- JMS export request
The JMS header received in the message will be propagated to target components by way of the context service. JMS properties received in the message will be propagated to target components by way of the context service.
- JMS export response
Any JMS header fields set in the context service will be used in the outbound message, if not overridden by static properties set on the JMS export binding. Any properties set in the context service will be used in the outbound message if not overridden by static properties set on the JMS export binding.
- JMS import request
Any JMS header fields set in the context service will be used in the outbound message, if not overridden by static properties set on the JMS import binding. Any properties set in the context service will be used in the outbound message if not overridden by static properties set on the JMS import binding.
- JMS import response

The JMS header received in the message will be propagated to target components by way of the context service. JMS properties received in the message will be propagated to target components by way of the context service.

External clients

The server can send messages to, or receive messages from, external clients using WebSphere MQ JMS bindings.

An external client (such as a Web portal or an enterprise information system) can send a message to an SCA component in the application by way of an export or it can be invoked by an SCA component in the application by way of an import.

The WebSphere MQ JMS export binding deploys message driven beans (MDBs) to listen to requests incoming to the receive destination specified in the export binding. The destination specified in the send field is used to send the reply to the inbound request if the invoked application provides a reply. Thus, an external client is able to invoke applications via the export binding.

WebSphere MQ JMS imports bind to, and can deliver message to, external clients. This message might or might not demand a response from the external client.

More information on how to interact with external clients using WebSphere MQ can be found at the WebSphere MQ information center.

Troubleshooting WebSphere MQ JMS bindings

You can diagnose and fix problems with WebSphere MQ JMS bindings.

Implementation exceptions

In response to various error conditions, the MQ JMS import and export implementation can return one of two types of exceptions:

- Service Business Exception: this exception is returned if the fault specified on the service business interface (WSDL port type) occurred.
- Service Runtime Exception: raised in all other cases. In most cases, the cause exception will contain the original exception (JMSEException).

For example, an import expects only one response message for each request message. If more than one response arrives, or if a late response (one for which the SCA response expiration has expired) arrives, a Service Runtime Exception is thrown. The transaction is rolled back, and the response message is backed out of the queue or handled by the failed event manager.

WebSphere MQ JMS-based SCA messages not appearing in the failed event manager

If SCA messages originated through a WebSphere MQ JMS interaction fail, you would expect to find these messages in the failed event manager. If such messages are not appearing in the failed event manager, ensure that the value of the maximum retries property on the underlying listener port is equal to or greater than 1. Setting this value to 1 or more enables interaction with the failed event manager during SCA invocations for the MQ JMS bindings.

Misusage scenarios: comparison with WebSphere MQ bindings

The WebSphere MQ JMS binding is designed to interoperate with JMS applications deployed against WebSphere MQ, which exposes messages according to the JMS

message model. The WebSphere MQ import and export, however, are principally designed to interoperate with native WebSphere MQ applications and expose the full content of the WebSphere MQ message body to mediations.

The following scenarios should be built using the WebSphere MQ JMS binding, not the WebSphere MQ binding:

- Invoking a JMS message-driven bean (MDB) from an SCA module, where the MDB is deployed against the WebSphere MQ JMS provider. Use a WebSphere MQ JMS import.
- Allowing the SCA module to be called from a Java EE component servlet or EJB by way of JMS. Use a WebSphere MQ JMS export.
- Mediating the contents of a JMS MapMessage, in transit across WebSphere MQ. Use a WebSphere MQ JMS export and import in conjunction with the appropriate data handler or data binding.

There are situations in which the WebSphere MQ binding and WebSphere MQ JMS binding might be expected to interoperate. In particular, when you are bridging between Java EE and non-Java EE WebSphere MQ applications, use a WebSphere MQ export and WebSphere MQ JMS import (or vice versa) in conjunction with appropriate data bindings or mediation modules (or both).

Handling exceptions

The way in which the binding is configured determines how exceptions that are raised by data handlers or data bindings are handled. Additionally, the nature of the mediation flow dictates the behavior of the system when such an exception is thrown.

A variety of problems can occur when a data handler or data binding is called by your binding. For example, a data handler might receive a message that has a corrupt payload, or it might try to read a message that has an incorrect format.

The way your binding handles such an exception is determined by how you implement the data handler or data binding. The recommended behavior is that you design your data binding to throw a `DataBindingException`.

The situation is similar for a data handler. Since the data handler is invoked by the data binding, any data handler exception is wrapped into a data binding exception. Therefore a `DataHandlerException` is reported to you as a `DataBindingException`.

When any runtime exception, including a `DataBindingException` exception, is thrown:

- If the mediation flow is configured to be transactional, the JMS message is stored in the Failed Event Manager by default for manual replay or deletion.

Note: You can change the recovery mode on the binding so that the message is rolled back instead of being stored in the failed event manager.

- If the mediation flow is not transactional, the exception is logged and the message is lost.

The situation is similar for a data handler. Because the data handler is called by the data binding, a data handler exception is produced inside a data binding exception. Therefore, a `DataHandlerException` is reported to you as a `DataBindingException`.

WebSphere MQ bindings

The WebSphere MQ binding provides Service Component Architecture (SCA) connectivity with WebSphere MQ applications.

Use the WebSphere MQ export and import bindings to integrate directly with a WebSphere MQ-based system from your server environment. This eliminates the need to use MQ Link or Client Link features of the Service Integration Bus.

When a component interacts with a WebSphere MQ service by way of an import, the WebSphere MQ import binding uses a queue to which data will be sent and a queue where the reply can be received.

When an SCA module provides a service to WebSphere MQ clients, the WebSphere MQ export binding uses a queue where the request can be received and the response can be sent. The function selector provides a mapping to the operation on the target component to be invoked.

Conversion of the payload data to and from an MQ message is done through the MQ body data handler or data binding. Conversion of the header data to and from an MQ message is done through the MQ header data binding.

For information about the WebSphere MQ versions supported, see the WebSphere Process Server system requirements Web site.

WebSphere MQ bindings overview

The WebSphere MQ binding provides integration with native MQ-based applications.

WebSphere MQ administrative tasks

The WebSphere MQ system administrator is expected to create the underlying WebSphere MQ Queue Manager, which the WebSphere MQ bindings will use, before running an application containing these bindings.

WebSphere administrative tasks

You must set the **Native library path** property of the MQ resource adapter in Websphere to the WebSphere MQ version supported by the server, and restart the server. This ensures that the libraries of a supported version of WebSphere MQ are being used. Detailed hardware and software requirements can be found on the IBM support pages

WebSphere MQ import bindings

The WebSphere MQ import binding allows components within your SCA module to communicate with services provided by external WebSphere MQ-based applications. You must be using a supported version of WebSphere MQ. Detailed hardware and software requirements can be found on the IBM support pages.

Interaction with external WebSphere MQ systems includes the use of queues for sending requests and receiving replies.

Two types of usage scenarios for the WebSphere MQ import binding are supported, depending on the type of operation being invoked:

- One-way: The WebSphere MQ import puts a message on the queue configured in the **Send destination queue** field of the import binding. Nothing is sent to the replyTo field of the MQMD header.
- Two-way (request-response): The WebSphere MQ import puts a message on the queue configured in the **Send destination queue** field. The receive queue is set in the replyTo MQMD header field. A message driven bean (MDB) is deployed to listen on the receive queue, and when a reply is received, the MDB passes the reply back to the component. The import binding can be configured (using the **Response correlation scheme** field) to expect the response message correlation ID to have been copied from the request message ID (the default) or from the request message correlation ID.

It is important to note that WebSphere MQ is an asynchronous binding. If a calling component invokes a WebSphere MQ import synchronously (for a two-way operation), the calling component is blocked until the response is returned by the WebSphere MQ service.

Figure 40 illustrates how the import is linked to the external service.

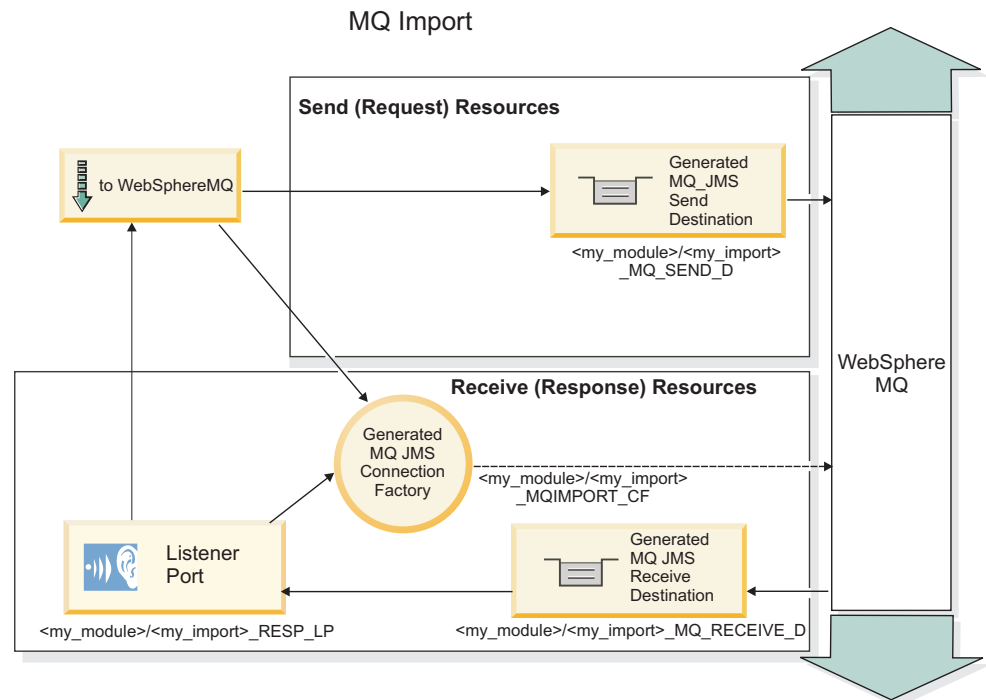


Figure 40. WebSphere MQ import binding resources

WebSphere MQ export bindings

The WebSphere MQ export binding provides the means for SCA modules to provide services to external WebSphere MQ-based applications.

An MDB is deployed to listen to requests incoming to the **Receive destination queue** specified in the export binding. The queue specified in the **Send destination queue** field is used to send the reply to the inbound request if the invoked component provides a reply. The queue specified in the replyTo field of the response message overrides the queue specified in the **Send destination queue** field.

Figure 41 illustrates how the external requestor is linked to the export.

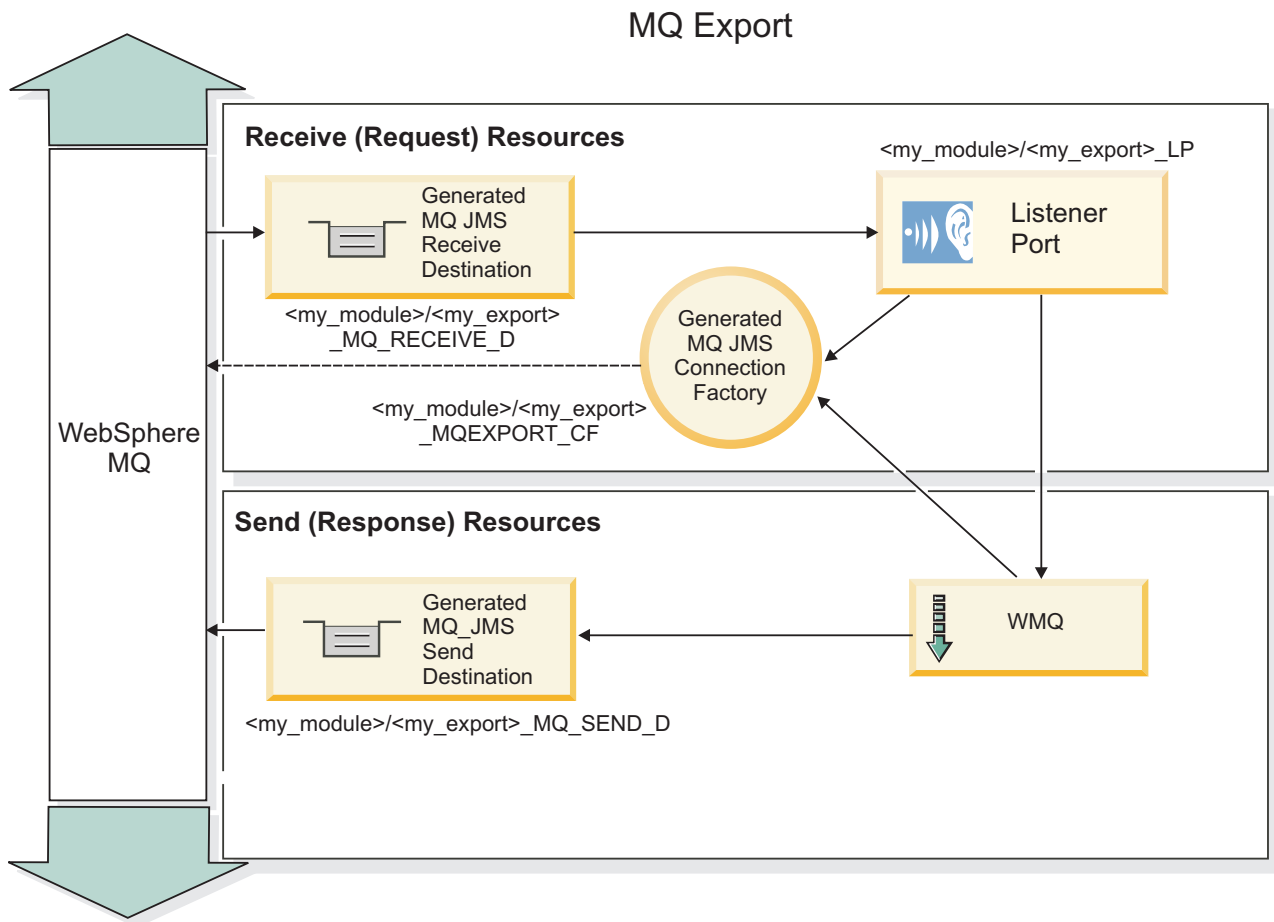


Figure 41. WebSphere MQ export binding resources

Note: Figure 40 on page 94 and Figure 41 illustrate how an application from a previous version of WebSphere Process Server is linked to an external service. For applications developed for WebSphere Process Server Version 7.0, the Activation specification is used instead of the Listener Port and Connection Factory.

Key features of a WebSphere MQ binding

Key features of a WebSphere MQ binding include headers, Java EE artifacts, and created Java EE resources.

Correlation schemes

A WebSphere MQ request/reply application can use one of a number of techniques to correlate response messages with requests, built around the MQMD MessageID and CorrelID fields. In the vast majority of cases, the requestor lets the queue manager select a MessageID and expects the responding application to copy this into the CorrelID of the response. In most cases, the requestor and responding application implicitly know which correlation technique is in use. Occasionally the responding application will honor various flags in the Report field of the request that describe how to handle these fields.

Export bindings for WebSphere MQ messages can be configured with the following options:

Response MsgId options:**New MsgID**

Allows the queue manager to select a unique MsgId for the response (default).

Copy from Request MsgID

Copies the MsgId field from the MsgId field in the request.

Copy from SCA message

Sets the MsgId to be that carried in WebSphere MQ headers in the SCA response message, or lets the queue manager define a new Id if the value does not exist.

As Report Options

Inspects the Report field of the MQMD in the request for a hint as to how to handle the MsgId. The MQRO_NEW_MSG_ID and MQRO_PASS_MSG_ID options are supported and behave like New MsgId and Copy from Request MsgID, respectively

Response CorrelId options:**Copy from Request MsgID**

Copies the CorrelId field from the MsgId field in the request (default).

Copy from Request CorrelID

Copies the CorrelId field from the CorrelId field in the request.

Copy from SCA message

Sets the CorrelId to be carried in WebSphere MQ headers in the SCA response message or leaves it blank if the value does not exist.

As Report Options

Inspects the Report field of the MQMD in the request for a hint as to how to handle the CorrelId. The MQRO_COPY_MSG_ID_TO_CORREL_ID and MQRO_PASS_CORREL_ID options are supported and behave like Copy from Request MsgID and Copy from Request CorrelID, respectively

Import bindings for WebSphere MQ messages can be configured with the following options:

Request MsgId options:**New MsgID**

Allows the queue manager to select a unique MsgId for the request (default)

Copy from SCA message

Sets the MsgId to be carried in WebSphere MQ headers in the SCA request message or lets the queue manager define a new Id if the value does not exist.

Response correlation options:**Response has CorrelID copied from MsgId**

Expects the response message to have a CorrelId field set, per the MsgId of the request (default).

Response has MsgID copied from MsgId

Expects the response message to have a MsgId field set, per the MsgId of the request.

Response has CorrelID copied from CorrelId

Expects the response message to have a CorrelId field set, per the CorrelId of the request.

Java EE resources

A number of Java EE resources are created when a WebSphere MQ binding is deployed to a Java EE environment.

Parameters

MQ Connection Factory

Used by clients to create a connection to the WebSphere MQ provider.

Response Connection Factory

Used by the SCA MQ runtime when the send destination is on a different Queue Manager than the receive destination.

Activation specification

An MQ JMS activation specification is associated with one or more message-driven beans and provides the configuration necessary for them to receive messages.

Destinations

- Send destination: where the request or outgoing message is sent (import); where the response message will be sent (export), if not superseded by the MQMD ReplyTo header field in the incoming message.
- Receive destination: where the response/request or incoming message should be placed.

WebSphere MQ headers

WebSphere MQ headers incorporate certain conventions for conversion to the service component architecture (SCA) messages.

WebSphere MQ messages consist of a system header (the MQMD), zero or more other MQ headers (system or custom), and a message body. If multiple message headers exist in the message, the order of the headers is significant.

Each header contains information describing the structure of the following header. The MQMD describes the first header.

How MQ headers are parsed

An MQ Header data binding is used to parse MQ headers. The following headers are supported automatically:

- MQRFH
- MQRFH2
- MQCIH
- MQIIH

Headers that start with MQH are handled differently. Specific fields of the header are not parsed; they remain as unparsed bytes.

For other MQ headers, you can write custom MQ header data bindings to parse those headers.

How MQ headers are accessed

MQ headers can be accessed in the product in one of two ways:

- Through the service message object (SMO) in a mediation

- Through the ContextService API

MQ headers are represented internally with the SMO MQHeader element. MQHeader is a container of header data that extends MQControl but contains a value element of anyType. It contains the MQMD, MQControl (MQ message body control information), and a list of other MQ headers.

- MQMD represents the contents of the WebSphere MQ message description, except for information determining the structure and encoding of the body.
- MQControl contains information determining the structure and encoding of a message body.
- MQHeaders contain a list of MQHeader objects.

The MQ header chain is unwound so that, inside the SMO, each MQ header carries its own control information (CCSID, Encoding, and Format). Headers can be added or deleted easily, without altering other header data.

Setting fields in the MQMD

You can update the MQMD using the Context API or through the service message object (SMO) in a mediation. The following fields are automatically propagated to the outbound MQ message:

- Encoding
- CodedCharacterSet
- Format
- Report
- Expiry
- Feedback
- Priority
- Persistence
- CorrelId
- MsgFlags

Configure the MQ binding on an Import or Export to propagate the following properties to the outbound MQ message:

MsgID

Set **Request Message ID** to copy from SCA message.

MsgType

Clear the **Set message type to MQMT_DATAGRAM or MQMT_REQUEST for request-response operation** check box.

ReplyToQ

Clear the **Override reply to queue of request message** check box.

ReplyToQMgr

Clear the **Override reply to queue of request message** check box.

From version 7.0 onwards, context fields can be overridden using a custom property on the JNDI destination definition. Set the custom property MDCTX with value SET_IDENTITY_CONTEXT on the send destination to propagate the following fields to the outbound MQ message:

- UserIdentifier
- AppIdentityData

Set the custom property MDCTX with value SET_ALL_CONTEXT on the send destination to propagate the following properties to the outbound MQ message:

- UserIdentifier
- AppIdentityData
- PutApplType
- PutApplName
- ApplOriginData

Some fields are not propagated to the outbound MQ message. The following fields are overridden during the send of the message:

- BackoutCount
- AccountingToken
- PutDate
- PutTime
- Offset
- OriginalLength

Adding MQCIH statically in a WebSphere MQ binding

WebSphere Process Server supports adding MQCIH header information statically without using a mediation module.

There are various ways to add MQCIH header information to a message (for example, by using the Header Setter mediation primitive). It might be useful to add this header information statically, without the use of an additional mediation module. Static header information, including the CICS[®] program name, the transaction ID, and other data format header details, can be defined and added as part of the WebSphere MQ binding.

WebSphere MQ, the MQ CICS Bridge, and CICS must be configured for MQCIH header information to be added statically.

You can use WebSphere Integration Developer to configure the WebSphere MQ import with the static values that are required for the MQCIH header information.

When a message arrives and is processed by the WebSphere MQ import, a check is made to see if MQCIH header information is already present in the message. If the MQCIH is present, the static values defined in the WebSphere MQ import are used to override the corresponding dynamic values in the message. If the MQCIH is not present, one is created in the message and the static values defined in the WebSphere MQ import are added.

The static values defined in the WebSphere MQ import are specific to a method. You can specify different static MQCIH values for different methods within the same WebSphere MQ import.

This facility is not used to provide default values if the MQCIH does not contain specific header information because a static value defined in the WebSphere MQ import will override a corresponding value provided in the incoming message.

External clients

WebSphere Process Server can send messages to, or receive messages from, external clients using WebSphere MQ bindings.

An external client (for example, a Web portal or an enterprise information system) can send a message to an SCA component in the application by way of an export or it can be invoked by an SCA component in the application by way of an import.

The WebSphere MQ export binding deploys message driven beans (MDBs) to listen to requests incoming to the receive destination specified in the export binding. The destination specified in the send field is used to send the reply to the inbound request if the invoked application provides a reply. Thus, an external client is able to invoke applications by way of the export binding.

WebSphere MQ imports bind to, and can deliver message to, external clients. This message might or might not demand a response from the external client.

More information on how to interact with external clients using WebSphere MQ can be found at the WebSphere MQ information center.

Troubleshooting WebSphere MQ bindings

You can diagnose and fix faults and failure conditions that occur with WebSphere MQ bindings.

Primary failure conditions

The primary failure conditions of WebSphere MQ bindings are determined by transactional semantics, by WebSphere MQ configuration, or by reference to existing behavior in other components. The primary failure conditions include:

- Failure to connect to the WebSphere MQ queue manager or queue.
A failure to connect to WebSphere MQ to receive messages will result in the MDB Listener Port failing to start. This condition will be logged in the WebSphere Application Server log. Persistent messages will remain on the WebSphere MQ queue until they are successfully retrieved (or expired by WebSphere MQ).
A failure to connect to WebSphere MQ to send outbound messages will cause rollback of the transaction controlling the send.
- Failure to parse an inbound message or to construct an outbound message.
A failure in the data binding causes rollback of the transaction controlling the work.
- Failure to send the outbound message.
A failure to send a message causes rollback of the relevant transaction.
- Multiple or unexpected response messages.
The import expects only one response message for each request message. If more than one response arrives, or if a late response (one for which the SCA response expiration has expired) arrives, a Service Runtime Exception is thrown. The transaction is rolled back, and the response message is backed out of the queue or handled by the failed event manager.

Misusage scenarios: comparison with WebSphere MQ JMS bindings

The WebSphere MQ import and export are principally designed to interoperate with native WebSphere MQ applications and expose the full content of the WebSphere MQ message body to mediations. The WebSphere MQ JMS binding, however, is designed to interoperate with JMS applications deployed against WebSphere MQ, which exposes messages according to the JMS message model.

The following scenarios should be built using the WebSphere MQ JMS binding, not the WebSphere MQ binding:

- Invoking a JMS message-driven bean (MDB) from an SCA module, where the MDB is deployed against the WebSphere MQ JMS provider. Use a WebSphere MQ JMS import.
- Allowing the SCA module to be called from a Java EE component servlet or EJB by way of JMS. Use a WebSphere MQ JMS export.
- Mediating the contents of a JMS MapMessage, in transit across WebSphere MQ. Use a WebSphere MQ JMS export and import in conjunction with the appropriate data binding.

There are situations in which the WebSphere MQ binding and WebSphere MQ JMS binding might be expected to interoperate. In particular, when you are bridging between Java EE and non-Java EE WebSphere MQ applications, use a WebSphere MQ export and WebSphere MQ JMS import (or vice versa) in conjunction with appropriate data bindings or mediation modules (or both).

Undelivered messages

If WebSphere MQ cannot deliver a message to its intended destination (because of configuration errors, for example), it sends the messages instead to a nominated dead-letter queue.

In doing so, it adds a dead-letter header to the start of the message body. This header contains the failure reasons, the original destination, and other information.

MQ-based SCA messages not appearing in the failed event manager

If SCA messages originated because of a WebSphere MQ interaction failure, you would expect to find these messages in the failed event manager. If these messages are not showing in the failed event manager, check that the underlying WebSphere MQ destination has a maximum failed deliveries value greater than 1. Setting this value to 2 or more allows interaction with the failed event manager during SCA invocations for the WebSphere MQ bindings.

MQ failed events are replayed to the wrong queue manager

When a predefined connection factory is to be used for outbound connections, the connection properties must match those defined in the activation specification used for inbound connections.

The predefined connection factory is used to create a connection when replaying a failed event and must therefore be configured to use the same queue manager from which the message was originally received.

Handling exceptions

The way in which the binding is configured determines how exceptions that are raised by data handlers or data bindings are handled. Additionally, the nature of the mediation flow dictates the behavior of the system when such an exception is thrown.

A variety of problems can occur when a data handler or data binding is called by your binding. For example, a data handler might receive a message that has a corrupt payload, or it might try to read a message that has an incorrect format.

The way your binding handles such an exception is determined by how you implement the data handler or data binding. The recommended behavior is that you design your data binding to throw a `DataBindingException`.

The situation is similar for a data handler. Since the data handler is invoked by the data binding, any data handler exception is wrapped into a data binding exception. Therefore a `DataHandlerException` is reported to you as a `DataBindingException`.

When any runtime exception, including a `DataBindingException` exception, is thrown:

- If the mediation flow is configured to be transactional, the JMS message is stored in the Failed Event Manager by default for manual replay or deletion.

Note: You can change the recovery mode on the binding so that the message is rolled back instead of being stored in the failed event manager.

- If the mediation flow is not transactional, the exception is logged and the message is lost.

The situation is similar for a data handler. Because the data handler is called by the data binding, a data handler exception is produced inside a data binding exception. Therefore, a `DataHandlerException` is reported to you as a `DataBindingException`.

Limitations of bindings

The bindings have some limitations in their use that are listed here.

Limitations of the MQ binding

The MQ binding has some limitations in its use that are listed here.

No publish-subscribe message distribution

The publish-subscribe method of distributing messages is not currently supported by the MQ binding though WMQ itself supports publish-subscribe. However, the MQ JMS binding does support this method of distribution.

Shared receive queues

Multiple WebSphere MQ export and import bindings expect that any messages present on their configured receive queue are intended for that export or import. Import and export bindings should be configured with the following considerations:

- Each MQ import must have a different receive queue because the MQ import binding assumes all messages on the receive queue are responses to requests that it sent. If the receive queue is shared by more than one import, responses could be received by the wrong import and will fail to be correlated with the original request message.
- Each MQ export should have a different receive queue, because otherwise you cannot predict which export will get any particular request message.
- MQ imports and exports can point to the same send queue.

Limitations of the JMS, MQ JMS, and generic JMS bindings

The JMS and MQ JMS bindings have some limitations.

Implications of generating default bindings

The limitations of using the JMS, MQ JMS, and generic JMS bindings are discussed in the following sections:

- Implications of generating default bindings
- Response correlation scheme
- Bidirectional support

When you generate a binding, several fields will be filled in for you as defaults, if you do not choose to enter the values yourself. For example, a connection factory name will be created for you. If you know that you will be putting your application on a server and accessing it remotely with a client, you should at binding creation time enter JNDI names rather than take the defaults since you will likely want to control these values through the administrative console at run time.

However, if you did accept the defaults and then find later that you cannot access your application from a remote client, you can use the administrative console to explicitly set the connection factory value. Locate the provider endpoints field in the connection factory settings and add a value such as <server_hostname>:7276 (if using the default port number).

Response correlation scheme

If you use the CorrelationId To CorrelationId response correlation scheme, used to correlate messages in a request-response operation, you must have a dynamic correlation ID in the message.

To create a dynamic correlation ID in a mediation module using the mediation flow editor, add an XSLT node before the import with the JMS binding. Open the XSLT mapping editor. The known service component architecture headers will be available in the target message. Drag and drop a field containing a unique ID in the source message onto the correlation ID in the JMS header in the target message.

Bidirectional support

Only ASCII characters are supported for Java Naming and Directory Interface (JNDI) names at runtime.

Shared receive queues

Multiple export and import bindings expect that any messages present on their configured receive queue are intended for that export or import. Import and export bindings should be configured with the following considerations:

- Each import binding must have a different receive queue because the import binding assumes all messages on the receive queue are responses to requests that it sent. If the receive queue is shared by more than one import, responses could be received by the wrong import and will fail to be correlated with the original request message.
- Each export should have a different receive queue, because otherwise you cannot predict which export will get any particular request message.
- Imports and exports can point to the same send queue.

Programming guides and techniques

This section includes programming guides and examples.

The following subtopics provide information for programming various components, applications, and business integration solutions.

Important: See the Reference section of the infocenter for details of the application programming interfaces (APIs) and system programming interfaces (SPIs) that are supported by WebSphere Process Server and WebSphere Enterprise Service Bus.

Service Component Architecture programming

Service Component Architecture (SCA) provides a simple, yet powerful programming model for constructing applications based on service-oriented architecture (SOA).

Service Component Definition Language

Service Component Definition Language (SCDL) is an XML-based language used to describe Service Component Architecture (SCA) elements such as modules, components, references, imports, and exports.

The various artifact types that exist in SCA were designed to support some of the basic requirements of this service-oriented architecture. To start with, SCA needs a mechanism for defining a basic service component. Once there is a mechanism for defining service components, it is important to be able to make these services available to clients both inside or outside of the current SCA module. In addition to this, a construct designed to import and reference services external to the current SCA module must exist. Finally, SCA provides constructs for composing services and modules into larger applications.

SCDL definitions are organized across several files. For example, in a credit approval application, we can store the SCDL for the interface and implementation in a file called `CreditApproval.component`. References can be included in the `CreditApproval.component` file (inline) or in a separate `sca.references` file located in the module root. Any stand-alone reference is placed in the `sca.references` file. Stand-alone references can be used by non-SCA artifacts (JSP) within the same SCA module to invoke the SCA component.

Table 14. Primary artifacts that make up an SCA service module

Artifact	SCDL Definition
Module Definition	<ul style="list-style-type: none">• Contained in the <code>sca.module</code> file at the root SCA project JAR
Service Components	<ul style="list-style-type: none">• A module can contain 0..n service definitions• Each component definition is contained in a <code><SERVICE_NAME>.component</code> file

Table 14. Primary artifacts that make up an SCA service module (continued)

Artifact	SCDL Definition
Imports	<ul style="list-style-type: none"> • A module can contain 0..n import definitions • Each import definition is contained in a <IMPORT_NAME>.import file
Exports	<ul style="list-style-type: none"> • A module can contain 0..n export definitions • Each export definition is contained in a <EXPORT_NAME>.export file
References	<ul style="list-style-type: none"> • Two types of references <ul style="list-style-type: none"> – Inline (contained within a service component definition) – Stand-alone • Each component definition is contained in a <SERVICE_NAME>.reference file
Other Artifacts	<ul style="list-style-type: none"> • Other artifacts include: Java Classes, WSDL files, Other Artifacts XSD files, BPEL.

When building an SCA application, WebSphere Integration Developer takes care of generating the appropriate SCDL definitions. However, a basic familiarity with SCDL can help you to understand the overall architecture and assist when debugging applications.

Module definition

Service Component Architecture (SCA) defines a standard deployment model for packaging components into a service module. The `sca.module` file contains the definition of the module.

An SCA module is not just another type of package. In WebSphere Process Server, an SCA service module is equivalent to a Java EE EAR file and several other Java EE sub-modules. Java EE elements, such as a WAR file, can be packaged along with the SCA module. Non-SCA artifacts (JSPs, and others) can also be packaged together with an SCA service module, enabling them to invoke SCA services through the SCA client programming model using a special type of reference called a stand-alone reference.

Here is an example of a `sca.module` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:module xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
name="CreditApproval"/>
```

The diagram shows a module in WebSphere Integration Developer along with its associated SCDL module definition that you can view in an editor. In this example, the module type is a mediation module.


```

<?xml version="1.0" encoding="UTF-8"?>
<scdl:module xmlns:mt="http://www.ibm.com/xmlns/prod/websphere/scdl/moduletype/7.0.0"
  xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0" name="StockQuote">
  <mt:moduleType type="com.ibm.ws.sca.scdl.moduletype.mediation" version="1.0.0"/>
</Scdl:module>

```

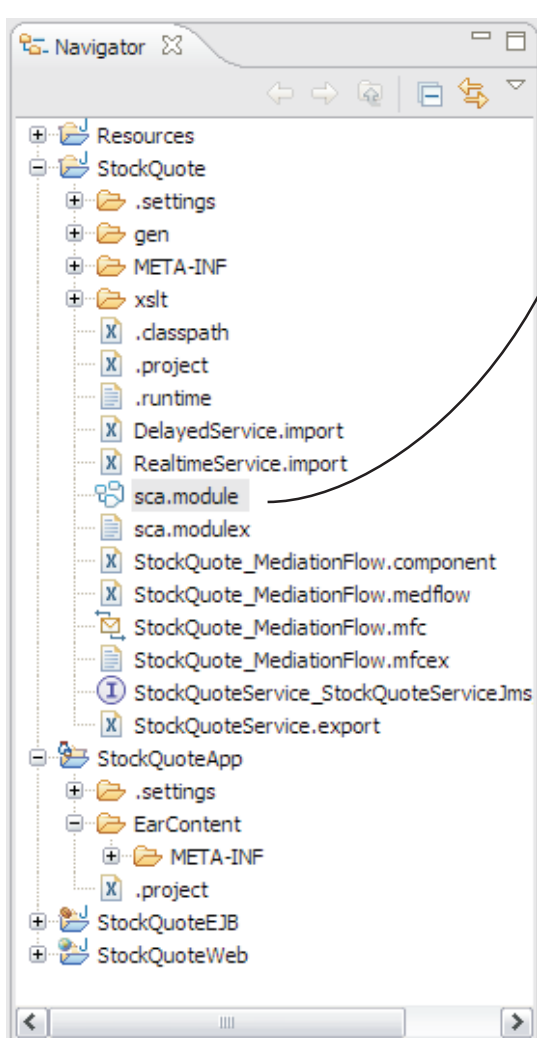


Figure 42. Showing the relationship between a module in WebSphere Integration Developer and the module definition

Component definition

The service component definition is included in a file called `<SERVICE_NAME>.component`. SCA components with their associated dependencies can be defined and packaged together into deployable units.

This figure provides a more detailed look at the service component definition. Each service component must have a unique name within the SCA module and it must match the file path relative to the module root. As noted on the previous slide, the service component definition is included in a file called `<SERVICE_NAME>.component`.

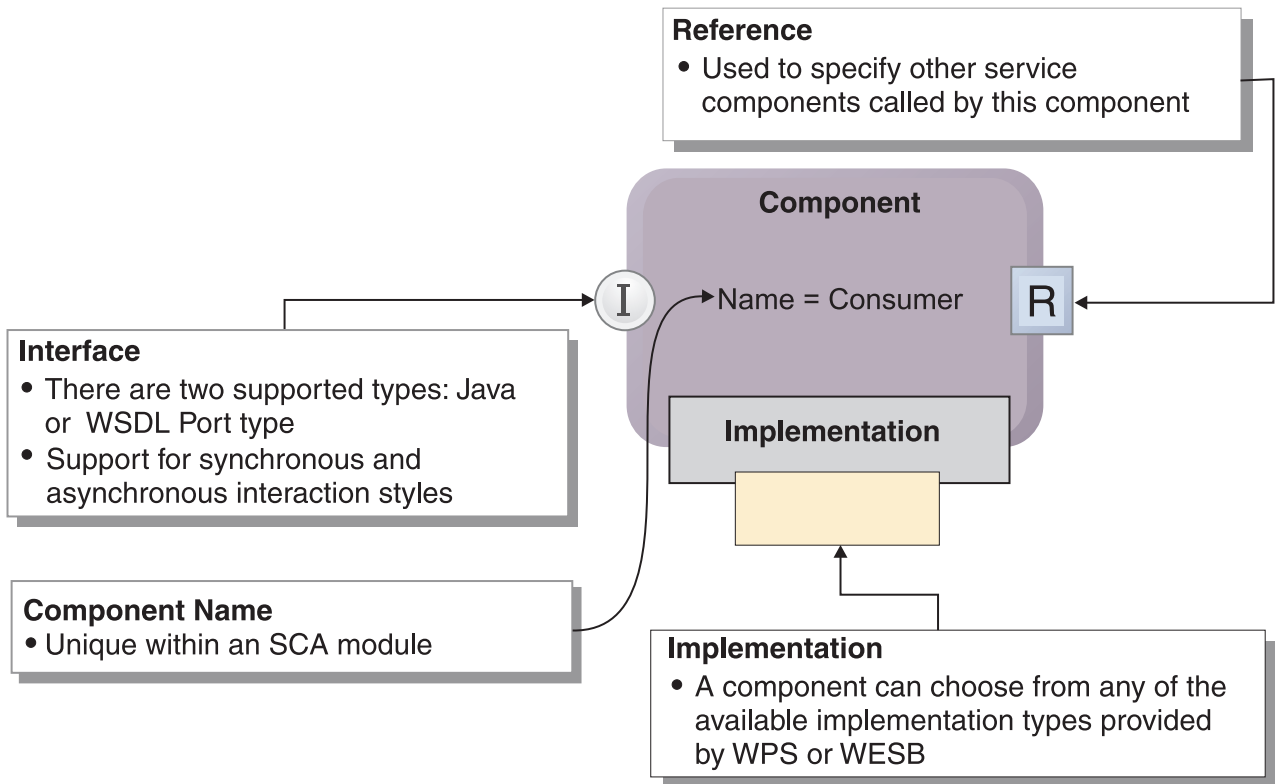
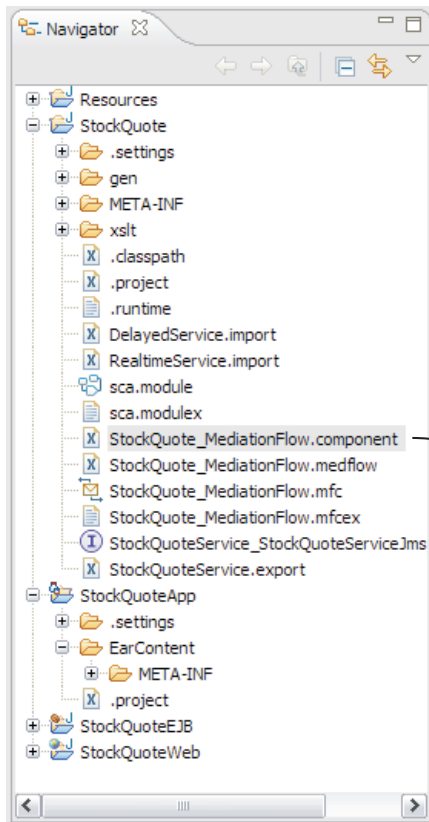


Figure 43. Service component definition including component name, implementation, interfaces, and references

Each service component must have a unique name within the SCA module and it must match the file path relative to the module root. Next, each service component can have one or more interfaces associated with it, which can be either Java or WSDL port type interface definitions. The interfaces associated with a service component can support either a synchronous or asynchronous interaction style with clients calling the service.

Each service component can be implemented in various ways, specified by the implementation definition. Finally, service components can invoke other service components or imports defined in the current service module. In this case, the appropriate reference must be defined to indicate which service is used. Often this type of reference is in lined in the service component definition, although it may alternatively be placed in the stand-alone references file. Each service component definition can have zero or more references to other services called by the service component being defined.

Here is an example that shows the definition for the StockQuote_MediationFlow component. Notice that the component includes definitions for a WSDL interface, two references, and an implementation.



```

<?xml version="1.0" encoding="UTF-8"?>
<scdl:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mfc="http://www.ibm.com/xmlns/prod/websphere/scdl/mfc/7.0.0"
  xmlns:ns1="http://Resources/StockQuoteService"
  xmlns:ns2="http://stockquote.samp.sibx.websphere.ibm.com/DelayedService/"
  xmlns:ns3="http://stockquote.samp.sibx.websphere.ibm.com/RealtimeService/"
  xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
  xmlns:wSDL="http://www.ibm.com/xmlns/prod/websphere/scdl/wSDL/7.0.0"
  DisplayName="StockQuote_MediationFlow" name="StockQuote_MediationFlow">
  <interfaces>
    <interface xsi:type="wSDL:WSDLPortType" portType="ns1:StockQuoteService"/>
  </interfaces>
  <references>
    <reference name="DelayedServicePortTypePartner">
      <interface xsi:type="wSDL:WSDLPortType" portType="ns2:DelayedServicePortType">
        <method name="getQuote"/>
      </interface>
      <wire target="DelayedService"/>
    </reference>
    <reference name="RealtimeServicePortTypePartner">
      <interface xsi:type="wSDL:WSDLPortType" portType="ns3:RealtimeServicePortType">
        <Method name="getQuote"/>
      </interface>
      <wire target="RealtimeService"/>
    </reference>
  </references>
  <implementation xsi:type="mfc:MediationFlowImplementation" mfcFile="StockQuote_MediationFlow.mfc"/>
</Scdl:component>

```

Figure 44. Example of a component definition in Service Component Definition Language

Import definition

The import definition is included in a file called `<IMPORT_NAME>.import`. SCA imports allow clients in an SCA module to access services that are outside the current SCA module.

Like service components, imports have a name and a set of 1..N interfaces with which they are associated. Imports also have a binding attribute, which is used to describe how the external service is bound to the current module. The common binding types are indicated in the diagram.

Imports can be thought of as a special type of service component in an SCA module. Imports are valid targets in a wire definition for a service reference. This means that to a client invoking a target service the client programming model is the same whether the reference points to an import or another service component.

```

<?xml version="1.0" encoding="UTF-8"?>
<scdl:import xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns1="http://stockquote.samp.sibx.websphere.ibm.com/DelayedService/"
  xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
  xmlns:webservice="http://www.ibm.com/xmlns/prod/websphere/scdl/webservice/7.0.0"
  xmlns:wSDL="http://www.ibm.com/xmlns/prod/websphere/scdl/wSDL/7.0.0"
  displayName="DelayedService" name="DelayedService">
  <interfaces>
    <interface xsi:type="wSDL:WSDLPortType" portType="ns1:DelayedServicePortType">
      <method name="getQuote"/>
    </interface>
  </interfaces>
  <esbBinding xsi:type="webservice:WebServiceImportBinding"
    endpoint="http://localhost:9080/DelayedService/services/DelayedServiceSOAP"
    port="ns1:DelayedServiceSOAP" service="ns1:DelayedService"/>
</scdl:import>

```

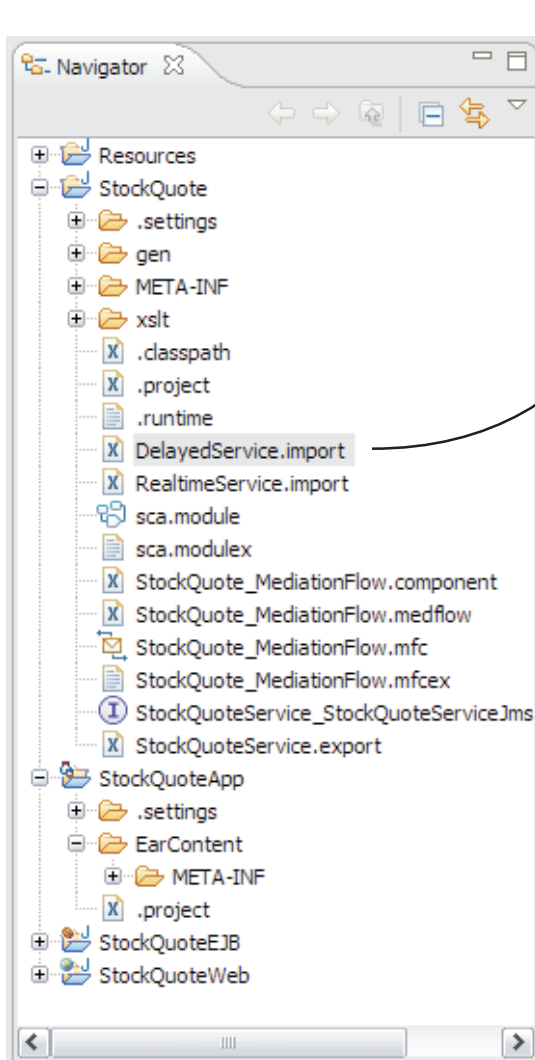
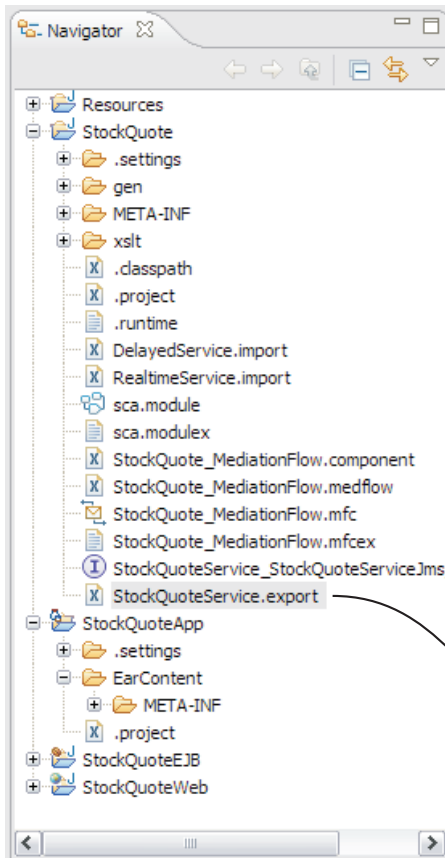


Figure 45. Example of an import definition in Service Component Definition Language

Export definition

The export definition is included in a file called <EXPORT_NAME>.export. SCA exports provide access to service components defined in an SCA module for use by clients outside of the current SCA module.

Export components include a name and a target attribute, which names the service component that is to be exported. Like import components, exports have a binding attribute that indicates how the service is bound externally. The common binding types are indicated in the diagram.



```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:export xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:_="http://Resources/StockQuoteService/Binding"
  xmlns:ns1="http://Resources/StockQuoteService"
  xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
  xmlns:webservice="http://www.ibm.com/xmlns/prod/websphere/scdl/webservice/7.0.0"
  xmlns:wSDL="http://www.ibm.com/xmlns/prod/websphere/scdl/wSDL/7.0.0"
  displayName="StockQuoteService" name="StockQuoteService" target="StockQuote_MediationFlow">
  <interfaces>
    <interface xsi:type="wSDL:WSDLPortType" portType="ns1:StockQuoteService">
      <method name="getQuote"/>
    </interface>
  </interfaces>
</export>
```

Figure 46. Example of an export definition in Service Component Definition Language

Reference definition

SCA and non-SCA clients calling a service component need a reference to that service in order to invoke it. References can be defined either as stand-alone in the `sca.reference` file or as inline within a service composition definition.

Each reference has a name, used to look up the appropriate service by a client using the client programming model. In addition to the name, a reference also includes an interface element. The multiplicity for a reference indicates how many wire definitions can name this reference as the source. Finally, the wire definition specifies the name of the target service component or import that resolves the reference.

There are two ways to define references. The first way is to inline the reference in the service component definition. Using this approach, the references are only available to the service component in which the references are included. Another approach is to include reference definitions within the stand-alone references file. For this approach, the references can be used by a non-SCA client or by another component within the module. An example of a non-SCA component that uses a reference in the stand-alone references file is a user interface component such as a JSP that needs the ability to invoke a particular service. In order to invoke a service component, the client needs a reference so that it can use the SCA runtime to look up the appropriate service to invoke.

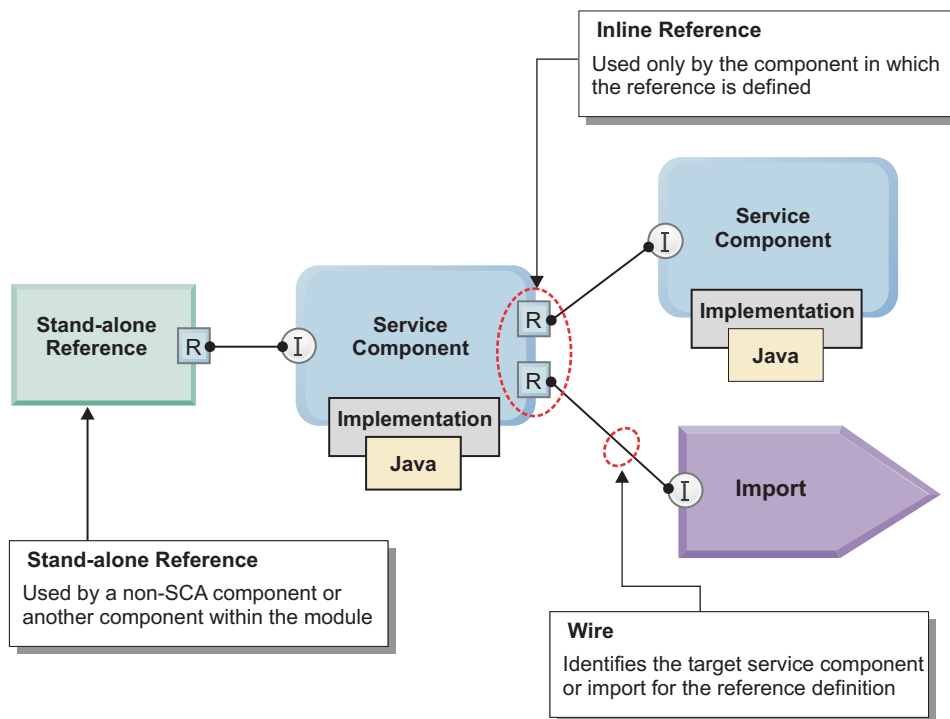


Figure 47. Clients can call a service component using either stand-alone or inline references

Here is an example that shows the definition for the component that includes two inline references: `DelayedServicePortTypePartner` and `RealtimeServicePortTypePartner`. Notice that the component includes definitions for a WSDL interface, two references, and an implementation.

```

<?xml version="1.0" encoding="UTF-8"?>
<scdl:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mfc="http://www.ibm.com/xmlns/prod/websphere/scdl/mfc/7.0.0"
  xmlns:ns1="http://Resources/StockQuoteService"
  xmlns:ns2="http://stockquote.samp.sibx.websphere.ibm.com/DelayedService/"
  xmlns:ns3="http://stockquote.samp.sibx.websphere.ibm.com/RealtimeService/"
  xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
  xmlns:wSDL="http://www.ibm.com/xmlns/prod/websphere/scdl/wSDL/7.0.0"
  DisplayName="StockQuote_MediationFlow" name="StockQuote_MediationFlow">
  <Interfaces>
    <interface xsi:type="wSDL:WSDLPortType" portType="ns1:StockQuoteService"/>
  </Interfaces>
  <references>
    <reference name="DelayedServicePortTypePartner">
      <interface xsi:type="wSDL:WSDLPortType" portType="ns2:DelayedServicePortType">
        <method name="getQuote"/>
      </interface>
      <wire target="DelayedService"/>
    </reference>
    <reference name="RealtimeServicePortTypePartner">
      <interface xsi:type="wSDL:WSDLPortType" portType="ns3:RealtimeServicePortType">
        <Method name="getQuote"/>
      </interface>
      <wire target="RealtimeService"/>
    </reference>
  </references>
  <implementation xsi:type="mfc:MediationFlowImplementation" mfcFile="StockQuote_MediationFlow.mfc"/>
</Scdl:component>

```

Figure 48. Example of inline reference definitions

SCA programming model fundamentals

The concept of a software *component* forms the basis of the Service Component Architecture (SCA) programming model. A component is a unit that implements some logic and makes it available to other components through an interface. A component may also require the services made available by other components. In that case, the component exposes a *reference* to these services.

In SCA, every component must expose at least one interface. The assembly diagram shown in Figure 49 on page 115 has three components. Each component has an interface that is represented by the letter I in a circle. A component can also refer to other components. References are represented by the letter R in a square. References and interfaces are then linked in an assembly diagram. Essentially, the integration developer “resolves” the references by connecting them with the interfaces of the components that implement the required logic.

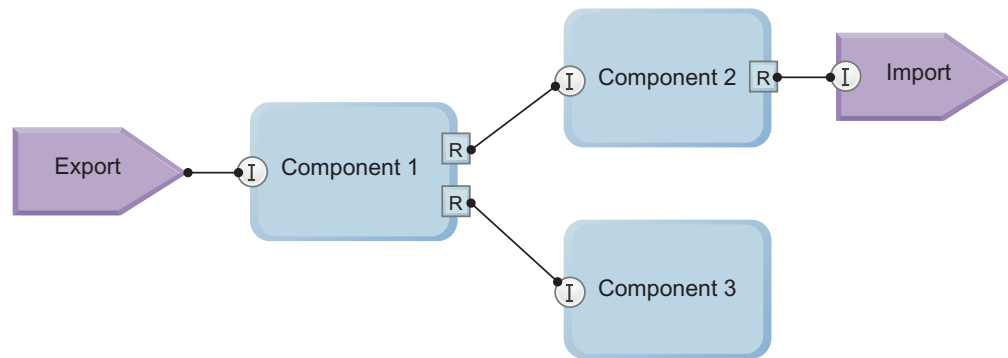


Figure 49. Assembly diagram

Invoking SCA Components

To provide access to the services to be invoked, the SCA programming model includes a *ServiceManager* class, which enables developers to look up available services by name. Here is a typical Java code fragment illustrating service lookup. The *ServiceManager* is used to obtain a reference to the *BOFactory* service, which is a system-provided service:

```
//Get service manager singleton
ServiceManager smgr = new ServiceManager();
//Access BOFactory service
BOFactory bof =(BOFactory)
    smgr.locateService("com/ibm/websphere/bo/BOFactory");
```

Note: The package for *ServiceManager* is `com.ibm.websphere.sca`.

Developers can use a similar mechanism to obtain references to their own services by specifying the name of the service referenced in the *locateService* method. After you have obtained a reference to a service using the *ServiceManager* class, you can invoke any of the available operations on that service in a way that is independent of the invocation protocol and the type of implementation.

SCA components can be called using three different invocation styles:

- **Synchronous invocation:** When using this invocation style, the caller waits synchronously for the response to be returned. This style is the classic invocation mechanism.
- **Asynchronous invocation:** This mechanism allows the caller to invoke a service without waiting for the response to be produced right away. Instead of getting the response, the caller gets a “ticket,” which can be used later to retrieve the response. The caller retrieves the response by calling a special operation that must be provided by the callee for this purpose.
- **Asynchronous invocation with callback:** This invocation style is like the preceding one, but it delegates the responsibility of returning the response to the callee. The caller needs to expose a special operation (the callback operation) that the callee can invoke when the response is ready.
- **Asynchronous invocation with deferred response:** In this invocation style, the client invokes a service and then continues processing until some later time when the client makes a request to capture the response.

Imports

Sometimes, business logic is provided by components or functions that are available on external systems, such as legacy applications, or other external implementations. In those cases, the integration developer cannot resolve the reference by connecting a reference to a component containing the implementation he or she needs to connect the reference to a component that “points to” the external implementation. Such a component is called an *import*. When you define an import, you need to specify how the external service can be accessed in terms of location and the invocation protocol.

Exports

Similarly, if your component has to be accessed by external applications, which is often the case, you must make it accessible. You make it accessible by using a special component that exposes your logic to the “outside world.” Such a component is called an *export*. Exports can also be invoked synchronously or asynchronously.

Stand-alone references

In WebSphere Process Server, an SCA service module is packaged as a Java EE EAR file that contains several other Java EE submodules. Java EE elements, such as a WAR file, can be packaged along with the SCA module. Non-SCA artifacts such as JSPs can also be packaged together with an SCA service module. This packaging lets them invoke SCA services through the SCA client programming model using a special type of component called a stand-alone reference.

The SCA programming model is strongly declarative. Integration developers can configure aspects such as transactional behavior of invocations, propagation of security credentials, whether an invocation should be synchronous or asynchronous in a declarative way, directly in the assembly diagram. The SCA runtime, not the developers, is responsible for taking care of implementing the behavior specified in these modifiers. The declarative flexibility of SCA is one of the most powerful features of this programming model. Developers can concentrate on implementing business logic, rather than focusing on addressing technical aspects, such as being able to accommodate asynchronous invocation mechanisms. All these aspects are automatically taken care of by the SCA runtime.

Qualifiers

The qualifiers govern the interaction between a service client and a target service. Qualifiers can be specified on service component references, interfaces, and implementations and are typically external to an implementation.

The different categories of qualifiers include the following:

- Transaction, which specifies the way transactional contexts are handled in an SCA invocation
- Activity session, which specifies how Activity Session contexts are propagated.
- Security, which specifies the permissions
- Asynchronous reliability provides rules for asynchronous message delivery

SCA allows these quality of service (QoS) qualifiers to be applied to components declaratively (without requiring programming or a change to the services implementation code). You can add service qualifiers using WebSphere Integration

Developer. Typically, you apply QoS qualifiers when you are ready to consider solution deployment.

Client programming model

The SCA client programming model is designed to locate a service, to create data objects, to invoke a service, and to handle exceptions that are raised by the invoked component.

The SCA client programming model provides two primary functions for clients. The programming model exposes an interface that allows clients to locate services within the current module, and once a service is located the client programming model provides a way for the client to invoke operations on that service.

Clients locate services by using the `ServiceManager` class. There are a few ways to instantiate the `ServiceManager` class, depending on the wanted lookup scope for the service.

The key interface that clients should be aware of for locating services is `com.ibm.websphere.sca.ServiceManager`. This interface includes a `locateService` method that returns a reference to the service implementation for the service requested. The string parameter that is passed into the `locateService` method represents the reference name for the service that the client wants to locate. The Java documentation for the SCA programming model is included in the WebSphere Process Server information center, and is also included if you choose to install the Java documentation as part of the WebSphere Process Server installation.

Once a client has located the appropriate service, there are two types of invocation models that can be used to make a call to an operation or method offered by the service. First, there is a *dynamic invocation* style of interaction. The key interface for this style of interaction is `com.ibm.websphere.sca.Service`. The `invoke()` method on this interface takes the name of the operation that you are going to invoke, along with the parameters needed to call that operation.

```
public Interface MyService {
    public String someMethod(String input);

    Service myService = (Service) serviceManager.locateService("myService");
    DataObject input = ...
    DataObject result = (DataObject) myService.invoke("someMethod", input);
```

Clients can also use a *static (type-safe) invocation* method to call a particular operation associated with a service. This type of invocation only works for interface definitions that are specified as Java. In this situation, the client casts the return from the `locateService()` call to the appropriate interface and can proceed calling the appropriate type safe method calls on that interface.

```
public Interface MyService {
    public String someMethod(String input);

    MyService myService = (MyService) serviceManager.locateService("myService");
    String input = ...
    String result = myService.someMethod(input);
```

Interfaces

A service component has one or more interfaces with which it is associated. The interfaces associated with a service component advertise the business operations associated with this service.

All components have interfaces of the WSDL type. Only Java components support Java-type interfaces. If a component, import or export, has more than one interface, all interfaces must be the same type.

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mfc="http://www.ibm.com/xmlns/prod/websphere/scdl/mfc/7.0.0"
  xmlns:ns1="http://HelloWorldLibrary/HelloWorld" xmlns:ns2="http://HelloService/HelloService"
  xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
  xmlns:wSDL="http://www.ibm.com/xmlns/prod/websphere/scdl/wSDL/7.0.0"
  displayName="HelloWorldMediation" name="HelloWorldMediation">
  <Interfaces>
    <interface xsi:type="wSDL:WSDLPortType" portType="ns1:HelloWorld">
      <scdl:interfaceQualifier xsi:type="scdl:JoinTransaction" value="true"/>
    </interface>
  </Interfaces>
  <references>
    <reference name="HelloServicePartner">
      <interface xsi:type="wSDL:WSDLPortType" portType="ns2:HelloService"/>
      <scdl:referenceQualifier xsi:type="scdl:SuspendTransaction" value="false"/>
      <scdl:referenceQualifier xsi:type="scdl:Reliability"/>
      <scdl:referenceQualifier xsi:type="scdl:DeliverAsyncAt" value="commit"/>
      <wire target="HelloServiceImport"/>
    </Reference>
  </references>
  <implementation xsi:type="mfc:MediationFlowImplementation" mfcFile="HelloWorldMediation.mfc">
    <scdl:implementationQualifier xsi:type="scdl:Transaction" value="global"/>
  </implementation>
</scdl:component>
```

Figure 50. Example of an interface definition within a SC DL component definition

These interfaces can be specified as either Java interfaces or WSDL port type interfaces. However, you can not mix Java and WSDL port type interfaces on the same service component definition. The arguments and return types for these interfaces are specified as simple Java types, Java classes, Service Data Objects, or XML Schema (for WSDL port type interfaces).

A component can be called synchronously or asynchronously; this is independent of whether the implementation is synchronous or asynchronous. The component interfaces are defined in the synchronous form and asynchronous support is also generated for them. You can specify a preferred interaction style as synchronous or asynchronous. The asynchronous type advertises to users of the interface that it contains at least one operation that can take a significant amount of time to complete. As a consequence, the calling service must avoid keeping a transaction open while waiting for the operation to complete and send its response. The interaction style applies to all the operations in the interface.

Components in different modules can be wired together by publishing the services as exports that have their interfaces and dragging the exports into the required assembly diagram to create imports. When wiring components, you can also specify quality of service qualifiers on the implementations, partner references, and interfaces of the component.

Imports have interfaces that are the same as or a subset of the interfaces of the remote service that they are associated with so that those remote services can be called. Imports are used in an application in exactly the same way as local components. This provides a uniform assembly model for all functions, regardless of their locations or implementations. The import binding does not have to be defined at development time; it can be done at deployment time.

Exports have interfaces that are the same as or a subset of the interfaces of the component that they are associated with so that the published service can be called. An export dragged from another module into an assembly diagram automatically creates an import.

Developing service modules

A service component must be contained in a service module. Developing service modules to contain service components is key to providing services to other modules.

Before you begin

This task assumes that an analysis of requirements shows that implementing a service component for use by other modules is beneficial.

About this task

After analyzing your requirements, you might decide that providing and using service components is an efficient way to process information. If you determine that reusable service components would benefit your environment, create a service module to contain the service components.

Procedure

1. Identify service components other modules can use.
After you have identified the service components, continue with “Developing service components”.
2. Identify service components in an application that could use service components in other service modules.
After you have identified the service components and their target components, continue with “Invoking components” or “Dynamically invoking components”
3. Connect the client components with the target components through wires.

Overview of developing modules:

A module is a basic deployment unit for a WebSphere Process Server application. A module can contain components, libraries, and staging modules used by the application.

Developing modules involves ensuring that the components, staging modules, and libraries (collections of artifacts referenced by the module) required by the application are available on the production server.

WebSphere Integration Developer is the main tool for developing modules for deployment to WebSphere Process Server. Although you can develop modules in other environments, it is best to use WebSphere Integration Developer.

WebSphere Process Server supports modules for business services and mediation modules. Both modules and mediation modules are types of Service Component Architecture (SCA) module. A mediation module allows communication between applications by transforming the service invocation to a format understood by the target, passing the request to the target and returning the result to the originator. A module for a business service implements the logic of a business process. However, a module can also include the same mediation logic that can be packaged in a mediation module.

The following sections address how to implement and update modules for WebSphere Process Server.

Components

SCA modules contain components, which are the basic building blocks to encapsulate reusable business logic. Components provide and consume services and are associated with interfaces, references, and implementations. The interface defines a contract between a service component and a calling component.

With WebSphere Process Server, a module can either export a service component for use by other modules or import a service component for use. To invoke a service component, a calling module references the interface to the service component. The references to the interfaces are resolved by configuring the references from the calling module to their respective interfaces.

To develop a module you must do the following activities:

1. Define or identify interfaces for the components in the module.
2. Define or manipulate business objects used by components.
3. Define or modify components through their interfaces.

Note: A component is defined through its interface.

4. Optional: Export or import service components.
5. Create an enterprise archive (EAR) file to deploy to the run time. You create the file using either the export EAR feature in WebSphere Integration Developer or the serviceDeploy command.

Development types

WebSphere Process Server provides a component programming model to facilitate a service-oriented programming paradigm. To use this model, a provider exports interfaces of a service component so that a consumer can import those interfaces and use the service component as if it were local. A developer uses either strongly-typed interfaces or dynamically-typed interfaces to implement or invoke the service component. The interfaces and their methods are described in the References section within this information center.

After installing service modules to your servers, you can use the administrative console to change the target component for a reference from an application. The new target must accept the same business object type and perform the same operation that the reference from the application is requesting.

Service component development considerations

When developing a service component, ask yourself the following questions:

- Will this service component be exported and used by another module?
If so, make sure the interface you define for the component can be used by another module.
- Will the service component take a relatively long time to run?
If so, consider implementing an asynchronous interface to the service component.
- Is it beneficial to decentralize the service component?

If so, consider having a copy of the service component in a service module that is deployed on a cluster of servers to benefit from parallel processing.

- Does your application require a mixture of 1-phase and 2-phase commit resources?

If so, make sure you enable last participant support for the application.

Note: If you create your application using WebSphere Integration Developer or create the installable EAR file using the serviceDeploy command, these tools automatically enable the support for the application. See the topic, “Using one-phase and two-phase commit resources in the same transaction” in the WebSphere Application Server for z/OS® information center.

Developing service components:

Develop service components to provide reusable logic to multiple applications within your server.

Before you begin

This task assumes that you have already developed and identified processing that is useful for multiple modules.

About this task

Multiple modules can use a service component. Exporting a service component makes it available to other modules that refer to the service component through an interface. This task describes how to build the service component so that other modules can use it.

Note: A single service component can contain multiple interfaces.

Procedure

1. Define the data object to move data between the caller and the service component.

The data object and its type is part of the interface between the callers and the service component.

2. Define an interface that the callers will use to reference the service component.

This interface definition names the service component and lists any methods available within the service component.

3. Generate the class that implements calling the service.

4. Develop the implementation of the generated class.

5. Save the component interfaces and implementations in files with a .java extension.

6. Package the service module and necessary resources in a JAR file.

See “Deploying a module to a production server” in this information center for a description of steps 6 through 8.

7. Run the serviceDeploy command to create an installable EAR file containing the application.

8. Install the application on the server node.

9. Optional: Configure the wires between the callers and the corresponding service component, if calling a service component in another service module.

The “Administering” section of this information center describes configuring the wires.

Examples of developing components

This example shows a synchronous service component that implements a single method, `CustomerInfo`. The first section defines the interface to the service component that implements a method called `getCustomerInfo`.

```
public interface CustomerInfo {
    public Customer getCustomerInfo(String customerID);
}
```

The following block of code implements the service component.

```
public class CustomerInfoImpl implements CustomerInfo {
    public Customer getCustomerInfo(String customerID) {
        Customer cust = new Customer();

        cust.setCustNo(customerID);
        cust.setFirstName("Victor");
        cust.setLastName("Hugo");
        cust.setSymbol("IBM");
        cust.setNumShares(100);
        cust.setPostalCode(10589);
        cust.setErrorMsg("");

        return cust;
    }
}

x
```

The following section is the implementation of the class associated with `StockQuote`.

```
public class StockQuoteImpl implements StockQuote {

    public float getQuote(String symbol) {

        return 100.0f;
    }
}
```

What to do next

Invoke the service.

Invoking components:

Components with modules can use components on any node of a WebSphere Process Server cluster.

Before you begin

Before invoking a component, make sure that the module containing the component is installed on WebSphere Process Server.

About this task

Components can use any service component available within a WebSphere Process Server cluster by using the name of the component and passing the data type the component expects. Invoking a component in this environment involves locating and then creating the reference to the required component.

Note: A component in a module can invoke a component within the same module, known as an intra-module invocation. Implement external calls (inter-module invocations) by exporting the interface in the providing component and importing the interface in the calling component.

Important: When invoking a component that resides on a different server than the one on which the calling module is running, you must perform additional configurations to the servers. The configurations required depend on whether the component is called asynchronously or synchronously. How to configure the application servers in this case is described in related tasks.

Procedure

1. Determine the components required by the calling module.
Note the name of the interface within a component and the data type that interface requires.
2. Define a data object.
Although the input or return can be a Java class, a service data object is optimal.
3. Locate the component.
 - a. Use the ServiceManager class to obtain the references available to the calling module.
 - b. Use the locateService() method to find the component.
Depending on the component, the interface can either be a Web Service Descriptor Language (WSDL) port type or a Java interface.
4. Invoke the component synchronously.
You can either invoke the component through a Java interface or use the invoke() method to dynamically invoke the component.
5. Process the return.
The component might generate an exception, so the client has to be able to process that possibility.

Example of invoking a component

The following example creates a ServiceManager class.

```
ServiceManager serviceManager = new ServiceManager();
```

The following example uses the ServiceManager class to obtain a list of components from a file that contains the component references.

```
InputStream myReferences = new FileInputStream("MyReferences.references");  
ServiceManager serviceManager = new ServiceManager(myReferences);
```

The following code locates a component that implements the StockQuote Java interface.

```
StockQuote stockQuote = (StockQuote)serviceManager.locateService("stockQuote");
```

The following code locates a component that implements either a Java or WSDL port type interface. The calling module uses the Service interface to interact with the component.

Tip: If the component implements a Java interface, the component can be invoked through either the interface or the `invoke()` method.

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
```

The following example shows `MyValue`, code that calls another component.

```
public class MyValueImpl implements MyValue {

    public float myValue throws MyValueException {

        ServiceManager serviceManager = new ServiceManager();

        // variables
        Customer customer = null;
        float quote = 0;
        float value = 0;

        // invoke
        CustomerInfo cInfo =
            (CustomerInfo)serviceManager.locateService("customerInfo");
        customer = cInfo.getCustomerInfo(customerID);

        if (customer.getErrorMsg().equals("")) {

            // invoke
            StockQuote sQuote =
                (StockQuote)serviceManager.locateService("stockQuote");
            Ticket ticket = sQuote.getQuote(customer.getSymbol());
            // ... do something else ...
            quote = sQuote.getQuoteResponse(ticket, Service.WAIT);

            // assign
            value = quote * customer.getNumShares();
        } else {

            // throw
            throw new MyValueException(customer.getErrorMsg());
        }
        // reply
        return value;
    }
}
```

What to do next

Configure the wires between the calling module references and the component interfaces.

Dynamically invoking a component:

When an module invokes a component that has a Web Service Descriptor Language (WSDL) port type interface, the module must invoke the component dynamically using the `invoke()` method.

Before you begin

This task assumes that a calling component is invoking a component dynamically.

About this task

With a WSDL port type interface, a calling component must use the `invoke()` method to invoke the component. A calling module can also invoke a component that has a Java interface this way.

Procedure

1. Determine the module that contains the component required.
2. Determine the array required by the component.
The input array can be one of three types:
 - Primitive uppercase Java types or arrays of this type
 - Ordinary Java classes or arrays of the classes
 - Service Data Objects (SDOs)
3. Define an array to contain the response from the component.
The response array can be of the same types as the input array.
4. Use the `invoke()` method to invoke the required component and pass the array object to the component.
5. Process the result.

Examples of dynamically invoking a component

In the following example, a module uses the `invoke()` method to call a component that uses primitive uppercase Java data types.

```
Service service = (Service)serviceManager.locateService("multiParamInf");

Reference reference = service.getReference();

OperationType methodMultiType =
    reference.getOperationType("methodWithMultiParameter");

Type t = methodMultiType.getInputType();

BOMFactory boFactory = (BOMFactory)serviceManager.locateService
    ("com/ibm/websphere/bo/BOMFactory");

DataObject paramObject = boFactory.createbyType(t);

paramObject.set(0,"input1")
paramObject.set(1,"input2")
paramObject.set(2,"input3")

service.invoke("methodMultiParamater",paramObject);
```

The following example uses the `invoke` method with a WSDL port type interface as the target.

```
Service serviceOne = (Service)serviceManager.locateService("multiParamInfWSDL");

DataObject dob = factory.create("http://MultiCallWSServerOne/bos", "SameBO");
dob.setString("attribute1", stringArg);

DataObject wrapBo = factory.createByElement
    ("http://MultiCallWSServerOne/wsdl/ServerOneInf", "methodOne");
wrapBo.set("input1", dob); //wrapBo encapsulates all the parameters of methodOne
wrapBo.set("input2", "XXXX");
wrapBo.set("input3", "yyyy");

DataObject resBo= (DataObject)serviceOne.invoke("methodOne", wrapBo);
```

Invocation styles

With SCA, you can invoke service components using synchronous and asynchronous programming styles. You can assemble modules into overall solutions where asynchronous channels between service components and modules can increase the overall throughput and flexibility of the system.

A component exposes business-level interfaces to its application business logic so that the service can be used or invoked. The interface of a component defines the operations that can be called and the data that is passed, such as input arguments, returned values, and exceptions. An import and export also has interfaces so that the published service can be invoked.

All components have interfaces of the WSDL type. Only Java components support Java-type interfaces. If a component, import or export, has more than one interface, all interfaces must be the same type.

A component can be called synchronously or asynchronously; independent of whether the implementation is synchronous or asynchronous. The component interfaces are defined in the synchronous form and asynchronous support is also generated for them. You can specify a preferred interaction style as synchronous or asynchronous. The asynchronous type advertises to users of the interface that it contains at least one operation that can take a significant amount of time to complete. As a consequence, the calling service must avoid keeping a transaction open while waiting for the operation to complete and send its response. The interaction style applies to all the operations in the interface.

When authoring applications in WebSphere Integration Developer, it is a best to explicitly set the invocation style that each of your components use to call each other. At a minimum, you want to know what invocation styles are used throughout your application while you do performance analysis or develop your error handling strategy. You certainly need to understand the interactions in your application when you consider/set your transaction boundaries. Users are often surprised to find that setting or determining invocation styles between components is not as easy a task as it seems. This section explains how to set or determine which invocation style is used at runtime, based on specific characteristics of your application.

The invocation styles that SCA provides are:

- Synchronous
- Asynchronous using one-way operation
- Asynchronous with callback
- Asynchronous with deferred response

The SCA API that is used in the Java implementation to make the invocation determines what the invocation style is at runtime:

- `invoke()`: Synchronous
- `invokeAsync()`: Asynchronous
- `invokeAsyncWithCallback()`: Asynchronous

In general, when considering an interaction from one component (source or client) to another (target), the service client determines what type of invocation is used. For example, if your source component is a Java™ component, the invocation style from source to target is determined by the particular SCA invocation API you use in the implementation, such as `invoke()`, `invokeAsync()`, or

invokeAsyncWithCallback(). Each of the other components provided in WebSphere Process Server has a set of rules that it uses to determine whether an invocation is synchronous or asynchronous.

Some components/imports are considered asynchronous:

- Long running BPEL
- Human tasks
- MQ/MQ imports
- JMS/Generic imports
- JMS/JMS imports

All invocations to components or imports of these types must be asynchronous invocations. If the calling component (or source) initiates the interaction synchronously, SCA switches the interaction to be asynchronous.

Synchronous invocation:

Service component interfaces (SCA) are always defined in the synchronous form. For each synchronous interface, one or more asynchronous interfaces can be generated.

When a service component is invoked synchronously, both the client (consumer) and the service provider execute in the same thread. The calling component within WebSphere Process Server is blocked until a response is received from the provider.

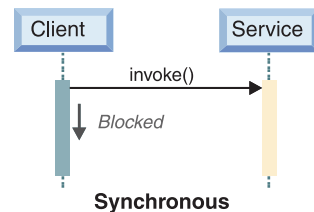


Figure 51. Synchronous invocation

Asynchronous invocation:

WebSphere Process Server delivers a powerful programming model for developing asynchronous applications. With asynchronous invocation in SCA, there are three types of asynchronous interaction styles available: one way, deferred response, and request with callback. With all three types of asynchronous invocation, the client receives control back immediately from the SCA runtime upon an `invokeAsync()` call.

In addition to its published APIs and tools to develop asynchronous programs using Java, WebSphere Process Server also comes with a number of built-in asynchronous messaging bindings and built-in asynchronous components.

There are three different ways that the client can capture the response at a later time. First, the client can choose to discard the response entirely or if it is a call to a void method. In this case, the asynchronous invocation is said to be *one way*. Another option is for the client to call `invokeAsync()` and then continue processing until some later time when the client makes a request to capture the response. This scenario is termed *deferred response*. Finally, the client also has the option of doing

an asynchronous *request with callback*. To do this, the client must first implement the ServiceCallback interface. Then, after calling `invokeAsync()`, the SCA runtime provides a callback to the ServiceCallback handler to provide the response to the client.

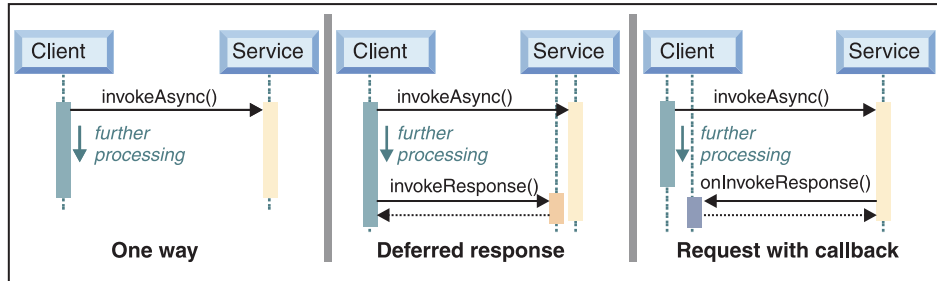


Figure 52. Asynchronous invocation models

SCA interfaces are always defined in the synchronous form. For each synchronous interface, one or more asynchronous interfaces can be generated. When a callback mechanism is chosen by the client, the client component needs to implement a class: `<interface name>.Callback.java`. The interface of this class is derived from the interface of the actual component that the client wants to use.

SCA interactions:

SCA supports synchronous and asynchronous invocation of modules. Developers have the option of selecting the appropriate interfaces and invocation methods for their SCA interactions.

The diagram summarizes the different interface types, the supported invocation methods and models, and how data is passed between client and service.

For synchronous invocation, data is passed by reference within the same SCA module, while for asynchronous calls the data is passed by value. The table also summarizes when it is possible to use either type safe or dynamic invocation based upon the interface type. The dynamic invocation methods are always available for either WSDL port type or Java interfaces. However, in order to have type safe invocation methods available to the client a Java interface type must be used for the interface definition on the appropriate client reference.

Interface Type	Invocation Model				Invocation Methods	
	Synchronous	One Way	Deferred Response	Request with Callback	Dynamic	Type Safe
WSDL Port Type	●	■	■	■	YES	NO
Java Interface	●	■	■	■	YES	YES

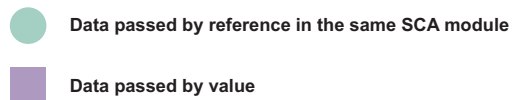


Figure 53. Summary of invocation models along with supported methods for passing data

Dynamic client invocation

There are a number of key methods and interfaces needed to support both synchronous and asynchronous interaction when using dynamic client invocation.

Table 15. Summary of key methods and interfaces for dynamic client invocation

Interface	Methods	Description
Service	Object invoke(Service String, Object)	Used to invoke synchronous service requests
	Ticket invokeAsync(String, Object)	Used to invoke one-way or deferred response asynchronous service requests
	Ticket invokeAsyncWithCallback(String, Object)	Used to invoke request with callback asynchronous service requests. The client must implement the ServiceCallback interface
	Object invokeResponse(Ticket, long)	Used to get response in the case of deferred response invocation
ServiceCallback	void onInvokeResponse(Ticket, Object, Exception)	Callback interface must be implemented by the client using a request with callback asynchronous service invocation

Exception handling for synchronous invocation:

When a service component is invoked synchronously, both the client and the service provider execute in the same thread. The target can return a response message, an exception, or nothing (in a one-way operation) to the client. If the result is an exception, it can be either a business exception or a system exception. The client in this case can be either application code or some form of system code.



Here is a sample client that invokes a Java component declared with a JType interface. The interface has one method declared as follows:

```
public interface StockQuote {
    float getQuote(String symbol) throws InvalidSymbolException;
}
```

The client code looks like this:

```
try {
    float quote = StockQuoteService.getQuote(String symbol);
} catch (InvalidSymbolException s) {
    System.out.println(This is business exception declared in the Java interface.);
} catch (ServiceRuntimeException e) {
    System.out.println(Unchecked system exception detected);
}
```

In the scenario, the first exception `InvalidSymbolException` indicates that the request has reached the service provider, which does not recognize the client input. The service provider then throws a business exception stating that the symbol supplied is invalid. This business exception is the only one declared by the method signature.

JType exceptions like `InvalidSymbolException` are only caught with clients using a JType reference.

In addition to the business exception declared, the client can receive system exceptions. For example, if the stock exchange system runs into a problem, the service might fail to obtain the quote with some unchecked exceptions. When such an exception is thrown by the service, a `ServiceRuntimeException` is returned to the client, and the client might then want to determine the underlying cause. The following code snippet shows how it can obtain this information:

```
try {
    float quote = StockQuoteService.getQuote(String symbol);
} catch (ServiceRuntimeException e) {
    Throwable t = e.getCause();
    if (t instanceof RemoteException) {
        system.out.println(System ran into RemoteException. Details as follows: + e.toString());
    }
}
```

Exception handling for asynchronous invocation:

When a service component is invoked asynchronously, the client and service provider are executed in different threads, and error conditions can occur in either thread. The client may experience a system exception during the invocation, or the service provider may experience a business or system exception while servicing the request.

In WebSphere Process Server, there is always an asynchronous counterpart of the synchronous interface.

Here is an example of an asynchronous interface:

```
public interface StockQuoteAsync {
    public Ticket getQuoteAsync(String arg0);
    public Ticket getQuoteAsyncWithCallback(String arg0);
    public float getQuoteResponse(Ticket ticket, long timeout) throws InvalidSymbolException;
}
```

Here is the client code for a call using the invocation pattern for deferred response:

```
Ticket ticket = stockQuote.getQuoteAsync(symbol);
try {
    quote = stockQuote.getQuoteResponse(ticket, Service.WAIT);
} catch (InvalidSymbolException s) {
```



```
System.out.println(This is business exception declared in the interface.);
} catch (ServiceRuntimeException e) {
System.out.println(Unchecked system exception detected);
}
```

Like a synchronous invocation, `InvalidSymbolException` indicates that the request has reached the service provider, which has thrown a business exception stating that the symbol is invalid. This business exception is the only one declared by the interface. JType exceptions like `InvalidSymbolException` are caught only with clients using a JType reference.

In addition to the business exception declared, the client can receive system exceptions such as connection error that happens while sending the message. The client cannot receive system exceptions that happen in the service thread (the thread on the service side of the asynchronous invocation). According to the SCA asynchronous programming model, runtime exceptions that occur at the target component are not returned to the source component.

Exception case on asynchronous exception handling

There is one exception to the SCA asynchronous programming model rule that runtime exceptions that occur at the target component are not returned to the source component. If the source component is a Business Process component or Staff Process, system exceptions that occur in the target service component are returned to the caller. This capability lets business process designers model and catch system exceptions, and execute error logic if a BPEL client returns a system exception.

Considerations when invoking services on different servers:

One of the benefits of service-oriented architecture is the ability for consumers to use services that exist in other service modules. To balance the workload equitably, you should install applications on different servers in a cell and those applications should reside on different physical servers.

One of the advantages of WebSphere Process Server is the ability to distribute the application workload across multiple servers in a cell. This distribution allows for better workload balancing among the various servers in the cell and maximizes the maintainability of the computing resources because there is only one copy of an application or service in the server. Thus, an application on server A could require a service installed in server B in the cell. To use services in this manner, you must configure communications between the servers. The type of configuration you perform depends on whether the calling service component invokes the service asynchronously or synchronously.

Related topics describe how to configure the systems for both asynchronous and synchronous invocations.

Configuring servers to invoke services asynchronously:

To enable service components on different servers to communicate, you have to configure the servers similarly. This topic describes the configuration you perform to enable the communication for applications that asynchronously invoke services on a different server.

Before you begin

The task assumes that you have already installed WebSphere Process Server on the systems for which you are configuring the communications but have not yet installed the applications involved. You are using an administrative console that can examine and change the configuration for both servers involved.

About this task

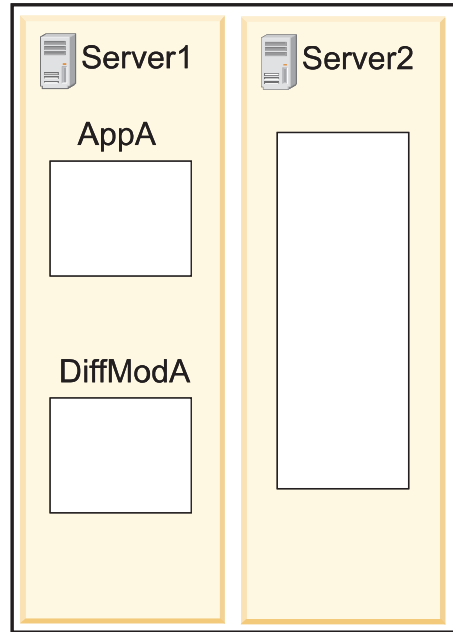
Before installing an application that requires the services of a service component installed on another system, you must configure the systems so they can communicate the requests. For service modules that use asynchronous invocations, the process involves foreign buses and Service Integration Bus (SIB) mediations.

Note: For the purposes of this task, the invoking service module resides on system A and the target resides on system B.

For the purposes of this task, Figure 54 on page 133 contains the information to use in the configuration.

System A

Host IP: 9.26.237.118
Node: WPSNodeServer: Server 1
Cell: WBINode01 Cell



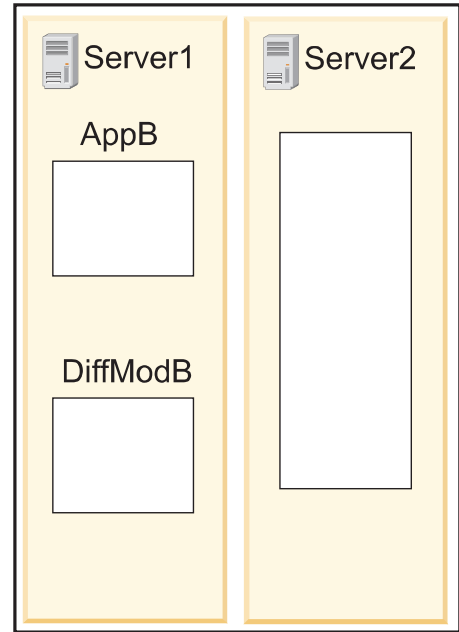
Bus Name: SCA.SYSTEM.WBINode01Cell.Bus

Messaging Engine: WPSNode.server1-SCA.SYSTEM.WBINode01Cell.Bus

Failed Event Queue: WBI.FailedEvent.WBINode01.server1

System B

Host IP: 9.26.237.144
Node: ABCNode1: Server 1
Cell: ABCNode01 Cell



Bus Name: SCA.SYSTEM.ABCNode01Cell.Bus

Messaging Engine: ABCNode01.server1-SCA.SYSTEM.ABCNode01Cell.Bus

Failed Event Queue: WBI.FailedEvent.ABCNode01.server1

Figure 54. Invoking a service on a different system

Note: For simplicity, only the servers involved in this communication in each cell is shown and each server resides on a different physical machine.

Procedure

1. Collect information about each server involved in the communication. You need the following information for both the originator and target servers:
 - Host IP address
 - Cell
 - Node
 - Server
 - Bus name
 - Messaging engine
 - Failed Event Queue name
2. Install the applications.
3. Create a foreign bus on each server pointing to the other server and set the routing definition type to Direct, service integration bus link.
See the Connecting service integration buses to use point-to-point messaging topic in the WebSphere Application Server Network Deployment, version 7 information center for more information.

From the example, the configuration of the foreign bus and SIB mediation link on System A would be:

```
Name of Service integration bus to connect to (the foreign bus):
SCA.SYSTEM.ABCNode01Cell.Bus
Gateway messaging engine in the foreign bus:
ABCNode01.server1-SCA.SYSTEM.ABCNode01Cell.Bus
Service integration bus link name: TestCrossCell
Bootstrap service integration bus provider endpoints:
9.26.237.144:7277:BootstrapBasicMessaging
```

The configuration of the foreign bus and SIB mediation link on System B would be:

```
Name of Service integration bus to connect to (the foreign bus):
SCA.SYSTEM.WBINode01Cell.Bus
Gateway messaging engine in the foreign bus:
WPSNode01.server1-SCA.SYSTEM.WBINode01Cell.Bus
Service integration bus link name: TestCrossCell
Bootstrap service integration bus provider endpoints:
9.26.237.118:7276:BootstrapBasicMessaging
```

Attention: The port number in the bootstrap is the SIB endpoint address port. If you enabled security, you must use the secure SIB endpoint address port.

4. Synchronize the SIB mediation links by restarting the servers.

You should see messages like:

```
[9/25/09 8:04:23:406 CDT] 00000034 SibMessage I [:] CWSIT0032I:
The service integration bus link TestCrossCell from messaging engine
WPSNode01.server1-SCA.SYSTEM.WPSNode01Cell.Bus in bus SCA.SYSTEM.WPSNode01Cell.Bus
to messaging engine ABCNode01.server1-SCA.SYSTEM.ABCNode01Cell.Bus in bus SCA.SYSTEM.
ABCNode01Cell.Bus started.
```

5. Display the destinations for each service module.
6. Modify the default forwarding path of outgoing destinations of the invoking service module that must be wired to targets on the other system.

Select **Applications > SCA modules**, choose module and then click on **SCA system bus destinations**.

The destination to wire has `importlink` in the destination name, for example on System A the destination would be `sca/AppA/importlink/test/sca/cros/simple/custinfo/CustomerInfo`. Modify the path by prefixing the foreign bus name to the destination name. From the example, the foreign bus name for the second system is `SCA.SYSTEM.ABCNode01Cell.Bus`. The result is

```
SCA.SYSTEM.ABCNode01Cell.Bus:sca/AppA/importlink/
test/sca/cros/simple/custinfo/CustomerInfo
```

7. Optional: Add sender roles to the foreign buses, if you enabled security on the systems. Make sure to define the user each application uses on both systems from the operating system command prompt. The command to add the role is:

```
wsadmin $AdminTask addUserToForeignBusRole -bus busName
      -foreignBus foreignBusName -role roleName -user userName
```

Where:

busName

Is the name of the bus on the system you enter the command.

foreignBusName

Is the foreign bus to which you are adding the user.

userName

Is the userid to add to the foreign bus.

What to do next

Start the applications.

Configuring servers to invoke services synchronously:

When a service component invokes another service component synchronously, you must configure the invoking service component to point to the system running the target so the target service can communicate results to the invoking service component.

Before you begin

The task assumes that you have already installed WebSphere Process Server on the systems for which you are configuring the communications but have not yet installed the applications involved. You are using an administrative console that can examine and change the configuration for both servers involved.

About this task

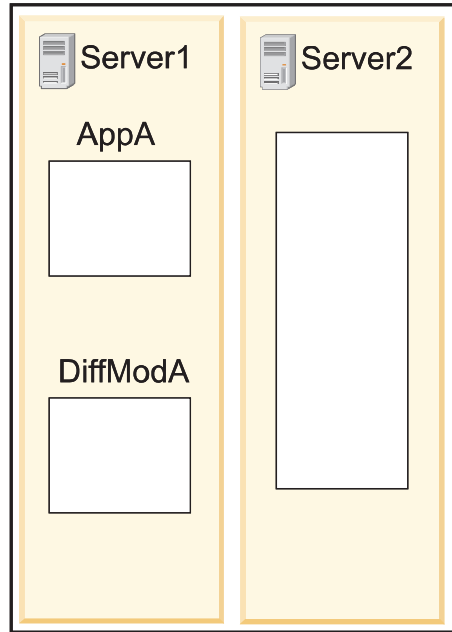
A service component invoking another service synchronously can communicate with the target only by configuring the export Java Naming and Directory Interface (JNDI) name on the target system to a JNDI name on the invoking system.

Note: For the purposes of this task, the invoking service module resides on system A and the target resides on system B.

For the purposes of this task, Figure 55 on page 136 contains the information to use in the configuration.

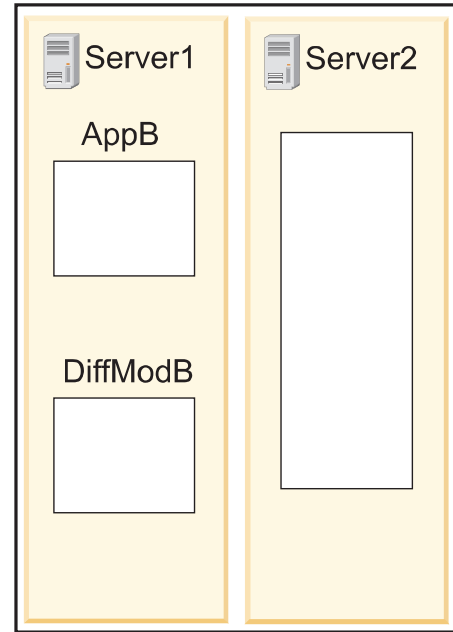
System A

Host IP: 9.26.237.118
Node: WPSNodeServer: Server 1
Cell: WBINode01 Cell



System B

Host IP: 9.26.237.144
Node: ABCNode1: Server 1
Cell: ABCNode01 Cell



Bus Name: SCA.SYSTEM.WBINode01Cell.Bus

Messaging Engine: WPSNode.server1-SCA.
SYSTEM.WBINode01Cell.Bus

Failed Event Queue: WBI.FailedEvent.
WBINode01.server1

Bus Name: SCA.SYSTEM.ABCNode01Cell.Bus

Messaging Engine: ABCNode01.server1-SCA.
SYSTEM.ABCNode01Cell.Bus

Failed Event Queue: WBI.FailedEvent.
ABCNode01.server1

Figure 55. Invoking a service on a different system

Note: For simplicity, only the servers involved in this communication in each cell is shown and each server resides on a different physical machine.

Procedure

1. Install the applications on each server.
2. Create a new namespace binding on the invoking system (System A, in the example) pointing to the export on the target system.

On the **Name Space Bindings** panel, select a scope of Cell and click **Apply**. With the changed scope, click **New** in the display to create the new binding.

In the wizard, specify the following (the values are appropriate for the example configuration):

- a. Binding type is CORBA
- b. The basic properties are:
 - Binding identifier is a unique string, in our example:
sca_import_test_sca_cross_simple_custinfo_CustomerInfo
 - Name in Name space is the JNDI name of the enterprise Java bean (EJB) you are invoking on the target system, for example:
sca/AppB/export/test/sca/cros/simple/custinfo/CustomerInfo

This names the export interface on the target system.

- Corbaname URL is the IP address and port number of the naming service on the target system:

```
corbaname:iiop:host:port/NameServiceServerRoot#<package.qualified.interface>
```

Note: For WebSphere Process Server, the port is the BOOTSTRAP_ADDRESS.

For example:

```
corbaname:iiop:9.26.237.144:2809/NameServiceServerRoot#sca/  
AppB/export/test/sca/cros/simple/custinfo/CustomInfo
```

When finished, click **Next** and verify the values on the **Summary** page.

After verifying, click **Finish**.

Your system displays your new binding.

3. Save your changes by clicking **Save**.
4. If your cross cell configuration consists of servers on the same host with the same name and you encounter JNDI lookup failures with a `NameNotFoundException`, then you need to set a system property.

Follow the directions in the Application access problems under the subheading *Two servers with the same name running on the same host are being used to interoperate*.

What to do next

Start the applications. The service component on System A can now synchronously invoke the service on System B.

Qualifiers

Qualifiers are an important part of SCA because they allow developers to place quality of service requirements on the WebSphere Process Server runtime.

There are several different categories of qualifiers available in SCA. These categories of qualifiers are transaction, activity session, security, and asynchronous reliability.

Each SCA qualifier has a particular scope within the SCDL definition for a component where the qualifier can be specified. For example, some qualifiers can be specified at the references level, while others may only be valid at the interfaces or implementation level. The table lists the various qualifiers that are available and the valid scope for each. The qualifiers are sorted by the type of quality of service they provide, such as transaction or security.

Table 16. Summary of qualifiers

Type	Qualifier	Scope	Description
Transaction	transaction	Implementation	<p>global – A global transaction must be present to run the component</p> <p>local – A global transaction must not exist to run the component</p> <p>any – Component is unaffected by transactional state</p> <p>local application – Component processing occurs within a WebSphere local transaction containment that is managed by the application</p>
	joinTransaction	Interface	<p>true – Hosting container joins client transaction</p> <p>false (default) – Hosting container does not join client transaction</p>
	suspendTransaction	Reference	<p>true – Synchronous invocations of target component do not run within client global transaction.</p> <p>false – Synchronous invocations of target component run within client global transaction</p>
	deliverAsyncAt	Reference	<p>call – Asynchronous invocations of a target service occur immediately</p> <p>commit – Asynchronous invocations of a target service occur as part of a global transaction</p>

Table 16. Summary of qualifiers (continued)

Type	Qualifier	Scope	Description
Asynchronous Response	reliability	Reference	Specifies the quality of service level for asynchronous message delivery. Reliability can be one of the following values: bestEffort or assured
	requestExpiration	Reference	Specifies the length of time (milliseconds) after which an asynchronous request is to be discarded if not delivered
	responseExpiration	Reference	Specifies the duration (milliseconds) between the time a request is sent and the time a response or callback is received
Security	securityIdentity	Implementation	The permission specifies a logical name for the identity under which the implementation executes at runtime.
	securityPermission	Interfaces, Interface, Method	The caller identity must have the role specified from this qualifier in order to have permission to run the interface or method

Table 16. Summary of qualifiers (continued)

Type	Qualifier	Scope	Description
Activity Session	activitySession	Implementation	<p>true – There must be an ActivitySession established in order to run this component</p> <p>false – The component runs under no ActivitySession</p> <p>any – The component does not depend on the presence or absence of an ActivitySession</p>
	joinActivitySession	Interface	<p>true – Hosting container joins client ActivitySession</p> <p>false – Hosting container does not join client ActivitySession</p>
	suspendActivitySession	Reference	<p>true – Methods on target component does NOT run as part of any client ActivitySession</p> <p>false – Methods on target component run as part of any client ActivitySession</p>

Transaction qualifiers

Transaction qualifiers allow developers to request a particular transactional environment for the service components in an SCA module. The following is a summary of these qualifiers:

transaction

The transaction qualifier is set at the implementation scope of a service component. This qualifier can be set to either 'global', 'local' (default), or 'any'. When set to global, the component runs in the context of a global transaction. If a global transaction is present on invocation, the component is added to this global transaction scope. If set to local, the component runs in the context of a local transaction. Finally, if the value is set to any then if a global transaction is present the component joins the current global transaction scope. However, if a global transaction is not present, the component runs in the context of a local transaction.

joinTransaction

The joinTransaction qualifier is set at the interface scope of a service component. This qualifier can be set to either true or false (false being the default). If set to true, it instructs the runtime not to suspend a global transaction (if present) at the interface boundary. If set to false, it instructs the runtime to suspend a global transaction (if present) at the interface

boundary. Exposing the `joinTransaction` transactional qualifier on an interface provides metadata that can be used by assemblers and deployers to ensure that the assembled application behaves as required. It is up to the assembler and deployer in addition to dynamic runtimes to reason about whether a target component federates with a propagated transaction.

suspendTransaction

The `suspendTransaction` qualifier is set at the reference level of a service component and identifies whether a global transaction should be suspended before invoking the target service associated with the reference. This qualifier can be set to either `true` or `false` (default).

deliveryAsyncAt

The `deliveryAsyncAt` qualifier is like the `suspendTransaction` qualifier, except that it pertains to asynchronous interactions rather than synchronous types as is the case with `suspendTransaction`. The `deliveryAsyncAt` qualifier can have the value of `call` (default) or `commit`. If set to `call`, it indicates to the runtime that the message for the asynchronous interaction should be committed to the queue immediately when the call has been made. The value of `commit` indicates that the message should be committed to the queue as part of a transaction associated with the current unit of work.

Asynchronous response qualifiers

There are three qualifiers available for indicating quality of service for asynchronous response. Each of the asynchronous response qualifiers are specified at the reference scope. The following is a summary of asynchronous response qualifiers:

reliability

The `reliability` qualifier is used to specify the quality of service level for asynchronous message delivery. The `reliability` can be set to either `bestEffort` or `assured` (default).

requestExpiration

The `requestExpiration` qualifier is used to specify the length of time the runtime should hold onto an asynchronous request if it has not yet been delivered. After the time indicated for this qualifier, given in milliseconds, this request is discarded.

responseExpiration

The `responseExpiration` qualifier is used to specify the length of time that the runtime must retain an asynchronous response or must provide a callback. The value for this qualifier is given in milliseconds.

Security qualifiers

There are two qualifiers available for indicating quality of service related to security. The following is a summary of these qualifiers:

securityIdentity

The `securityIdentity` qualifier is used to specify the security identity under which the implementation for the service component should run at runtime. This qualifier must be placed at the implementation scope for the service component and the value given must match the logical name for the identity under which the component will run.

securityPermission

The securityPermission qualifier is specified at the interfaces level, including interface, or method level. The value for this qualifier indicates that a caller of this service must have the role that is specified in order to invoke the service.

For both the securityPermission and the securityIdentity, the underlying implementation for these qualifiers is based on existing Java EE concepts.

Activity session qualifiers

The set of activity session qualifiers are similar to the transaction qualifiers introduced earlier. The ActivitySession service is a WebSphere programming model extension that can provide an alternative unit of work when compared with global transactions. In fact, an activity session context can be longer lived than a global transaction and can even include global transactions. The following is a summary of the activity session qualifiers:

joinActivitySession

The joinActivitySession qualifier is set at the interface level, and indicates whether or not the component should join the activity session of a client caller. There are two values for this qualifier, true and false (default). If set to true it indicates that the runtime should not suspend an activity session if present when the component is invoked. If set to false it indicates that an activity session should be suspended before invoking the component.

activitySession

The activitySession qualifier is specified at the implementation level and is used to indicate whether or not an activity session should or should not exist in order to run the service component with which it is associated. This qualifier can be set to either 'true', 'false', or 'any' (default). If set to true, it indicates that the component will run as part of an activity session. If set to false, the component should not run as part of an activity session. This means that the joinActivitySession should also be set to false for any interfaces specified for the component. Finally, if this qualifier is set to any, the component will run as part of an activity session if it is present, otherwise it will not.

suspendActivitySession

The suspendActivitySession qualifier is set at the reference level and is used to indicate whether or not a target service associated with a reference will get called as part of the calling activity session or not. If set to true, the activity session is suspended and the methods on the target component will not run as part of the client activity session. If set to false (default) the activity session is not suspended and methods on the target component will run as part of the client ActivitySession.

SCA programming techniques

This section provides examples of Service Component Architecture programming techniques.

Runtime rules used for Java to Service Data Objects conversion

To correctly override generated code, or to determine possible runtime exceptions related to Java to Service Data Object (SDO) conversions, an understanding of the rules involved is important. The majority of the conversions are straightforward, but there are some complex cases where the runtime provides the best possibility when it converts the generated code.

Basic types and classes

The runtime performs a straightforward conversion between Service Data Objects and basic Java types and classes. Basic types and classes include:

- Char or `java.lang.Character`
- Boolean
- `Java.lang.Boolean`
- Byte or `java.lang.Byte`
- Short or `java.lang.Short`
- Int or `java.lang.Integer`
- Long or `java.lang.Long`
- Float or `java.lang.Float`
- Double or `java.lang.Double`
- `Java.lang.String`
- `Java.math.BigInteger`
- `Java.math.BigDecimal`
- `Java.util.Calendar`
- `Java.util.Date`
- `Java.xml.namespace.QName`
- `Java.net.URI`
- `Byte[]`

User-defined Java classes and arrays

When converting from a Java class or array to an SDO, the runtime creates a data object that has a URI that is generated by inverting the package name of the Java type and has a type equal to the name of the Java class. For example, the Java class `com.ibm.xsd.Customer` is converted to an SDO and URI `http://xsd.ibm.com` with type `Customer`. The runtime then inspects the contents of the Java class members and assigns the values to properties in the SDO.

When converting from an SDO to a Java type, the runtime generates the package name by inverting the URI and the name of the type equals the type of the SDO. For example, the data object with type `Customer` and URI `http://xsd.ibm.com` generates an instance of the Java package `com.ibm.xsd.Customer`. The runtime then extracts values from the properties of the SDO and assign those properties to fields in the instance of the Java class.

When the Java class is a user-defined interface, you must override the generated code and provide a concrete class that the runtime can instantiate. If the runtime cannot create the concrete class, an exception occurs.

Java.lang.Object

When a Java type is `java.lang.Object` the generated type is `xsd:anyType`. A module can invoke this interface with any SDO. The runtime attempts to instantiate a concrete class the same way it does for user-defined Java classes and arrays, if the runtime can find that class. Otherwise, the runtime passes the SDO to the Java interface.

Even if the method returns a `java.lang.Object` type, the runtime converts to an SDO only if the method returns a concrete type. The runtime uses a similar conversion to that for converting user-defined Java classes and arrays to SDOs, as described by the next paragraph.

When converting from a Java class or array to an SDO, the runtime creates a data object that has a URI that is generated by inverting the package name of the Java type and has a type equal to the name of the Java class. For example, the Java class `com.ibm.xsd.Customer` is converted to an SDO and URI `http://xsd.ibm.com` with type `Customer`. The runtime then inspects the contents of the Java class members and assigns the values to properties in the SDO.

In either case, if the runtime is unable to complete the conversion an exception occurs.

Java.util container classes

When converting to a concrete Java container class such as `Vector`, `HashMap`, `HashSet` and the like, the runtime instantiates the appropriate container class. The runtime uses a method similar to that used for user-defined Java classes and arrays to populate the container class. If the runtime cannot locate a concrete Java class, the runtime populates the container class with the SDO.

When converting concrete Java container classes to SDOs, the runtime uses the generated schemas shown in “Java to XML conversion.”

Java.util interfaces

For certain container interfaces in the `java.util` package, the runtime instantiates the following concrete classes:

Table 17. WSDL type to Java class conversion

Interface	Default concrete classes
Collection	HashSet
Map	HashMap
List	ArrayList
Set	HashSet

Overriding a Service Data Object to Java conversion

Sometimes, the conversion the system creates between a Service Data Object (SDO) and a Java type object may not meet your needs. Use this procedure to replace the default implementation with your own.

Before you begin

Make sure that you have generated the WSDL to Java type conversion using either WebSphere Integration Developer or the `genMapper` command.

About this task

You override a generated component that maps a WSDL type to a Java type by replacing the generated code with code that meets your needs. Consider using your own map if you have defined your own Java classes. Use this procedure to make the changes.

Procedure

1. Locate the generated component. The component is named `java_classMapper.component`.

2. Edit the component using a text editor.
3. Comment out the generated code and provide your own method.
Do not change the file name that contains the component implementation.

Example

This is an example of a generated component to replace:

```
private Object datatojava_get_customerAcct(DataObject myCustomerID,
    String integer)
{

    // You can override this code for custom mapping.
    // Comment out this code and write custom code.

    // You can also change the Java type that is passed to the
    // converter, which the converter tries to create.

    return SDOJavaObjectMediator.data2Java(customerID, integer) ;
}
```

What to do next

Copy the component and other files to the directory in which the containing module resides, and either wire the component in WebSphere Integration Developer or generate an enterprise archive (EAR) file using the serviceDeploy command.

Overriding the generated Service Component Architecture implementation

Sometimes, the conversion the system creates between a Java code and a Service Data Object (SDO) may not meet your needs. Use this procedure to replace the default Service Component Architecture (SCA) implementation with your own.

Before you begin

Make sure that you have generated the Java to Web Services Definition Language (WSDL) type conversion using either WebSphere Integration Developer or the genMapper command.

About this task

You override a generated component that maps a Java type to a WSDL type by replacing the generated code with code that meets your needs. Consider using your own map if you have defined your own Java classes. Use this procedure to make the changes.

Procedure

1. Locate the generated component. The component is named `java_classMapper.component`.
2. Edit the component using a text editor.
3. Comment out the generated code and provide your own method.
Do not change the file name that contains the component implementation.

Example

This is an example of a generated component to replace:

```
private DataObject javatodata_setAccount_output(Object myAccount) {  
  
    // You can override this code for custom mapping.  
    // Comment out this code and write custom code.  
  
    // You can also change the Java type that is passed to the  
    // converter, which the converter tries to create.  
  
    return SD0JavaObjectMediator.java2Data(myAccount);  
  
}
```

What to do next

Copy the component and other files to the directory in which the containing module resides, and either wire the component in WebSphere Integration Developer or generate an enterprise archive (EAR) file using the serviceDeploy command.

Protocol header propagation from non-SCA export bindings

The context service is responsible for propagating the context (including the protocol headers, such as the JMS header, and the user context, such as account ID) along a Service Component Architecture (SCA) invocation path. The context service offers a set of APIs and configurable settings.

When the context service propagation is bi-directional, the response context always overwrites the current context. When you are running an invocation from one SCA component to another, a response contains a different context. A service component has an incoming context, but when you invoke another service, the other service overwrites the original outgoing context. The response context becomes the new context.

When the context service propagation is one way, the original context remains the same.

The lifecycle of the context service is associated with an invocation. A request has associated context, and the lifecycle of that context is bound to the processing of that particular request. When that request is finished processing, then the lifecycle of that context ends.

For a short-running Business Process Execution Language (BPEL) process, the response context overwrites the request context. It takes back the response context from the first request and pushes it to the next request. For a long-running BPEL process, the response context is discarded by the BPEL framework. It stores the original context and uses that context when making other outgoing calls.




Context services have configurable rules and tables that dictate the binding behavior. For more information, see the Generated API and SPI documentation that is available in the Reference section. During development in WebSphere® Integration Developer, you can set the context service on import-export properties. For more details, see the import and export bindings information in the WebSphere Integration Developer information center.

Business objects programming

Business objects are containers for application data, such as a customer or an invoice. Data is exchanged between components by way of business objects. The underlying structure of a business object is an XML schema definition (XSD), and programmatic access to business objects is provided via business object interfaces in WebSphere. Collectively, these aspects of the business object, its structural representation, its programmatic interfaces, and its behavior and manipulation within the service component architecture (SCA), are the business object framework, which provides a powerful, consistent means for describing and delivering business data in your solution.

This guide provides information about programming business objects, including descriptions of problem areas in handling the schema constructs for some features. For information about how a business object is defined, business object development guidelines, and how to use business object programming APIs, refer to the articles in the "Related information" section.

Related information

-  [Web Services Description Language \(WSDL\) 1.1](#)
-  [Introduction to Service Data Objects](#)
-  [Examining business objects in WebSphere Process Server](#)

Programming model

The business object programming model section describes how the basic types of data are encapsulated within the IBM business object framework. To facilitate the creation and manipulation of business objects, the business object framework extends Service Data Objects specifications by providing a set of Java services.

Working with the IBM business object framework

The business object framework describes how data in the run time is modeled by applications, integrated into the run time, and represented in memory.

Table 1 summarizes how the basic types of data are implemented in the business object framework.

Table 18. Data abstractions and the corresponding implementations

Data Abstraction	Implementation	Description
Instance data	Business object	Business objects are the primary mechanism for representing business entities, or document literal message definitions, enabling everything from a simple basic object with scalar properties to a large complex hierarchy or graph of objects. A business object is a direct corollary to the SDO DataObject concept.

Table 18. Data abstractions and the corresponding implementations (continued)

Data Abstraction	Implementation	Description
Instance metadata	Business graph	Business graphs are wrappers that are added around a simple business object or a hierarchy of business objects to provide additional capabilities, such as carrying change summary and event summary information related to the business objects in the business graph. A business graph is a direct corollary of the SDO DataGraph concept, except that it provides more than just the single change summary header.
Type metadata	Enterprise metadata Business object type metadata	Business object type metadata is the metadata that can be added to business object definitions to enhance their value in the runtime. These metadata items are added to the business object's XML schema definition as well known <code>xs:annotation</code> and <code>xs:appinfo</code> elements.
Services	Business object services (APIs)	Business object services are a set of capabilities provided on top of the basic capabilities provided by Service Data Objects. Examples are services such as create, copy, equality, and serialization. These APIs are found in the <code>com.ibm.websphere.bo</code> package.

The WebSphere Process Server business object framework is an extension of the SDO standard. Therefore, business objects exchanged between WebSphere Process Server components are instances of the `commonj.sdo.DataObject` class. However, the WebSphere Process Server business object framework adds several services and functions that simplify and enrich the basic `DataObject` functionality.

To facilitate the creation and manipulation of business objects, the WebSphere business object framework extends SDO specifications by providing a set of Java services. These services are part of the package named `com.ibm.websphere.bo`.

- **BOFactory:** The key service that provides various ways to create instances of business objects.
- **BOXMLSerializer:** Provides ways to "inflate" a business object from a stream or to write the content of a business object, in XML format, to a stream.
- **BOCopy:** Provides methods that make copies of business objects ("deep" and "shallow" semantics).

- **BODataObject**: Gives you access to the data object aspects of a business object, such as the change summary, the business graph, and the event summary.
- **BOXMLDocument**: The front end to the service that lets you manipulate the business object as an XML document.
- **BOChangeSummary** and **BOEventSummary**: Simplifies access to and manipulation of the change summary and event summary portion of a business object.
- **BOEquality**: A service that enables you to determine whether two business objects contain the same information. It supports both deep and shallow equality.
- **BOType** and **BOTypeMetaData**: These services materialize instances of the `commonj.sdo.Type` and let you manipulate the associated metadata. Instances of `Type` can then be used to create business objects "by type".
- **BOInstanceValidator**: Validates the data in a business object to see if it conforms to the XSD.

Modeling business objects

Business object data that flows through the WebSphere Process Server runtime is modeled using XML schemas. XML schemas are an alternative to document type definitions (DTDs) and can be used to extend functionality in the areas of data typing, inheritance, and presentation. XML schema provides a form for modeling the data types that is industry standard, widely adopted, and is platform and language neutral.

Target namespace definition:

Most business and communications problems that XML can solve require a combination of several XML vocabularies. XML has a mechanism for qualifying names to be allocated into different namespaces, such as namespaces that apply to different industries. In XML, a uniform resource identifier (URI) provides a unique name to associate with the element, attribute, and type definitions in an XML schema.

There are two requirements for the business object target namespace:

- The business process management run time and tools prefer target namespaces to look like `http://www.foo.com/xyz` versus `urn:foo:com:xyz`.
- The business object framework requires a target namespace for business objects.

Business object definition:

WebSphere Process Server provides a flexible mechanism for defining or importing business objects.

There are essentially three different forms of XML schema that WebSphere Process Server recognizes as a business object definition:

- A top-level complex type definition
- A top-level anonymous complex type definition
- A top-level element that references a named complex type

Top-level complex type definition

This is an example of a top-level complex type definition:

```

<complexType name="ProductType">
  <sequence>
    <element name="name" type="string"/>
    <element name="color" type="string" maxOccurs="unbounded"/>
  </sequence>
</complexType>

```

Importing or defining business objects that are all defined using complex type definitions only is the most flexible and manageable scheme. The upside of this model is a type library that enables reuse. Reuse can be by three different methods.

- First, new types can be created using the complex type derivation model (of extension or restriction).
- Second, new aggregate types can be created using the existing complex types and available simple types as primitives.
- Third, new complex document definitions can be created, like the aggregate complex types.

The other implication of business objects defined as complex types is that when the complex type is used by the JService component kinds to transfer data within the runtime, in order to maintain WS-I compliance, an element needs to be created that references the named complex type.

Top-level anonymous complex type definition

This is an example of a top-level anonymous complex type definition:

```

<element name="Product">
  <complexType>
    <sequence>
      <element name="name" type="string"/>
      <element name="color" type="string" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>

```

If the imported business objects are all anonymous element definitions, they are ready made to be included in JService invocations. However, they are not inherently reusable.

Top-level element referencing a named type

This is an example of a top-level element referencing a named type:

```

<element name="product" type="prod:ProdType"/>

```

Business objects that reference named complex types might be frequent in an environment that has already defined WSDL operations that require element definitions. In this scenario, it is important to consider the possible disposition of the complex type and element definitions:

- Elements can cohabitate with their complex type definitions in the same XML schema file.
- Elements can cohabitate with their complex type definitions embedded in a WSDL file.
- Elements can be defined in XML schema A.xsd, while their complex type definition is defined in XML schema file B.xsd.
- Elements can be embedded in a WSDL file, referencing a complex type definition defined in an XML schema file.

Example

The example demonstrates all the mechanisms for defining a business object combined together.

```
<schema
  targetNamespace="http://www.app.com/Address"
  xmlns:addr="http://www.app.com/Address"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema">

  <complexType name="Address">
    <sequence>
      <element name="street1" type="string"/>
      <element name="street2" type="string"/>
      <element name="city" type="string"/>
      <element name="state" type="string"/>
      <element name="zip" type="string"/>
    </sequence>
  </complexType>

  <element name="homeAddress" type="addr:Address"/>
  <element name="workAddress" type="addr:Address"/>
  <element name="otherAddress" type="addr:Address"/>

  <element name="individualContact">
    <complexType>
      <sequence>
        <element name="firstName" type="string"/>
        <element name="lastName" type="string"/>
        <element ref="addr:HomeAddress"/>
        <element ref="addr:WorkAddress"/>
        <element ref="addr:OtherAddress"/>
      </sequence>
    </complexType>
  </element>

  <element name="businessContact">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element ref="addr:WorkAddress"/>
      </sequence>
    </complexType>
  </element>

  <element name="chairmanOfTheBoard">
    <complexType>
      <sequence>
        <element name="startDate" type="date"/>
        <element ref="addr:IndividualContact"/>
        <element ref="addr:BusinessContact"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

The following guidelines are the preferred way to define business objects:

- Elements are defined using named types; anonymous types are discouraged.
- Elements and complex type definitions do not cohabitate the same XML schema or WSDL file. This practice discourages type reuse.
- Complex types are defined in XML schema files, not WSDL definitions, creating a type library like concept. Again, this type of definition enables and encourages complex type reuse.

- Element definitions are built as necessary to reference a single complex type definition. For example, the definition of an element inside the WSDL is a pattern that is encouraged.
- Element definitions typically use the same target namespace as their complex type definition.

Business object property definition:

XML schema provides complex types, simple types, and attributes, which are used to build business objects.

Complex type definitions, anonymous complex type definitions, and elements referencing complex type definitions are used to define the outer business objects. The term property is used to define the data inside a business object. The term is derived from the Service Data Object term property, and is defined by the `commonj.sdo.Property` interface. It is synonymous with the concept of an attribute.

A property can either be simple or complex. A simple property can be defined either as an XML schema attribute, or as an XML schema element with a type that is an XML schema simple type. A complex property can either reference another business object, or it can define a complex structure within the current business object.

The full XML schema type system is supported.

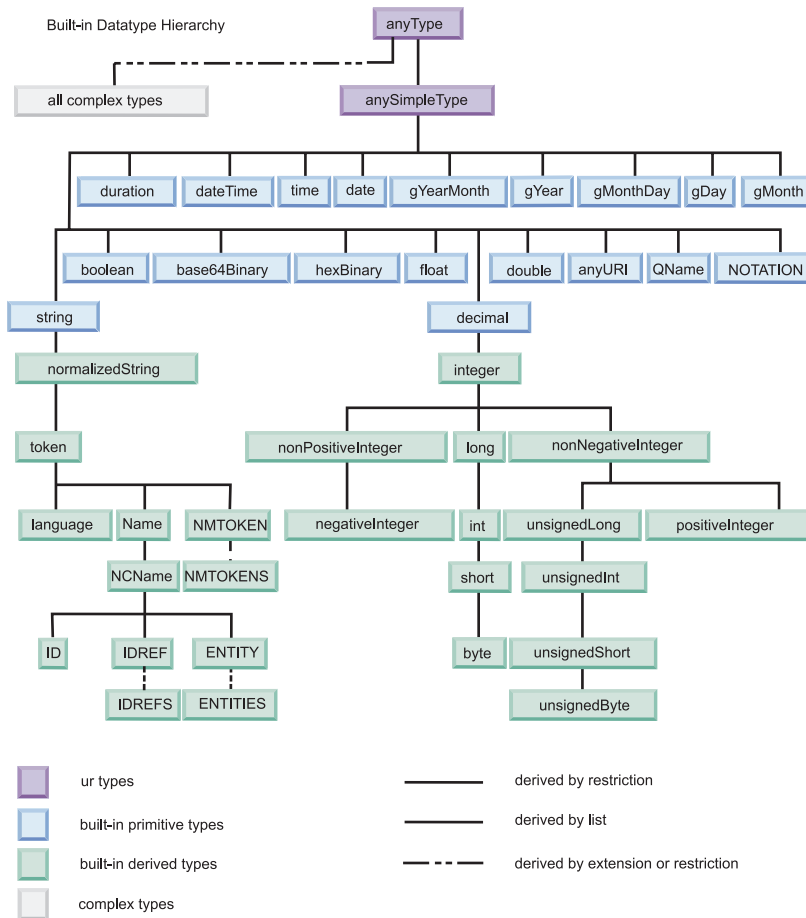


Figure 56. XML schema simple types

Supported XSD and WSDL artifacts:

When a WSDL or a schema is imported into a project in WebSphere Integration Developer, the business objects rendered from the WSDL or schema can then be used to develop a module. It is important to note however, that only certain artifacts from a schema are rendered as *business objects* (for example, root/top level elements and named complex types). Certain artifacts, such as nested anonymous complex types, are *not* rendered as business objects. These restrictions are a result of which artifacts are accessible in the XML schema. For example, if you import a schema which resulted in only one business object, it is most likely that the rest of the elements were anonymous complex types. The following information details which XSD and WSDL artifacts result in business objects.

Business objects from imported XSD definitions

When an XML schema is imported into a project, only certain artifacts are rendered as business objects. The following lists show which artifacts are supported at authoring time and at runtime:

XSD artifacts resulting in business objects at authoring time:

- Complex types defined at the root level
- Elements defined at the root level with anonymous complex types

These artifacts result in user-defined simple types at authoring time which can be referenced by business objects:

- Simple types defined at the root level
- Elements defined at the root level with anonymous simple types

Business objects from imported WSDL files

When a WSDL definition that includes an inline XSD schema is imported into a project, only certain artifacts are rendered as business objects. The following lists show which artifacts are supported at authoring time and at runtime:

Inline XSD artifacts resulting in business objects at authoring time:

- Complex types defined at the root level
- Elements defined at the root level with anonymous complex types AND the name of the element does not contain the names of any operations/messages (as these elements could be doc-lit-wrapped elements which WebSphere Integration Developer unwraps automatically)

These artifacts result in user-defined simple types at authoring time which can be referenced by business objects:

- Simple types defined at the root level
- Elements defined at the root level with anonymous simple types

Runtime business objects from XSD artifacts

These artifacts result in business objects at runtime:

- Complex types defined at the root level
- Elements defined at the root level with anonymous complex types
- Elements defined at the root level which reference a complex type

Runtime business objects from WSDL files

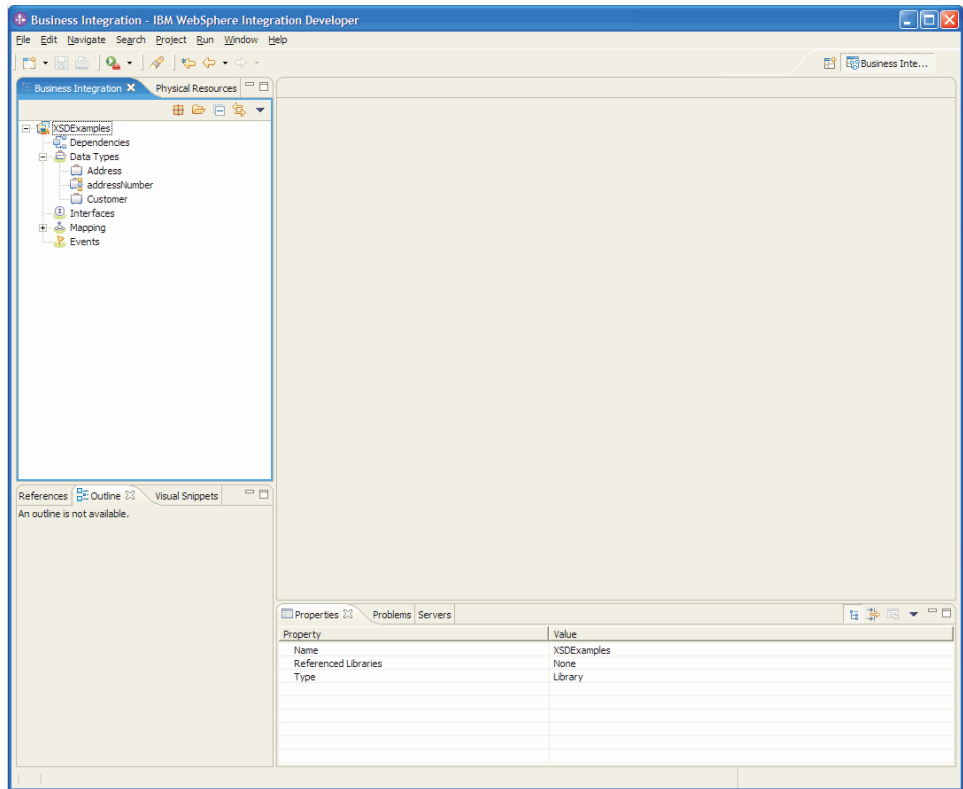
These artifacts result in business objects at runtime:

- Complex types defined at the root level
- Elements defined at the root level with anonymous complex types AND the name of the element does not contain the names of any operations/messages (as these elements could be doc-lit-wrapped elements which WebSphere Integration Developer unwraps automatically)
- Elements defined at the root level which reference a complex type

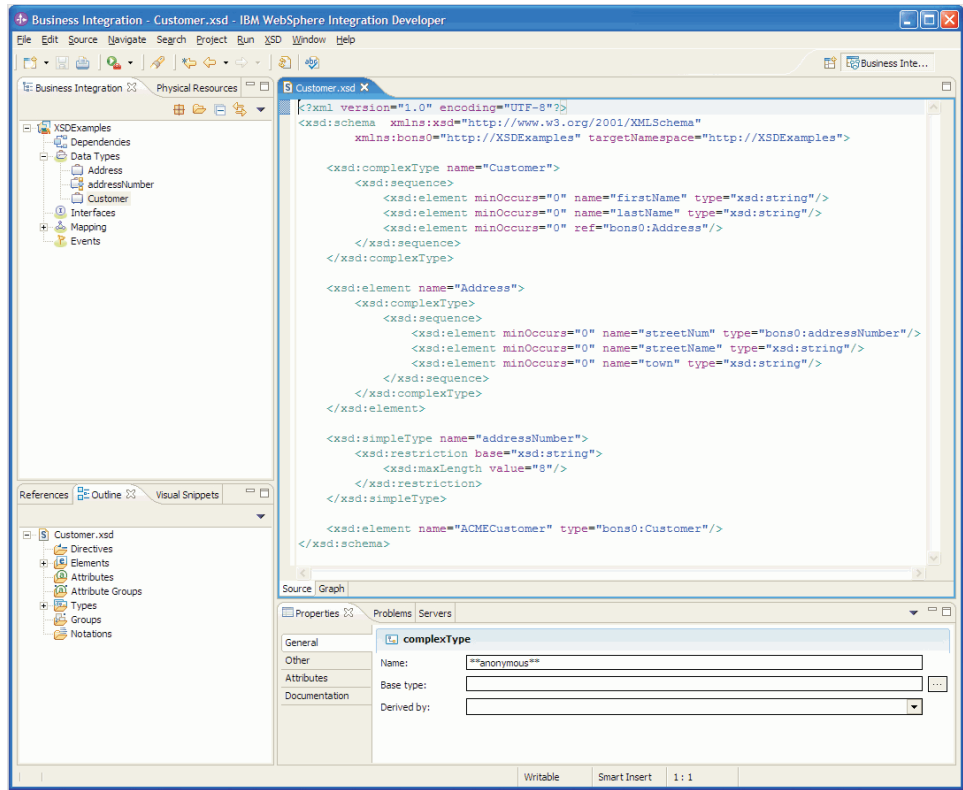
Examples

1. XSD example (supported at authoring time)

This example shows a project (XSDExamples) in the Business Integration view with the business objects shown:



This shows the Customer.xsd file in the XSD Schema editor:



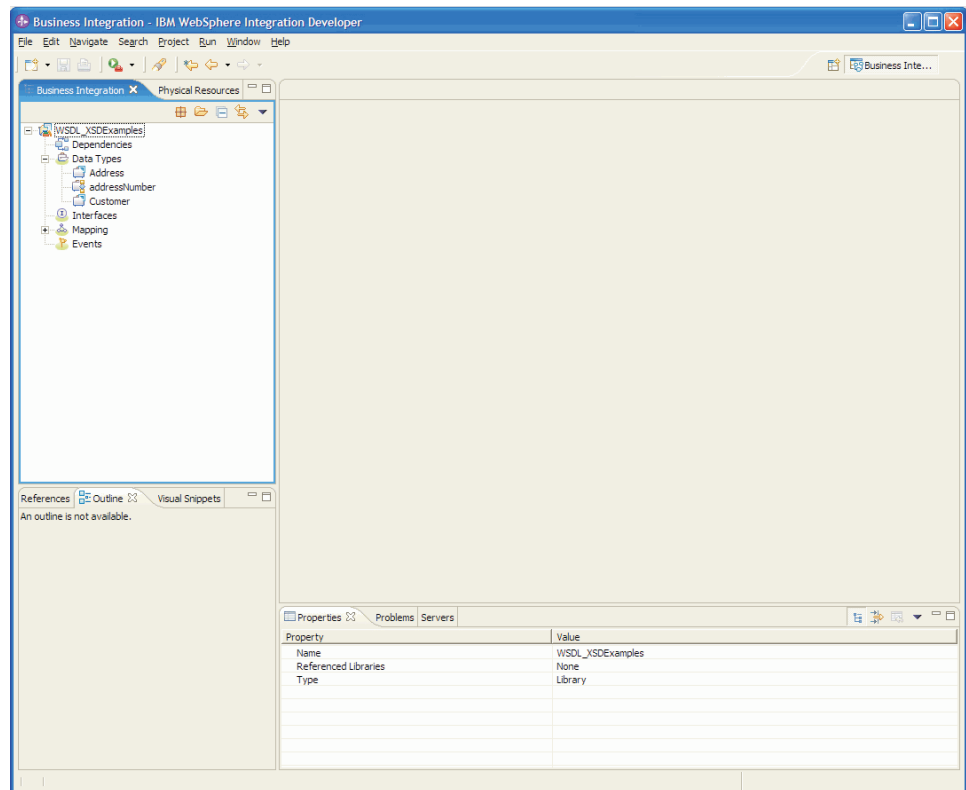
The above examples illustrate the following support:

Table 19. XSD artifact support

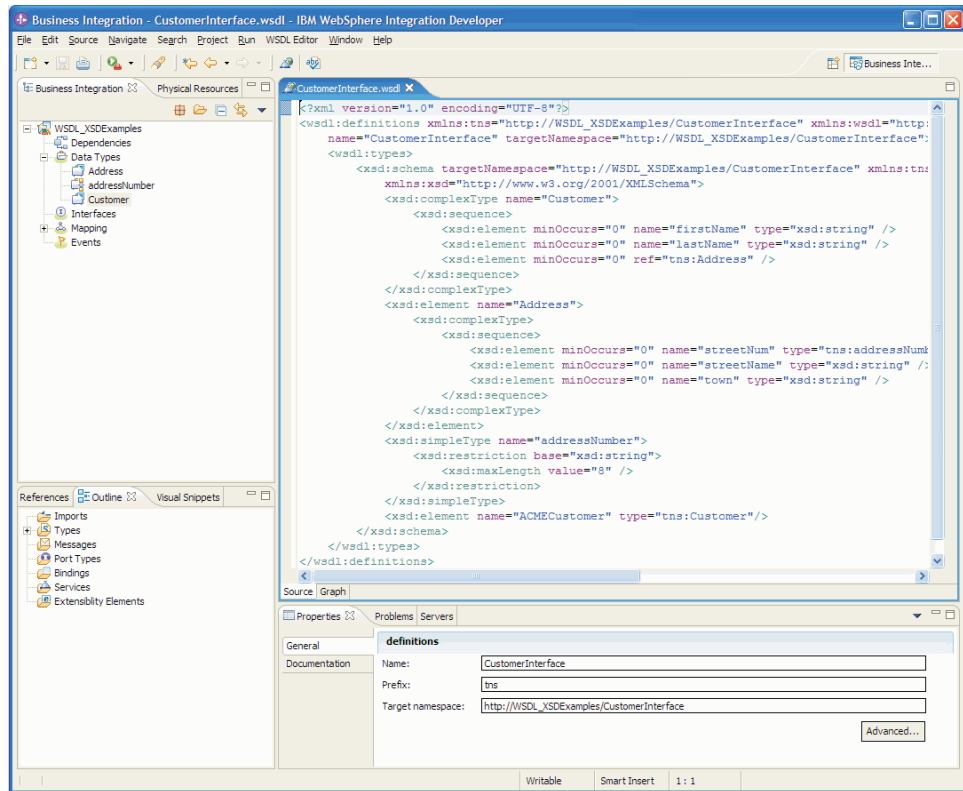
XSD support	XSD artifact in the above example
Complex types defined at the root level	Customer
Elements defined at the root level with anonymous complex types	Address
Elements defined at the root level with user-defined simple types	addressNumber

2. WSDL example (supported at authoring time)

This example shows a project (WSDL_XSDEExamples) in the Business Integration view with the business objects shown:



This screen capture shows the CustomerInterface.wsdl file opened in the WSDL editor:



The above examples illustrate the following support:

Table 20. WSDL artifact support

WSDL support	Inline XSD Artifact in the above example
Complex types defined at the root level	Customer
Elements defined at the root level with anonymous complex types AND the name of the element does <i>not</i> contain the names of any operations/messages (as these could be doc-lit-wrapped elements which WebSphere Integration Developer will unwrap automatically)	Address
Elements defined at the root level with user-defined simple types	addressNumber

3. Runtime example

The above examples illustrate the following runtime support:

Table 21. Runtime artifact support

Runtime support	XSD or Inline XSD artifact in the above examples
All the above in examples 1 and 2 (except for addressNumber as simple types are not business objects)	See above (examples 1 and 2)
Elements defined at the root level which reference a complex type	ACMECustomer (shown in examples 1 and 2)

Flat and hierarchical business objects:

Business objects can be modeled as flat or as hierarchical.

Flat business object

A *flat business object* contains one or more simple attributes and a list of supported verbs. A simple attribute represents one value, such as a String or Integer or Date. All simple attributes have single cardinality. If the business object is an application-specific business object, a flat business object can represent one entity in an application or in a technology standard.

Hierarchical business object

Hierarchical business object definitions define the structure of multiple related entities, encapsulating not only each individual entity but also aspects of the relationship between entities. In addition to containing at least one simple attribute, a hierarchical business object has one or more attributes that are complex (that is, the attribute itself contains one or more business objects, called *child business objects*). The business object that contains the complex attribute is called the *parent business object*.

There are two types of relationships between parent and child business objects:

- Single cardinality - When an attribute in a parent business object represents a single child business object. The type of the attribute is set to the name of the child business object, and the cardinality is set to one.
- Multiple cardinality - When an attribute in the parent business object represents an array of child business objects. The type of the attribute is set to the name of the child business object, and the cardinality is set to *n*.

In turn, each child business object can contain attributes that contain a child business object, or an array of business objects, and so on. The business object at the top of the hierarchy, which itself does not have a parent, is called the *top-level business object*. Any single business object, independent of its child business objects that it might contain (or that might contain it), is called an *individual business object*.

Example

The following example helps illustrate the difference between a flat business object and hierarchical business object. The diagram contains a flat business object, named Product. The business object is represented in memory with the Service Data Object type `commonj.sdo.DataObject` (unless it was statically generated). This flat business object has a set of attributes that are modeled as XML schema simple types as well as an attribute that is modeled as a list of simple types.

The diagram also illustrates a Product business object, in combination with the ProductCategory business object, to create a more complex hierarchical business object. This business object has both a top-level business object (ProductCategory), as well as a contained business object (Product).

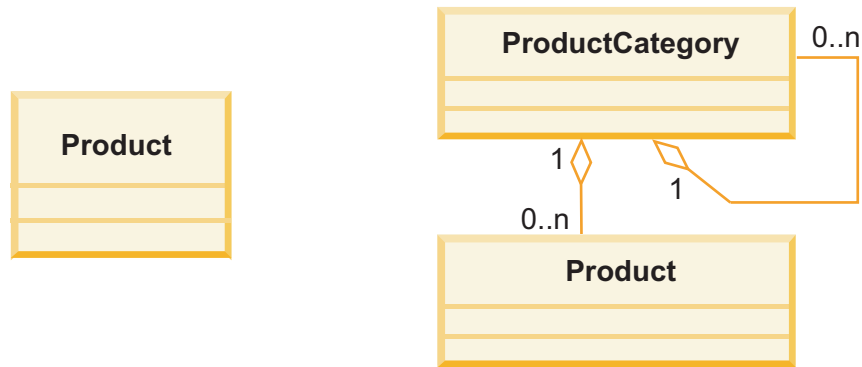


Figure 57. Comparison of flat and hierarchical business objects

Here is example of the flat business object definition for the Product business object. The Product business object defines two properties, *Name* and *Inventory*, that are typed by the XML schema simple types `xs:string` and `xs:int`. In addition, Product also demonstrates the definition of List property *Color*, which is a List of `xs:string` simple types.

```
<schema>
  targetNamespace="http://www.scm.com/ProductTypes"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema">

  <complexType name="Product">
    <sequence>
      <element name="name" type="string"/>
      <element name="inventory" type="int"/>
      <element name="color" type="string" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</schema>
```

Here is example of the hierarchical business object ProductCategory. The definition defines two different business objects, ProductCategory and Product. The hierarchical ProductCategory business object defines the property, *Name* and also defines a List of business objects of type Product or ProductCategory.

```
<schema>
  targetNamespace="http://www.scm.com/ProductCategoryTypes"
  xmlns:pc="http://www.scm.com/ProductCategoryTypes"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema">
  elementFormDefault="qualified">

  <complexType name="ProductCategory">
    <sequence>
      <element name="name" type="string"/>
      <choice>
        <element name="productCategory"
          type="pc:ProductCategory"
          maxOccurs="unbounded"/>
        <element name="product"
          type="pc:Product"
          maxOccurs="unbounded"/>
      </choice>
    </sequence>
```

```

</complexType>

<complexType name="Product">
  <sequence>
    <element name="name" type="string"/>
    <element name="inventory" type="int"/>
    <element name="color" type="string" maxOccurs="unbounded"/>
  </sequence>
</complexType>

</schema>

```

Business object characteristics:

Business objects have inherent characteristics that enhance their use within the business object framework.

Cardinality

The cardinality of properties is defined by standard XML schema `minOccurs` and `maxOccurs` facets for simple and complex types and the `use` attribute for attributes.

Default property values

The capability to provide default values in XML schema for attributes and simple types in a business object is supported by the business object framework. This support is visible at creation time when the simple property types of a business object reflect their default values.

Nilable

An element can be defined in XML schema to be nilable. The business object framework enables properties that are *nilable* to have their value set to `Null` at runtime.

Key definition

Business object key information can be used by multiple subsystems, such as relationship, sequencing, and isolation. However, each of these subsystems can define their own key mechanism independent of the business object's key definition. Since the underlying model language leveraged by business objects is XML schema, first class support for key definitions exists within the modeling language. However, this support within the modeling language is not fully supported in the SDO runtime.

`xs:ID`, `xs:IDREF`, and `xs:IDREFS`

These types were added to XML schema primarily to provide an upgrade path for DTDs. Each complex type can have 0 or 1 elements/attributes typed as an `xs:ID`. IDs must be unique to an entire document, as opposed to a primary key in a database, for example, that must be unique with respect to the scope of the table. As an example, a conformant document cannot use the same ID value to identify both a `Product` and a `ProductCategory`. Often elements get around this restriction by prepending the complex type name to the key value. An attribute typed `IDREF` must contain a value that matches one of the ID values in the current document. In addition, XML schema provides for a construct which is an element that can be typed to contain a list of ID references, `xs:IDREFS`.

xs:unique, xs:key, xs:keyref

XML schema introduced a new style enabling key definitions and key references. The `xs:unique` construct enables a user to define that 1 or more fields in an element that must be unique within a particular scope of the element (which represent the entire document). The `xs:key` construct is a variant of `xs:unique` with the additional constraint that the elements referenced are required. The `xs:keyref` construct is used to identify that the value of an element must be named key or unique construct.

The `unique`, `key`, and `keyref` constructs have several advantages over the `ID`, `IDREF`, and `IDREFS` set, including:

- They can define compound keys.
- They can define unique constraints that are relative to a portion of the document.

Although a business object is not required to have a defined key, it is highly recommended. Business objects that do not define a key can be used by applications. This scenario is a common use model in many Java EE centric applications use models, where JavaBeans are passed back and forth between the servlet and EJB containers without the specification of a key. However, those business objects that do not define a key are unable to interact with the subsystems that require a key. This situation limits their ability to take advantage of WebSphere Process Server qualities of service.

Modeling business graphs

Business graphs relate to business objects in much the same way SDO DataGraphs relate to SDO DataObjects. When a top-level business object requires enrichment to be able to use the services provided by WebSphere Process Server, it is wrapped with a business graph. The business graph wrapper provides the additional value add by adding data headers for storing information logically in memory, or physically when the business graph is serialized.

Note: If you are migrating an application from WebSphere InterChange Server or migrating adapters, you may need to use business graphs.

Business graph use models:

Two primary use models represent the fundamental capabilities provided by the business graph: delta support and after image.

Delta support is the capability enabled by the SDO 1.0, where changes to a business object graph are captured in a special header called the *change summary*.

An *after image* is a business graph that captures the current state of business data in an EIS system, typically as a result of a change to that data in the EIS. An after image enables changes in EIS systems to be captured and published to the runtime.

To provide these two fundamental concepts, business graphs introduce and provide for the following concepts:

- *Templated business graph* is a business graph that is typed specifically for a type of business object graph that it wraps.
- *Change summary* is provided to capture implicit changes, explicit changes, for both Delta, as well as After Image use models.

- *Explicit change summary* support is provided by the business graph programming interfaces enabling BPM component, such as adapters, mediators, maps, and relationships to explicitly modify the Change Summary header.
- *Event summary* is provided to support capturing instance-based annotations about data in the business object graph, and specifically, to contain object event identifiers
- *Verb* support is provided by the business graph enabling the components in the runtime to key off the event type to perform value added function.
- *Supported verbs* is the notion of constraining and extending the set of allowable verbs that can be specified for a business graph.
- *Object event identifiers* are supported by the business graph enabling all the objects in a graph to be uniquely identified. This capability is required by some of the components that provide value added features.

Business graph model definition:

To remain non-intrusive to an externally developed model of a business object, the business graph capability is wrapped around the original business object. A pattern named the templated business graph is used to wrap the original business object with the enriched business graph schema.

The *templated business graph* is created by extending the business graph complex type that is provided by the business object framework runtime and adding an element delegating to the original business object. The diagram shows a UML model for the business graph. The business graph is abstract, providing just the standard set of headers that are added to the top-level business object.

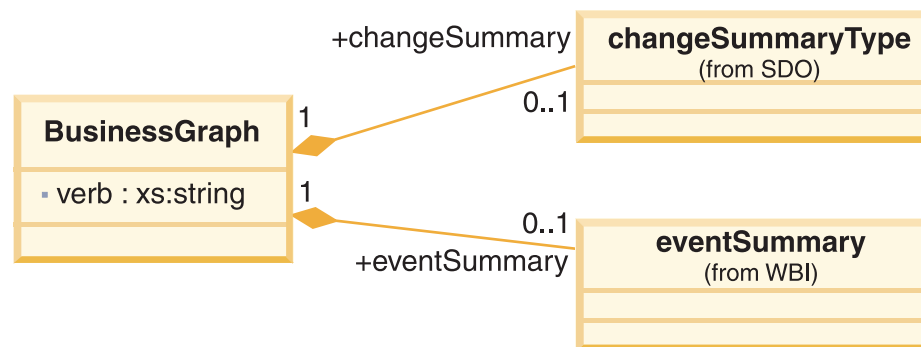


Figure 58. Business graph complex type

The XML schema model for the abstract business graph XML schema complex type:

```

<schema
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
  xmlns:bo="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
  xmlns:sdo="commonj.sdo"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  
```



```

<import namespace="commonj.sdo" schemaLocation="DataGraph.xsd"/>

<complexType name="BusinessGraph" abstract="true">
  <sequence>
    <element name="changeSummary" type="sdo:ChangeSummaryType"
      minOccurs="0" maxOccurs="1"/>
    <element name="eventSummary" type="bo:EventSummary"
      minOccurs="0" maxOccurs="1"/>
    <element name="property" type="bo:ValueType"
      minOccurs="0"/>
  </sequence>
  <anyAttribute namespace="##other" processContents="lax"/>
</complexType>

<complexType name="EventSummary">
  <sequence>
    <any namespace="##any" processContents="lax"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="ValueType">
  <complexContent>
    <extension base="ecore:EClass"/>
  </complexContent>
</complexType>

<attribute name="name" type="string"/>
</schema>

```

Business graphs are only a top-level concept because they exist to add a set of headers to an existing top-level business object, and cannot be modeled in a recursive design pattern like business objects can. Business graphs can be applied to any business object, but upon application, that business object becomes a top-level business object.

This is an example of a business graph that wraps a business object named ProductCategory, ProductCategory is a hierarchical business object that contains a child business object named Product.

```

<schema
  targetNamespace="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
  xmlns:pcbg="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
  xmlns:pc="http://www.scm.com/ProductCategoryTypes"
  xmlns:bo="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <import namespace="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
    schemaLocation="BusinessGraph.xsd"/>

  <import namespace="http://www.scm.com/ProductCategoryTypes"
    schemaLocation="ProductCategoryTypes.xsd"/>

  <complexType name="ProductCategoryBG">
    <complexContent>
      <extension base="bo:BusinessGraph">
        <sequence>
          <element name="verb" minOccurs="0" maxOccurs="1"/>
          <element name="productCategory"
            type="pc:ProductCategory"
            minOccurs="0" maxOccurs="1"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

```

```
</extension>
<complexContent>
<complexType>
</schema>
```

The recommended pattern for a generated templated business graph is:

- The templated business graph are defined using a named complex type that extends the `bo:BusinessGraph` schema by restriction (this pattern provides restrictions on the allowable Verb value).
- The name of the templated business graph is the name of the top-level business object, with the string "BG" appended.
- The target namespace of the templated business graph is composed of the target namespace of the business object being wrapped, followed by a "/", named by the complex type of the templated business graph.
- The templated business graph complex type definition are placed in its own XML schema file whose name corresponds to the name of the complex type.

Business graph model instance:

The top-level business graph is represented in memory much the same as a business object, by an SDO 1.0 DataObject, specifically, the class `commonj.sdo.DataObject`.

The business graph is composed of more than just the business graph containment object. It also contains two headers, plus the top-level business object. None of the headers is represented in memory as a DataObject, and neither of the headers enables a DataObject API access mechanism.

Change summary header

A capability provided by business graphs is the ability to implicitly track changes to the business object in the business graph as the business graph is passed between several different business processes. As each process changes the business graph, a change log is generated in memory. Once the business graph is serialized, the change log is written out in a format that enables the next process to see the types of changes that have been made to the business graph. This technique enables adapter and Data Mediator services to efficiently update their persistence data stores by optimizing the data they have to focus on.

In addition, the change summary is also used when an adapter generates an After Image event to describe data that was updated in an EIS system. In particular, the ability of the change summary to annotate object and property changes is used in the After Image use model (as well as the verb for the business graph).

Implicit change summary usage

When applications cause changes to the business object, the applications can turn on change logging so change events are automatically logged to the change summary. The change summary is included in the business graph at a peer level of the top-level business object. Change events are defined at two levels, for business objects, and for business object properties. Business objects have create, update, and delete change types, while properties have set and unset change types. Change events are only tracked for modifications to the business object portion of the business graph. A change in the event summary does not result in an implicit update of the change summary for the business graph.

Explicit change summary modification

There are several use models in the WebSphere Process Server components that require the ability to explicitly write to the change summary header. For example, an adapter that generates an EIS event explicitly creates the object change types, and potentially the property change types. An application-specific business object/general business object (ASBO/GBO) map transforms the change summary from one business graph to another, creating a new version of the change summary in an output business graph. This capability is provided by the business object framework.

Event summary header

The event summary provides the **ObjectEventID**, which is the mechanism used to uniquely identify an instance of an object that appears in the runtime. This information is carried in the event summary, where the unique identifier is associated with a given DataObject in the business object hierarchy of the business graph.

Event information can also be carried in the event summary. This information is a string that can be used to add additional metadata associated with each object in the business object hierarchy for the business graph. One potential use model for event information is to mark up contained business objects with a verb other than the standard Create, Update, and Delete verbs supported by change summary.

Verb header

If the verb is set on the business graph, then the business object data portion of the business graph carries an EIS After Image data set. If the verb contains a value, there are three possibilities with respect to the granularity of the after image event:

- The change summary is empty. This situation occurs when an EIS knows about the type of update to a set of data, but does not have specifics on which objects in the graph have been created, updated, or deleted. The result is that a downstream mediation, map, relationship, or adapter, must use the information in the business graph plus additional data to determine the actual update to perform.
- The change summary has object level change event annotations. This case is typical where the EIS system recognizes what has happened to each object in the business graph, but lacks the granularity to determine whether specific properties of the objects have been updated.
- The change summary has object level change event annotations and property level get/set annotations. This situation is the most granular case, and all adapters strive to obtain this level of an After Image if their EIS system makes the appropriate data available. The advantage of a fully specified After Image is that it enables property level implicit property change event management to occur. Therefore, it is much easier for After Image business graphs to interoperate with the disconnected Delta-based Business Graphs.

Modeling business object type metadata

Business object type metadata can be added to business object definitions to enhance their value in the runtime. The business object framework allows you to design, annotate, and convert business object type metadata.

The business object framework provides a mechanism that enables:

- Metadata to be mixed into a business object in a consistent, and relatively non-intrusive fashion
- A prescriptive policy for developers to define complex annotation structures
- A prescriptive policy for business object developers and deployers to annotate a business object with instance metadata that conforms to the predefined complex annotation structures
- A set of APIs for transforming the annotations at runtime into a simple to use DataObject structure

This process involves at least three different roles:

- The first role is that of the business object type metadata designer. This role designs the metadata structure. For example, the business object framework plays this role and defines a couple of metadata characteristics for annotating a business object. It is expected that adapters such as PeopleSoft, Siebel, and SAP, play the role of the metadata designer to annotate the business object with application-specific information.
- The second role is the business object designer or deployer. This role uses the structure of the business object type metadata, and the policy defined by the business object type metadata framework, to annotate the business object definitions with metadata.
- If the business object is annotated correctly, the third role can use the business object metadata APIs to validate and inspect the metadata at runtime, and turn it into a navigable and useful DataObject graph structure.

In addition, the business object framework provides for the following metadata features:

- Compound primary key and foreign key property definitions
- Top-level supported verb metadata annotations

For more details, refer to these other topics.

Representing business object type metadata:

The business object framework defines a mechanism by which business object type metadata can be mixed into either a top down, or externally developed, imported schema.

The mechanism that is used to mix in the business object type metadata, is XML schema annotations and the appInfo structure. This policy, although it requires modification of the original schema definition, does so in a way that enables the annotations and appinfo to be easily view toggled, or removed completely. This identification scheme is done by using the source attribute on the xs:appinfo tag whose value is the target namespace that defines the business object type metadata.

For example, assume that the following schema was imported into the runtime. The schema describes a business object, named Product, that includes customer annotations.

```
<schema>
  targetNamespace="http://www.scm.com/ProductTypes"
  xmlns:p="http://www.scm.com/ProductTypes"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
```

```

<complexType name="Product">
  <annotation>
    <appInfo>
      <SCMEditor value="Bottom" type="Anchor"/>
    </appInfo>
    <documentation>
      Describes the SCM Product
    </documentation>
  </annotation>

  <sequence>
    <element name="id" type="ID"/>
    <element name="description" type="string" default="DefaultDescription"/>
    <element name="sku" type="p:Sku"/>
  </sequence>
</complexType>

<simpleType name="Sku">
  <restriction base="string">
    <pattern value="\d{3}-[A-Z]{2}"/>
  </restriction>
</simpleType>
</schema>

```

Designing business object type metadata:

You can design a metadata structure to house the business object metadata.

About this task

To design the metadata structure follow these steps.

Procedure

1. Create an XML schema file with a valid target namespace. This step is used when the instance metadata is added to the business object definition.
2. Define a named complex type to define each separate piece of metadata that can be added to a business object. The complex type definition is used to define the structure of the dynamically typed DataObject that is used to read the instance metadata. The name of the complex type is used when the instance metadata is added to the business object definition.

Example

This example shows a portion of business object type metadata developed for a PeopleSoft adapter.

```

<schema>
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/adapter/psft/
  PSFTB0DefinitionASI/7.0.0"
  xmlns:psft="http://www.ibm.com/xmlns/prod/websphere/adapter/psft/
  PSFTB0DefinitionASI/7.0.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <complexType name="PSFTB0DefinitionASI">
    <sequence>
      <element name="hostname" type="string"/>
      <element name="ipaddress" type="string"/>
    </sequence>
  </complexType>
</schema>

```

Annotating a business object definition:

You can use the syntax supported by the business object framework to annotate a business object definition.

About this task

The business object framework does not attempt to define the mechanism by which instance metadata data is attached to the business object definition. However, it does define the syntax of the instance metadata.

To annotate the business object definition follow these guidelines.

Procedure

1. If the part of the business object definition that the instance metadata is to be attached to does not already have an annotation, add one. If it already has an annotation, utilize the existing one.
2. Create a separate `xs:appinfo` tag within the `xs:annotation` tag, and define the source attribute to the namespace defined by the business object type metadata being added.
3. Inside the `xs:appinfo` tag, define a `QName`, using a target namespace prefix, followed by the name of the complex type definition. Append the attribute `QName` defined with `xmlns:` followed by the previously used target namespace prefix. Set the value of this `QName` attribute to the target namespace of the business object type metadata being used.
4. Add the structure and instance data mandated by the business object type metadata definition. Close all tags as appropriate.

Example

Once the object is imported in, it is identified to need a set of PeopleSoft annotations. This example shows the same business object definition with the PeopleSoft metadata added.

```
<schema>
  targetNamespace="http://www.scm.com/ProductTypes"
  xmlns:p="http://www.scm.com/ProductTypes"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified

  <complexType name="Product">
    <annotation>
      <appinfo source="http://www.ibm.com/xmlns/prod/websphere/Adapter/PSFT">
        <psft:PSFTMetadata
          xmlns:psft="http://www.ibm.com/xmlns/prod/websphere/Adapter/PSFT">
          <hostname>mumbai</hostname>
          <ipaddress>9.29.1.1</ipaddress>
        </psft:PSFTMetadata>
      </appinfo>
    </annotation>
    <documentation>
      Describes the SCM Product
    </documentation>
  </complexType>

  <sequence>
    <element name="id" type="ID"/>
  </sequence>
</schema>
```

```

    <element name="description" type="string" default="DefaultDescription"/>
    <element name="sku" type="p:Sku"/>
  </sequence>
</complexType>

<simpleType name="Sku">
  <restriction base="string">
    <pattern value="\d{3}-[A-Z]{2}"/>
  </restriction>
</simpleType>
</schema>

```

Converting annotation into DataObjects:

The business object framework provides the capability to transform annotations into a usable DataObject structure.

About this task

The annotations associated with a business object definition can be read at runtime by using the SDO implementation-specific set of APIs. However, the problem with these APIs is that they return a binary large object (BLOB). However, the business object framework provides a utility that, if the recommended annotation patterns have been followed, reads the BLOB, validates it, and transforms it into a usable DataObject structure.

To convert an annotation into a DataObject.

Procedure

1. Obtain an annotation.
2. Use the BOTypeMetadata to convert that annotation into an SDO DataObject. The BOTypeMetadata implementation is available as a singleton using the BOTypeMetadata.INSTANCE instance.

Example

The following example demonstrates how to use the APIs to obtain an annotation and then the use the BOTypeMetadata API to convert that annotation into an SDO DataObject. The metadata is defined in BOTypeMetadata.xsd.

```

<schema>
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <complexType name="VerbInfo">
    <sequence>
      <element name="verbInfo" type="string"/>
    </sequence>
  </complexType>
</schema>

```

One of the capabilities of the business object framework is the ability to add additional supported verbs and accompanying metadata to enable meta-driven capabilities at runtime. The capability is supported through the use of verbs and the VerbInfo metadata to annotate the verbs with additional metadata. The following example demonstrates the places where the VerbInfo metadata would be added for each of the possible Verb values.

```

<schema
  targetNamespace="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
  xmlns:pcbg="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
  xmlns:pc="http://www.scm.com/ProductCategoryTypes"
  xmlns:sdo="commonj.sdo"
  xmlns:bo="http://www.ibm.com/xmlns/prod/websphere/bo/7.0.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <import namespace="http://www.ibm.com/xmlns/prod/websphere/bo/7.0.0"
    schemaLocation="BusinessGraph.xsd"/>

  <import namespace="commonj.sdo"
    schemaLocation="DataGraph.xsd"/>

  <import namespace="http://www.scm.com/ProductCategoryTypes"
    schemaLocation="ProductCategoryTypes.xsd"/>

  <complexType name="ProductCategoryBG">
    <complexContent>
      <extension base="bo:BusinessGraph">
        <sequence>

          <element name="verb"
            minOccurs="0" maxOccurs="1">
            <simpleType>
              <restriction base="string">
                <enumeration value="Create">
                  <annotation>
                    <appinfo source="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
                      <botm:VerbInfo
                        xmlns:botm="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
                        <verbInfo>Metadata relating to Create</verbInfo>
                      </botm:VerbInfo>
                    </appinfo>
                  </annotation>
                </enumeration>
                <enumeration value="Retrieve">
                  <annotation>
                    <appinfo source="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
                      <botm:VerbInfo
                        xmlns:botm="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
                        <verbInfo>Metadata relating to Retrieve</verbInfo>
                      </botm:VerbInfo>
                    </appinfo>
                  </annotation>
                </enumeration>
              </restriction>
            </simpleType>
          </element>

          <element name="productCategory" type="pc:ProductCategory"
            minOccurs="0" maxOccurs="1"/>

        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <element name="productCategoryBG" type="pcbg:ProductCategoryBG"/>
</schema>

```


Programming using business object services

To facilitate the creation and manipulation of business objects, the business object framework extends SDO specifications by providing a set of Java services. These services are part of the package named `com.ibm.websphere.bo`.

This is a brief description of the business object services.

Table 22. Business object services

Service	Description
BOChangeSummary	Provides enhancements to the SDO change summary interface to manage the business graph's change summary header.
BOCopy	Facilitates copying a graph of business objects or a business graph that contains a graph of business objects.
BODataObject	Provides enhancements to the SDO Data Object interface.
BOEquality	Provides the ability to determine if two business graphs or business objects are equivalent.
BOEventSummary	Provides the interface for managing the content of the business graph's event summary header.
BOFactory	Provides the capability to create a business graph or a business object.
BOType	Provides a mechanism to obtain the SDO type of a business graph or business object that mirrors what <code>Class.forName()</code> provides for Java class names.
BOTypeMetadata	Provides the capability of taking an annotation binary large object (BLOB) that conforms to the BO Type Metadata pattern and transforms it into a set of SDO DataObjects (and performs the reserve transform).
BOXMLDocument/BOXMLSerializer	Provides the mechanisms for creating and representing an XML Document in memory, and for serializing and de-serializing an XML document.

Table 22. Business object services (continued)

Service	Description
BOInstanceValidator	<p>Provides a facility to validate a business object instance against its XSD definition. Business objects can be present with various forms. They can be simple business objects or wrapped by the enriched business graph model. In certain business integration scenarios, the business objects are in the deleted section of change summary. These business objects drive the downstream business logics. The accuracy of the business objects need to be ensured in all cases. There are two supported styles for BOInstanceValidator:</p> <ul style="list-style-type: none"> • Explicit Programmatic Validation: A system service is provided to validate business objects via a set of programming APIs. • Implicit Interface Validation: This validation is enabled/disabled via WebSphere Integration Developer on the target interfaces via a SCA interface qualifier.

XML document validation

XML documents and business objects can be validated using the validation service.

In addition, other services require certain minimum standards or they throw a runtime exception. One of these is `BOXMLSerializer`.

You can use the `BOXMLSerializer` to validate XML documents before they are processed by a service request. The `BOXMLSerializer` validates the structure of XML documents to determine if any of the following types of errors are present:

- Invalid XML documents, such as those that are missing certain element tags.
- Not well-formed XML documents, such as those that contain missing closing tags.
- Documents containing parsing errors, such as errors in entity declaration.

When an error is discovered by the `BOXMLSerializer`, an exception will be thrown with problem details.

The validation can be performed for import and/or export of XML documents for the following services:

- HTTP
- JAXRPC web services
- JAX-WS web services
- JMS services
- MQ services

For the HTTP, JAXRPC, and JAX-WS services, the `BOXMLSerializer` will generate exceptions in the following manner:

- Imports –
 1. The SCA component invokes the service.

2. The service invokes a destination URL.
 3. The destination URL responds with an invalid XML exception.
 4. The service fails with a runtime exception and message.
- Exports –
 1. The service client invokes the service export.
 2. The service client sends an invalid XML
 3. The export fails for the service and generates an exception and message.

For the JMS and MQ messaging services, the exceptions are generated in the following manner:

- Imports –
 1. The import invokes the JMS or MQ service.
 2. The service returns a response.
 3. The service returns an invalid XML exception.
 4. The import fails and generates a message.
- Exports –
 1. The MQ or JMS client invokes an export.
 2. The client sends invalid XML.
 3. The export fails and generates an exception and message.

You can view the logs for any messages generated by an XML validation exception. The examples below are messages generated by improper XML coding that was validated by the `BOXMLSerializer`

- JAXWS import

```
javax.xml.ws.WebServiceException: org.apache.axiom.om.OMException:
javax.xml.stream.XMLStreamException: Element type "TestResponse" must be
followed by either attribute specifications, ">" or "/>".
```

```
javax.xml.ws.WebServiceException: org.apache.axiom.soap.SOAPProcessingException:
First Element must contain the local name, Envelope
```

- JAXRPC import

```
[9/11/08 15:16:27:417 CDT] 0000003e ExceptionUtil E
CNTR0020E: EJB threw an unexpected (non-declared)
exception during invocation of method
"transactionNotSupportedActivitySessionNotSupported" on bean
"BeanId(WXXMLValidationApp#WXXMLValidationEJB.jar#Module, null)".
Exception data: WebServicesFault
faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
faultString: org.xml.sax.SAXParseException: Element type "TestResponse"
must be followed by either
attribute specifications, ">" or "/>". Message being parsed:
<?xml version="1.0"?><TestResponse
xmlns="http://WXXMLValidation"><firstName>Bob</firstName>
<lastName>Smith</lastName></TestResponse>
faultActor: null
faultDetail:
[9/11/08 15:16:35:135 CDT] 0000003f ExceptionUtil E CNTR0020E: EJB threw an
unexpected (non-declared) exception during invocation of method
"transactionNotSupportedActivitySessionNotSupported" on bean
"BeanId(WXXMLValidationApp#WXXMLValidationEJB.jar#Module, null)".
Exception data: WebServicesFault
faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
faultString: org.xml.sax.SAXException: WWS3066E: Error: Expected 'envelope'
but found TestResponse
Message being parsed: <?xml version="1.0"?><TestResponse
xmlns="http://WXXMLValidation">
```

```

    <firstName>Bob</firstName><middleName>John</middleName>
    <lastName>Smith</lastName>
  </TestResponse>
  faultActor: null
  faultDetail:

```

- JAXRPC/JAXWS export

```

[9/11/08 15:35:13:401 CDT] 00000064 WebServicesSe E
com.ibm.ws.webservices.engine.transport.http.WebServicesServlet
getSoapAction WSWS3112E:
  Error: Generating WebServicesFault due to missing SOAPAction.
      WebServicesFault
  faultCode: Client.NoSOAPAction
  faultString: WSWS3147E: Error: no SOAPAction header!
  faultActor: null
  faultDetail:

```

For more information about validation services, see the `BOInstanceValidator` interface in the Generated API and SPI documentation in the Reference section.

Programming techniques

These techniques illustrate how to effectively program business objects using the business object framework.

Arrays in business objects

You can define arrays for an element in a business object so that the element can contain more than one instance of data.

You can use a List type to create an array for a single named element in a business object. This will allow you to use that element to contain multiple instances of data. For example, you can use an array to store several telephone numbers within an element named `telephone` that is defined as a string in its business object wrapper. You can also define the size of the array by specifying the number of data instances using the `maxOccurs` value. The following example code shows how you would create such an array that will hold three instances of data for that element:

```
<xsd:element name="telephone" type="xsd:string" maxOccurs="3"/>
```

This will create a list index for the element `telephone` that can hold up to three data instances. You may also use the value `minOccurs` if you are only planning to have one item in the array.

The resulting array consists of two items:

- the contents of the array
- the array itself.

In order to create this array, however, you need perform an intermediate step by defining a wrapper. This wrapper, in effect, replaces the property of the element with an array object. In the example above, you can create an `ArrayOfTelephone` object to define the element `telephone` as an array. The following code example shows how you accomplish this task:

```

<?xml version="1.0" encoding="UTF-8"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Customer">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="ArrayOfTelephone" type="ArrayOfTelephone"/>
        </xsd:sequence>
      </xsd:complexType>

```

```

</xsd:element>

<xsd:complexType name="ArrayOfTelephone">
  <xsd:sequence maxOccurs="3">
    <xsd:element name="telephone" type="xsd:string" nillable="true"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

The telephone element now appears as a child of the ArrayOfTelephone wrapper object.

Note that in the example above, the telephone element contains a property named `nillable`. You can set this property to `true` if you want to certain items in the array index to contain no data. The following example code shows how the data in an array may be represented:

```

<Customer>
  <name>Bob</name>
  <ArrayOfTelephone>
    <telephone>111-1111</telephone>
    <telephone xsi:nil="true"/>
    <telephone>333-3333</telephone>
  </ArrayOfTelephone>
</Customer>

```

In this case, the first and third items in the array index for the telephone element contain data, while the second item does not contain any data. If you had not used the `nillable` property for the telephone element, then you would have had to have the first two elements contain data.

You can use the Service Data Object (SDO) Sequence APIs in WebSphere Process Server as an alternative method to handle sequences in business object arrays. The following example code will create an array for the telephone element with data identical to that shown above:

```

DataObject customer = ...
customer.setString("name", "Bob");

DataObject tele_array = customer.createDataObject("ArrayOfTelephone");
Sequence seq = tele_array.getSequence(); // The array is sequenced
seq.add("telephone", "111-1111");
seq.add("telephone", null);
seq.add("telephone", "333-3333");

```

You can return the data for a given element array index by using code similar to the example below:

```
String tele3 = tele_array.get("telephone[3]"); // tele3 = "333-3333"
```

In this example, a string named `tele3` will return the data "333-3333".

You can enter the data items for the array in the list index by using fixed width or delimited data placed in a JMS or MQ message queue. You can also accomplish this task by using a flat text file that contains the properly formatted data

Creating nested business objects

You can use the `setWithCreate` function to create nested business objects within a parent business object.

You can create nested business objects from a parent business object without having to write code that details intermediate child objects. For instance, you can

set a nested business object two levels below the parent object without having to define a dependent business object one level below the parent object. Use the `setWithCreate` function to accomplish this task for:

- a single instance
- multiple instances
- a wildcard value
- a model group

The following topics describe how you can do each of these.

Single instance of a nested business object:

Use the `setWithCreate` function to create a single instance of nested business object.

Before you begin

The example code below shows how you would normally have to create code for an intermediate (child) object from a higher level (parent) object in order to create a third-level (grandchild) object. The XSD file would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="Child"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Child">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="grandChild" type="GrandChild"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="GrandChild">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

About this task

If you used the traditional "top-down" method to set the business object data, you would have to process the following code specifying the child and grandchild objects before setting the data in the grandchild object:

```
DataObject parent = ...
DataObject child = parent.createDataObject("child");
DataObject grandchild = child.createDataObject("grandChild");
grandchild.setString("name", "Bob");
```

You can use a more efficient method by using the `setWithCreate` function to simultaneously define the grandchild object and set its data, without having to specify the intermediate child object. The following example code shows how you would accomplish this task:

```
DataObject parent = ...
parent.setString("child/grandchild/name", "Bob");
```

Results

The lower-level business object data is set without having to reference the intermediate-level business object. An exception occurs if the path is not valid.

Creating multiple instances of nested business objects:

Use the `setWithCreate` function to create a multiple instances of nested business object.

Before you begin

The example XSD file below contains nested objects one (child) and two (grandchild) levels below the top (parent) business object:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="Child" maxOccurs="5"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Child">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="grandChild" type="GrandChild"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="GrandChild">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Note that the parent object can have up to five child objects, as specified in the `maxOccurs` value.

About this task

You can create a list with a more stringent policy that will not allow for missing sequences in an array. You can use the `setWithGet` method, and at the same time specify the data that will appear in a particular list index item:

```
DataObject parent = ...
parent.setString("child[3]/grandchild/name", "Bob");
```

In this case, the resulting array would be of size three, but the values for `child[1]` and `child[2]` list index items are undefined. You may want the items to either be a null value or have an associated data value. In the scenario above, an exception will be thrown because the values for the first two array index items are undefined.

You can remedy this situation by defining the values in the index of the list. If the index item refers to an existing element in the array and if that element is not null (that is, it contains data), it will be used. If it is null, it will be created and used. If the index of the list is one greater than the size of the list, a new value will be created and added. The following example code shows what will happen in a list that of size two, where `child[1]` is designated null and `child[2]` contains data:

```
DataObject parent = ...
// child[1] = null
// child[2] = existing Child
// This code will work because child[1] is null and will be created.
parent.setString("child[1]/grandchild/name", "Bob");

// This code will work because child[2] exists and will be used.
parent.setString("child[2]/grandchild/name", "Dan");

// This code will work because the child list is of size 2, and adding
// one more list item will increase the list size.
parent.setString("child[3]/grandchild/name", "Sam");
```

Results

You have overridden the values for the two existing items and added a third item to the list index. If, however, you then add another item that is not of size four, or which is greater than the size specified in `maxOccurs`, then an exception will be thrown. The more stringent policy of this method is demonstrated in the following example code.

Note: The code below is assumed to be appended to the existing code above:

```
// This code will throw an exception because the list is of size 3
// and you have not created an item to increase the size to 4.
parent.setString("child[5]/grandchild/name", "Billy");
```

Using a nested business object defined by a wildcard:

You can specify the type `xsd:any` in a parent object to specify a child object, but only if the child object already exists.

About this task

The `setWithCreate` function used to define nested business objects for single and multiple instances does not work if you are using a wildcard value of `xsd:any` in the Service Data Object. This is illustrated in the following example code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="xsd:anyType"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Results

An exception will be thrown if the child data object does not exist.

Using business objects in model groups:

You create the model group path patterns when working with nested business objects that are part of a model group.

About this task

Model groups use the tag `xsd:choice` that you can use to create business objects from a parent business object. The business object framework, however, can cause naming conflicts that can generate an exception. The following example code illustrates how naming conflicts can occur:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://MultipleGroup">
  <xsd:complexType name="MultipleGroup">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="child1" type="Child"/>
        <xsd:element name="child2" type="Child"/>
      </xsd:choice>
      <xsd:element name="separator" type="xsd:string"/>
      <xsd:choice>
        <xsd:element name="child1" type="Child"/>
        <xsd:element name="child2" type="Child"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Note: there can be multiple instances of the elements named "child1" and "child2",

Use the Service Data Object (SDO) path patterns for model groups to resolve these conflicts.

Results

You would get arrays that use the SDO path pattern that is used to handle model groups, as shown in the following example code:

```
set("child1/grandchild/name", "Bob");

set("child11/grandchild/name", "Joe");
```

Differentiating identically named elements

You must provide unique names for business object elements and attributes.

In the Service Data Object (SDO) framework, elements and attributes are created as properties. In the following code examples, the XSDs create types that have one property named `foo`:

```
<xsd:complexType name="ElementFoo">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string" default="elem_value"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="AttributeFoo">
  <xsd:attribute name="foo" type="xsd:string" default="attr_value"/>
</xsd:complexType>
```

In these cases, you can access the property using the XML Path Language (XPath). However, valid schema types can have an attribute and element of the same name, as in the following example:

```

<xsd:complexType name="DuplicateNames">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string" default="elem_value"/>
  </xsd:sequence>
  <xsd:attribute name="foo" type="xsd:string" default="attr_value"/>
</xsd:complexType>

```

In XPath, you must be able to differentiate identically named elements from attributes. This is achieved by beginning one of the names with an at sign (@). The following snippet shows how to access the identically named element and attribute:

```

1 DataObject duplicateNames = ...

2 // Displays "elem_value"
3 System.out.println(duplicateNames.get("foo"));

4 // Displays "attr_value"
5 System.out.println(duplicateNames.get("@foo"));

```

Use this naming scheme for all methods that take a String value that is an SDO XPath.

Model group support (all, choice, sequence, and group references):

The SDO specification requires model groups (all, choice, sequence, and group references) to be expanded in place and not describe types or properties.

Basically, this means that any of those structures that are within the same containing structures are "flattened". This "flattening" puts all the child structures at the same level, which can produce duplicate naming issues in an SDO whose structure is derived from the flattened data. When an XSD does not flatten the groups, there is still a separation of duplicates that are contained by different parents.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://MultipleGroup">
  <xsd:complexType name="MultipleGroup">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
      <xsd:element name="separator" type="xsd:string"/>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

Since the multiple occurrences of option1 and option2 are contained in different choice blocks and even have a separating element between them, XSD and XML have no problem distinguishing between them. But when SDO flattens these groups, all the option properties are now under the same container of MultipleGroup.

Even without duplicate names, there is also the semantic issue that the flattening of these groups cause. Take the following XSD for example:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://SimpleChoice">
  <xsd:complexType name="SimpleChoice">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

Asking the user to rename duplicate names or add special annotations to XSDs is impractical because in many cases, like standards and industry schemas, the user does not control the XSDs they are working with.

To create consistency for all properties, business objects include a method to access each individual occurrence of the duplicate named properties through XPath. Following the business object framework naming convention, any duplicate property names encountered have the next unused digit appended to their name. So for example, the following XSD:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://TieredGroup">
  <xsd:complexType name="TieredGroup">
    <xsd:sequence>
      <xsd:choice minOccurs="0">
        <xsd:sequence>
          <xsd:element name="low" minOccurs="1"
            maxOccurs="1" type="xsd:string"/>
          <xsd:choice minOccurs="0">
            <xsd:element name="width" minOccurs="0"
              maxOccurs="1" type="xsd:string"/>
            <xsd:element name="high" minOccurs="0"
              maxOccurs="1" type="xsd:string"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:element name="high" minOccurs="1"
          maxOccurs="1" type="xsd:string"/>
        <xsd:sequence>
          <xsd:element name="width" minOccurs="1"
            maxOccurs="1" type="xsd:string"/>
          <xsd:element name="high" minOccurs="0"
            maxOccurs="1" type="xsd:string"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="center" minOccurs="1"
            maxOccurs="1" type="xsd:string"/>
          <xsd:element name="width" minOccurs="0"
            maxOccurs="1" type="xsd:string"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

The preceding XSD produces the following DataObject model:

```

DataObject - TieredGroup
Property[0] - low - string
Property[1] - width - string
Property[2] - high - string
Property[3] - high1 - string

```

```
Property[4] - width1 - string
Property[5] - high2 - string
Property[6] - center - string
Property[7] - width2 - string
```

Where **width**, **width1**, and **width2** are the names of the properties named width starting from the first one in the XSD going down, likewise with **high**, **high1**, **high2**.

These new property names are just the names used for reference and XPath and do not affect serialized content. The "true" names of each of these properties that appear in the serialized XML are the values given in the XSD. So for the XML instance:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:TieredGroup xsi:type="p:TieredGroup"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://TieredGroup">
  <width>foo</width>
  <high>bar</high>
</p:TieredGroup>
```

In order to access those properties you would use the following code:

```
DataObject tieredGroup = ...

// Displays "foo"
System.out.println(tieredGroup.get("width1"));

// Displays "bar"
System.out.println(tieredGroup.get("high2"));
```

Differentiating identically named properties

When multiple XSDs with the same namespace define the same named types, an incorrect type can be accidentally referenced.

Address1.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="city" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Address2.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="state" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Business objects do not support duplicate names for any global XSD structures (such as `complexType`, `simpleType`, `element`, `attribute`, and so on) through the `BOFactory.create()` APIs. These duplicate global structures can still be created as the child to other structures if the proper APIs are used, as shown in the following examples

Customer1.xsd:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://Customer1"
  targetNamespace="http://Customer1">
  <xsd:import schemaLocation="./Address1.xsd"/>
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="address" type="Address"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

Customer2.xsd:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://Customer2"
  targetNamespace="http://Customer2">
  <xsd:import schemaLocation="./Address2.xsd"/>
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="address" type="Address"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

When populating both of the Customer address fields and then calling `BOFactory.create()` to make the Address, the resulting child business object types can be incorrectly set. You can avoid this by calling the `createDataObject("address")` API on the Customer DataObject. This will be guaranteed to produce a child of the correct type because business objects will follow the import's schemaLocation.

```
DataObject customer1 = ...
```

```

// Incorrect way to create Address child
// This may create a type of Address1.xsd Address or maybe Address2.xsd Address
DataObject incorrect = boFactory.create("", "Address");
customer1.set("address", incorrect);

```

```

// Correct way to create Address child
// This is guaranteed to create a type of Address1.xsd Address
customer1.createDataObject("address");

```

Resolving property names that contain periods

Property names in an XSD may contain a period (".") as one of many valid characters, while in a SDO they are also used to show indexing in a property of multiple cardinality. This may cause resolution problems in certain situations.

Property names in Service Data Objects (SDOs) are based on the names of the elements and attribute that are generated from in the XSD. Business objects will handle the "." character properly, with one exception: if an XSD has a single cardinality property named "<name>.<#>" and a multiple cardinality property named "<name>".

An XPath such as "foo.0" would not resolve properly if there is a single cardinality property named "foo.0" and multiple cardinality property named "foo". In this case, the single cardinality Property named "foo.0" would be the one resolved. Although this should be a rare occurrence, you can avoid it entirely if you use the "foo[1]" syntax to access their multiple cardinality property. SDOs will not support the "." syntax for indexing, so you should use the "[]" for indexing.

Serializing and deserializing unions with `xsi:type`:

In XSD, a union is a way to merge the lexical spaces of several simple datatypes known as members.

The following example XSD shows a union that has the members of an integer and a date.

```
<xsd:simpleType name="integerOrDate">
  <xsd:union memberTypes="xsd:integer xsd:date"/>
</xsd:simpleType>
```

This multiple typing can cause confusion during deserialization and when manipulating the data.

Business objects support SDO's using `xsi:type` for serialization and will follow the same algorithm for determining the type on a deserialization if the `xsi:type` is not present in the XML data.

So to guarantee that the data (the number "42" in this example) would be deserialized as an integer, you can use the `xsi:type` specified in the input XML. You can also order the member list of the union in the XSD so that the integer comes before the string. The following example shows how both methods are implemented:

```
<integerOrString xsi:type="xsd:integer">42</integerOrString>

<xsd:simpleType name="integerOrString">
  <xsd:union memberTypes="xsd:integer xsd:string"/>
</xsd:simpleType>
```

Likewise, if the user wanted the data to be deserialized as a string, then either of the following changes would cause that behavior:

```
<integerOrString xsi:type="xsd:string">42</integerOrString>

<xsd:simpleType name="integerOrString">
  <xsd:union memberTypes="xsd:string xsd:integer"/>
</xsd:simpleType>
```

Note that if a string type is the first member of the union, it never has any information loss. It can also hold any data that will always be chosen by the no `xsi:type` algorithm. If you want to use a type other than string, you must either use `xsi:type` in the XML or reorder the member types in the XSD to give the other members a chance to accept the data.

Support for null business objects

This scenario involves an outside system communicating with WebSphere Process Server through XML wrapped inside of a SOAP message. When the enclosed element is nillible and has `xsi:nil="true"`, then the resulting DataObject which is created in WebSphere Process Server is null.

Here is an example which illustrates an XML message with a nillible element.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <p:Employee xmlns:p="http://www.mycompany.com" xmlns:xsi="http://www.w3.org"
      xsi:nil="true"/>
  </soap:Body>
</soap:Envelope>
```

Where Employee is defined as:

```
<element name="Employee" nillable="true">
...
</element>
```

The business object that is generated and sent from the export is null in this case. For example, if any downstream components have operations invoked on them, the input to that operation is null.

Note: These objects cannot be passed into business object maps because business object maps are unable to map fields from a null object.

Using the Sequence object to set data order

Some XSDs are defined in a way that makes the order that the data occurs in the XML have special significance.

One example of order significance in XSDs is mixed content. If the text data appears before or after an element, it may have different meaning than if it occurs in a different location. For these situations, SDO generates an object known as a Sequence, which is used to set the data in an ordered fashion.

SDO Sequences should not be confused with XSD sequences. XSD sequences are just model groups that are flattened out before SDO model generation. The presence of an XSD sequence does not relate to the presence of an SDO Sequence.

The following conditions in an XSD cause an SDO Sequence to be generated:

A complexType with mixed content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://MixedContent"
  targetNamespace="http://MixedContent">
  <xsd:complexType name="MixedContent" mixed="true">
    <xsd:sequence>
      <xsd:element name="element1" type="xsd:string" minOccurs="0"/>
      <xsd:element name="element2" type="xsd:string" minOccurs="0"/>
      <xsd:element name="element3" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="MixedContent" type="tns:MixedContent"/>
</xsd:schema>
```

A schema that has 1 or more <any/> tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://AnyElemAny"
  targetNamespace="http://AnyElemAny">
  <xsd:complexType name="AnyElemAny">
    <xsd:sequence>
      <xsd:any/>
      <xsd:element name="marker1" type="xsd:string"/>
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

A model group array (an all, choice, sequence, or group reference with maxOccurs > 1):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
```

```

<xsd:complexType name="ModelGroupArray">
  <xsd:sequence maxOccurs="3">
    <xsd:element name="element1" type="xsd:string"/>
    <xsd:element name="element2" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

An `<all/>` tag of `maxOccurs <= 1` that contains more than one element:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://A11">
  <xsd:complexType name="A11">
    <xsd:all>
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:schema>

```

Specific information about using `<any/>` and sequence together will be discussed in the topic listed at the bottom of this page.. The general information that follows in the remainder of this section will describe how to work with the other Sequence conditions, but will still apply to `<any/>` as well.

How do I know if my `DataObject` has a sequence?:

There are two simple APIs to choose from that can determine if a `DataObject` is sequenced: `DataObject noSequence` and `DataObject withSequence`.

You would use `DataObject noSequence` and `DataObject withSequence` as shown in the following example:

```

DataObject noSequence = ...
DataObject withSequence = ...

// Displays false
System.out.println(noSequence.getType().isSequenced());

// Displays true
System.out.println(withSequence.getType().isSequenced());

// Displays true
System.out.println(noSequence.getSequence() == null);

// Displays false
System.out.println(withSequence.getSequence() == null);

```

Why do I need to know a `DataObject` has a Sequence?:

If you are working on a `DataObject` that has a Sequence, it is important to know the order in which the data is set. Because of this, care must be taken in the order in which the values are set.

A `DataObject` that is not sequenced allows random order set access. This functions like a `Map` where all the keys are set to the same values. It does not matter in what order the keys were set, the data in the map is the same, and would be serialized to XML identically.

When a `DataObject` is sequenced, the order in which the data was set is recorded in the Sequence, much like adding data to a `List`. This provides two ways to access the data, by name/value pairs (the `DataObject` APIs) and by order in which it was

set (the Sequence APIs). You can use the DataObject set(...) or Sequence add(...) APIs to preserve the structure. This ordering affects the way that the XML is serialized.

Take for example, the <all/> tag XSD. When the set methods are called in the following order it produces the following XML when serialized:

```
DataObject all = ...
all.set("element1", "foo");
all.set("element2", "bar");

<?xml version="1.0" encoding="UTF-8"?>
<p:A11 xsi:type="p:A11"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:p="http://A11">
  <element1>foo</element1>
  <element2>bar</element2>
</p:A11>
```

If instead, the set methods are called in the opposite order, then the following XML is produced when the business object is serialized:

```
DataObject all = ...
all.set("element2", "bar");
all.set("element1", "foo");

<?xml version="1.0" encoding="UTF-8"?>
<p:A11 xsi:type="p:A11"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:p="http://A11">
  <element2>bar</element2>
  <element1>foo</element1>
</p:A11>
```

If the order of the Sequence is ever changed, then the Sequence class has basic add, remove, and move methods to allow the user to alter the order of Sequence.

How do I work with mixed content?:

For mixed content, Sequence has a specific API for adding text: addText(...).

All other APIs work equally with text as they do with Properties. The getProperty(int) API will return null for mixed content text data. The following example of mixed content code can be used to print all the mixed content text from a DataObject:

```
DataObject mixedContent = ...
Sequence seq = mixedContent.getSequence();

for (int i=0; i < seq.size(); i++)
{
    Property prop = seq.getProperty(i);
    Object value = seq.getValue(i);

    if (prop == null)
    {
        System.out.println("Found mixed content text: "+value);
    }
    else
    {
        System.out.println("Found Property "+prop.getName()+": "+value);
    }
}
```

How do I work with a model group array?:

A model group array is created when a model group has a value for maxOccurs > 1.

Since model groups are flattened and not expressed in a DataObject, the properties inside of the model group become multiple cardinality properties so that their isMany() methods return true if they are not already true. Their minOccurs and maxOccurs facets become multiplied by that of the containing model group. Choice will multiply the maxOccurs facet in the same way as the other model groups, but will always use 0 as the multiplication value for minOccurs because any data in a choice may not be selected.

For example, the following XSD has a model group array:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:sequence minOccurs="2" maxOccurs="5">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"
        minOccurs="0" maxOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

As stated, **element1** and **element2** will now be multiple cardinality so that a get(...) accessor would return a List. **Element1** has a default minOccurs of 1 and a default maxOccurs of 1. **Element2** has a minOccurs of 0 and a maxOccurs of 3. In the following example, their new minOccurs and maxOccurs will be as follows:

```
DataObject - ModelGroupArray
  Property[0] - element1 - minOccurs=(2*1)=2 - maxOccurs=(5*1)=5
  Property[1] - element2 - minOccurs=(2*0)=0 - maxOccurs=(5*3)=15
```

If the type were Choice:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:choice minOccurs="2" maxOccurs="5">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"
        minOccurs="0" maxOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Would generate the following minOccurs due to the exclusion of choice that only **element1** can be picked each time or only **element2** could be picked each time, so to pass validation both need to be able to have 0 occurrences:

```
DataObject - ModelGroupArray
  Property[0] - element1 - minOccurs=(0*1)=0 - maxOccurs=(5*1)=5
  Property[1] - element2 - minOccurs=(0*0)=0 - maxOccurs=(5*3)=15
```

Using Any data types

This section provides programming techniques for using Any data types.

Using AnySimpleType for simple types:

AnySimpleType is handled no differently from any other simple type (string, int, boolean, and so on) by the SDO APIs.

The only differences between `anySimpleType` and the other simple types are in its instance data and serialization/deserialization. These should be internal concepts to business object only, and used to determine if data being mapped to or from the field is valid. If a string type were to have a `set(...)` method called on it, the data would first be converted to a string and the original data type would be lost:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://StringType">
  <xsd:complexType name="StringType">
    <xsd:sequence>
      <xsd:element name="foo" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

DataObject stringType = ...

// Set the data to a String
stringType.set("foo", "bar");

// The instance data will always be type String, regardless of the data set
// Displays "java.lang.String"
System.out.println(stringType.get("foo").getClass().getName());

// Set the data to an Integer
stringType.set("foo", new Integer(42));

// The instance data will always be type String, regardless of the data set
// Displays "java.lang.String"
System.out.println(stringType.get("foo").getClass().getName());
```

An `anySimpleType` instead does not lose the original data type of what is being set:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://AnySimpleType">
  <xsd:complexType name="AnySimpleType">
    <xsd:sequence>
      <xsd:element name="foo" type="xsd:anySimpleType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

DataObject anySimpleType = ...

// Set the data to a String
stringType.set("foo", "bar");

// The instance data will always be of the type of data used in the set
// Displays "java.lang.String"
System.out.println(stringType.get("foo").getClass().getName());

// Set the data to an Integer
stringType.set("foo", new Integer(42));

// The instance data will always be of the type of data used in the set
// Displays "java.lang.Integer"
System.out.println(stringType.get("foo").getClass().getName());
```

This data type is also preserved across serialization and deserialization by `xsi:type`. Consequently, any time you serialize an `anySimpleType` element, it will have an `xsi:type` that matches that defined in the SDO specification based on its Java type:

In the following example, you serialize the business object above so that the data would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:StringType xsi:type="p:StringType"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:p="http://StringType">
  <foo xsi:type="xsd:int">42</foo>
</p:StringType></p:StringType>
```

The `xsi:type` will be used during deserialization to load the data as the appropriate Java instance class. If no `xsi:type` is specified, then the default deserialization type will be string.

For the other simple types, determining mappability is a constant. For instance, A boolean can always map to a string. `AnySimpleType` can contain any of the simple types, however, so a mapping may or may not be possible based on the instance data in the field.

Use the property `Type's URI and Name` to determine if a property is of type `anySimpleType`. They will be "commonj.sdo" and "Object". To determine if data is valid to be inserted into `anySimpleType`, check to see if it is not an instance of a `DataObject`. All data that can be represented as a `String` and is not a `DataObject` is allowed to be set into an `anySimpleType` field.

This leads to the following mapping rules:

- `anySimpleType` can always map to `anySimpleType`.
- any other simple type can always map to `anySimpleType`.
- `anySimpleType` can always map to string because all simple types must be able to be converted into a string.
- `anySimpleType` may or may not be able to map to any of the other simple types depending on its value in the business object. This means that this mapping cannot be determined at design time, only at runtime.

Related information

 [Assigning from and to xs:any](#)

Using AnyType for complex types:

The `anyType` tag is not handled differently from any other complex type by the SDO APIs.

The only differences between `anyType` and any other complex types are in its instance data and serialization/deserialization, which should be internal concepts to business object only, and determining if data being mapped to or from the field is valid. Complex types are limited to a single type: Customer, Address, and so on. The `anyType`, however, allows any `DataObject` regardless of type. If `maxOccurs > 1`, then each `DataObject` in the list can of a different type.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://AnyType">
  <xsd:complexType name="AnyType">
    <xsd:sequence>
      <xsd:element name="person" type="xsd:anyType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://Customer">
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://Employee" targetNamespace="http://Employee">
  <xsd:complexType name="Employee">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

DataObject anyType = ...
DataObject customer = ...
DataObject employee = ...

// Set the person to a Customer
anyType.set("person", customer);

// The instance data will be a Customer
// Displays "Customer"
System.out.println(anyType.getDataObject("person").getName());

// Set the person to an Employee
anyType.set("person", employee);

// The instance data will be an Employee
// Displays "Employee"
System.out.println(anyType.getDataObject("person").getName());

```

Just like `anySimpleType`, `anyType` uses `xsi:type` during serialization to assure that the intended type of `DataObject` is maintained when deserialized. So when you set to "Customer," the XML would look like:

```

<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:customer="http://Customer"
  xmlns:p="http://AnyType">
  <person xsi:type="customer:Customer">
    <name>foo</name>
  </person>
</p:AnyType>

```

And when set to "Employee":

```

<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:employee="http://Employee"
  xmlns:p="http://AnyType">
  <person xsi:type="employee:Employee">
    <id>foo</id>
  </person>
</p:AnyType>

```

`AnyType` also allows for the setting of simple type values through wrapper `DataObjects`. These wrapper `DataObjects` have a single property named "value"

(element) that holds the simple type value. The SDO APIs have been overridden to automatically wrap and unwrap these simple types and wrapper DataObjects when using the get<Type>/set<Type> APIs. The non-type casting get/set APIs will not perform this wrapping.

```
DataObject anyType = ...

// Calling a set<Type> API on an anyType Property causes automatic
// creation of a wrapper DataObject
anyType.setString("person", "foo");

// The regular get/set APIs are not overridden, so they will return
// the wrapper DataObject
DataObject wrapped = anyType.get("person");

// The wrapped DataObject will have the "value" Property
// Displays "foo"
System.out.println(wrapped.getString("value"));

// The get<Type> API will automatically unwrap the DataObject
// Displays "foo"
System.out.println(anyType.getString("person"));
```

When the wrapper DataObject is serialized, it will be serialized just like anySimpleType mapping of Java instance classes to XSD types in the xsi:type field. So this setting would serialize as:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:p="http://AnyType">
  <person xsi:type="xsd:string">foo</person>
</p:AnyType>
```

If no xsi:type is given or if an incorrect xsi:type is given, then an exception will be thrown. In addition to automatic wrapping, the wrapper can be manually created for use with the set() API through BOFactory createDataTypeWrapper(Type, Object) where Type is the SDO simple type of the data to be wrapped and Object is the data to be wrapped.

```
Type stringType = boType.getType("http://www.w3.org/2001/XMLSchema", "string");
DataObject stringType = boFactory.createByMessage(stringType, "foo");
```

To determine if a DataObject is a wrapper type, the BOType isDataTypeWrapper(Type) can be called.

```
DataObject stringType = ...
boolean isWrapper = boType.isDataTypeWrapper(stringType.getType());
```

For the other complex types, in order to move data from one field to another, the data must be of the same type. AnyType can contain any complex types, however, so a direct move with no mapping may or may not be possible based on the instance data in the field.

You can use the property Type's URI and Name to determine if a property is of type anyType. They will be "commonj.sdo" and "DataObject". All data is valid to be inserted into an anyType. This leads to the following mapping rules:

- anyType can always map to anyType.
- any complex type can always map to anyType.
- any simple type can always map to anyType.

- anyType may or may not be able to map to any of the other simple or complex types depending on its value in the BO instance. This means that this mapping cannot be determined at design time, only at runtime.

Using Any to set global elements for complex types:

You can use the <any/> tag to set global elements to a complex type.

An occurrence of the any tag makes the DataObject Type isOpen() method and the isSequenced() method return true. If the value for maxOccurs is > 1 on an any tag, it has no effect on the structure of the DataObject; it is only used as information during validation. Similarly, the occurrence of multiple any tags in a type does not change the structure of the DataObject; they are used only for validating the location of open data that was set.

How do I know if my DataObject has an any tag?:

You can easily determine if instances of a DataObject have any values set within them by checking the instance properties to see if any of the open properties are attributes.

DataObject does not provide a mechanism for determining if a DataObject Type has an any tag. DataObjects only have the concept of "open" that applies to both any and anyAttribute and allows the free additional of any properties. While the presence of an any tag causes a DataObject to have isOpen() = true and isSequenced() = true, it might just have an anyAttribute tag and one of the reasons for being sequenced discussed in the Sequences topics. The following example demonstrates these concepts:

```
DataObject dobj = ...

// Check to see if the type is open, if it isn't then it can't have
// any values set in it.
boolean isOpen = dobj.getType().isOpen();

if (!isOpen) return false; // Does not have any values set

// Open Properties are added to the Instance Property list, but not
// the Property list, so comparing their sizes can easily determine
// if any open data is set
int instancePropertyCount = dobj.getInstanceProperties().size();
int definedPropertyCount = dobj.getType().getProperties().size();

// If equal, does not have any open content set
if (instancePropertyCount == definedPropertyCount) return false;

// Check the open content Properties to determine if any are Elements
for (int i=definedPropertyCount; i < instancePropertyCount; i++)
{
    Property prop = (Property)dobj.getInstanceProperties().get(i);
    if (boXsdHelper.isElement(prop))
    {
        return true; // Found an any value
    }
}

return false; // Does not have any values set
```

How do I get/set any values?:

Performing a get on data that was set in an any field can be done in the same manner as any other element value if the name is known.

You can perform a get with the XPath "<name>" and it will be resolved. If the name is unknown, then the value can be found by checking the instance properties as in above. If there are multiple any tags, or an any tag with maxOccurs > 1, then the DataObject sequence will have to be used instead if it is important to determine which any tag the data originated from.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://AnyElemAny"
  targetNamespace="http://AnyElemAny">
  <xsd:complexType name="AnyElemAny">
    <xsd:sequence>
      <!-- Handle all these any one way -->
      <xsd:any maxOccurs="3"/>
      <xsd:element name="marker1" type="xsd:string"/>
      <!-- Handle this any in another -->
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Because the <any/> tag causes the DataObject to be sequenced determining which any value was set can be done by checking the Sequence for the position of the any properties.

You can determine which any tag the instance data for the following XSD belongs to by using the following code:

```
DataObject anyElemAny = ...
Sequence seq = anyElemAny.getSequence();

// Until we encounter the marker1 element, all the open data
// found belongs to the first any tag
boolean foundMarker1 = false;

for (int i=0; i<seq.size(); i++)
{
  Property prop = seq.getProperty(i);

  // Check to see if the property is an open property
  if (prop.isOpenContent())
  {
    if (!foundMarker1)
    {
      // Must be the first any because it occurs
      // before the marker1 element
      System.out.println("Found first any data: "+seq.getValue(i));
    }
    else
    {
      // Must be the second any because it occurs
      // after the marker1 element
      System.out.println("Found second any data: "+seq.getValue(i));
    }
  }
  else
  {
    // Must be the marker1 element
    System.out.println("Found marker1 data: "+seq.getValue(i));
    foundMarker1 = true;
  }
}
```

Setting an <any/> value is done by creating a global element property and adding that value to the sequence.


```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://GlobalElems"
  targetNamespace="http://GlobalElems">
  <xsd:element name="globalElement1" type="xsd:string"/>
  <xsd:element name="globalElement2" type="xsd:string"/>
</xsd:schema>

DataObject anyElemAny = ...
Sequence seq = anyElemAny.getSequence();

// Get the global element Property for globalElement1
Property globalProp1 = boXsdHelper.getGlobalProperty(http://GlobalElems,
"globalElement1", true);

// Get the global element Property for globalElement2
Property globalProp2 = boXsdHelper.getGlobalProperty(http://GlobalElems,
"globalElement2", true);

// Add the data to the sequence for the first any
seq.add(globalProp1, "foo");
seq.add(globalProp1, "bar");

// Add the data for the marker1
seq.add("marker1", "separator"); // or anyElemAny.set("marker1", "separator")

// Add the data to the sequence for the second any
seq.add(globalProp2, "baz");

// The data can now be accessed by a get call
System.out.println(dobj.get("globalElement1"); // Displays "[foo, bar]"
System.out.println(dobj.get("marker1"); // Displays "separator"
System.out.println(dobj.get("globalElement2"); // Displays "baz"

```

What are valid mappings for data in an any?:

An `<any/>` tag is a set of name/value pairs. The only valid mapping that can be determined at design time for `<any/>` is another `<any/>` or `anyType` that has the same `maxOccurs` value.

Individually, the values contained in an instance of a `DataObject` for any are basic complex types that follow all the rules of complex type mapping. Some of these complex types may be wrapped simple types, so these will follow the rules of simple type mapping.

Using AnyAttribute to set global attributes for complex types:

The `<anyAttribute/>` tag allows a complex type to have any number of global attributes set to it.

Similar to the `<any/>` tag, the occurrence of the `<anyAttribute/>` tag makes the `DataObject` `Type.isOpen()` method return true. Unlike the `<any/>` tag, however, the `<anyAttribute/>` tag does not cause a `DataObject` to be sequenced because attributes in XSD are not ordered constructs.

How do I tell if my DataObject has an anyAttribute tag?:

You can easily determine if instances of a `DataObject` have any `AnyAttribute` values set within them by checking the instance properties to see if any of the open properties are attributes.

DataObject does not provide a mechanism for determining if a DataObject Type has an anyAttribute tag. DataObjects only have the concept of "open" that applies to both any and <anyAttribute/> and allows the free additional of any Properties. While it is true that if a DataObject has isOpen() = true and isSequenced() = false, then it must have an anyAttribute tag, if isOpen() = true and isSequenced() = true, the DataObject Type might or might not have an anyAttribute tag.

DataObject provides metadata query methods to programmatically answer this and other questions about the XSD structure that was used to generate the DataObject. The InfoSet model can be queried if it is necessary to know if the anyAttribute tag is present. Because anyAttribute is singular and either is or is not true, business objects will also provide a BOXSDHelper hasAnyAttribute(Type) method to allow determination as to if setting an open attribute on this DataObject would produce a valid result. The following example code demonstrates these concepts:

```
DataObject dobj = ...

// Check to see if the type is open, if it isn't then it can't have
// anyAttribute values set in it.
boolean isOpen = dobj.getType().isOpen();

if (!isOpen) return false; // Does not have anyAttribute values set

// Open Properties are added to the Instance Property list, but not
// the Property list, so comparing their sizes can easily determine
// if any open data is set
int instancePropertyCount = dobj.getInstanceProperties().size();
int definedPropertyCount = dobj.getType().getProperties().size();

// If equal, does not have any open content set
if (instancePropertyCount == definedPropertyCount) return false;

// Check the open content Properties to determine if any are Attributes
for (int i=definedPropertyCount; i<instancePropertyCount; i++)
{
    Property prop = (Property)dobj.getInstanceProperties().get(i);
    if (boxsdHelper.isAttribute(prop))
    {
        return true; // Found an anyAttribute value
    }
}

return false; // Does not have anyAttribute values set
```

How do I get/set anyAttribute values?:

Setting an <anyAttribute/> value is done in the same way as setting an <any/>, but instead of a global element a global attribute is used.

Performing a get on data that was set in an anyAttribute field can be done in the same manner as any other attribute value if the name is known. You can perform a get with the XPath "@<name>" and it will be resolved. If the name is unknown, using the above code the values can be iterated and accessed one by one. The example code below shows this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://AnyAttrOnlyMixed"
  targetNamespace="http://AnyAttrOnly">
  <xsd:complexType name="AnyAttrOnly">
    <xsd:sequence>
      <xsd:element name="element" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

    </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://GlobalAttrs">
  <xsd:attribute name="globalAttribute" type="xsd:string"/>
</xsd:schema>

DataObject dobj = ...

// Get the global attribute Property that is going to be set
Property globalProp = boXsdHelper.getGlobalProperty(http://GlobalAttrs,
"globalAttribute", false);

// Set the value on the dobj, just like any other data
dobj.set(globalProp, "foo");

// The data can now be accessed by a get call
System.out.println(dobj.get("@globalAttribute")); // Displays "foo"

```

What are valid mappings for data in an anyAttribute?:

The AnyAttribute tag is similar to the any tag, and is a set of name/value pairs. Consequently, the only valid mapping for anyAttribute is another anyAttribute.

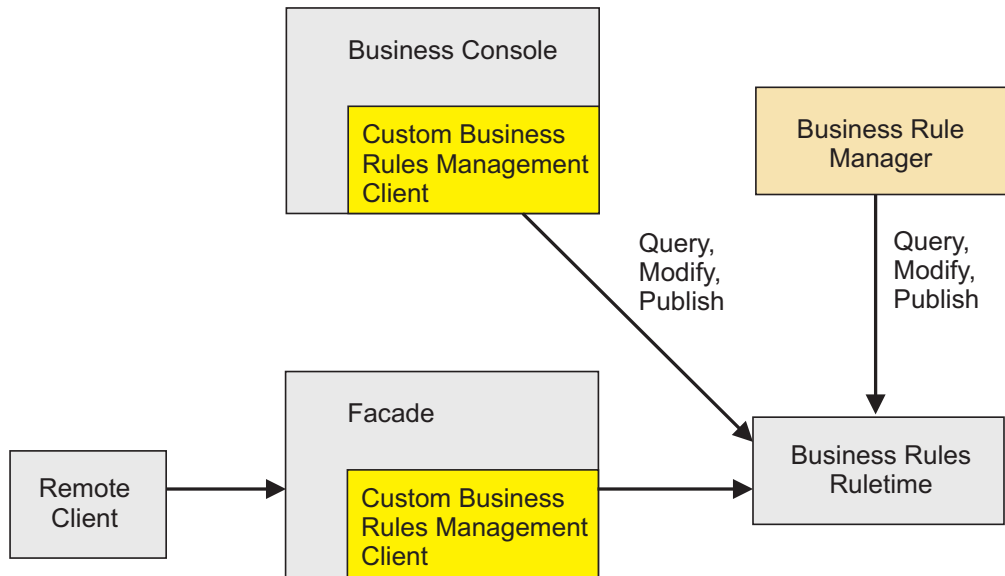
Individually, the values contained in the anyAttribute data are basic simple types that follow all the rules of simple type mapping

Business rule management programming

Public business rule management classes are provided for the building of custom management clients or to automate changes to business rules.

Business rules management classes could be used in a Web application where they are combined with other management capabilities for things such as business process or human tasks in order to manage all components from a single client. Any custom management clients can be used along side the Business Rule Manager Web application included with WebSphere Process Server. The classes could also be used to automate changes to business rules within an application. For example business rules could be changed as the results of a business process that is using the business rules exceeds some threshold or limit.

The business rule management classes must be used in an application installed on WebSphere Process Server. The classes do not provide a remote interface, however they can be wrapped in a facade which is then exposed over a specific protocol for remote execution.



This programming guide is composed of two main sections and an appendix. The first section explains the programming model and how to use the different classes. Class diagrams are provided to show the relationship between classes. The second section provides examples on using the classes to perform actions such as searching for business rule groups, scheduling a new rule destination, and modifying a rule set or decision table. The appendix contains additional classes that were used in the examples to simplify common operations and additional examples of creating complex queries for searching for business rule groups using wildcards.

Besides this programming guide information on the classes is also available in Javadoc HTML format included with both the WebSphere Process Server v6.1 and test environment included with WebSphere Integration Developer v6.1. This Javadoc documentation is available at `${WebSphere Process Server Install Directory}\web\apidocs` or `${WebSphere Integration Developer Install Directory}\runtimes\bi_v61\web\apidocs`. The packages `com.ibm.wbiserver.brules.mgmt.*` contain all of the information.

Programming model

WebSphere Business Integration Business Rules are authored with two different authoring tools and issued by the rule runtime. All three share the same model for the business rule artifacts.

Sharing of the model was deemed critical for not only ease of future maintenance, but for a consistent programming model for the end user. Sharing this model required compromises between the needs of desktop tooling and runtime execution and authoring -- all have clear sets of requirements to meet for their respective environments and these requirements at times conflicted with each other. The artifacts described below as part of the overall programming model represent a balance in meeting the requirements of these different environments.

Modification of business rules is limited to only those items that are defined with templates in the rule sets and decision tables as well as the operation selection table (effective dates and targets). Creation of new rule sets and decision tables is only supported through the copy of an existing rule set or decision table. The business rule group component itself is not eligible for dynamic authoring in the

runtime with the exception of the user defined properties and description values. Changes that need to be made to the component (for example, adding a new operation) must be done using WebSphere Integration Developer and then redeployed or reinstalled in the server.

Business Rule Group

The `BusinessRuleGroup` class represents the business rule group component. The `BusinessRuleGroup` class can be considered the root object which contains rule sets and decision tables.

Rule sets and decision tables can only be reached through the business rule group that they are associated with. Methods are provided on the class to retrieve information about the business rule group and to reach the rule sets and decision tables. Through the methods the following information can be retrieved:

- Target name space
- Name of business rule group
- Display name
- Name/Display name synchronization
- Description
- Presentation time zone which indicates whether dates should be displayed in UTC format or local to the system
- Operations defined in the interface associated with the business rule group
- Custom properties defined on the business rule group

The different rule sets and decision tables associated with the business rule group can be reached through the business rule group's operation.

There are also methods that allow for information to be updated on the business rule group. Through the methods, the following information can be updated:

- Description
- Display name
- Name/Display name synchronization
- Custom properties defined on the business rule group

The Display name for the business rule group can be set explicitly or it can be set to the value of the Name using the `setDisplayNamesIsSynchronizedToName` method.

Other values cannot be modified as these are part of the business rule group component definition and changes to these values would require a redeploy as well as reinstallation.

The business rule group class also provides a refresh method. This method will make a call to the persistent storage or repository where the business rules are stored and return the business rule group and all of the associated rule sets and decision tables with the persisted information. The returned business rule group is the latest copy and the previous object is obsolete.

The `isShell` method can be used to tell if a business rule group instance is of a version that is not supported by the current runtime. For example, if a web client was created with the current business rule management classes, and in the future new capabilities are added to the business rule group that are not supported by the classes, a shell business rule group will be created when the business rule

group is retrieved. This allows the web client to continue to work with business rules that are supported and still retrieve business rule groups with limited attributes and capabilities. When `isShell` is true, only the methods `getName`, `getTargetNameSpace`, `getProperties`, `getPropertyValue`, and `getProperty` will return values. All other methods will result in an `UnsupportedOperationException`. Besides using the `isShell` method, the type of the `BusinessRuleGroup` can also be checked if it is an instance of `BusinessRuleGroupShell` in order to determine if it is of a supported version.

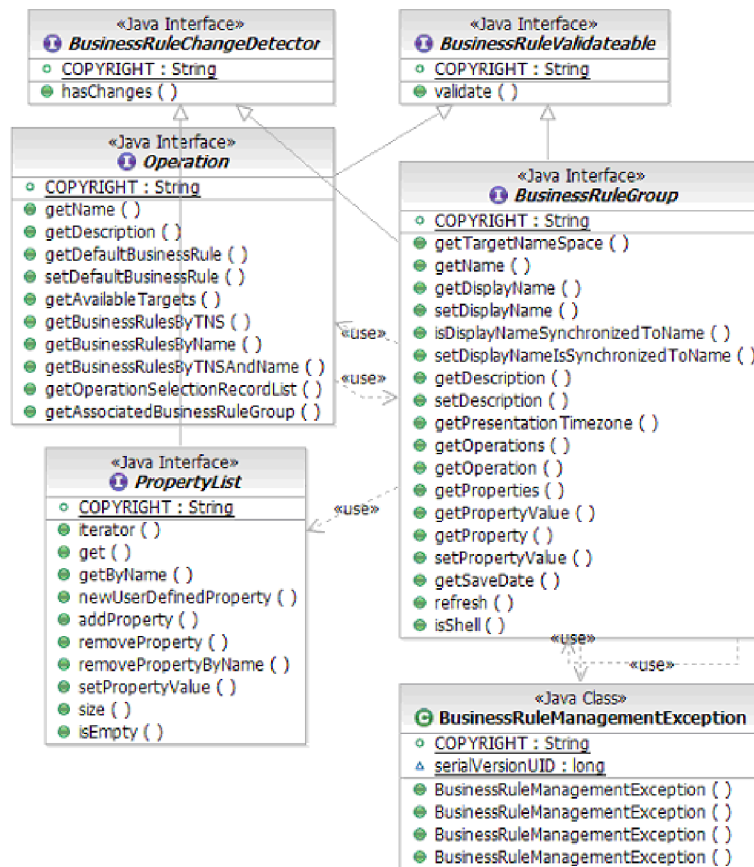


Figure 59. Class diagram of `BusinessRuleGroup` and related classes

Business Rule Group Properties

The properties on business rule groups are intended to be used for management of business rule groups. Properties set on business rule groups can be used in queries to return only a subset of business rule groups which are to be displayed and then modified.

All properties are of type string and defined as name-value pairs. Each property can only be defined once in a business rule group. For each property defined, it must have a value also defined. The property value can be an empty string or zero in length, but not null. Setting a property to null is the same as deleting the property.

The properties on a business rule group can also be accessed in a rule set or decision table at runtime. This allows a single value to be set at the business rule

group to be used within multiple rule sets or decision tables in the business rule group. Only those properties defined on the business rule group are available to enclosed rule sets and decision tables.

There are two types of properties, system and user-defined. The number of system or user-defined properties is not limited on a business rule group. The system properties are used to hold specific component information such as the version of the rule model used in defining the rule logic. This system information is exposed in properties to allow for query across these fields. The system properties begin with a prefix `IBMSystem` and are read-only through the business rule group and property classes. System properties can not be added, changed or deleted. An example of a system property is:

Property Name	Property Value
<code>IBMSystemVersion</code>	6.2.0

Note: The values of name, namespace and display name for a business rule group are treated as system properties for query purposes, and will be part of the list of properties that can be retrieved for a business rule group with the `getProperties` method. These properties are not however, defined as actual property elements in the business rule group artifact and are not seen as properties in WebSphere Integration Developer as they are defined with separate and unique elements on the business rule group. They are solely provided to offer more query options.

User-defined properties are available to be used for holding any customer specific information and can also be used in queries for business rule groups. User-defined properties are available for read-write.

The properties for a business rule group can be retrieved either individually or as a list (`PropertyList` object). With the `PropertyList`, methods are provided for retrieving individual properties and adding and removing user-defined properties.

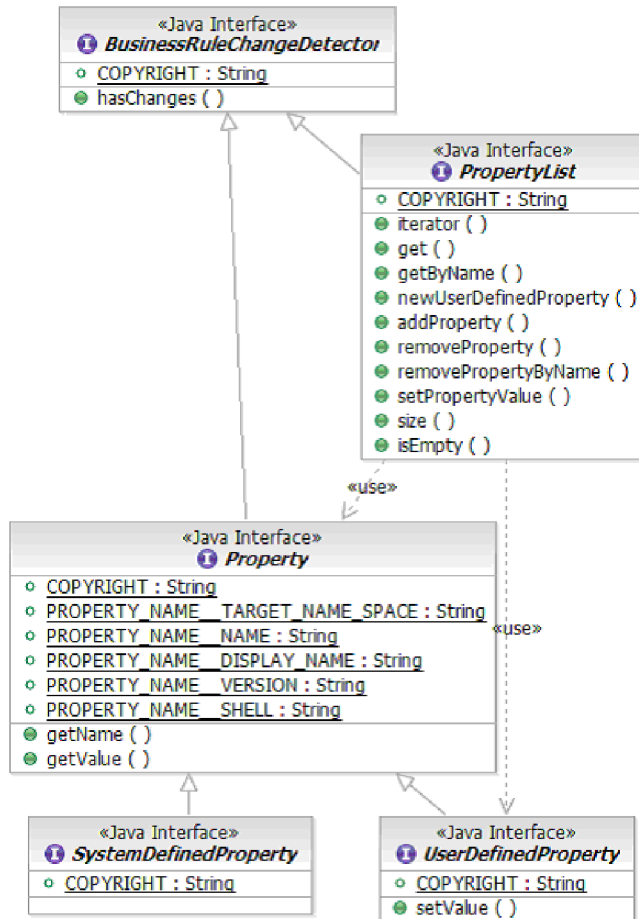


Figure 60. Class diagram of Property and related classes

Operation

Operations are starting points for reaching individual rule sets and decision tables to modify. The operations of a business rule group match the operations listed in the WSDL which is associated with the business rule group component.

For each operation, there are different targets, each of which is a business rule (rule set or decision table):

- Default target (optional)
- List of targets scheduled by date/time ranges (OperationSelectionRecord)
- List of all available targets that can be used for that operation

Each operation must have at least one business rule target specified. This target can be an OperationSelectionRecord with a specific start date and end date when the target should be scheduled to be active. The operation can also have a single default target set which is used during execution when no matching scheduled business rule target is found. The Operation class provides methods for retrieving and setting the default business rule target as well as retrieving the list (OperationSelectionRecordList) of scheduled business rule targets. Besides the default business rule target and the scheduled business rule targets, there is a list of all available business rule targets for the operation. This list will include those business rules targets which are scheduled and the default business rule target as

well as any other rule sets or decision tables which are not scheduled for this operation. An unscheduled rule set or decision table is associated with the operation through the available target list by the fact that it implicitly shares the operation information. All business rule targets must support the input and output messages for their operation. With each operation unique on an interface, the rule sets and decision tables for an operation are unique from those rule sets and decision tables of another operation.

Any of the different rule sets and decision tables in the available targets list can be scheduled to be active through the creation of an `OperationSelectionRecord`. Along with the particular rule set or decision table from the available targets list, a start date and end date must be specified. The start date must be before the end date. The dates can be for a time which covers the current date as well as the past and the future. The time span of the dates cannot overlap with any other `OperationSelectionRecords` once it is added to the `OperationSelectionRecordList` and published. The start date and end date values are of type `java.util.Date`. Any values which are specified will be treated as UTC values according to the `java.util.Date` class. With the `OperationSelectionRecord` complete, it can be added to the `OperationSelectionRecordList` to be scheduled along with other business rule targets. Gaps may exist between the time spans of different `OperationSelectionRecords`. When a gap is encountered during execution, the default target is used. If no default target has been specified, an exception will be thrown. It is recommended to always specify a default business rule target.

A scheduled business rule target can be removed from the list of scheduled targets by removing the `OperationSelectionRecord` from the `OperationSelectionRecordList`. Removing an `OperationSelectionRecord` will not remove the business rule target from the list of available business rule targets and it will not remove any other `OperationSelectionRecords` which have the same business rule target scheduled.

Besides retrieving a rule set or decision table through the `OperationSelectionRecordList` or list of available targets, the `Operation` class also allows for business rule targets to be retrieved by name and target namespace property values. Through the methods on the `Operation` class, those rule sets and decision tables which are listed in the available targets for that operation can be queried. Rule sets and decision tables which might have matching name and target namespace values, but are part of the available target lists of other operations, will not be included in the result set. As a convenience, the `getBusinessRulesByName`, `getBusinessRulesByTNS`, and `getBusinessRulesByTNSAndName` methods are provided to simplify retrieving specific rule sets and decision tables.

The `Operation` class provides methods that support the following:

- Retrieve the operation name
- Retrieve the operation description
- Retrieve and set the default business rule target
- Retrieve the scheduled business rule targets (`OperationSelectionRecordList`)
- Retrieve the list of all available business rule targets
- Retrieve a rule set or decision table from the list of all available targets by name or target namespace
- Retrieve the business rule group with which the operation is associated

The `OperationSelectionRecordList` class provides methods that support the following:

- Retrieve a specific `OperationSelectionRecord` by index value

- Remove a specific OperationSelectionRecord by index value
- Add a new OperationSelectionRecord to the list

The OperationSelectionRecord class provides methods that support the following:

- Retrieve and set the start date
- Retrieve and set the end date
- Retrieve and set the business rule target
- Retrieve the operation with which the OperationSelectionRecord is associated

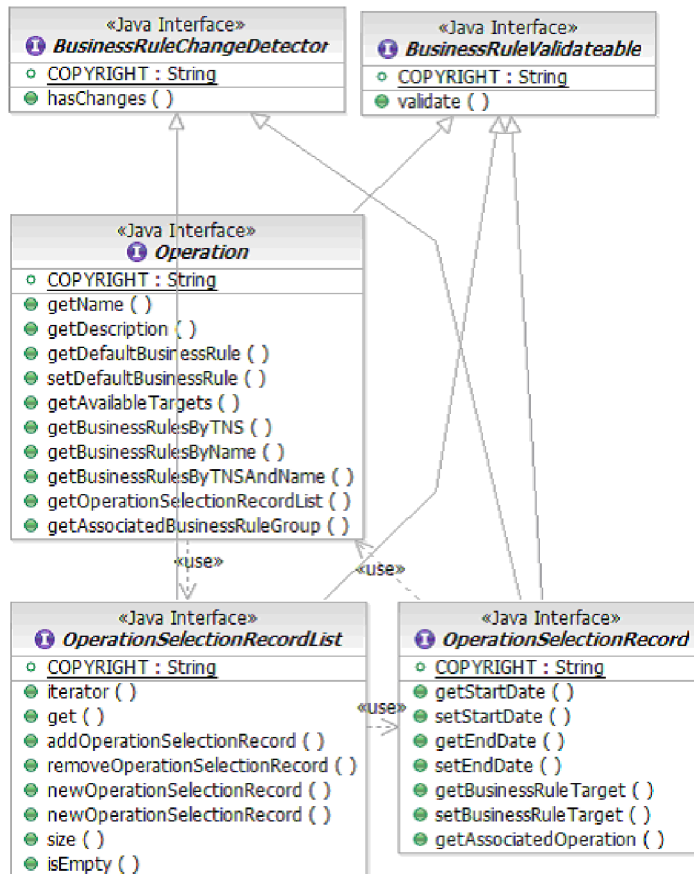


Figure 61. Class diagram for Operation and related classes

Business Rule

The RuleSet and DecisionTable classes are based off a generic BusinessRule class with methods that provide information that is available on both rule sets and decision tables.

Similar to business rule group artifacts, rule sets and decision tables have a name and a target namespace. The combination of these values must be unique when compared to other rule sets and decision tables. For example, two rule sets can share the same target namespace value, but must have different names or a rule set and decision table could have the same name, but have different target namespace values.

A copy of a business rule can be made from an existing business rule for situations where a similar rule is required to be scheduled at a specific time with different

parameter values for rules constructed from templates. New rules cannot be fully created from scratch as there must be a backing Java class to provide the implementation for the business rule. The backing Java class is only created at deploy time. When a new rule is created, it is added to the list of available targets for the operation which is associated with the original rule. The additional rule is not persisted however, until the business rule group with which the operation is associated is published.

The new business rule must have either a different target namespace or name from the original rule. The display name for the new business rule can remain the same as the original rule as the combination of the name and namespace provide a key value to identify the business rule. Within the business rule, the different parameter values which have been defined with a template can be modified. Scheduling the business rule at a certain time can be done with the `OperationSelectionRecordList` or as a default destination with the `Operation` associated with the business rule.

The `BusinessRule` class provides methods that support the following:

- Retrieve the target namespace
- Retrieve the name of the rule set or decision table
- Retrieve and set the display name of the rule set or decision table
- Retrieve the type of the business rule, either rule set or decision table
- Retrieve and set the description for the business rule
- Retrieve the operation that the business rule is associated with.
- Create a copy of the business rule with a different name and/or target namespace

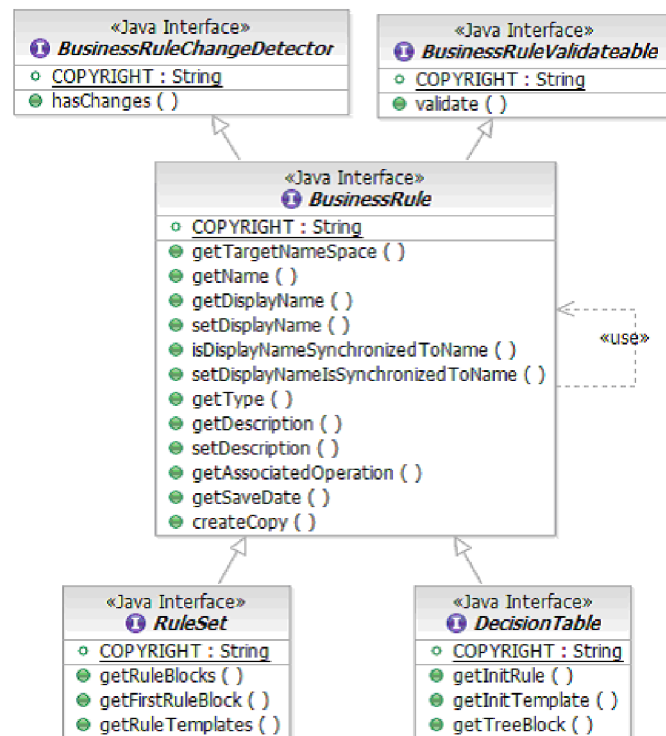


Figure 62. Class diagram of `BusinessRule` and related classes

Rule set

A rule set is one type of business rule. Rule sets are typically used when multiple rules may need to be executed based on different conditional values. Rule sets are composed of a rule block and rule templates. The rule block (`RuleBlock`) contains the different if-then and action rules which make up the logic of the rule set.

The `RuleSet` class provides methods that support the following:

- Retrieve a list of rule blocks for the rule set
- Retrieve a list of rule templates defined in the rules set

Currently each rule set can only have one rule block, while there can be multiple rule templates defined in the rule set. The rule block contains the set of rules that will be executed when the rule set is invoked. The rule block allows for the order of the rules to be modified. A rule block must have at least one rule defined. The rules (`Rule`) can be defined as template instance rules (`TemplateInstanceRule`) or hard-coded. If an if-then or action rule has been defined with a template, it can be removed from the rule block. If a new instance of rule was created with a template, it can be added to the rule block.

If a rule is hard-coded and was not defined with a template, it cannot be modified or removed from the rule block. The expectation with these rules is that they have been designed to always be part of the rule set logic and are not to be changed or repeated within the logic.

When a new rule is created with a template, it must have a unique name value. The list of existing rules can be retrieved and checked first before creating the rule.

For hard-coded if-then and action rules, only the name and presentation can be retrieved. The presentation is a string which can be used to display information about the rule in client applications. For if-then or action rules that are defined with a template, the name and presentation can be retrieved as well as additional information. Specific parameter values can be retrieved and changed. With a template (`RuleSetRuleTemplate`) defined in the rule set, another instance of the rule can be created within the rule set and parameter values can be set. For example, if you have a rule saying that a customer of a particular status level receives a discount of a specific amount. This logic could be defined with a single rule template and then repeated with parameter values changed for the status level (gold, silver, bronze, and so on) and the discount amount (15%, 10%, 5%, and so on).

The parameters for a rule defined with a template are specific to the instance of the rule. The template only defines a standard presentation and the number of parameters for the rule. Each rule defined with a template can have different values as explained in the example on discounts for different customer status.

The `RuleBlock` class provides methods that support the following:

- Retrieve a rule by index
- Add a rule that was defined with a template
- Remove a rule defined with a template
- Modify the order of a rule by one place or to a specific index location

The `RuleSetRule` class provides methods that support the following:

- Retrieve the name of the rule
- Retrieve the display name of the rule

- Retrieve the user presentation
- Retrieve the rule block

The RuleSetRuleTemplate class provides methods that support the following:

- Create a rule template instance from this template definition
- Retrieve the parent rule set

The TemplateInstanceRule class provides methods that support the following:

- Retrieve the parameters for the rule
- Retrieve the template definition which defined the rule

The Template class provides methods that support the following:

- Retrieve the template ID
- Retrieve the name
- Retrieve and set the display name
- Retrieve and set the description
- Retrieve the parameters for this template
- Retrieve the user presentation

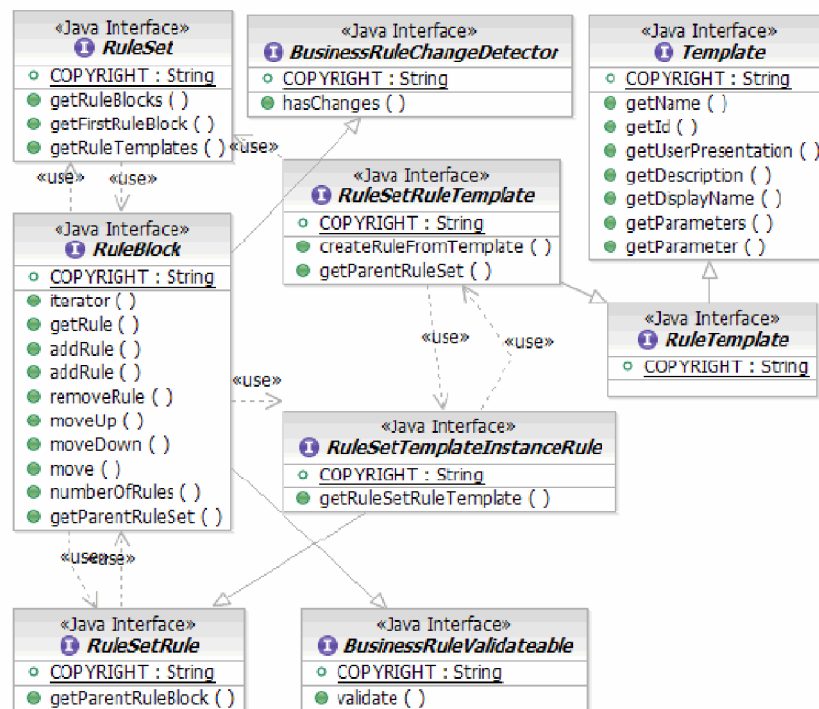


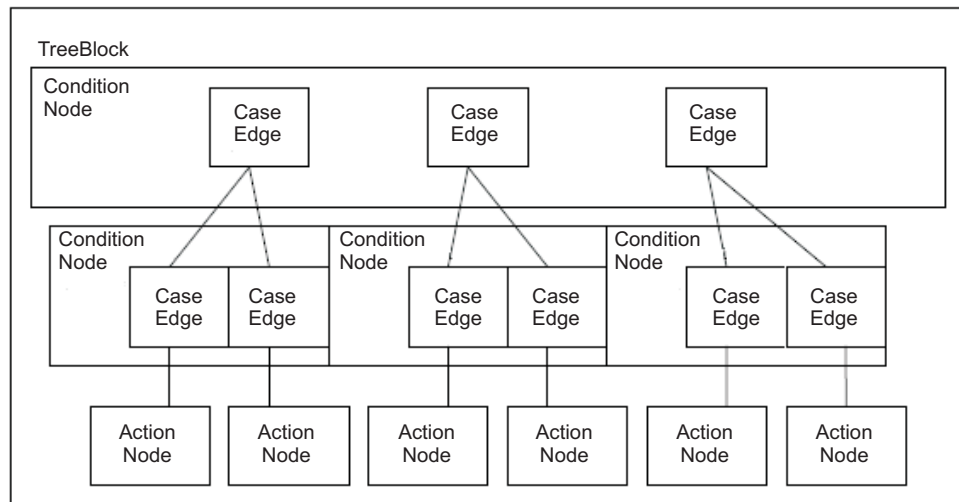
Figure 63. Class diagram of BusinessRule and related classes

Decision table

Decision tables are another type of business rule which can be managed and modified. Decision tables are typically used when there are a consistent number of conditions which must be evaluated and a specific set of actions to be issued when the conditions are met.

Decision tables are similar to decision trees, however they are balanced. Decision tables always have the same number of conditions to be evaluated and actions to be performed no matter what set of branches are resolved to true. A decision tree may have one branch with more conditions to evaluate than another branch.

Decision tables are structured as a tree of nodes and defined by a TreeBlock. There are different TreeNodes which make up the TreeBlock. TreeNodes can be condition nodes or action nodes. Condition nodes are the evaluation branches. At the end of branches, there are action nodes that have the appropriate tree actions to issue should all of the conditions evaluate to true. There can be any number of levels of condition nodes, but only one level of action nodes.



Decision tables might also have an initialization rule (init rule) which can be issued before the conditions in the table are checked.

The DecisionTable class provides methods that support the following:

- Retrieve the tree block of tree nodes (condition and action nodes)
- Retrieve the init rule instance
- Retrieve the init rule template if defined

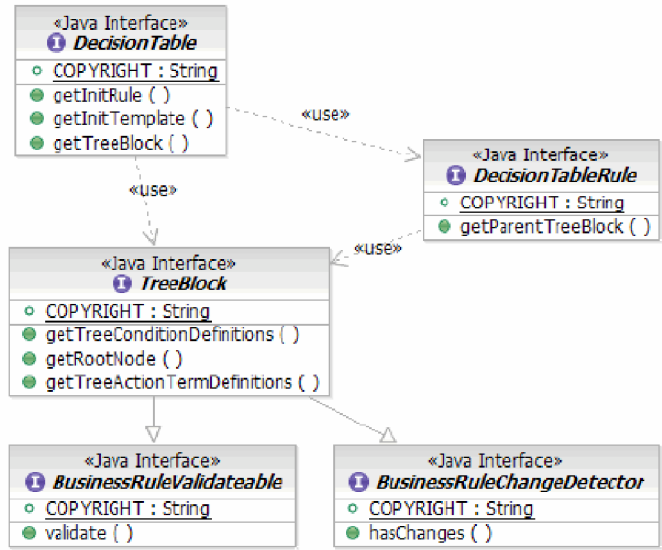
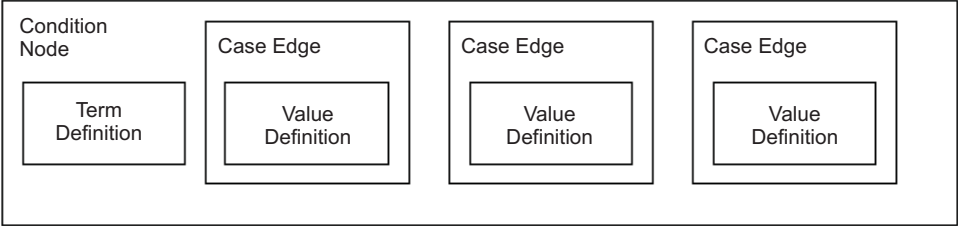


Figure 64. Class diagram of DecisionTable and related classes

The TreeBlock of a decision table contains the different condition and action nodes. Each condition node (ConditionNode) has a term definition (TreeConditionTermDefinition) and one to n case edges (CaseEdge). The term definition contains the left operand for the condition expression. The case edges contain the value definitions which are the different right operands to be used in the condition expression. For example, in the expression (status == "gold") the term definition would be "status" and "gold" would be the value definition in the case edge. For all of the case edges in a condition node, they share the term definition and are only different by the value (TreeConditionValueDefinition). Continuing with the example, another case edge in the condition node could have a value "silver". This would be used in an expression too (status == "silver"). The only exception to this behavior is if an otherwise has been defined for the condition node. With an otherwise, there is no value definition as it is used if all other case edges within the condition node evaluate to false. While an otherwise is not a case edge, it does have a TreeNode that can be retrieved.



For the term definition, the user presentation can be retrieved and used in client applications. The presentation for the term definition is typically only a representation of the left operand (status in our example) and does not contain any placeholders. For the case edges, a template can be used to define the value definition (TreeConditionValueTemplate). A template value definition instance (TemplateInstanceExpression) holds the parameter values which are used for execution and can be modified. If an attempt is made to retrieve the value

template definition for a `TreeConditionValueDefinition` that was not defined with a template, a null value will be returned. If a template has not been used to define the value condition, a user presentation can still be retrieved and used in client applications if it was specified at authoring time.

The `TreeBlock` class provides methods that support the following:

- Retrieve the root node of the tree
- Retrieve the condition term definitions for the tree block
- Retrieve the action term definitions for the tree block

The root node of the tree is of type `TreeNode` and from here, navigation of the decision table can occur. The `TreeNode` class provides methods that support the following:

- Determine if a node is an otherwise clause
- Retrieve the parent node for the current tree node (condition or action node)
- Retrieve the root node of the tree containing the current tree node

The `ConditionNode` class provides methods that support the following:

- Retrieve the case edges
- Retrieve the term definition
- Retrieve the otherwise case
- Retrieve the templates for the value conditions of the case edges for the condition node
- Add a condition value based on a template to the node
- Remove a condition value based on a template

The `CaseEdge` class provides methods that support the following:

- Retrieve the list of value templates which are available for the value definition
- Retrieve the child node (condition or action node)
- Retrieve the instance of the template definition associated with the value definition
- Retrieve the value definition directly without retrieving the template
- Set the value for the definition to use a specific template instance definition

The `TreeConditionTermDefinition` class provides methods that support the following:

- Retrieve the value definition templates defined for the condition node
- Retrieve the user presentation of the condition term

The `TreeConditionDefinition` class provides methods that support the following:

- Retrieve the term definition for the condition node
- Retrieve the condition value definitions for the condition node from all of the case edges
- Retrieve the orientation (row or column)

The `TreeConditionValueDefinition` class provides methods that support the following:

- Retrieve the specific template instance expression defined for the value
- Retrieve the user

The `Template` class provides methods that support the following:

- Retrieve the system ID for the template
- Retrieve the name of the template
- Retrieve the parameters defined for the template
- Retrieve the presentation for the template

The `TreeConditionValueTemplate` class provides a method that supports the following:

- Create a new template condition value instance

The `TemplateInstanceExpression` class provides methods that support the following:

- Retrieve the parameters for the template instance
- Retrieve the template (`TreeConditionValueTemplate` in the case of a case edge in a decision table) that was used to define the instance

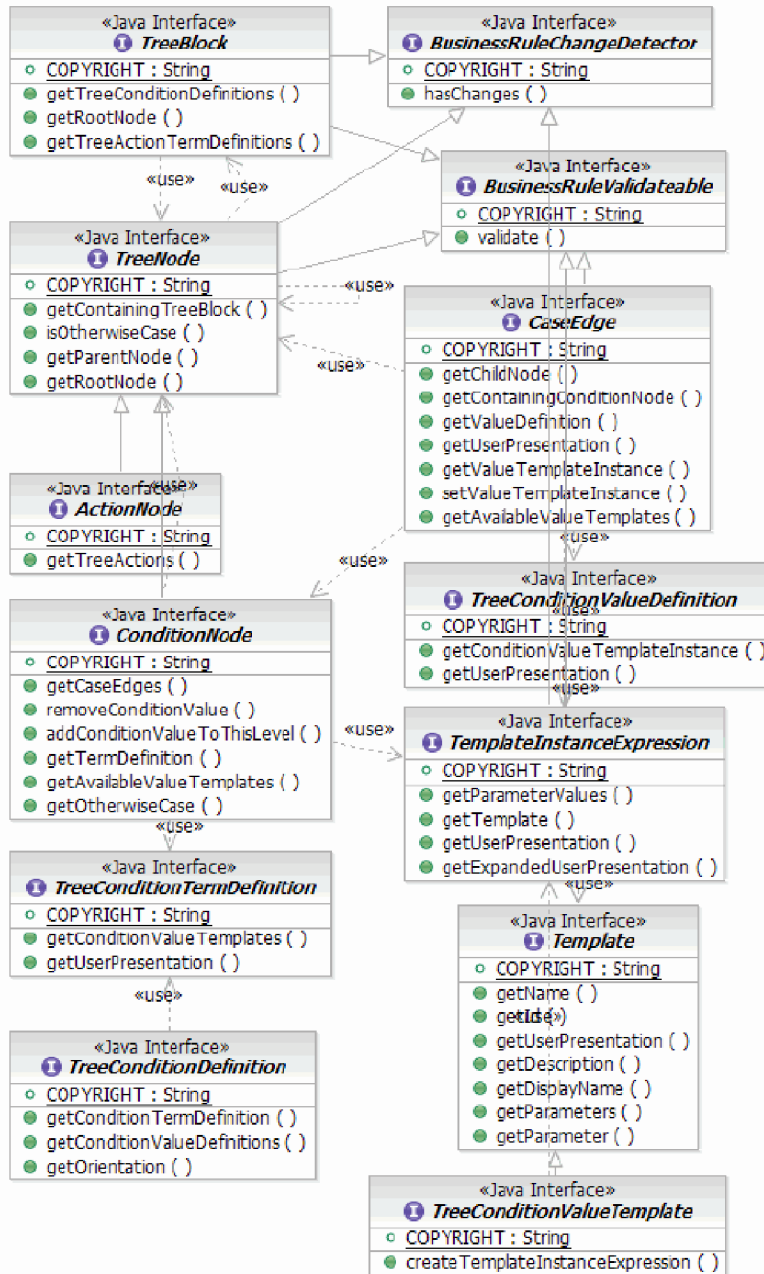


Figure 65. Class diagram for *TreeNode* and related classes

When a new case edge is added to a condition node, the new case edge must use a template to define the value. For example if a new case edge of “bronze” was to be added for checking ‘status’, the appropriate template (*TreeConditionValueTemplate*) would need to be used to create a new *TemplateInstanceExpression*, setting the parameter value to “bronze”.

When a new case edge is added, it will also have a child condition node added to it automatically. This child condition node will contain case edges which are based on the case edge definitions that have been defined for condition nodes at that same level. If templates or hard-coded values are used in case edges, they will then be used in the child condition node's case edges as well. The child condition node that is added automatically will also have its own child condition nodes created

automatically. These child condition nodes will also have child condition nodes and so on until all levels of condition nodes have been re-created.

Besides the condition nodes, a decision table and more specifically tree block, also contains a level of action nodes (`ActionNode`). The action nodes are leaf nodes and reside at the end of the branch of condition nodes and the case edges. Should all of the condition values in a line of case edges resolve to true, an action node is reached. The action node will have at least one action (`TreeAction`) defined. For the action, there will be a term definition and value definition. Just as with the condition nodes, the term definition (`TreeActionTermDefinition`) is to the left of the expression and the value definition (`TemplateInstanceExpression`) is to the right side of the expression. For example, for the different condition nodes which were checking on the status, there might be actions to define the discount. If the condition was (`status == "gold"`), the action can be (`discountValue = 0.90`). For the action the "discountValue" would be the term definition and the "`= 0.90`" would be the value definition.

The term definition of a tree action is shared with other tree actions in other action nodes. Since every branch of case edges reaches an action, the same term definitions are used. The value definitions however, can be different per tree action and action node. For example the discountValue for a status of "gold" can be "0.90"; however the "discountValue" for a status of "silver" can be "0.95".

Action nodes can have multiple tree actions which have a separate term definition and separate value definition. For example, if the discount was being determined for a rental car, besides setting the discountValue, you can also want to assign a specific level of car. Another tree action could be created to set the "carSize" term to "full size" if the status was "gold" as well as set the "discountValue" to "0.90".

The value definition in a tree action can be created from a template (`TreeActionValueTemplate`). The template definition contains an expression (`TemplateInstanceExpression`) which has the parameters for the expression.

Besides changing the parameters, the entire value definition can be modified with a new value definition instance which is created with another template which was defined for the tree action.

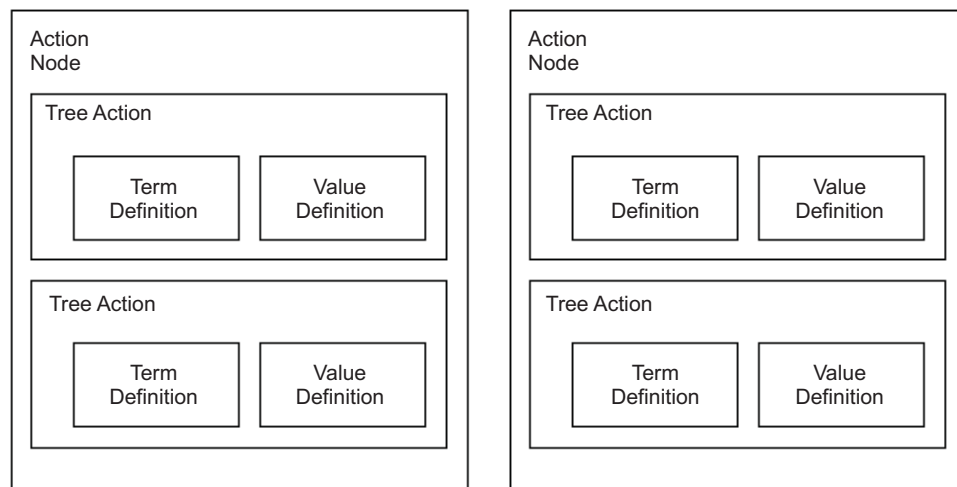
If a value definition is not created with a template, it cannot be changed. For client applications, the user presentation can be used in display if it was specified at author time.

For term definitions in tree actions, if a user presentation has been specified, it can also be used by client applications.

When a new case edge is added to a condition node and the different child condition nodes are created, action nodes will also be created. Unlike the child condition nodes and case edges which are created based on the definition of the case edges already defined for that level, action nodes do not automatically inherit an existing design. Only empty placeholder `TreeActions` are created in the action node. A template (`TreeActionValueTemplate`) must be used to complete the action definition by creating a `TemplateInstanceExpression` for at least one term definition for the action node. Until the tree action is set with a `TemplateInstanceExpression`, the tree action will have null values specified for the user presentation value and template instance value.

When creating a new condition that results in new ActionNodes, the action nodes will be added to the right of existing actions for the immediate parent condition node. For example if a status of “ruby” is added to the decision table and should have a specific discount, the condition to check the status is added at the right of “gold”, “silver”, and “bronze”. The action node for the discount for “ruby” will be added to the right of the action nodes that correspond to the “gold”, “silver” and “bronze” case edges.

When setting new tree actions for action nodes, an algorithm that looks to the rightmost action node at the lowest case edge will return the action node with an empty tree action. The tree action can also be checked that it has null values for the user presentation value and template instance value. Once the tree action is obtained, it can be set with the correct instance of a TreeActionValueTemplate.



The ActionNode class provides a method that supports the following:

- Retrieve a list of the defined tree actions

The TreeAction class provides methods that support the following:

- Retrieve a list of the available value templates defined for the tree action
- Retrieve the term definition
- Retrieve the value template instance defined for the tree action
- Retrieve the user presentation for the value if a value template was not used
- Check if the action is a SCA service invocation (isValueNotApplicable method)
- Replace the value template instance with a new instance

The TreeActionTermDefinition class provides methods that support the following:

- Retrieve the user presentation for the term value definition
- Retrieve a list of the value templates available for the tree action
- Check if the action is a SCA service invocation (isTermNotApplicable method)

The Template class provides methods that support the following:

- Retrieve the system ID for the template
- Retrieve the name of the template

- Retrieve the parameters defined for the template
- Retrieve the presentation for the template

The `TreeActionValueTemplate` class provides a method that supports the following:

- Create a new value template instance from the template definition

The `TemplateInstanceExpression` class provides methods that support the following:

- Retrieve the parameters for the template instance
- Retrieve the template (`TreeActionValueTemplate` in the case of a tree action in a decision table) which was used to define the instance

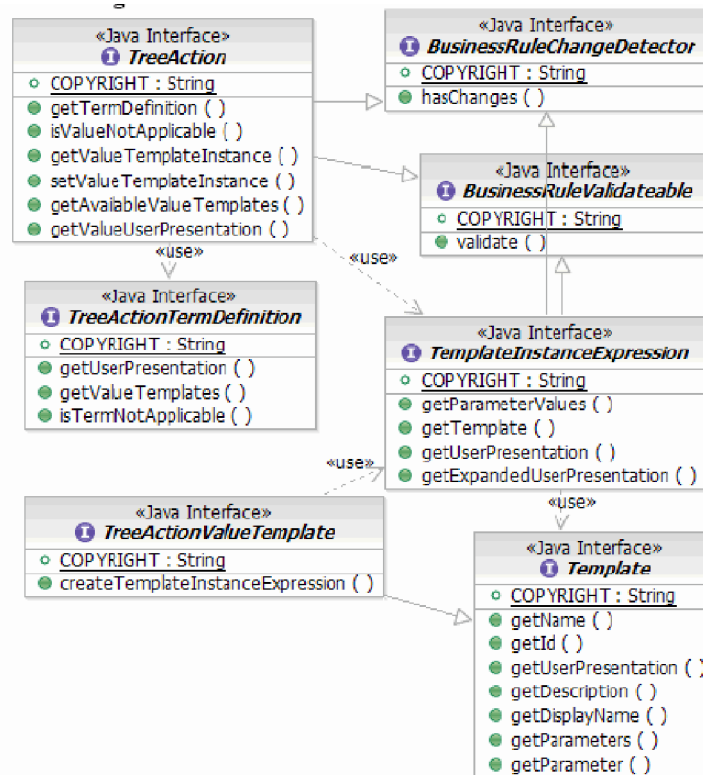


Figure 66. Class diagram of `TreeAction` and related classes

The definition of an init rule for a decision table follows the same structure as a rule in a rule set. The init rule can be defined with a template (`DecisionTableRuleTemplate`).

If an init rule was not created at authoring time, it can not be added once the rule is deployed.

The `Rule` class provides methods that support the following:

- Retrieve the name of the rule
- Retrieve the user presentation for the rule
- Retrieve the user presentation for the rule with the different parameters for the rule filled in

The DecisionTableRule class provides a method that supports the following:

- Retrieve the tree block containing the init rule

The DecisionTableRuleTemplate class provides a method that supports the following:

- Retrieve the decision table containing the template

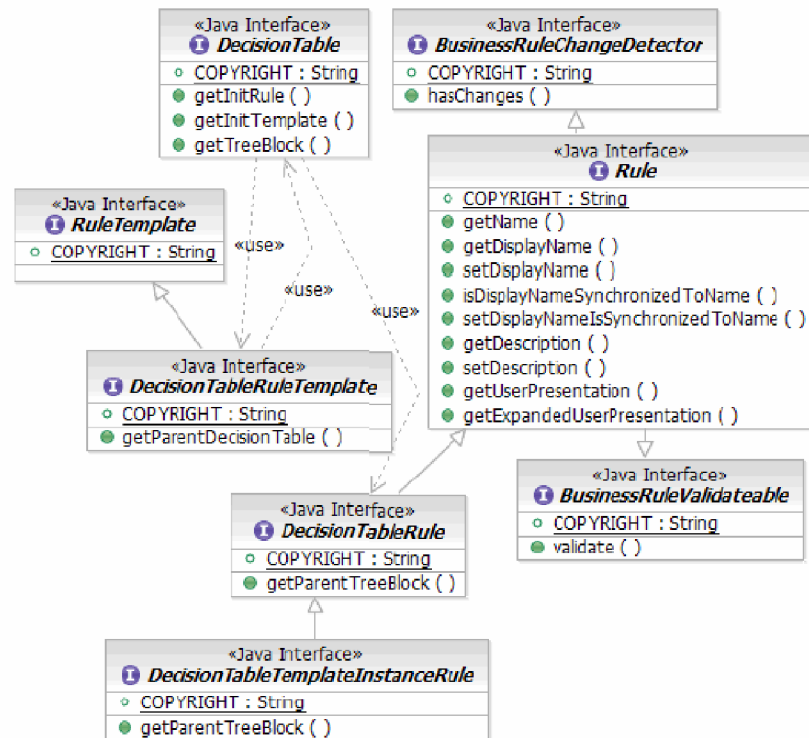


Figure 67. Class diagram for DecisionTableRule and related classes

Templates and Parameters

Templates in rule sets and decision tables are based off of a common definition. Templates have parameters and a user presentation. The template parameter values are defined to allow for changes to be made to the rule once it has been deployed.

The user presentation value provides a string value which can be used for displaying the rule and different parameters in a user-friendly manner. The user presentation, which is a string, has placeholders to allow for the different parameter values to be replaced and displayed correctly. The placeholders are in the format {<parameter index>}. For example, if the presentation string for the init rule is “Base discount is {0} %”, the placeholder {0} could be substituted with the parameter value. The presentation string cannot be changed for the rule or the template definition. The placeholder values however, can be modified with the parameter values in a client application per the definition of the template. The different templates include a convenience method (getExpandedUserPresentation) that returns a string which has all of the parameter values correctly placed in the string.

All parameter values have a specific data type, however when retrieving and setting a parameter value, a string object is used. The parameter value can be

treated as string when substituting the value into the user presentation and also when setting the parameter with a new value. The parameter is converted to the correct data type at runtime in order to correctly issue the rule at execution time. During validation, the parameter value will be compared to the data type to ensure it is correct. For example, if a parameter is of type `boolean` and is set to "T", validation will not recognize this value and will return a problem.

In the template definition, the parameter values can be restricted by constraints. The constraints can be defined as a range or an enumeration. The constraints for the parameter will be enforced when the rule is validated. If a template was not used to define the value definition, only a user presentation will be available. A value definition can not have both a template and a user presentation. Should a template be used, the presentation from the template definition is the only presentation which is available.

The `Template` class provides methods that support the following:

- Retrieve the template ID
- Retrieve the name
- Retrieve the parameters
- Retrieve the user presentation

The `Parameter` class provides methods that support the following:

- Retrieve the parameter name
- Retrieve the parameter data type
- Retrieve the constraint for the parameter
- Retrieve the template defining the parameter
- Create a parameter value

The `ParameterValue` class provides methods that support the following:

- Retrieve the parameter name
- Retrieve the parameter value
- Set the parameter value

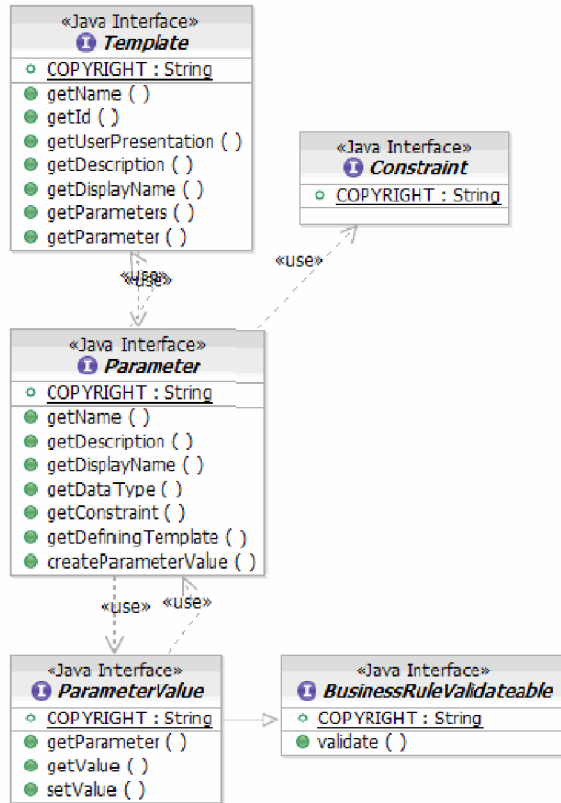


Figure 68. Class diagram for Template and Parameter and related classes

Validation

Many of the main objects have a validate method which allows for the artifacts to be checked for correctness and completeness before publishing the artifacts.

The validation that occurs when making changes through the API classes is only a proper subset of the overall validation that occurs during serviceDeploy or when editing the artifacts in WebSphere Integration Developer. This is due to the constraints that are already placed on the business rule group in limiting which aspects are editable at runtime. The user of the classes can validate the business rule group selection table, rule set or decision table whenever it is needed (the rule group component itself is not editable at runtime). When a business rule group is published the rule group selection table, rule sets and decision tables will be validated before being published to the repository.

If the artifacts are invalid, a ValidationException will be thrown with a list of the validation problems. The different validation problems are documented in the Exception Handling section.

Tracking Changes

For all objects, a hasChanges method is available to check if there have been any modifications which have occurred to the object and any containing objects.

This method can be used to check for changes and only publish a business rule group if it has items which changed.

BusinessRuleManager

The BusinessRuleManager class is the main class for working with the business rule groups, rule sets and decision tables.

The BusinessRuleManager has methods which allow for retrieving business rule groups by name, target namespace, or custom properties. It also has a method for publishing changes which have been made to business rule groups, rule sets, or decision tables.

The BusinessRuleManager class provides methods that support the following:

- Retrieve all of the business rule groups
- Retrieve business rule groups of a specific target namespace
- Retrieve business rule groups of a specific name
- Retrieve business rule groups of a specific name and target namespace
- Retrieve business rule groups which contain a specific property
- Retrieve business rule groups which contain specific properties
- Publish business rule groups

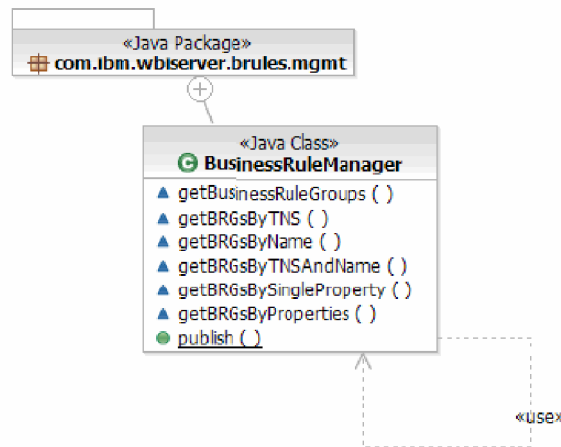


Figure 69. Class diagram for BusinessRuleManager and package

Rule Group Component Query

The rule group component can have user defined properties (name/value pairs) that can be used to narrow the list of business rule groups returned from the class. The fields that can be used in the query and in any combination are as follows:

- Business rule group component target name space
- Business rule group component name
- Property name
- Property value

Each property name can only be defined once per business rule group component.

The query function supported by this class is a small subset of the full SQL language. The user does not provide the SQL statement, but rather provides the values as parameters for a single property or a tree structure containing the information for a multi-property query in the form of nodes. There are logical

operator nodes and property query nodes which all implement the QueryNode interface. The logical operator nodes specify the boolean operators (AND, OR, NOT). These are created through the QueryNodeFactory. As part of the creation of these logical operator nodes, the left and right operators must be specified with additional QueryNode classes. These nodes can either be a property query node or another logical operator node. If a property query node is passed, it will contain the property name, value and operator (EQUAL (==), NOT_EQUAL (!=), LIKE, or NOTLIKE). The overall QueryNode is parsed by the class and a query is performed on the underlying data in persistent storage.

Wildcard searches are supported when the LIKE and NOTLIKE operators are used. Both the '%' and '_' characters are supported in wildcard searches. The '%' character is used when there are an infinite number of characters which are not known or should not be considered when searching. For example if a search was to be performed for all business rule groups that have a property with a name of Department and value that begins with "North", the value would be specified as "North%". Another example, suppose that all Departments with a value ending in "Region" was desired. The value would be "%Region". The '%' character can also be used in the middle of a string. For example, if there were business rule groups with properties that had values of "NorthCentralRegion", "NorthEastRegion", and "NorthWestRegion", a value of "North%Region" could be specified.

The '_' character is used when there is a single character which is unknown or should not be considered when searching. For example, if a search for all business rule groups with Department properties with values of "Dept1North", "Dept2North", "Dept3North", and "Dept4North" was desired, a value of "Dept_ North" could be specified and all 4 of the business rule groups with these properties will be returned. The '_' character can be used multiple times in a search value with each instance indicating a single character to ignore. The '_' character can be used at the beginning or end of a value. For example if two characters were to be ignored in a value, two '_' could be used such as "Dept__outh".

In order to treat '%' and '_' as literal characters and not wildcards a '\' escape character must be specified preceding the '%' or '_'. For example if the property name was "%Discount", in order to use this in a query, "%Discount" would need to be specified. If the '\' character is to be used as a literal character, another '\' escape character must be used such as "Orders\\Customer". If a single '\' character is found without a following '%', '_', or '\', an IllegalArgumentException will be thrown.

Wildcard characters can only be used on the left operator (property value). Wildcard characters can not be used in property name.

During searches on the value of a specific property or a search for values which do not match a property, the absence of a property causes the artifact to be ignored from consideration in the search. For example, if there are 3 business rule groups (A, B, and C) and only two (A and B) have a property named "Department" with different values ("Accounting" and "Shipping" respectively) a search for all business rule groups which do not have a "Department" property of "Accounting" will only return the business rule group which has the "Department" property defined but does not equal "Accounting" (business rule group B). The business rule group (C) which does not have the "Department" property, will not be returned as it does not have the property defined in any way.

When using properties for searching, there are two special properties named *IBMSysName* and *IBMSysTargetNameSpace* which can be used for searching based on the name and namespace of an artifact. These values can also be retrieved with the *getName* and *getTargetNameSpace* methods.

The class supports the following methods for query:

- List *getBRGsByTNS* (string tNSName, Operator op, int skip, int threshold)
- List *getBRGByName*(string Name, Operator op, int skip, int threshold)
- List *getBRGsByTNSAndName* (string tNSName, Operator, tNSOp, string name, Operator nameOp, int skip, int threshold)
- List *getBRGsBySingleProperty* (string propertyName, string propertyValue, Operator op, int skip, int threshold)
- List *getBRGsByProperties* (QueryNode queryTree, int skip, int threshold)

The 'skip' and 'threshold' parameters provide the user the capability of fetching a partial result list up to the specified threshold. A value of zero for both of these parameters will return the full result list. The cursor is not maintained in the result set from a query call. If a skip value is used, it is possible that additions or deletions could have been made to the result set such that a subsequent request will return business rule groups which were in an earlier result set.

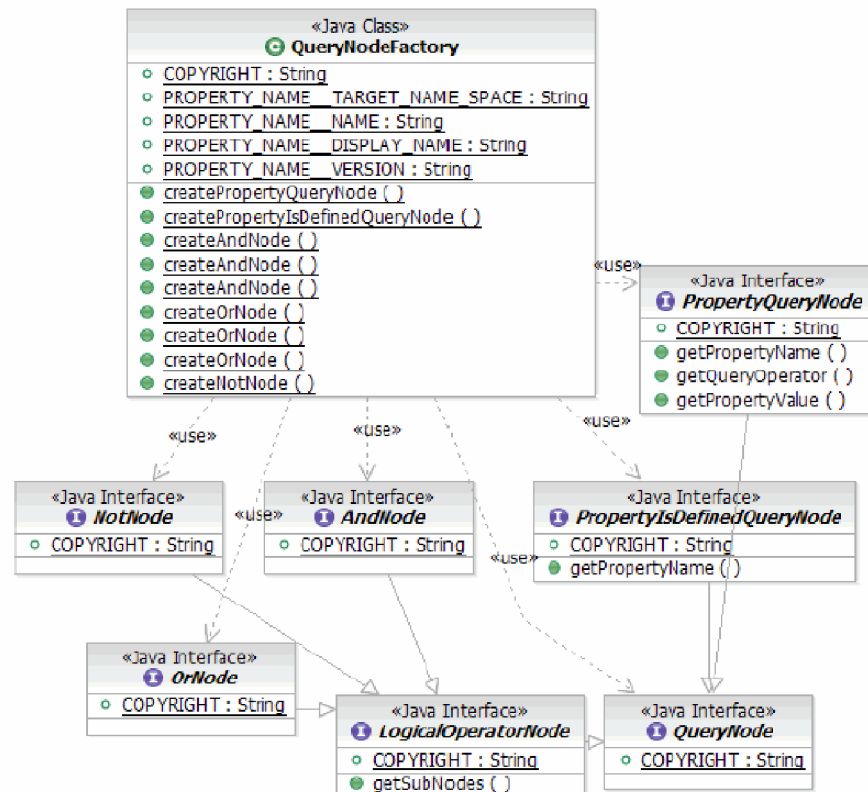


Figure 70. Class diagram for QueryNodeFactory and related classes

The nodes in the tree allow the user to specify a search expression using the boolean operators, wild cards (% and escape) and the property/value pair. The operator is only valid for the values, the operator for the property is always equals (==).

Publishing

Publishing of business rule changes is done at the business rule group component level. The user can publish 1...n business rule group components. Before a publish operation is performed, a validate action is performed on the business rule group and the different objects contained in the business rule group (operation selection table, rule sets, decision table, and so on). Each publish request will occur within a single transaction and if any exceptions are encountered during validation or database publishing the transaction is rolled back and no changes for any business rule group are published to the repository. This allows changes that are dependent on each other within a single component (for example, operation selection table and a rule set) or dependencies between components to occur within one atomic operation.

At publishing time, a check will be made to ensure that the items which are to be published have not been changed by another transaction. To reduce the possibilities of a conflict, the publish method will give the user the ability to choose to publish all artifacts whether they are changed or not or only those artifacts that were changed within the business rule group. The default behavior will be to publish all artifacts. If the option is set to publish all artifacts and another transaction had changed the artifacts in the meantime, a `ChangeConflictException` will be thrown. Specifying to only publish those artifacts which have changed will reduce the chance of conflict. Publishing only those artifacts that were changed could result in two users pushing changes to the repository for two different artifacts within a business rule group (for example, two rule sets) which could introduce incompatible changes within the business rule group. Because this potential situation, this option should be used with caution.

Exception Handling

Exceptions can occur when validation is called on an artifact or when an artifact is published. When a validation error occurs, the `ValidationException` is thrown with a list of problems. If there is a problem during publishing due to another transaction publishing the same artifacts, a `ChangeConflictException` is thrown. Anytime another transaction is detected as changing an artifact, the `ChangeConflictException` exception is thrown.

There is a `SystemPropertyNotChangeableException` which is thrown if a property which duplicates a system property name is attempted to be changed. System properties cannot be changed.

There is a `ChangesNotAllowedException` which is thrown if a set operation is attempted on an artifact as it is being published.

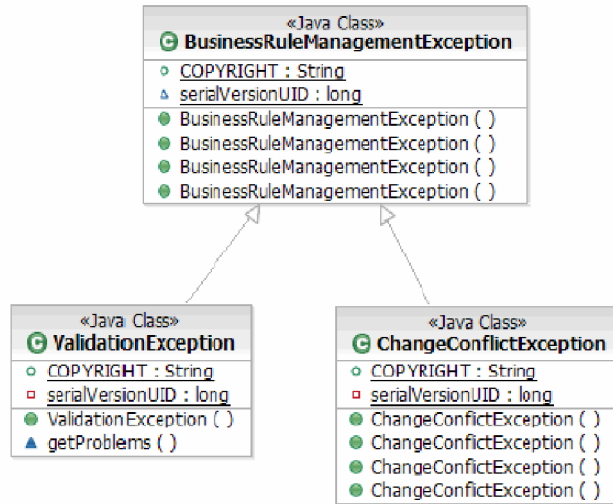


Figure 71. Class diagram for BusinessRuleManagementException and related classes

Business Rule Group problems

Problems which can occur when a business rule group is validated or an attempt to publish the business rule group is made and a portion of the business rule group is not valid.

Table 23. Business Rule Group problems

Exception	Description
ProblemBusRuleNotInAvailTargetList	Problem that occurs when a rule is specified as the default business rule for an operation selection table but the rule artifact is not in the list of available targets for that operation. A valid business rule from the list of available targets on the operation should be specified to avoid this problem.
ProblemDuplicatePropertyName	This problem occurs when a property is attempted to be created which is a duplicate of a system or user-defined property on a business rule group. A unique property name should be used to avoid this problem.
ProblemOperationContainsNoTargets	Problem that occurs when an operation does not have a default rule destination or any scheduled rule destinations set. The operation should be set with at least one rule destination as either the default or at a scheduled time to avoid this problem.
ProblemOverlappingRanges	Problem that occurs when an operation selection record has a start date or end date which overlaps with the range of another operation selection record start date and end date. This overlap in date ranges prevents the business rule runtime from finding the correct rule destination to invoke. The start date or end date of the other operation selection records for an operation should be checked to ensure there is not an overlap to avoid this problem.
ProblemStartDateAfterEndDate	This problem occurs when the start date for an operation selection record is after the end date for that selection record. This problem can occur for any operation selection record except the default record which does not have a start date or end date. Specify a start date after the end date on an operation selection record to avoid this problem.

Table 23. Business Rule Group problems (continued)

Exception	Description
ProblemTargetBusRuleNotSet	Problem that occurs when an operation selection record has a rule specified which is not in the list of available target rules. A rule from the available target list should be specified to avoid this problem.
ProblemTNSAndNameAlreadyInUse	Problem that occurs when a new business rule is created with a target name space and name which is already in use by a rule set or decision table. A check is done on all rule sets and decision tables associated with the current business rule group as well as any rule artifact stored in the repository. A different target name space or name should be used to avoid this problem.
ProblemWrongOperationForOpSelectionRecord	Problem that occurs when a new operation selection record is added to an operation selection record list and the operation of the new record does not match the operation of the records in the list. A new operation should be created using the <code>newOperationSelectionRecord</code> method on the correct operation selection record list object to avoid this problem.

Rule set and Decision Table problems

Table 24. Rule set and Decision Table problems

Exception	Description
ProblemInvalidBooleanValue	Problem that occurs when a parameter for a rule template in a rule set or an action value or condition value in decision table receives a different value than "true" or "false" for a parameter of type Boolean. Examples of incorrect parameter values would be "T" or "F". Use values of "true" or "false" when working with a parameter of type Boolean to avoid this problem.
ProblemParmNotDefinedInTemplate	Problem that occurs when a value is specified for a template parameter and the parameter is not defined in the list of valid parameters for the template. The parameters should be checked before setting in the template. It can occur for <code>RuleTemplate</code> , <code>TreeActionValueTemplate</code> , or <code>TreeConditionValueTemplate</code> templates.
ProblemParmValueListContainsUnexpectedValue	Problem that occurs when valid parameters are passed with a template, however, there are too many parameters for the parameter. The number of parameters should be reduced. It can occur for <code>RuleTemplate</code> , <code>TreeActionValueTemplate</code> , or <code>TreeConditionValueTemplate</code> templates.
ProblemRuleBlockContainsNoRules	This problem occurs when all rules in a rule block in a rule set are removed and the rule set is attempted to be validated or published. The rule block in a rule set must have at least one rule.
ProblemTemplateNotAssociatedWithRuleSet	Problem that occurs when a rule is attempted to be added to a rule set and the rule was created with a template which is not defined with that rule set. When creating a new rule, a template which has been defined in the rule set should be used to avoid this problem.
ProblemRuleNameAlreadyInUse	Problem that occurs when a rule is attempted to be added to a rule block in a rule set and it has the same name as an existing rule in the rule block. The names of the rules should be checked prior to adding a new rule to avoid this problem.

Table 24. Rule set and Decision Table problems (continued)

Exception	Description
ProblemTemplateParameterNotSpecified	This problem occurs when a parameter is not included when a template is updated for a rule in a rule set or action or condition value in a decision table. All parameters for a template should be specified to avoid this problem.
ProblemTypeConversionError	This problem occurs when a parameter for a template cannot be converted to the appropriate type All parameters are treated as string objects and then converted to the parameter type (boolean, byte, short, int, long, float, and double). If the parameter value string cannot be converted to the specified type for this parameter, then this error occurs. To avoid this problem, a string that can be converted to the parameter's type (boolean, byte, short, int, long, float, and double) should be specified.
ProblemValueViolatesParmConstraints	This problem occurs when a parameter is not within the enumeration or range of values which have been defined within the template for that parameter. This problem can occur for parameters restricted with enumerations or ranges in rule templates in a rule set or action value or condition value templates in a decision table. A value which is within the enumeration should be used to avoid this problem.
ProblemInvalidActionValueTemplate	Problem that occurs when a template instance is attempted to be set on the value definition in a tree action but the corresponding template is not available to that tree action. Use the correct template to create a value definition in a tree action in order to avoid this problem.
ProblemInvalidConditionValueTemplate	Problem that occurs when a template instance is attempted to be set on the condition definition in a case edge but the corresponding template is not available to that case edge. Use the correct template to create a condition definition in a case edge in order to avoid this problem.
ProblemTreeActionIsNull	This problem occurs when a new condition value is created and an action is not set with a template instance. Using a template from the ActionNode, create a new template instance and set it in the list of TreeActions.

Authorization

The classes do not support any level of authorization. It is up to the client application using the classes to add its own form of authorization.

Examples

A number of examples are provided that show how the different classes can be used to retrieve business rule groups and to make modifications to rule sets and decision tables. The examples are provided in a project interchange file (ZIP) that can be imported into WebSphere Integration Developer where they can be browsed and reused.

The project interchange contains a number of projects.

- **BRMgmtExamples** – Module project with business rules artifacts that are used in the various examples.
- **BRMgmt** – Java project with the examples located in the `com.ibm.websphere.sample.brules.mgmt` package.
- **BRMgmtDriverWeb** – Web project with interface for executing the samples.

The examples are also provided as an EAR file (BRMgmtExamples.ear) that can be issued once after installed into WebSphere Process Server. A Web interface is provided with the examples. The Web interface is purposely simple as the examples focus on using the classes to retrieve artifacts, make modifications, and publish changes. It is not meant to be a high-functioning Web interface. The classes can however, be easily used to build robust Web interfaces or used in other Java applications focused on modifying the business rules.

Note: You can download the example project interchange and EAR file from Business Rule Management Programming Guide for WebSphere Process Server V6.1.

The example application can be installed on WebSphere Process Server v6.1 and the index page can be accessed at:

`http://<hostname>:<port>/BRMgmtDriverWeb/`

For example, `http://localhost:9080/BRMgmtDriverWeb/`

As the examples are issued, changes will be made to the rule artifacts. If all examples are issued, the application will need to be reinstalled to see the same results for all examples again.

The examples are explained in detail with complete code samples as well as the result as displayed in a Web browser.

A number of additional classes were created in order to perform common operations and assist with displaying the information within the example Web application. See the appendix for more information on the Formatter and RuleArtifactUtility classes.

To fully understand these examples, a study of the different artifacts within WebSphere Integration Developer will greatly help.

Example 1: Retrieve and print all business rule groups

This example will retrieve all business rule groups and print out the attributes, the properties, and the operations for each business rule group.

```
package com.ibm.websphere.sample.brules.mgmt;
```

```
import java.util.Iterator;  
import java.util.List;
```

For the business rules management classes, be sure to use those classes in the com.ibm.wbiserver.brules.mgmt package and not the com.ibm.wbiserver.brules package or other package. These other packages are for IBM internal classes.

```
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;  
import  
com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;  
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;  
import com.ibm.wbiserver.brules.mgmt.Operation;  
import com.ibm.wbiserver.brules.mgmt.Property;  
import com.ibm.wbiserver.brules.mgmt.PropertyList;
```

```
public class Example1 {  
    static Formatter out = new Formatter();  
    static public String executeExample1()
```



```

{
try
{
out.clear();

```

The `BusinessRuleManager` class is the main class to retrieve business rule groups and to publish changes to business rule groups. This includes working with and changing any rule artifacts such as rule sets and decision tables. There are a number of methods on the `BusinessRuleManager` class that simplify the retrieval of specific business rule groups by name and namespace and properties.

```

// Retrieve all business rule groups
List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBusinessRuleGroups(0, 0);

Iterator<BusinessRuleGroup> iterator = brgList.iterator();

BusinessRuleGroup brg = null;
// Iterate through the list of business rule groups
while (iterator.hasNext())
{
    brg = iterator.next();
    // Output attributes for each business rule group
    out.printlnBold("Business Rule Group");

```

The basic attributes of the business rule group can be retrieved and displayed.

```

out.println("Name: " + brg.getName());
out.println("Namespace: " +
brg.getTargetNameSpace());
out.println("Display Name: " +
brg.getDisplayName());
out.println("Description: " + brg.getDescription());
out.println("Presentation Time zone: "
    + brg.getPresentationTimezone());
out.println("Save Date: " + brg.getSaveDate());

```

The properties for the business rule group can also be retrieved and modified.

```

PropertyList propList = brg.getProperties();

Iterator<Property> propIterator =
propList.iterator();
Property prop = null;
// Output property names and values
while (propIterator.hasNext())
{
    prop = propIterator.next();
    out.println("Property Name: " +
prop.getName());
    out.println("Property Value: " +
prop.getValue());
}

```

The operations for the business rule group are also available and are the way to retrieve the business rule artifacts such as rule sets and decision tables.

```

List<Operation> opList = brg.getOperations();

Iteration<Operation> opIterator = opList.iterator();
Operation op = null;
// Output operations for the business rule group
while (opIterator.hasNext())
{
    op = opIterator.next();
    out.println("Operation: " + op.getName());
}

```

```

out.println("");
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
return out.toString();
}
}

```

Web browser output for example 1.

Executing example1

Business Rule Group

Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSYSTEMVERSION
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSYSTEMTARGETNAMESPACE
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSYSTEMNAME
Property Value: ApprovalValues
Property Name: IBMSYSTEMDISPLAYNAME
Property Value: ApprovalValues
Operation: getApprover

Business Rule Group

Name: ConfigurationValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ConfigurationValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSYSTEMVERSION
Property Value: 6.2.0
Property Name: Department
Property Value: General
Property Name: RuleType
Property Value: messages
Property Name: IBMSYSTEMTARGETNAMESPACE
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSYSTEMNAME
Property Value: ConfigurationValues
Property Name: IBMSYSTEMDISPLAYNAME
Property Value: ConfigurationValues
Operation: getMessages

Business Rule Group

Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSYSTEMVERSION
Property Value: 6.2.0

```

Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules
Operation: calculateOrderDiscount
Operation: calculateShippingDiscount

```

Example 2: Retrieve and print business rule groups, rule sets and decision tables

Besides the function in example 1, this example will print out the selection table for each operation and then the default business rule destination (either rule set or decision table) and the other business rules scheduled for the operation. It prints out both rule sets and decision tables.

The majority of the example is the same, but provided for completeness.

```

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
public class Example2
{
    status Formatter out = new Formatter();
    static public String executeExample2()
    {
        try
        {
            out.clear();

```

A specific business rule group is retrieved by name for this example.

```

// Retrieve all business rule groups
List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByName("DiscountRules",
        QueryOperator.EQUAL, 0, 0);

Iterator<BusinessRuleGroup> iterator = brgList.iterator();

BusinessRuleGroup brg = null;
// Iterate through the list of business rule groups
while (iterator.hasNext())
{
    brg = iterator.next();
    // Output attributes for each business rule group
    out.printlnBold("Business Rule Group");
    out.println("Name: " + brg.getName());
    out.println("Namespace: " +
        brg.getTargetNameSpace());
    out.println("Display Name: " +
        brg.getDisplayName());
    out.println("Description: " + brg.getDescription());
    out.println("Presentation Time zone: "

```

```

    + brg.getPresentationTimezone());
out.println("Save Date: " + brg.getSaveDate());

PropertyList propList = brg.getProperties();

Iterator<Property> propIterator =
propList.iterator();
Property prop = null;
// Output property names and values
while (propIterator.hasNext())
{
    prop = propIterator.next();
    out.println("Property Name: " +
prop.getName());
    out.println("Property Value: " +
prop.getValue());
}

```

For each operation, a selection table has a list of the different rule artifacts and the schedule on when they are active. A default business rule can be specified for each operation. There is no requirement that a default business rule be specified or that there is a scheduled business rule, however there must be at least a default business rule or one scheduled business rule. Because of this support, it is best to check for null before using the default business rule or check the size of the OperationSelectionRecordList.

```

List<Operation> opList = brg.getOperations();

Iterator<Operation> opIterator = opList.iterator();
Operation op = null;
out.println("");
out.printlnBold("Operations");
// Output operations for the business rule group
while (opIterator.hasNext())
{
    op = opIterator.next();
    out.printBold("Operation: ");
    out.println(op.getName());

    // Retrieve the default business rule for the operation
    BusinessRule defaultRule =
op.getDefaultBusinessRule();
    // If the default rule is found, print out the business rule
    // using the appropriate method for rule type
    if (defaultRule != null)
    {
        out.printlnBold("Default Destination:");
    }
}

```

The default business rule is of type RuleSet or DecisionTable and can be cast to the correct type in order to process the rule artifact.

```

    if (defaultRule instanceof RuleSet)
        out.println(RuleArtifactUtility.
intRuleSet(defaultRule));
    else
        out.print(RuleArtifactUtility.
tDecisionTable(defaultRule));
}
OperationSelectionRecordList
opSelectionRecordList = op
.getOperationSelectionRecordList()
;

Iterator<OperationSelectionRecord>

```

```

opSelRecordIterator = opSelectionRecordList
    .iterator();
OperationSelectionRecord record = null;

```

The OperationSelectionRecord is composed of the rule artifact and the schedule on when the rule artifact is active.

```

while (opSelRecordIterator.hasNext())
{
    out.printlnBold("Scheduled
Destination:");
    record = opSelRecordIterator.next();

    out.println("Start Date: " +
record.getStartDate()
+ " - End Date: " +
record.getEndDate());
    BusinessRule ruleArtifact = record
.getBusinessRuleTarget();

    if (ruleArtifact instanceof RuleSet)
        out.println(RuleArtifactUtility.pr
intRuleSet(ruleArtifact));
    else
        out.print(RuleArtifactUtility.prin
tDecisionTable(ruleArtifact));
}
}
}
}
out.println("");
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
return out.toString();
}
}
}

```

Example

Web browser output for example 2.

Business Rule Group

```

Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules

```

Operations

Operation: calculateOrderDiscount

Default Destination:

Rule Set

Name: calculateOrderDiscount

Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: CopyOrder
 Display Name: CopyOrder
 Description: null
 Expanded User Presentation: null
 User Presentation: null
Rule: FreeGiftInitialization
 Display Name: FreeGiftInitialization
 Description: null
 Expanded User Presentation: Product ID for Free Gift = 5001AE80 Quantity = 1 Cost = 0.0 Description = Free gift for discounted order
 User Presentation: Product ID for Free Gift = {0} Quantity = {1} Cost = {2}
 Description = {3}Parameter Name: param0
 Parameter Value: 5001AE80
 Parameter Name: param1
 Parameter Value: 1
 Parameter Name: param2
 Parameter Value: 0.0
 Parameter Name: param3
 Parameter Value: Free gift for discounted order
Rule: Rule1
 Display Name: Rule1
 Description: null
 Expanded User Presentation: If customer is gold status, then apply a discount of 20.0 and include a free gift
 User Presentation: If customer is {0} status, then apply a discount of {1} and include a free gift
 Parameter Name: param0
 Parameter Value: gold
 Parameter Name: param1
 Parameter Value: 20.0
Rule: Rule2
 Display Name: Rule2
 Description: null
 Expanded User Presentation: If customer.status == silver, then provide a discount of 15.0
 User Presentation: If customer.status == {0}, then provide a discount of {1}
 Parameter Name: param0
 Parameter Value: silver
 Parameter Name: param1
 Parameter Value: 15.0
Rule: Rule3
 Display Name: Rule3
 Description: Template for non-gold customers
 Expanded User Presentation: If customer.status == bronze, then provide a discount of 10.0
 User Presentation: If customer.status == {0}, then provide a discount of {1}
 Parameter Name: param0
 Parameter Value: bronze
 Parameter Name: param1
 Parameter Value: 10.0

Operation: calculateShippingDiscount
Default Destination:
Decision Table
 Name: calculateShippingDiscount
 Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Init Rule: Rule1
 Display Name: Rule1
 Description: null
 Extended User Presentation: null
 User Presentation: null

Example 3: Retrieve business rule groups by multiple properties with AND

This example is also similar to example 1, but will only retrieve those business rule groups which have a property named Department and a value of "accounting" and a property named RuleType and a value of "regulatory".

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.AndNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example3
{
    static Formatter out = new Formatter();
    static public String executeExample3()
    {
        try
        {
            out.clear();
```

Queries for business rule groups are composed of query nodes that follow a tree structure. Each query node has a left side term and a right side term and condition. Each term and right term can be another query node. For this example the business rule group is retrieved by the combination of two property values.

```
        // Retrieve business rule groups based on two conditions
        // Create PropertyQueryNodes for each one condition
        PropertyQueryNode propertyNode1 = QueryNodeFactory
            .createPropertyQueryNode("Department",
                QueryOperator.EQUAL,"Accounting");
        PropertyQueryNode propertyNode2 = QueryNodeFactory
            .createPropertyQueryNode("RuleType", QueryOperator.EQUAL,
                "regulatory");
        // Combine the two PropertyQueryNodes with an AND node
        AndNode andNode =
            QueryNodeFactory.createAndNode(propertyNode1, propertyNode2);

        // Use andNode in search for business rule groups
        List<BusinessRuleGroup> brgList = BusinessRuleManager
            .getBRGsByProperties(andNode, 0, 0);

        Iterator<BusinessRuleGroup> iterator = brgList.iterator();

        BusinessRuleGroup brg = null;
        // Iterate through the list of business rule groups
        while (iterator.hasNext())
        {
            brg = iterator.next();
            // Output attributes for each business rule group
            out.printlnBold("Business Rule Group");
            out.println("Name: " + brg.getName());
            out.println("Namespace: " +
                brg.getTargetNameSpace());
            out.println("Display Name: " + brg.getDisplayName());
            out.println("Description: " + brg.getDescription());
            out.println("Presentation Time zone: "
```

```

+ brg.getPresentationTimezone());
out.println("Save Date: " + brg.getSaveDate());

PropertyList propList = brg.getProperties();

Iterator<Property> propIterator =
propList.iterator();
Property prop = null;
// Output property names and values
while (propIterator.hasNext())
{
    prop = propIterator.next();
    out.println("\t Property Name: " +
prop.getName());
    out.println("\t Property Value: " +
prop.getValue());
}
}
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
return out.toString();
}
}

```

Example

Web browser output for example 3.

Executing example3

```

Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSYSTEMVERSION
Property Value: 6.2.0
Property Name: DEPARTMENT
Property Value: ACCOUNTING
Property Name: RULETYPE
Property Value: REGULATORY
Property Name: IBMSYSTEMTARGETNAMESPACE
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSYSTEMNAME
Property Value: ApprovalValues
Property Name: IBMSYSTEMDISPLAYNAME
Property Value: ApprovalValues

```

Example 4: Retrieve business rule groups by multiple properties with OR

This example is similar to example 3; however it will only retrieve those business rule groups which have a property named Department and a value of "accounting" or a property named RuleType and a value of "monetary".

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;

```



```

import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.OrNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example4
{
    static Formatter out = new Formatter();
    static public String executeExample4()
    {
        try
        {
            out.clear();

```

Different properties make up the query and return different business rule groups.

```

// Retrieve business rule groups based on two conditions
// Create PropertyQueryNodes for each one condition
PropertyQueryNode propertyNode1 = QueryNodeFactory
    .createPropertyQueryNode("Department",
        QueryOperator.EQUAL,"Accounting");
PropertyQueryNode propertyNode2 = QueryNodeFactory
    .createPropertyQueryNode("RuleType",
        QueryOperator.EQUAL,"monetary");
// Combine the two PropertyQueryNodes with an OR node
OrNode orNode =
    QueryNodeFactory.createOrNode(propertyNode1,
        propertyNode2);
// Use orNode in search for business rule groups
List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByProperties(orNode, 0, 0);

Iterator<BusinessRuleGroup> iterator = brgList.iterator();

BusinessRuleGroup brg = null;
// Iterate through the list of business rule groups
while (iterator.hasNext())
{
    brg = iterator.next();
    // Output attributes for each business rule group
    out.printlnBold("Business Rule Group");
    out.println("Name: " + brg.getName());
    out.println("Namespace: " +
        brg.getTargetNameSpace());
    out.println("Display Name: " + brg.getDisplayName());
    out.println("Description: " + brg.getDescription());
    out.println("Presentation Time zone: "
        + brg.getPresentationTimezone());
    out.println("Save Date: " + brg.getSaveDate());

    PropertyList propList = brg.getProperties();

    Iterator<Property> propIterator =
        propList.iterator();
    Property prop = null;
    // Output property names and values
    while (propIterator.hasNext())
    {
        prop = propIterator.next();
        out.println("\t Property Name: " +
            prop.getName());
        out.println("\t Property Value: " +
            prop.getValue());
    }
}

```

```

        out.println("");
    }
} catch (BusinessRuleManagementException e)
{
    e.printStackTrace();
    out.println(e.getMessage());
}
return out.toString();
}
}

```

Example

Web browser output for example 4.

Executing example4

Business Rule Group

```

Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues

```

Business Rule Group

```

Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules

```

Example 5: Retrieve business rule groups with a complex query

This example is a combination of examples 3 and 4 and it is meant to show how more complex queries can be created. In this example a search is performed with a query that combines 2 query conditions. The first query condition is to retrieve those business rule groups which have a property named Department and a value of "General" or a property named MissingProperty and a value of "somevalue". This query condition is then combined with an AND to a condition where the property is named RuleType and a value of "messages".

More examples of business rule group queries are available in the appendix.

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.AndNode;
import com.ibm.wbiserver.brules.mgmt.query.OrNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example5
{
    static Formatter out = new Formatter();
    static public String executeExample5()
    {
        try
        {
            out.clear();

            // Retrieve business rule groups based on three conditions where
            // two of the conditions are combined in an OR node
            // Create PropertyQueryNodes for each condition for the OR node
            PropertyQueryNode propertyNode1 = QueryNodeFactory
                .createPropertyQueryNode("Department",
                    QueryOperator.EQUAL, "General");
            PropertyQueryNode propertyNode2 = QueryNodeFactory
                .createPropertyQueryNode("MissingProperty",
                    QueryOperator.EQUAL, "SomeValue");
            // Combine the two PropertyQueryNodes with an OR node
            OrNode orNode =
                QueryNodeFactory.createOrNode(propertyNode1, propertyNode2);
            // Create the third PropertyQueryNode
            PropertyQueryNode propertyNode3 = QueryNodeFactory
                .createPropertyQueryNode("RuleType",
                    QueryOperator.EQUAL, "messages");
```

The left condition is combined to the right condition with an AND node. The AndNode is the root of the query tree.

```
            // Combine OR node with third PropertyQueryNode with
            AndNode andNode =
                QueryNodeFactory.createAndNode(propertyNode3, orNode);

            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsByProperties(andNode, 0, 0);

            Iterator<BusinessRuleGroup> iterator = brgList.iterator();

            BusinessRuleGroup brg = null;
            // Iterate through the list of business rule groups
            while (iterator.hasNext())
            {
                brg = iterator.next();
                // Output attributes for each business rule group
                out.printlnBold("Business Rule Group");
                out.println("Name: " + brg.getName());
                out.println("Namespace: " +
                    brg.getTargetNameSpace());
                out.println("Display Name: " + brg.getDisplayName());
                out.println("Description: " + brg.getDescription());
```

```

        out.println("Presentation Time zone: "
            + brg.getPresentationTimezone());
        out.println("Save Date: " + brg.getSaveDate());
        PropertyList propList = brg.getProperties();

        Iterator<Property> propIterator =
            propList.iterator();
        Property prop = null;
        // Output property names and values
        while (propIterator.hasNext())
        {
            prop = propIterator.next();
            out.println("\t Property Name: " +
                prop.getName());
            out.println("\t Property Value: " +
                prop.getValue());
        }
    } catch (BusinessRuleManagementException e)
    {
        e.printStackTrace();
        out.println(e.getMessage());
    }
    return out.toString();
}
}

```

Example

Web browser output for example 5.

Executing example5

Business Rule Group

```

Name: ConfigurationValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ConfigurationValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSYSTEMVERSION
Property Value: 6.2.0
Property Name: DEPARTMENT
Property Value: General
Property Name: RULETYPE
Property Value: messages
Property Name: IBMSYSTEMTARGETNAMESPACE
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSYSTEMNAME
Property Value: ConfigurationValues
Property Name: IBMSYSTEMDISPLAYNAME
Property Value: ConfigurationValues

```

Example 6: Update a business rule group property and publish

In this example, a property in a business rule group is updated and then the business rule group is published.

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.UserDefinedProperty;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

```

```

public class Example6
{
    static Formatter out = new Formatter();

    static public String executeExample6()
    {
        try
        {
            out.clear();
            out.printlnBold("Business Rule Group before publish:");
            // Retrieve business rule groups by a single property value
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsBySingleProperty("Department",
                    QueryOperator.EQUAL,"General", 0, 0);

            if (brgList.size() > 0)
            {
                // Get the first business rule group from the list
                BusinessRuleGroup brg = brgList.get(0);
                // Retrieve the property from the business rule group
                UserDefinedProperty userDefinedProperty =
                    (UserDefinedProperty) brg
                        .getProperty("Department");

                out.println("Business Rule Group: " + brg.getName());
                out.println("Department Property value: "
                    + brg.getProperty("Department").getValue());
            }
        }
    }
}

```

The `getProperty` method returns a property by reference and changes made to the property are directly made to the business rule group.

```

// Modify the property value in the brg
// This updates the property value directly in the
// brg object
userDefinedProperty.setValue("GeneralConfig");
// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();
// Add the changed business rule group to the list
publishList.add(brg);

```

The `BusinessRuleManager` class is used to publish the changes made to a business rule group. To publish the change, a list is passed to the `BusinessRuleManager` `publish` method even if there is only one item is being published.

```

// Publish the list with the updated business rule group
BusinessRuleManager.publish(publishList, true);

out.println("");

// Retrieve the business rule group again to verify the
// changes were published
out.printlnBold("Business Rule Group after publish:");
brgList = BusinessRuleManager
    .getBRGsBySingleProperty("Department",
        QueryOperator.EQUAL, "GeneralConfig", 0, 0);

brg = brgList.get(0);

out.println("Business Rule Group: " + brg.getName());
// Display the property value to show the change
out.println("Department Property value: "
    + brg.getProperty("Department").getValue());
}
} catch (BusinessRuleManagementException e)

```

```

    {
    e.printStackTrace();
    out.println(e.getMessage());
    }
return out.toString();
}
}

```

Example

Web browser output for example 6.

Executing example6

Business Rule Group before publish:

```

Business Rule Group: ConfigurationValues
Department Property value: General

```

Business Rule Group after publish:

```

Business Rule Group: ConfigurationValues
Department Property value: GeneralConfig

```

Example 7: Update properties in multiple business rule groups and publish

In this example, properties in multiple business rule groups are updated before publish.

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.UserDefinedProperty;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example7
{
    static Formatter out = new Formatter();

    static public String executeExample7()
    {
        try
        {
            out.clear();
            out.printlnBold("Business Rule Group before publish:");
            // Retrieve business rule groups by a single property value
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsBySingleProperty("Department",
                    QueryOperator.EQUAL, "Accounting", 0, 0);

            Iterator<BusinessRuleGroup> iterator = brgList.iterator();

            BusinessRuleGroup brg = null;

            // Use the original list or create a new list
            // of business rule groups
            List<BusinessRuleGroup> publishList = new
                ArrayList<BusinessRuleGroup>();

            // Iterate through all of the business rule groups and
            // modify the property
            while (iterator.hasNext())
            {

```

```

// Retrieve the property from the business rule group
brg = iterator.next();

out.println("Business Rule Group: " + brg.getName());

// Retrieve the property from the business rule group
UserDefinedProperty prop = (UserDefinedProperty) brg
    .getProperty("Department");
out.println("Department Property value: "
+
brg.getProperty("Department").getValue()
);

// Modify the property value in the brg
// This updates the property value directly in the
brg object
prop.setValue("Finance");

```

Each changed business rule group is added to the list.

```

// Add the changed business rule group to the list
publishList.add(brg);
}

// Publish the list with the updated business rule
group
BusinessRuleManager.publish(publishList, true);

out.println("");

// Retrieve the business rule groups again to verify the
// changes were published
out.printlnBold("Business Rule Group after
publish:");

brgList = BusinessRuleManager
    .getBRGsBySingleProperty("Department",
        QueryOperator.EQUAL,
        "Finance", 0, 0);
iterator = brgList.iterator();

while (iterator.hasNext())
{
    brg = iterator.next();
    out.println("Business Rule Group: " +
brg.getName());
    out.println("Department Property value: "
+
brg.getProperty("Department").getVa
lue());
}
} catch (BusinessRuleManagementException e)
{
    e.printStackTrace();
    out.println(e.getMessage());
}
return out.toString();
}
}

```

Example

Web browser output for example 7.

Executing example7

Business Rule Group before publish:

```
Business Rule Group: ApprovalValues
Department Property value: Accounting
Business Rule Group: DiscountRules
Department Property value: Accounting
```

Business Rule Group after publish:

```
Business Rule Group: ApprovalValues
Department Property value: Finance
Business Rule Group: DiscountRules
Department Property value: Finance
```

Example 8: Change the default business rule for a business rule group

In this example, the default business rule is changed with another business rule that is part of the available targets list for a specific operation.

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example8
{
    static Formatter out = new Formatter();

    static public String executeExample8()
    {
        try
        {
            out.clear();

            // Retrieve a business rule group by target namespace and name
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsByTNSAndName(
                    "http://BRSamples/com/ibm/websphere
                    /sample/brules",
                    QueryOperator.EQUAL,
                    "DiscountRules",
                    QueryOperator.EQUAL, 0, 0);

            if (brgList.size() > 0)
            {
                out.printlnBold("Business Rule Group before publish:");
                // Get the first business rule group from the list
                // This should be the only business rule group in the list as
                // the combination of target namespace and name are unique
                BusinessRuleGroup brg = brgList.get(0);

                out.print("Business Rule Group: ");
                out.println(brg.getName());

                // Get the operation of the business rule group that
                // will have its default business rule updated
                Operation op =
                    brg.getOperation("calculateShippingDiscount");
            }
        }
    }
}
```

The default business rule is retrieved before it is updated with another rule that is part of the available target list for the operation. Rule sets and decision tables are

specific to operations and only those business rule artifacts that are for an operation can be set to be default or be scheduled for another time on the operation.

```
// Retrieve the default business rule for the operation
BusinessRule defaultRule =
op.getDefaultBusinessRule();
out.print("Default Rule: ");
out.println(defaultRule.getName());

// Get the list of available business rules for this
operation
List<BusinessRule> ruleList =
op.getAvailableTargets();

Iterator<BusinessRule> iterator =
ruleList.iterator();
BusinessRule rule = null;

// Find a business rule that is different from the
current
// default
// business rule
while (iterator.hasNext())
{
    rule = iterator.next();
    if
    (!defaultRule.getName().equals(rule.getName()))
    {
```

The default business rule is set on the operation object. Setting the default business rule to null will remove any default business rule from the operation, however it is recommended that every operation have a default business rule specified.

```
    // Set the default business rule to be a
    // different business rule
    // This change is to the operation object
    // directly
    op.setDefaultBusinessRule(rule);
    break;
}
}
// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();
// Add the changed business rule group to the list
publishList.add(brg);
// Publish the list with the updated business rule
group
BusinessRuleManager.publish(publishList, true);

out.println("");

// Retrieve the business rule groups again to verify the
// changes were published

out.printlnBold("Business Rule Group after publish:");
brgList = BusinessRuleManager
.getBRGsByTNSAndName(
    "http://BRSamples/com/ibm/websphere/sample/brules",
    QueryOperator.EQUAL, "DiscountRules",
    QueryOperator.EQUAL, 0, 0);

brg = brgList.get(0);
out.println("Business Rule Group: " + brg.getName());
op = brg.getOperation("calculateShippingDiscount");
```

```

        // Retrieve the default business rule for the operation
        defaultRule = op.getDefaultBusinessRule();
        out.print("Default Rule: ");
        out.println(defaultRule.getName());
    }
} catch (BusinessRuleManagementException e)
{
    e.printStackTrace();
    out.println(e.getMessage());
}
return out.toString();
}
}

```

Example

Web browser output for example 8.

Executing example8

Business Rule Group before publish:

```

Business Rule Group: DiscountRules
Default Rule: calculateShippingDiscount

```

Business Rule Group after publish:

```

Business Rule Group: DiscountRules
Default Rule: calculateShippingDiscountHoliday

```

Example 9: Schedule another rule for an operation in a business rule group

In this example, a business rule is scheduled to be active for 1 hour from the time of publish for a specific operation.

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example9 {
    static Formatter out = new Formatter();

    static public String executeExample9()
    {
        try
        {
            out.clear();

            // Retrieve a business rule group by target namespace and name
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsByTNSAndName(
                    "http://BRSamples/com/ibm/websphere
                    /sample/brules",
                    QueryOperator.EQUAL,
                    "DiscountRules",

```

```

    QueryOperator.EQUAL, 0, 0);

if (brgList.size() > 0)
{
    out.println("");
    out.printlnBold("Business Rule Group before publish:");
    // Get the first business rule group from the list
    // This should be the only business rule group in the
    list as
    // the combination of target namespace and name are unique
    BusinessRuleGroup brg = brgList.get(0);

    // Get the operation of the business rule group that
    // will have a new business rule scheduled
    Operation op =
    brg.getOperation("calculateShippingDiscount");

    printOperationSelectionRecord(op);
    // Get the list of available business rules for this operation
    List<BusinessRule> ruleList =
    op.getAvailableTargets();

    // Get the first rule in the list as this will be scheduled
    // for the operation
    BusinessRule rule = ruleList.get(0);

    // Get the list of scheduled business rules
    OperationSelectionRecordList opList = op
    .getOperationSelectionRecordList();
    // Create an end date in the future for the business rule
    Date future = new Date();
    long futureTime = future.getTime() + 3600000;

```

For a new scheduled rule, a start date and end date can be specified along with the rule. If the start date is set to null, this indicates that the rule will be active immediate upon publish. If an end date is set to null, the rule will not have an end date. An overlap of schedules is not allowed and can be checked by calling the validate method on the operation.

```

    // Create the new scheduled business rule with the current
    // date which means this rule will become active immediately
    // upon
    // publish and the future date.
    newOperationSelectionRecord(new Date(),
    new Date(futureTime), rule);
    // Add the new scheduled business rule to the list of
    // scheduled rule
    opList.addOperationSelectionRecord(newRecord);

```

Validate operation to ensure that an overlap does not exist.

```

    // Validate the list to insure there isn't an overlap
    List<Problem> problems = op.validate();
    if (problems.size() == 0)
    {
        // Use the original list or create a new list
        // of business rule groups
        List<BusinessRuleGroup> publishList = new
        ArrayList<BusinessRuleGroup>();
        // Add the changed business rule group to the list
        publishList.add(brg);
        // Publish the list with the updated business
        rule group
        BusinessRuleManager.publish(publishList, true);
        out.println("");

        // Retrieve the business rule groups again to

```

```

        verify the
        // changes were published
        out.printlnBold("Business Rule Group after
        publish:");

        brgList =
        BusinessRuleManager.getBRGsByTNSAndName(
            "http://BRSamples/com/ibm/websphere
            /sample/brules",
            QueryOperator.EQUAL,
            "DiscountRules",
            QueryOperator.EQUAL, 0, 0);
        brg = brgList.get(0);

        op =
        brg.getOperation("calculateShippingDiscount");

        printOperationSelectionRecord(op);
    }
    // else handle the validation error
}
} catch (BusinessRuleManagementException e)
{
    e.printStackTrace();
    out.println(e.getMessage());
}
return out.toString();
}
/*
Method to print the operation selection record for an operation. The
start date and end date are printed as well as the name of the rule
artifact for the scheduled time.
*/
private static void printOperationSelectionRecord(Operation op)
{
    OperationSelectionRecordList opSelectionRecordList = op
        .getOperationSelectionRecordList();
    Iterator<OperationSelectionRecord> opSelRecordIterator =
    opSelectionRecordList
        .iterator();
    OperationSelectionRecord record = null;
    while (opSelRecordIterator.hasNext())
    {
        out.printlnBold("Scheduled Destination:");
        record = opSelRecordIterator.next();
        out.println("Start Date: " + record.getStartDate()
            + " - End Date: " + record.getEndDate());
        BusinessRule ruleArtifact = record.getBusinessRuleTarget();
        out.println("Rule: " + ruleArtifact.getName());
    }
}
}
}

```

Example

Web browser output for example 9.

Executing example9

Business Rule Group before publish:

Scheduled Destination:

Start Date: Thu Dec 01 00:00:00 CST 2005 - End Date: Sun Dec 25 00:00:00 CST 2005

Rule: calculateShippingDiscountHoliday

Business Rule Group after publish:

Scheduled Destination:

Start Date: Thu Dec 01 00:00:00 CST 2005 - End Date: Sun Dec 25 00:00:00 CST 2005

Rule: calculateShippingDiscountHoliday

Scheduled Destination:

Start Date: Mon Jan 07 21:08:31 CST 2008 - End Date: Mon Jan 07 22:08:31 CST 2008

Rule: calculateShippingDiscount

Example 10: Modify a parameter value in a template in a rule set

In this example a rule instance defined with a template is modified by changing a parameter value and then publish.

```
package com.ibm.websphere.sample.brules.mgmt;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;

public class Example10
{
    static Formatter out = new Formatter();

    static public String executeExample10()
    {
        try
        {
            out.clear();

            // Retrieve a business rule group by target namespace and
            // name
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsByTNSAndName(
                    "http://BRSamples/com/ibm/websphere
                    /sample/brules",
                    QueryOperator.EQUAL,
                    "ApprovalValues",
                    QueryOperator.EQUAL, 0, 0);
            if (brgList.size() > 0)
            {
                // Get the first business rule group from the list
                // This should be the only business rule group in the
                // list as
                // the combination of target namespace and name are
                // unique
                BusinessRuleGroup brg = brgList.get(0);
                // Get the operation of the business rule group that
                // has the business rule that will be modified as
                // the business rules are associated with a specific
                // operation
                Operation op = brg.getOperation("getApprover");

                // Get the business rule on the operation that will
                // be modified
                List<BusinessRule> ruleList =
                    op.getBusinessRulesByName(
                        "getApprover", QueryOperator.EQUAL, 0,
                        0);

                if (ruleList.size() > 0)
```

```

{
    out.println("");
    out.printlnBold("Rule set before publish:");
    // Get the rule to be modified. Rules are
    // unique by
    // target namespace and name, but for this
    // example
    // there is only one business rule named
    "getApprover"
    RuleSet ruleSet = (RuleSet) ruleList.get(0);
    out.print(RuleArtifactUtility.printRuleSet(rule
    Set));
}

```

All of the rules in a rule set are in a rule block. Only one rule block is supported and the `getFirstRuleBlock` method should be used to retrieve the rule block.

```

// A rule set has all of the rules defined in a
// rule block
RuleBlock ruleBlock =
ruleSet.getFirstRuleBlock();

Iterator<RuleSetRule> ruleIterator =
ruleBlock.iterator();

// Iterate through the rules in the rule block
// to find the
// rule instance called "LargeOrderApprover"
while (ruleIterator.hasNext())
{
    RuleSetRule rule = ruleIterator.next();
}

```

If a rule is not defined with a rule template, it only has a Web presentation that can be retrieved. No updates can be made to a rule that is not defined with a template. It is best to check if a rule has been defined with a template if the name of the rule is unknown.

```

// The rule must have been defined with a
// template
// in order for it to be changed. Check
// if the current
// rule is even based on a template.
if (rule instanceof
RuleSetTemplateInstanceRule)
{
}

```

Use the `TemplateInstance` object to create the rule.

```

// Get the rule template instance
RuleSetTemplateInstanceRule
templateInstance =
(RuleSetTemplateInstanceRule) rule;

// Check for the rule instance
// which matches
// the rule to modify
if
(templateInstance.getName().equals(
"LargeOrderApprover"))
{
}

```

For the template instance, only parameter values can be modified. The parameters are modified by retrieving the `ParameterValue` and setting it to the appropriate value. Because the `ParameterValue` is passed by reference, the update is made directly on the rule, rule set, and business rule group.

```

        // Get the parameter from the
        rule instance
        ParameterValue parameter =
        templateInstance
        .getParameterValue("par
        am2");

        // Modify the value of the
        parameter
        parameter.setValue("superviso
        r");
        break;
    }
}
// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// Add the changed business rule group to the list
publishList.add(brg);

// Publish the list with the updated business rule
group
BusinessRuleManager.publish(publishList, true);

out.println("");
// Retrieve the business rule groups again to verify
the
// changes were published
out.printlnBold("Rule set after publish:");

brgList = BusinessRuleManager
.getBRGsByTNSAndName(
    "http://BRSamples/com/ibm/websphere/sample/brules",
    QueryOperator.EQUAL, "ApprovalValues",
    QueryOperator.EQUAL, 0, 0);

brg = brgList.get(0);
op = brg.getOperation("getApprover");
ruleList = op.getBusinessRulesByName(
    "getApprover", QueryOperator.EQUAL, 0,0);

ruleSet = (RuleSet) ruleList.get(0);
out.print(RuleArtifactUtility.printRuleSet(ruleSet));
}
} catch (BusinessRuleManagementException e)
{
    e.printStackTrace();
    out.println(e.getMessage());
}
return out.toString();
}
}

```

Example

Web browser output for example 10.

Executing example10

Rule set before publish:

Rule Set

Name: getApprover

Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Rule: LargeOrderApprover
 Display Name: LargeOrderApprover
 Description: null
 Expanded User Presentation: If the number of items order is above 10 and the order is above \$5000, then it requires the approval of manager
 User Presentation: If the number of items order is above {0} and the order is above \${1}, then it requires the approval of {2}
 Parameter Name: param0
 Parameter Value: 10
 Parameter Name: param1
 Parameter Value: 5000
 Parameter Name: param2
 Parameter Value: manager
Rule: DefaultApprover
 Display Name: DefaultApprover
 Description: null
 Expanded User Presentation: approver = peer
 User Presentation: approver = {0}
 Parameter Name: param0
 Parameter Value: peer

Rule set after publish:

Rule Set

Name: getApprover
 Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
 Display Name: LargeOrderApprover
 Description: null
 Expanded User Presentation: If the number of items order is above 10 and the order is above \$5000, then it requires the approval of supervisor
 User Presentation: If the number of items order is above {0} and the order is above \${1}, then it requires the approval of {2}
 Parameter Name: param0
 Parameter Value: 10
 Parameter Name: param1
 Parameter Value: 5000
 Parameter Name: param2
 Parameter Value: supervisor
Rule: DefaultApprover
 Display Name: DefaultApprover
 Description: null
 Expanded User Presentation: approver = peer
 User Presentation: approver = {0}
 Parameter Name: param0
 Parameter Value: peer

Example 11: Add a new rule from a template to a rule set

In this example, a new rule is added from a template to a rule set. Before the new rule instance is created, parameters for the new rule instance are created.

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRuleTemplate;
```



```

import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example11
{
static Formatter out = new Formatter();

static public String executeExample11()
{
try
{
out.clear();
// Retrieve a business rule group by target namespace and
name
List<BusinessRuleGroup> brgList = BusinessRuleManager
.getBRGsByTNSAndName(
"http://BRSamples/com/ibm/websphere
/sample/brules",
QueryOperator.EQUAL,
"ApprovalValues",
QueryOperator.EQUAL, 0, 0);

if (brgList.size() > 0)
{
// Get the first business rule group from the list
// This should be the only business rule group in the
list as
// the combination of target namespace and name are
unique
BusinessRuleGroup brg = brgList.get(0);
// Get the operation of the business rule group that
// has the business rule that will be modified as
// the business rules are associated with a specific
// operation
Operation op = brg.getOperation("getApprover");

// Get the business rule on the operation that will
be modified
List<BusinessRule> ruleList =
op.getBusinessRulesByName(
"getApprover", QueryOperator.EQUAL, 0,0);

if (ruleList.size() > 0)
{
out.println("");
out.printlnBold("Rule set before publish:");
// Get the rule to be modified. Rules are unique by
// target namespace and name, but for this example
// there is only one business rule named
"getApprover"
RuleSet ruleSet = (RuleSet) ruleList.get(0);
out.print(RuleArtifactUtility.printRuleSet(rule
Set));
}
}
}
}
}

```

In order to add a new rule to the rule set, the appropriate template must be located in the rule set and an instance created from the template. The template can be located by name.

```

// Get the list of rule templates
ListRuleSetRuleTemplate> ruleTemplates =
ruleSet
.getRuleTemplates();

Iterator<RuleSetRuleTemplate> templateIterator
= ruleTemplates
.iterator();

```

```

while (templateIterator.hasNext())
{
    RuleSetRuleTemplate template =
    templateIterator.next();

    // Locate the template to use to create a
    new rule
    if
    (template.getName().equals("Template_LargeOrder"))
    {

```

For a template instance, a list of parameters must be created.

```

// Create a list for the parameters
for this template
// rule instance
List<ParameterValue> paramList =
new ArrayList<ParameterValue>();

// From the template definition,
get a specific parameter
// and set the value
Parameter param =
template.getParameter("param0");
ParameterValue paramValue = param
.createParameterValue("
20");

// Add parameter to the list
paramList.add(paramValue);

// Get the next parameter and set
the value
param = template.getParameter("param1");
paramValue =
param.createParameterValue("7500");

// Add parameter to the list
paramList.add(paramValue);

// Get the next parameter and set
the value
param =
template.getParameter("param2");
paramValue = param
.createParameterValue("
2nd-line manager");

// Add parameter to the list
paramList.add(paramValue);

```

With the parameters created, the template instance can be created.

```

// Create the template rule
instance with the parameter
// list
RuleSetTemplateInstanceRule
templateInstance = template
.createRuleFromTemplate
("ExtraLargeOrder",
paramList);
// Get the ruleblock for the rule
set
RuleBlock ruleBlock =
ruleSet.getFirstRuleBlock();

```

Once the template instance is created, it can be added to the ruleblock. Once it is added to the rule block it can be ordered among other template rule instances.

```
// Add the template rule to the
ruleblock
ruleBlock.addRule(templateInstance)
;

break;
}
}

// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// Add the changed business rule group to the
list
publishList.add(brg);

// Publish the list with the updated business
rule group
BusinessRuleManager.publish(publishList, true);

out.println("");

// Retrieve the business rule groups again to
verify the
// changes were published
out.printlnBold("Rule set after publish:");

brgList = BusinessRuleManager
.getBRGsByTNSAndName(
"http://BRSamples/com/ibm/websphere
/sample/brules",
QueryOperator.EQUAL,
"ApprovalValues",
QueryOperator.EQUAL, 0, 0);

brg = brgList.get(0);
op = brg.getOperation("getApprover");
ruleList = op.getBusinessRulesByName(
"getApprover", QueryOperator.EQUAL,
0, 0);

ruleSet = (RuleSet) ruleList.get(0);
out.print(RuleArtifactUtility.printRuleSet(rule
Set));
}
}
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
return out.toString();
}
}
```

Example

Web browser output for example 11.

Executing example11

Rule set before publish:

Rule Set

Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and the order is above \$5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order is above \${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer

Rule set after publish:**Rule Set**

Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and the order is above \$5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order is above \${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer
Rule: ExtraLargeOrder
Display Name:
Description: null
Expanded User Presentation: If the number of items order is above 20 and the order is above \$7500, then it requires the approval of 2nd-line manager
User Presentation: If the number of items order is above {0} and the order is above \${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 20
Parameter Name: param1
Parameter Value: 7500
Parameter Name: param2
Parameter Value: 2nd-line manager

Example 12: Modify a template in a decision table by changing a parameter value and then publish

In this example, a condition and action, both defined with templates, are modified in a decision table by changing the parameter values before it is published.

The easiest way to modify conditions and actions in a decision table is to use unique names for the templates at each condition level and for each action. The unique names can be searched for and then changes can be made to template instances defined with the template. When changes are made to a template instance of a particular template, all of the condition values defined with that template at that level will be updated. For action expressions, each instance is unique and a change to one does not change others.

For this example, there are a number of additional methods that were created to simplify the locating of a specific case edge for update, finding the specific parameter value, and finding the action expression defined with a specific template.

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Vector;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example12 {
    static Formatter out = new Formatter();

    static public String executeExample12()
    {
        try
        {
            out.clear();
            // Retrieve a business rule group by target namespace and
            name
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsByTNSAndName(
                    "http://BRSamples/com/ibm/websphere
                    /sample/brules",
                    QueryOperator.EQUAL,
                    "ConfigurationValues",
                    QueryOperator.EQUAL, 0, 0);

            if (brgList.size() > 0)
            {
                // Get the first business rule group from the list
                // This should be the only business rule group in the
                list as
                // the combination of target namespace and name are
                unique
                BusinessRuleGroup brg = brgList.get(0);
            }
        }
    }
}
```

```

// Get the operation of the business rule group that
// has the business rule that will be modified as
// the business rules are associated with a specific
// operation
Operation op = brg.getOperation("getMessages");

// Get all business rules available for this
operation
List<BusinessRule> ruleList =
op.getAvailableTargets();

// For this operation there is only 1 business rule
and
// it is the business that we want to update
DecisionTable decisionTable = (DecisionTable)
ruleList.get(0);
out.println("");
out.printlnBold("Decision table before publish:");
out
.print(RuleArtifactUtility
.printDecisionTable(decisionT
able));

```

The init rule and condition and actions are contained in a tree block. With the tree block, the root node can be retrieved.

```

// Get the tree block that contains all of the
conditions
// and actions for the decision table
TreeBlock treeBlock = decisionTable.getTreeBlock();
// From the tree block, get the tree node which is
the
// starting point for navigating through the decision
table
TreeNode treeNode = treeBlock.getRootNode();

```

The condition to be updated was defined with a template by the name of "Condition Value Template 2.1". The method `getCaseEdge` will recursively search from the `TreeNode` down to the appropriate case edge to find the case edge where the template is defined. The method expects that the level at which the template is defined is known as well as the current level. This method can be used to find the case edge with a template by a specific name in case the same name is used at multiple case edges.

```

// Find the case edge at level 1 below the root with
// specific template with a parameter value that has
// a specific name. Since we are starting at the top,
// the current depth is 0
CaseEdge caseEdge = getCaseEdge(treeNode, "param0",
"Condition Value Template 2.1", 1, 0);

```

With the case edge found, the `ConditionValueTemplateInstance` for the condition can be retrieved.

```

if (caseEdge != null)
{
// Case edge was found. Get the value
definition of the
// case edge
TreeConditionValueDefinition condition =
caseEdge
.getValueDefinition();
// Get the condition expression defined with a
template

```

```

TemplateInstanceExpression conditionExpression
= condition
    .getConditionValueTemplateInstance(
);

```

With the `ConditionValueTemplateInstance`, the appropriate parameter value can be retrieved and then updated with the `getParameterValue` method.

```

// Get the template for the expression
Template conditionTemplate =
conditionExpression
    .getTemplate();

// Check that template is correct as it is
// possible to have
// multiple templates for a condition value,
// but only one
// applied
if (conditionTemplate.getName().equals(
"Condition Value Template 2.1"))
{
// Get the parameter value
ParameterValue parameterValue =
getParameterValue("param0",
    conditionExpression);

// Set the new parameter value
parameterValue.setValue("info");
}

```

The different action expressions defined with templates that need to be updated can then be retrieved. The `getActionExpressions` method will return all actions that are defined with the template by name `Action Value Template 1`.

```

ConditionNode conditionNode = (ConditionNode)
treeNode;

// Get the case edges tree node
List<CaseEdge> caseEdges =
conditionNode.getCaseEdges();

// Create a list to hold all of the action
// expressions that
// also need to be updated. Because every
// action is
// independent of other action even though the
// template is
// shared, all must be updated.
List<TemplateInstanceExpression> expressions =
new Vector<TemplateInstanceExpression>();

// Retrieve all of the expressions
for (CaseEdge edge : caseEdges)
{
    getActionExpressions("Action Value
    Template 1", edge,
    expressions);
}

```

With the list of action expressions, each item can be updated. For action expressions defined with templates the correct parameter value can be updated.

```

// Update the correct parameter in each
// expression
for (TemplateInstanceExpression expression
expressions)
{

```

```

        for (ParameterValue parameterValue :
            expression
                .getParameterValues())
        {
            // Check for correct parameter
            // although there is
            // only one parameter in our
            // template
            if
            (parameterValue.getParameter().getName().equals("param0")) {
                String value =
                    parameterValue.getValue();
                parameterValue.setValue("Info
                    "
                    +
                    value.substring(value.
                        indexOf(":"),
                        value.length()));
            }
        }
    }
    // With the condition value and actions
    // updated, the
    // business rule group can be published.
    // Use the original list or create a new list
    // of business rule groups
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // Add the changed business rule group to the
    // list
    publishList.add(brg);

    // Publish the list with the updated business
    // rule group
    BusinessRuleManager.publish(publishList, true);

    out.println("");

    // Retrieve the business rule groups again to
    // verify the
    // changes were published
    out.printlnBold("Decision table after
    publish:");

    brgList =
    BusinessRuleManager.getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere
        /sample/brules",
        QueryOperator.EQUAL,
        "ConfigurationValues",
        QueryOperator.EQUAL, 0, 0);

    brg = brgList.get(0);
    op = brg.getOperation("getMessages");
    ruleList = op.getAvailableTargets();

    decisionTable = (DecisionTable)
    ruleList.get(0);
    out.print(RuleArtifactUtility
        .printDecisionTable(decisionTable))
    ;
    }
}
} catch (BusinessRuleManagementException e)
{

```



```

        e.printStackTrace();
        out.println(e.getMessage());
    }
    return out.toString();
}

/*
Method to recursively navigate through a decision table and locate a
case
edge that has a template with a specific name and contains a specific
parameter to change. This method assumes that the level(depth) in the
decision table of the value that is to be changed is known and the
current level(currentDepth) is tracked *
*/
static private CaseEdge getCaseEdge(TreeNode node, String pName,
    String templateName, int depth, int currentDepth)
{
    // Check if the current node is an action. This is an indication
    // that this branch of the decision table has been exhausted
    // looking for the case edge
    if (node instanceof ActionNode)
    {
        return null;
    }

    // Get the case edges for this node
    List<CaseEdge> caseEdges = ((ConditionNode) node).getCaseEdges();
    for (CaseEdge caseEdge : caseEdges)
    {

        // Check if the correct level has been reached
        if (currentDepth < depth)
        {
            // Move down one level and then call getCaseEdge
            // again
            // to process that level
            currentDepth++;
            return getCaseEdge(caseEdge.getChildNode(), pName,
                templateName, depth, currentDepth);
        } else
        {
            // The correct level has been reached. Get the
            // condition in
            // order to check the templates on that condition on
            // whether
            // they match the template sought
            TreeConditionValueDefinition condition = caseEdge
                .getValueDefinition();

            // Get the expression for the condition which has
            // been defined
            // with a template
            TemplateInstanceExpression expression = condition
                .getConditionValueTemplateInstance();
            // Get the template from the expression
            Template template = expression.getTemplate();

            // Check if this is the template sought
            if (template.getName().equals(templateName))
            {
                // The template is found to match
                return caseEdge;
            } else
            {
                caseEdge = null;
            }
        }
    }
    return null;
}

```

```

}

/*
This method will check the different parameter values for an expression
and if the correct one is found, return that parameter value.
*/
private static ParameterValue getParameterValue(String pName,
    TemplateInstanceExpression expression)
{
    // Check that the expression is not null as null would indicate
    // that the expression that was passed in was probably not
    // defined
    // with a template and does not have any parameters to check.
    if (expression != null) {
        // Get the parameter values for the expression
        List<ParameterValue> parameterValues = expression
            .getParameterValues();

        for (ParameterValue parameterValue : parameterValues)
        {
            // For the different parameters, check that it
            // matches the
            // parameter value sought

            if
                (parameterValue.getParameter().getName().equals(pName
                ))
            {
                // Return the parameter value that matched
                return parameterValue;
            }
        }
    }
    return null;
}
/*
This method finds all of the action expressions that are
defined with a specific template. It recursively works through
a case edge and adds action expressions that match to the
expressions parameter.
*/

private static void getActionExpressions(String templateName,
CaseEdge next, List<TemplateInstanceExpression>
expressions)
{
    ActionNode actionNode = null;
    TreeNode treeNode = next.getChildNode();

    // Check if the current node is at the action node level
    if (treeNode instanceof ConditionNode)
    {
        List<CaseEdge> caseEdges = ((ConditionNode) treeNode)
            .getCaseEdges();

        Iterator<CaseEdge> caseEdgesIterator =
            caseEdges.iterator();

        // Work through all case edges to find the action
        // expressions
        while (caseEdgesIterator.hasNext())
        {
            getActionExpressions(templateName,
                caseEdgesIterator.next(),
                expressions);
        }
    } else {

```

```

// ActionNode found
actionNode = (ActionNode) treeNode;

List<TreeAction> treeActions = actionNode.getTreeActions();
// Check that there is at least one treeAction specified
for
// the expression and work through the expressions checking
// if the expressions have been created with the specific
// template.
if (!treeActions.isEmpty())
{

    Iterator<TreeAction> iterator =
treeActions.iterator();

while (iterator.hasNext())
{
    TreeAction treeAction = iterator.next();
    TemplateInstanceExpression expression =
treeAction
    .getValueTemplateInstance();

    Template template = expression.getTemplate();

    if (template.getName().equals(templateName))
    {
        // Expression found with matching
        template
        expressions.add(expression);
    }
}
}
}
}
}
}
}
}

```

Example

Web browser output for example 12.

Executing example12

Rule set before publish:

Decision Table

Name: getMessages

Namespace: <http://BRSamples/com/ibm/websphere/sample/brules>

Decision table after publish:

Decision Table

Name: getMessages

Namespace: <http://BRSamples/com/ibm/websphere/sample/brules>

Example 13: Add a condition value and actions to a decision table

In this example, a condition value and action are added to a decision table. Condition values can be added to a decision table through the use of a template.

When adding a condition value to a condition node, you are adding a case edge. The new case edge is added at the end of the list of case edges. For the condition value, you must specify a template instance expression that has the appropriate parameter values set. In order to specify the template instance expression you will have to use a specific template. It is recommended to give the template names at each condition node level unique names in order to retrieve the correct templates

for that type of condition. If a single template definition is used, it may make it difficult to determine at which level the condition is being added.

When setting the condition value in a condition node, this will add the condition value with the same template instance to all condition nodes at the same level. This is done as the decision table is balanced. Also as part of the adding a new condition value, new action nodes will be added. These action nodes have tree actions that have null values specified for the user presentation and template instance expression. Because the condition value can be added to a condition node that does not have an action node as a child node, the addition of a condition node may result in a large number of action nodes. The number of action nodes is based upon the level the condition node is added and the number of condition nodes at that level and the number of condition nodes at each child level.

In order to find the action nodes that have been created, a search of action nodes with tree actions that have null user presentation and template instance expression may be performed. A `TreeActionValueTemplate` can be used to create an expression that can be set into the `TreeAction`. This pattern would need to be repeated for all new action nodes.

For this example two methods were provided to assist in setting up the new tree actions. `getEmptyActionNode` recursively looks for an empty action node from the current condition node and `getParameterValue` returns the value of a parameter that was specified by name.

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionTermDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionValueTemplate;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueTemplate;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example13
{
    static Formatter out = new Formatter();

    static public String executeExample13()
    {
        try
        {
            out.clear();
        }
    }
}
```

```

// Retrieve a business rule group by target namespace
// and name
List<BusinessRuleGroup> brgList = BusinessRuleManager
.getBRGsByTNSAndName(
"http://BRSamples/com/ibm/websphere/sample/brules",
QueryOperator.EQUAL,"ConfigurationValues",
QueryOperator.EQUAL, 0, 0);

if (brgList.size() > 0)
{
// Get the first business rule group from the
// list. This should be the only business
// rule group in the list as the combination
// of target namespace and name are unique
BusinessRuleGroup brg = brgList.get(0);

// Get the operation of the business rule
// group that has the business rule that will
// be modified as the business rules are
// associated with a specific operation
Operation op = brg.getOperation("getMessages");

// Get all business rules available for
// this operation
List<BusinessRule> ruleList =
op.getAvailableTargets();

// For this operation there is only 1 business
// rule and it is the business that we want
// to update

DecisionTable decisionTable = (DecisionTable)
ruleList.get(0);
out.printlnBold("Decision table before
publish:");
out.print(RuleArtifactUtility
.printDecisionTable(decisionTable));

```

The level at which the condition value is going to be added needs to be located. This is typically passed as parameter as the user interface or application using the classes knows where to add the condition.

```

// Get the tree block that contains all of
// the conditions and actions for the decision
// table
TreeBlock treeBlock =
decisionTable.getTreeBlock();

// From the tree block, get the tree node which
// is the starting point for navigating through
// the decision table
ConditionNode conditionNode = (ConditionNode)
treeBlock.getRootNode();

// Get the case edges for this node which is
// the first level of conditions
List<CaseEdge> caseEdges =
conditionNode.getCaseEdges();

// Get the case edge which will have the new
// condition added
CaseEdge caseEdge = caseEdges.get(0);

// For the case edge get the condition node in
// order to retrieve the templates for the
// condition

```

```

conditionNode = (ConditionNode)
    caseEdge.getChildNode();

// Get the templates for the condition
List<TreeConditionValueTemplate>
treeValueConditionTemplates = conditionNode
    .getAvailableValueTemplates();

Iterator<TreeConditionValueTemplate>
treeValueConditionTemplateIterator =
treeValueConditionTemplates.iterator();

TreeConditionValueTemplate conditionTemplate =
    null;

```

By using unique template names at each condition node level in the decision table, you can more easily ensure the condition value is being added at the correct condition node value.

```

// Find the template that should be used
while
(treeValueConditionTemplateIterator.hasNext())
{
conditionTemplate =
treeValueConditionTemplateIterator
    .next();
if (conditionTemplate.getName().equals(
"Condition Value Template
2.1"))
{
// Template found
break;
}
conditionTemplate = null;
}
if (conditionTemplate != null)
{

```

With the correct template found, an instance can be created and the appropriate parameter value set before it is added to the condition node.

```

// Get the parameter definition from the
// template
Parameter conditionParameter =
conditionTemplate.getParameter("param0");

// Create a parameter value instance to
// be used in a new condition template
// instance
ParameterValue conditionParameterValue =
conditionParameter
    .createParameterValue("fatal");

List<ParameterValue>
conditionParameterValues = new
ArrayList<ParameterValue>();

// Add the parameter value to a list

conditionParameterValues
    .add(conditionParameterValue);

// Create a new condition template
// instance with the parameter value
TemplateInstanceExpression
newConditionValue =
conditionTemplate

```

```

        .createTemplateInstanceExpression(c
            onditionParameterValues);
// Add the condition template instance to
// this condition node
conditionNode

.addConditionValueToThisLevel(newConditionValue);
// When a condition node is added there
// are new action nodes that are created
// and empty. These must be filled with
// action template instances. By
// searching for each empty action
// node from the parent level, all of the
// new empty action nodes can be found.
conditionNode = (ConditionNode)
conditionNode.getParentNode();

```

With the condition value added to the condition node, the tree actions in the new action nodes must be set with a `TreeActionValueTemplate`. First locate the empty action node for the case edges. Use the parent condition node to ensure that as you iterate through the condition nodes, you will pick up all action nodes.

```

// Get the case edges for the parent node
caseEdges = conditionNode.getCaseEdges();

Iterator<CaseEdge> caseEdgesIterator =
caseEdges.iterator();

while (caseEdgesIterator.hasNext())
{
    // For each case edge, retrieve an
    // empty action node if it exists
    ActionNode actionNode =
    getEmptyActionNode(caseEdgesIterator
        .next());

    // Check if all actions are filled
    if (actionNode != null)
    {

```

When an action node is found with empty tree actions, the tree action must be set with a `TreeActionValueTemplate`. First locate the template and then specify the parameters before creating a template instance. With the template instance created, the tree action can be updated. For this example the parameter was set with a value from another tree action in another action node under the same condition node. For other decision tables where another tree action might not have a value that may be used to create the new parameter values, the value will have to be passed as a parameter from the application.

```

// Get the list of tree
// actions. These
// are not the actual
// actions, but the
// placeholders for the
// actions
List<TreeAction>
treeActionList = actionNode
    .getTreeActions();

List<TreeActionTermDefinition>
treeActionTermDefinitions =
    treeBlock
    .getTreeActionTermDefinitions();

List<TreeActionValueTemplate>
treeActionValueTemplates =

```

```

treeActionTermDefinitions
.get(0).getValueTemplates();

TreeActionValueTemplate
actionTemplate = null;

for (TreeActionValueTemplate
tempActionTemplate :
treeActionValueTemplates)
{

    if
(tempActionTemplate.get
Name().equals(
"Action Value
Template 1"))
    {
        actionTemplate =
tempActionTemplate;
        break;
    }
}

if (actionTemplate != null)
{
    // Get another action
    // that is under
    // the parent condition
    // node in order
    // to use the value as
    // the basis for
    // the error message in
    // the new
    // action node. Move up
    // to the
    // parent condition
    // node first
    ConditionNode
parentNode =
(ConditionNode)
actionNode
.getParentNode();

    // Get the first case
    // edge of the
    // parent node as this
    // action will
    // always be filled in
    // as new actions
    // are added to the end
    // of the case
    // edge list.
    CaseEdge caseE =
parentNode.getCas
eEdges().get(
0);

    // The child node is an
    // action node
    // and at the same
    // level as the new
    // action node.
    ActionNode aNode =
(ActionNode) caseE
.getChildNode();

    // Get the list of tree

```



```

// actions
TreeAction
existingTreeAction =
aNode
.getTreeActions()
.get(0);

// Get the template
// instance
// expression for the
// tree action
// from which you can
// retrieve the
// parameter

TemplateInstanceExpression
existingExpression =
existingTreeAction
.getValueTemplateInstance();

ParameterValue
existingParameterValue =
getParameterValue(
"param0",
existingExpression);

String actionValue =
existingParameterValue
.getValue();

// Create the new
// message from the
// message of the
// existing
// tree action
actionValue = "Fatal"
+
actionValue.substring(actionValue
.indexOf(":"), actionValue
.length());
Parameter
actionParameter =
actionTemplate
.getParameter("param0");

// Get the parameter
// from the template
ParameterValue
actionParameterValue =
actionParameter
.createParameterValue(actionValue);

// Add the parameter to
// a list of templates
List<ParameterValue>
actionParameterValues = new
ArrayList<ParameterValue>();

actionParameterValues.add(actionParameterValue);

// Create a new tree
// action instance

TemplateInstanceExpression
treeAction = actionTemplate
.createTemplateInstanceExpression(actionParameterValues);

```

```

        // Set the tree action
        // in the action node
        // by setting it in the
        // tree action list

```

Here the tree action in the action node is updated.

```

        treeActionList.get(0)
        .setValueTemplateInstance(
        treeAction);
    }
}
// With the condition value and actions
// updated, the business rule group can be
// published.
// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

// Add the changed business rule group to the
// list
publishList.add(brg);

// Publish the list with the updated business
// rule group

BusinessRuleManager.publish(publishList, true);

brgList =
    BusinessRuleManager.getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere/sample/brules",
        QueryOperator.EQUAL, "ConfigurationValues",
        QueryOperator.EQUAL, 0, 0);
brg = brgList.get(0);
op = brg.getOperation("getMessages");
ruleList = op.getAvailableTargets();
decisionTable = (DecisionTable)
    ruleList.get(0);
out.printlnBold("Decision table after
    publish:");
out
    .print(RuleArtifactUtility
        .printDecisionTable(decisionTable));
}
} catch (ValidationException e)
{
    List<Problem> problems = e.getProblems();

    out.println("Problem = " +
        problems.get(0).getErrorType().name());

    e.printStackTrace();
    out.println(e.getMessage());
} catch (BusinessRuleManagementException e)
{
    e.printStackTrace();
    out.println(e.getMessage());
}
}
return out.toString();
}

/*
 * This method searches from the current case edge for any
 * action nodes that have empty tree actions. An empty

```

```

* action node is found by looking at the end of the list
* of case edges and checking if the action node has tree
* actions that have both a null user presentation and
* TemplateInstanceExpression.
*/
private static ActionNode getEmptyActionNode(CaseEdge next)
{
    ActionNode actionNode = null;
    TreeNode treeNode = next.getChildNode();

    if (treeNode instanceof ConditionNode)
    {
        List<CaseEdge> caseEdges = ((ConditionNode) treeNode)
            .getCaseEdges();

        if (caseEdges.size() > 1)
        {
            // Get right-most case-edge as the new
            // condition and thus empty actions are at the
            // right-end of the case edges
            actionNode = getEmptyActionNode(caseEdges
                .get(caseEdges.size() - 1));

            if (actionNode != null)
            {
                return actionNode;
            }
        }
        else
        {
            actionNode = (ActionNode) treeNode;

            List<TreeAction> treeActions =
                actionNode.getTreeActions();

            if (!treeActions.isEmpty())
            {
                if
                ((treeActions.get(0).getValueUserPresentation() == null)
                    &&
                    (treeActions.get(0).getValueTemplateInstance() == null))
                {
                    return actionNode;
                }
            }
            actionNode = null;
        }
        return actionNode;
    }
}
/*
* This method will check the different parameter values for an
* expression and if the correct one is found, return that
* parameter value.
*/
private static ParameterValue getParameterValue(String pName,
    TemplateInstanceExpression expression)
{
    ParameterValue parameterValue = null;

    // Check that the expression is not null as null would
    // indicate that the expression that was passed in was
    // probably not defined with a template and does not have
    // any parameters to check.
    if (expression != null)
    {
        // Get the parameter vlues for the expression
        List<ParameterValue> parameterValues = expression

```

```

        .getParameterValues();
    Iterator<ParameterValue> parameterIterator =
        parameterValues
        .iterator();

    // For the different parameters, check that it
    // matches the parameter value sought
    while (parameterIterator.hasNext())
    {
        parameterValue = parameterIterator.next();

        if
        (parameterValue.getParameter().getName().equals(pName))
        {
            // Return the parameter value that
            // matched
            return parameterValue;
        }
    }
    return parameterValue;
}
}

```

Example

Web browser output for example 13.

Executing example13

Decision table before publish:

Decision Table

Name: getMessages

Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Decision table after publish:

Decision Table

Name: getMessages

Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Example 14: Handle errors in a rule set

This example focuses on how to catch problems in a rule set and find out what problem has occurred such that the appropriate message can be displayed or action can be taken to correct the situation.

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import
com.ibm.wbiserver.brules.mgmt.problem.ProblemStartDateAfterEndDate;
import com.ibm.wbiserver.brules.mgmt.problem.ValidationError;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import

```

```

com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example14 {
    static Formatter out = new Formatter();

    static public String executeExample14() {
        try {
            out.clear();

            // Retrieve a business rule group by target namespace and
            // name
            List<BusinessRuleGroup> brgList = BusinessRuleManager
                .getBRGsByTNSAndName(
                    "http://BRSamples/com/ibm/websphere
                    /sample/brules",
                    QueryOperator.EQUAL,
                    "ApprovalValues",
                    QueryOperator.EQUAL, 0, 0);

            if (brgList.size() > 0) {
                // Get the first business rule group from the list
                // This should be the only business rule group in the
                // list as
                // the combination of target namespace and name are
                // unique
                BusinessRuleGroup brg = brgList.get(0);
                out.println("Business Rule Group retrieved");

                // Get the operation of the business rule group that
                // has the business rule that will be modified as
                // the business rules are associated with a specific
                // operation
                Operation op = brg.getOperation("getApprover");

                // Retrieve specific rule by name
                List<BusinessRule> ruleList =
                    op.getBusinessRulesByName(
                        "getApprover", QueryOperator.EQUAL, 0,
                        0);

                // Get the specific rule
                RuleSet ruleSet = (RuleSet) ruleList.get(0);
                out.println("Rule Set retrieved");

                RuleBlock ruleBlock = ruleSet.getFirstRuleBlock();

                Iterator<RuleSetRule> ruleIterator =
                    ruleBlock.iterator();

                // Search through the rules to find the rule to
                // change
                while (ruleIterator.hasNext()) {
                    RuleSetRule rule = ruleIterator.next();

                    // Check that the rule was defined with a
                    // template
                    // as it can be changed.
                    if (rule instanceof
                        RuleSetTemplateInstanceRule) {
                        // Get the template rule instance
                        RuleSetTemplateInstanceRule
                            templateInstance =
                                (RuleSetTemplateInstanceRule) rule;
                        // Check for the correct template rule
                        // instance
                        if (templateInstance.getName().equals(
                            "LargeOrderApprover")) {

```

To cause a problem, this example sets a parameter to a value that is not compatible for the expression. The parameter is expecting an integer, but a string is passed in.

```

        // Get the parameter from the
        template instance
        ParameterValue parameter =
        templateInstance
            .getParameterValue("par
            am1");

        // Set an incorrect value for this
        parameter
        // This will cause a validation
        error
        parameter.setValue("$3500");
        out.println("Incorrect parameter
        value set");
        break;
    }
}
// This code should never be reached because of the
error
// introduced
// above

// With the condition value and actions updated, the
business
// rule
// group can be published.
// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// Add the changed business rule group to the list
publishList.add(brg);

// Publish the list with the updated business rule
group
BusinessRuleManager.publish(publishList, true);
}

```

A `ValidationException` can be caught and from the exception, the problems can be retrieved. For each problem, the error can be checked to determine which error has occurred. A message can be printed out or the appropriate action can be taken.

```

    } catch (ValidationException e) {
        out.println("Validation Error");

        List<Problem> problems = e.getProblems();

        Iterator<Problem> problemIterator = problems.iterator();

        // Check the list of problems for the appropriate error and
        // perform the appropriate action, for example report error
        // or correct error
        while (problemIterator.hasNext()) {
            Problem problem = problemIterator.next();
            ValidationError error = problem.getErrorType();

            // Check for specific error value
            if (error == ValidationError.TYPE_CONVERSION_ERROR) {
                // Handle this error by reporting the problem
                out
                    .println("Problem: Incorrect value
                    entered for a parameter");
            }
        }
    }
}

```

```

        return out.toString();
    }
    // else if...
    // Checks can be done for other errors and the
    // appropriate error message or action can be
    // performed
    // correct the problem
}
} catch (BusinessRuleManagementException e) {
    out.println("Error occurred.");
    e.printStackTrace();
}
return out.toString();
}
}

```

Example

Web browser output for example 14.

Executing example14

```

Business Rule Group retrieved
Rule Set retrieved
Validation Error
Problem: Incorrect value entered for a parameter

```

Example 15: Handle errors in a business rule group

This example is similar to example 14 as it shows how to handle problems that occur when a business rule group is published. It shows how the problem can be determined and the correct message can be printed or action performed.

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import com.ibm.wbiserver.brules.mgmt.problem.ProblemStartDateAfterEndDate;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example15
{
    static Formatter out = new Formatter();

    static public String executeExample15()
    {
        try
        {
            out.clear();

```

```

// Retrieve a business rule group by target namespace and
name
List<BusinessRuleGroup> brgList = BusinessRuleManager
.getBRGsByTNSAndName(
"http://BRSamples/com/ibm/websphere
/sample/brules",
QueryOperator.EQUAL,
"ApprovalValues",
QueryOperator.EQUAL, 0, 0);
if (brgList.size() > 0)
{
// Get the first business rule group from the list
// This should be the only business rule group in the
list as
// the combination of target namespace and name are
unique
BusinessRuleGroup brg = brgList.get(0);
out.println("Business Rule Group retrieved");

// Get the operation of the business rule group that
// has the business rule that will be modified as
// the business rules are associated with a specific
// operation
Operation op = brg.getOperation("getApprover");

// Retrieve specific rule by name
List<BusinessRule> ruleList =
op.getBusinessRulesByName(
"getApprover", QueryOperator.EQUAL, 0,
0);

// Get the specific rule
RuleSet ruleSet = (RuleSet) ruleList.get(0);
out.println("Rule Set retrieved");

RuleBlock ruleBlock = ruleSet.getFirstRuleBlock();

Iterator<RuleSetRule> ruleIterator =
ruleBlock.iterator();

// Search through the rules to find the rule to
change
while (ruleIterator.hasNext())
{
RuleSetRule rule = ruleIterator.next();

// Check that the rule was defined with a
template
// as it can be changed.
if (rule instanceof
RuleSetTemplateInstanceRule)
{
// Get the template rule instance
RuleSetTemplateInstanceRule
templateInstance =
(RuleSetTemplateInstanceRule) rule;

// Check for the correct template rule
instance
if (templateInstance.getName().equals(
"LargeOrderApprover"))
{
// Get the parameter from the
template instance
ParameterValue parameter =
templateInstance

```



```

        .getParameterValue("parameter1");

        // Set the value for this parameter
        // This value is in the correct
        // format and will
        // not cause a validation error
        parameter.setValue("4000");
        out.println("Rule set parameter
        value on set correctly");
        break;
    }
}
}

```

To ensure a rule set is correct, the validate method can be called. The validate method is available on all objects and will return a list of problems that can be checked to determine the problem. When calling validate on an object, the validate method is called on all contained objects as well.

```

// Validate the changes made the rule set
List<Problem> problems = ruleSet.validate();
out.println("Rule set validated");

// No errors should occur for this test case, however,
// check if there are problems and then
// perform the correct action to recover or report
// the error
if (problems != null)
{
    Iterator<Problem> problemIterator =
    problems.iterator();

    while (problemIterator.hasNext())
    {
        Problem problem = problemIterator.next();

        if (problem instanceof
        ProblemStartDateAfterEndDate)
        {
            out
            .println("Incorrect
            value entered for a
            parameter");
            return out.toString();
        }
    }
} else
{
    out.println("No problems found for the rule
    set");
}

// Get the list of available rule targets
List<BusinessRule> ruleList2 =
op.getAvailableTargets();

// Get the first rule that will be scheduled
incorrectly
BusinessRule rule = ruleList2.get(0);

// The error condition will be to set the end time
for a
// scheduled rule to be 1 hour before the start time
// This will cause a validation error
Date future = new Date();
long futureTime = future.getTime() - 360000;

```

```

// Get the operation selection list to add the
incorrectly
// scheduled item
OperationSelectionRecordList opList = op
.getOperationSelectionRecordList();

// Create a new scheduled rule instance
// No error is thrown until validated or a publish
// occurs as more changes might be made
OperationSelectionRecord newRecord = opList
.newOperationSelectionRecord(new Date(),
new Date(
futureTime), rule);

```

When the record is added with an incorrect set of dates, this does not cause an error. It is possible overlaps might occur or no selection records are set for the operation as things are in the process of being changed. The error will be found when the business rule group with the operation selection record is published. The validate method is called before the objects are published and exceptions will be thrown if any errors exists.

```

// Add the scheduled rule instance to the operation
// No error here either
opList.addOperationSelectionRecord(newRecord);
out.println("New selection record added with
incorrect schedule");

// With the condition value and actions updated, the
business
// rule
// group can be published.
// Use the original list or create a new list
// of business rule groups
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// Add the changed business rule group to the list
publishList.add(brg);

// Publish the list with the updated business rule
group
BusinessRuleManager.publish(publishList, true);
}
} catch (ValidationException e) {
out.println("Validation Error");

List<Problem> problems = e.getProblems();

Iterator<Problem> problemIterator = problems.iterator();
// There might be multiple problems
// Go through the problems and handle each one or
// report the problem
while (problemIterator.hasNext())
{
Problem problem = problemIterator.next();

// Each problem is a different type that can be
compared
if (problem instanceof ProblemStartDateAfterEndDate)
{
out
.println("Rule schedule is
incorrect. Start date is after end
date.");
return out.toString();
}
}

```

```

        // else if...
        // Checks can be done for other errors and the
        // appropriate error message or action can be
        // performed
        // correct the problem
    }
} catch (BusinessRuleManagementException e)
{
    out.println("Error occurred.");
    e.printStackTrace();
}
return out.toString();
}
}

```

Example

Web browser output for example 15.

Executing example15

```

Business Rule Group retrieved
Rule Set retrieved
Rule set parameter value on set correctly
Rule set validated
Validation Error
Rule schedule is incorrect. Start date is after end date.

```

Additional Query Examples

The following examples are not included with the application containing examples 1-15, however they provide more examples on creating queries to retrieve business rule groups.

In these examples different properties and wildcard values ('_', '%') are used with different operators (AND, OR, LIKE, NOT_LIKE, EQUAL and NOT_EQUAL).

Example

For the examples, queries will be performed that return between different combinations of 4 business rule groups. It is important to understand the different attributes and properties of the business rule groups as these are used in the queries.

```

Name: BRG1
Targetnamespace : http://BRG1/com/ibm/br/rulegroup
Properties:
organization, 8JAA
department, claims
ID, 00000567
region: SouthCentralRegion
manager: Joe Bean

```

```

Name: BRG2
Targetnamespace : http://BRG2/com/ibm/br/rulegroup
Properties:
organization, 7GAA
department, accounting
ID, 0000047
ID_cert45, ABC
region: NorthRegion

```

```

Name: BRG3
Targetnamespace : http://BRG3/com/ibm/br/rulegroup
Properties:
organization, 7FAB

```

```
department, finance
ID, 0000053
ID_app45, DEF
region: NorthCentralRegion
```

```
Name: BRG4
Targetnamespace : http://BRG4/com/ibm/br/rulegroup
Properties:
organization, 7HAA
department, shipping
ID, 0000023
ID_app45, GHI
region: SouthRegion
```

Query by a single property:

This is an example of a query by a single property.

```
List<BusinessRuleGroup> brgList = null;

brgList = BusinessRuleManager.getBRGsBySingleProperty(
    "department", QueryOperator.EQUAL,
    "accounting", 0, 0);
// Returns BRG2
```

Query business rule groups by properties and wildcard (%) at the beginning and at the end of the value:

This is an example of a query of business rule groups by properties and wildcard (%) at the beginning and at the end of the value.

```
// Query Prop AND Prop
QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
    "region", QueryOperator.LIKE,
    "%Region");

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode(
    "ID", QueryOperator.LIKE,
    "000005%");

QueryNode queryNode =
QueryNodeFactory.createAndNode(leftNode,
    rightNode);

brgList =
BusinessRuleManager.getBRGsByProperties(queryNode, 0, 0);
// Returns BRG1 and BRG3
```

Query business rule groups by properties and wildcard ('_'):

This is an example of a query of business rule groups by properties and wildcard (%).

```
brgList = BusinessRuleManager.getBRGsBySingleProperty("ID",
QueryOperator.LIKE, "00000_3", 0, 0);

// Returns BRG3 and BRG4
```

Query business rule group by properties with multiple wildcards ('_' and '%'):

This is an example of a query of business rule group by properties with multiple wildcards ('_' and '%').

```

brgList =
BusinessRuleManager.getBRGsBySingleProperty("region",
QueryOperator.LIKE, "__uth%Region",
0, 0);

// Returns BRG1 and BRG4

```

Query business rule groups by NOT_LIKE operator and wildcard ('_'):

This is an example of a query of business rule group by NOT_LIKE operator and wildcard ('_').

```

brgList =
BusinessRuleManager.getBRGsBySingleProperty("organization",
QueryOperator.NOT_LIKE,
"7__A", 0, 0);

// Returns BRG1 and BRG3

brgList =
BusinessRuleManager.getBRGsBySingleProperty("organization",
QueryOperator.NOT_LIKE,
"7%", 0, 0);

// Returns BRG1

```

Query business rule groups by NOT_EQUAL operator:

This is an example of a query of business rule group by the NOT_EQUAL operator.

```

brgList =
BusinessRuleManager.getBRGsBySingleProperty("department",
QueryOperator.NOT_EQUAL,
"claims", 0, 0);
// Returns BRG1

```

Query business rule groups by PropertyIsDefined:

This is an example of a query of business rule groups by PropertyIsDefined.

```

PropertyIsDefinedQueryNode node =
QueryNodeFactory.createPropertyIsDefinedQueryNode("manager"
);

brgList = BusinessRuleManager.getBRGsByProperties(node, 0,
0);

// Returns BRG1

```

Query business rule groups by NOT PropertyIsDefined:

This is an example of a query of business rule groups by NOT PropertyIsDefined.

```

// NOT Prop
QueryNode node =
QueryNodeFactory.createPropertyIsDefinedQueryNode("manager"
);

NotNode notNode = QueryNodeFactory.createNotNode(node);

brgList = BusinessRuleManager.getBRGsByProperties(notNode,
0, 0);

// Returns BRG1

```

Query business rule groups by multiple properties with a single NOT node:

This is an example of a query of business rule groups by multiple properties with a single NOT node.

```
// Prop AND NOT Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.EQUAL, "accounting");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID",
    QueryOperator.LIKE, "00000%");

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
    notNode);

brgList = BusinessRuleManager.getBrgsByProperties(andNode,
    0, 0);

// Returns BRG2
```

Query business rule groups by multiple properties with multiple NOT nodes combined with AND operator:

This is an example of a query business rule groups by multiple properties with multiple NOT nodes combined with AND operator.

```
// NOT Prop AND NOT Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.EQUAL, "accounting");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE, "cla%");

NotNode notNode2 =
QueryNodeFactory.createNotNode(leftNode);

AndNode andNode = QueryNodeFactory.createAndNode(notNode,
    notNode2);

brgList = BusinessRuleManager.getBrgsByProperties(andNode,
    0, 0);

// Returns BRG1 and BRG2
```

Query business rule groups by multiple properties with multiple NOT nodes combined with OR operator:

This is an example of a query business rule groups by multiple properties with multiple NOT nodes combined with OR operator.

```
// NOT Prop OR NOT Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE, "acc%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);
```

```

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
    "department", QueryOperator.EQUAL,
    "claims");

NotNode notNode2 =
QueryNodeFactory.createNotNode(leftNode);

OrNode orNode = QueryNodeFactory.createOrNode(notNode,
notNode2);

brgList = BusinessRuleManager.getBrgsByProperties(orNode,
0, 0);

//Returns BRG1, BRG2, BRG3, and BRG4

```

Query business rule groups by multiple properties combined with multiple AND operators:

This is an example of a query business rule groups by multiple properties combined with multiple AND operators.

```

// (Prop AND Prop) AND (Prop AND Prop)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE, "acc%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.EQUAL, "7GAA");

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(leftNode,rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("ID",
QueryOperator.LIKE,"000004_");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
QueryOperator.EQUAL,
    "NorthRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, rightNode2);

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft,andNodeRight);

brgList = BusinessRuleManager.getBrgsByProperties(andNode,
0, 0);

// Returns BRG2

```

Query business rule groups by multiple properties combined with AND and OR operators:

This is an example of a query business rule groups by multiple properties combined with AND and OR operators.

```

// (Prop AND Prop) OR (Prop AND NOT Prop)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE, "acc%");

QueryNode leftNode =

```

```

QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.EQUAL, "7GAA");

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(leftNode, rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.EQUAL, "8JAA");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE, "%1Region");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, notNode);

OrNode orNode = QueryNodeFactory.createOrNode(andNodeLeft,
andNodeRight);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// Returns BRG2 and BRG3

```

Query business rule groups by multiple properties combined with AND and NOT operators:

This is an example of a query business rule groups multiple properties combined with AND and NOT operators.

```

// Prop AND NOT (Prop AND Prop)
QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID",
QueryOperator.LIKE, "000005%");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.EQUAL,
    "8JAA");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region", QueryOper
ator.LIKE,
    "%1Region");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, rightNode2);

NotNode notNode =
QueryNodeFactory.createNotNode(andNodeRight);

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
notNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// Returns BRG3

```

Query business rule groups by multiple properties combined with NOT and OR operators:

This is an example of a query business rule groups by multiple properties combined with NOT and OR operators.

```
// NOT (Prop AND Prop) OR Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "8_A_");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
    "%1Region");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

NotNode notNode =
QueryNodeFactory.createNotNode(andNodeRight);

OrNode orNode = QueryNodeFactory.createOrNode(notNode,
rightNode);

brgList = BusinessRuleManager.getBrgsByProperties(orNode,
0, 0);

// Returns BRG3
```

Query business rule groups by multiple properties combined with nested AND operators:

This is an example of a query business rule groups by multiple properties combined with nested AND operators.

```
// Prop AND (Prop AND (Prop AND Prop))
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "__thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

PropertyIsDefinedQueryNode node2 =
QueryNodeFactory.createPropertyIsDefinedQueryNode("ID_cert4
5");

AndNode andNode = QueryNodeFactory.createAndNode(node2,
andNodeLeft);
```

```
brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);
// Returns BRG2
```

Query business rule groups by multiple properties combined with nested AND operators:

This is an example of a query business rule groups by multiple properties combined with nested AND operators.

```
// (Prop AND (Prop AND Prop)) AND Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
"__thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.LIKE,
"7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
QueryOperator.LIKE,
"%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_app45",QueryOp
erator.LIKE, "GH_");

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft, leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// Returns BRG4
```

Query business rule groups by multiple properties combined with nested AND operators and a NOT node:

This is an example of a query business rule groups by multiple properties combined with nested AND operators and a NOT node.

```
// Prop AND (Prop AND (Prop AND NOT Prop))
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.LIKE,
"7%");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
QueryOperator.LIKE,
"%1Region");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
```

```

QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,notNode);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
    QueryOperator.LIKE,
    "AB_");

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
andNodeLeft);

brgList = BusinessRuleManager.getBrgsByProperties (andNode,
0, 0);

// Returns BRG2

```

Query business rule groups by multiple properties combined with nested AND operators:

This is an example of a query business rule groups by multiple properties combined with nested AND operators.

```

// (Prop AND (Prop AND Prop)) AND Prop - Return empty
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "__thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
    QueryOperator.LIKE,
    "GH_");

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft, leftNode);

brgList = BusinessRuleManager.getBrgsByProperties (andNode,
0, 0);

//Returns no BRGs

```

Query business rule groups by multiple properties combined with nested OR operators:

This is an example of a query business rule groups by multiple properties combined with nested OR operators.

```
// (Prop OR (Prop OR Prop)) OR Prop

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "__thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(leftNode2, rightNode2);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(rightNode, orNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
    QueryOperator.LIKE,
    "GH_");

OrNode orNode = QueryNodeFactory.createOrNode(orNodeLeft,
leftNode);

brgList = BusinessRuleManager.getBrgsByProperties(orNode,
0, 0);

// Returns BRG1
```

Query business rule groups by multiple properties combined with nested OR operators:

This is an example of a query business rule groups by multiple properties combined with nested OR operators.

```
// (Prop OR (Prop OR NOT Prop)) OR Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "__thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(leftNode2, notNode);
```

```

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(rightNode,orNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
    QueryOperator.LIKE,
    "GH_");

OrNode orNode = QueryNodeFactory.createOrNode(orNodeLeft,
leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// Returns BRG3

```

Query business rule groups by multiple properties combined with nested OR operators and a NOT node:

This is an example of a query business rule groups by multiple properties combined with nested OR operators and a NOT node.

```

// Prop OR NOT(Prop OR Prop)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "__thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode(
    "organization",
    QueryOperator.LIKE,
    "7%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
    "department",
    QueryOperator.LIKE,
    "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(rightNode2,
    rightNode);

NotNode notNode =
QueryNodeFactory.createNotNode(orNodeRight);

OrNode orNodeLeft = QueryNodeFactory.createOrNode(leftNode,
notNode);

brgList =
BusinessRuleManager.getBRGsByProperties(orNodeLeft, 0, 0);

// Returns BRG3

```

Query business rule groups by multiple properties combined with nested OR operators and a NOT node:

This is an example of a query business rule groups by multiple properties combined with nested OR operators and a NOT node.

```

// NOT(Prop OR Prop) OR Prop
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "%1Region");

```

```

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode(
    "organization",
    QueryOperator.LIKE,
    "7%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
    "department",
    QueryOperator.LIKE,
    "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(rightNode2, rightNode);

NotNode notNode =
QueryNodeFactory.createNotNode(orNodeRight);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(notNode, leftNode);

brgList =
BusinessRuleManager.getBRGsByProperties(orNodeLeft, 0, 0);

// Returns BRG2 and BRG4

```

Query business rule groups by a list of nodes that are combined with an AND operator:

This is an example of a query business rule groups by a list of nodes that are combined with an AND operator.

```

// AND list
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7%");

list.add(rightNode2);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "7H%");

list.add(leftNode2);

AndNode andNode = QueryNodeFactory.createAndNode(list);

```

```
brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// Returns BRG4
```

Query business rule groups by a list of nodes and NOT node combined with an AND operator:

This is an example of a query business rule groups by a list of nodes and NOT node combined with an AND operator.

```
// AND list with a notNode
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "8%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

list.add(notNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",

list.add(leftNode2);

AndNode andNode = QueryNodeFactory.createAndNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// Return BRG4
```

Query business rule groups by a list of nodes that are combined with an OR operator:

This is an example of a query business rule groups by a list of nodes that are combined with an OR operator.

```
// OR list
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "%thRegion");

list.add(rightNode);
```

```

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "8%");

list.add(rightNode2);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

list.add(leftNode);

OrNode orNode = QueryNodeFactory.createOrNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

//Returns BRG3

```

Query business rule groups by a list of nodes and Not node combined with an OR operator:

This is an example of a query business rule groups by a list of nodes and Not node combined with an OR operator.

```

// OR list with Not node
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
    QueryOperator.LIKE,
    "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "8%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

list.add(notNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
    QueryOperator.LIKE,
    "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
    QueryOperator.LIKE,
    "8%");

list.add(leftNode2);

OrNode orNode = QueryNodeFactory.createOrNode(list);

```



```
brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

//Returns BRG1, BRG2, BRG3, and BRG4
```

Common operations classes

This section contains additional classes that were used in the examples to simplify common operations.

Formatter class

This class provides different methods to help with displaying the different examples. It adds different HTML tags to format the output.

```
package com.ibm.websphere.sample.brules.mgmt;

public class Formatter {

private StringBuffer buffer;

public Formatter()
{
    buffer = new StringBuffer();
}

public void println(Object o)
{
    buffer.append(o);
    buffer.append("<br>\n");
}

public void print(Object o)
{
    buffer.append(o);
}

public void printlnBold(Object o)
{
    buffer.append("<b>");
    buffer.append(o);
    buffer.append("</b><br>\n");
}

public void printBold(Object o)
{
    buffer.append("<b>");
    buffer.append(o);
    buffer.append("</b>");
}

public String toString()
{
    return buffer.toString();
}

public void clear()
{
    buffer = new StringBuffer();
}
}
```

RuleArtifactUtility class

This utility class has two public methods. The first public method is for printing out a decision table. This method makes use of a private method that uses recursion to print out the conditions and actions for the decision table. The second public method is for printing out a rule set.

```

package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.RuleTemplate;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTableRule;
import
com.ibm.wbiserver.brules.mgmt.dtable.DecisionTableTemplateInstanceRule;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionTermDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionTermDefinition;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRuleTemplate;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class RuleArtifactUtility
{
    static Formatter out = new Formatter();

    /*
    Method to print out a decision table with the conditions and
    actions printed out in a HTML tabular format. The conditions
    and actions are printed out with a separate method that
    recursively works through the case edges of the decision
    tables.
    */

    public static String printDecisionTable(BusinessRule
ruleArtifact)
    {
        out.clear();
        out.printlnBold("Decision Table");
        DecisionTable decisionTable = (DecisionTable)
ruleArtifact;
        out.println("Name: " +
decisionTable.getName());
        out.println("Namespace: " +
decisionTable.getTargetNameSpace());

        // Output the init rule for the decision table
        before
        // working through the table of conditions and
        actions
        DecisionTableRule initRule =
decisionTable.getInitRule();
        if (initRule != null)
        {
            out.printBold("Init Rule: ");
            out.println(initRule.getName());
        }
    }
}

```

```

out.println("Display Name: " +
initRule.getDisplayName());
out.println("Description: " +
initRule.getDescription());
// The expanded user presentation
will automatically populate the
// presentation with the parameter
values and can be used for
// display if the init rule was
defined with a template. If no
// template was defined the
expanded user presentation
// is the same as the regular
presentation.
out.println("Extended User
Presentation: "
+
initRule.getExpandedUse
rPresentation());
// The regular user presentation
will have placeholders in the
// string where the
// parameter can be substituted if
the init rule was defined with a
// template
// If the rule was not defined with
a template, the user
// presentation will only
// be a string without
placeholders. The placeholders are
of a
// format of {n} where
// n is the index (zero-based) of
the parameter in the template. This
// value
// can be used to create an
interface for editing where there
are
// fields with
// the parameter values available
for editing
out.println("User Presentation: " +
initRule.getUserPresentation());
// Init rules might be defined with
or without a template
// Check to make sure a template
was used before trying
// to access the parameters
if (initRule instanceof
DecisionTableTemplateInstanceRule)
{
    DecisionTableTemplateIn
stanceRule
templateInstance =
    (DecisionTableTemplateI
nstanceRule) initRule;

    RuleTemplate template =
templateInstance.getRul
eTemplate();

    List<Parameter>
parameters =
template.getParameters(
);
    Iterator<Parameter>
paramIterator =

```

```

        parameters.iterator();

        Parameter parameter =
            null;

        while
            (paramIterator.hasNext(
            )) {
            parameter =
                paramIterator.next();

            out.println("Parameter
            Name: " +
                parameter.getName());
            out.println("Parameter
            Value: "
                +
                templateInstance.getPar
                ameterValue(parameter
                .getName()));
            }
        }
    }
    // For the rest of the decision table, start at
    the root and
    // recursively work through the different case
    edges and
    // actions
    TreeBlock treeBlock =
        decisionTable.getTreeBlock();
    TreeNode treeNode = treeBlock.getRootNode();

    printDecisionTableConditionsAndActions(treeNode
        , 0);
    out.println("");
    return out.toString();
}
/*Method to recursively work through the case edges and print
out the conditions and actions.
*/
static private void printDecisionTableConditionsAndActions(
    TreeNode treeNode, int indent)
{
    out.print("<table border=\"1\">");
    if (treeNode instanceof ConditionNode)
    {
        // Get the case edges for the
        current TreeNode
        // and for each case edge print out
        the conditions
        ConditionNode conditionNode =
            (ConditionNode) treeNode;

        List<CaseEdge> caseEdges =
            conditionNode.getCaseEdges();
        Iterator<CaseEdge> caseEdgeIterator
            = caseEdges.iterator();

        CaseEdge caseEdge = null;

        while (caseEdgeIterator.hasNext())
        {
            out.print("<tr>");
            // If this is the start
            of the conditions for the
            // condition node,
            print out the condition term

```

```

if (indent == 0)
{
out.print("<td>");

TreeConditionTermDefinition
termDefinition =
conditionNode
.getTermDefinition();

out.print(termDefinitio
n.getUserPresentatio
n());
out.print("</td>");
indent++;
} else {
// After the condition
term has been printed
for a
// case edge skip for
the rest of the case
edges
out.print("<td></td>");
}

caseEdge =
caseEdgeIterator.next()
;

out.print("<td>");

// Check if the
caseEdge is defined by
a template
if
(caseEdge.getValueDefin
ition() != null)
{
TemplateInstanceExpress
ion templateInstance =
caseEdge
.getValueTemplateInstan
ce();

out.println(templateIns
tance.getExpandedUserPr
esentation());

TreeConditionValueDefin
ition valueDef =
caseEdge
.getValueDefinition();

out.println(valueDef.ge
tUserPresentation());

Template template =
templateInstance.getTem
plate();

// Get the parameters
for the template
definition and
// print out the
parameter names and
values
List<Parameter>
parameters =

```

```

template.getParameters(
);
Iterator<Parameter>
paramIterator =
parameters.iterator();

List<ParameterValue>
parameterValues =
templateInstance
.getParameterValues();
Iterator<ParameterValue
> paramValues =
parameterValues
.iterator();

Parameter parameter =
null;
ParameterValue
parameterValue = null;

while
(paramIterator.hasNext(
) &&
paramValues.hasNext())
{
parameter =
paramIterator.next();
parameterValue =
paramValues.next();

out.println("Parameter
Name: " +
parameter.getName());
out.println("Parameter
Value: "
+
parameterValue.getValue
());
}
}

out.print("</td><td>");
// Print the child node
for the caseEdge
printDecisionTableCondi
tionsAndActions(caseEdg
e.getChildNode(),
0);

out.print("</td></tr>")
;
}

// Add Otherwise condition if it
exists
TreeNode otherwise =
conditionNode.getOtherwiseCase();

if (otherwise != null)
{
out.print("<tr><td></td>
<td>Otherwise</td><td>
");
// Print the Otherwise
ConditionNode
printDecisionTableCondi

```

```

        tionsAndActions(otherwi
se, 0);
        out.print("</td></td>")
        ;
    }
    out.print("</table>");
} else {
    // ActionNode has been found and
    different logic is needed
    // to print out the TreeActions
    ActionNode actionNode =
    (ActionNode) treeNode;
    List<TreeAction> treeActions =
    actionNode.getTreeActions();

    Iterator<TreeAction>
    treeActionIterator =
    treeActions.iterator();

    TreeAction treeAction = null;

    // The ActionNode can contain
    multiple TreeActions to
    // print out
    while
    (treeActionIterator.hasNext())
    {
        out.print("<tr>");
        treeAction =
        treeActionIterator.next
        ();

        TreeActionTermDefinitio
n treeActionTerm =
        treeAction
        .getTermDefinition();

        if (indent == 0) {
            out.print("<td>");
            out.print(treeActionTer
m.getUserPresentation()
            );
            out.print("</td>");
        }
        out.print("<td>");
        TemplateInstanceExpress
ion templateInstance =
        treeAction
        .getValueTemplateInstan
ce();

        // Check that a
        template was specified
        for
        // the TreeAction
        before working with the
        // parameter name and
        values
        if (templateInstance !=
        null) {
            out.println(templateIns
tance.getExpandedUserPr
esentation());

            Template template =
            templateInstance.getTem
plate();

```

```

List<Parameter>
parameters =
template.getParameters(
);

Iterator<Parameter>
paramIterator =
parameters.iterator();

List<ParameterValue>
parameterValues =
templateInstance
.getParameterValues();
Iterator<ParameterValue
> paramValues =
parameterValues
.iterator();

Parameter parameter =
null;
ParameterValue
parameterValue = null;

while
(paramIterator.hasNext(
) &&
paramValues.hasNext())
{
parameter =
paramIterator.next();
parameterValue =
paramValues.next();

out.println(" Parameter
Name: " +
parameter.getName());
out.println(" Parameter
Value: "
+
parameterValue.getValue
());

}
} else
{
// If a template was
not used, the only item
that is
// available is the
UserPresentation if it
was
// specified when the
rule was created
out.print(treeAction.ge
tValueUserPresentation(
));
}

out.print("</td></tr>")
;
}
out.print("</table>");
}
}
/*
Method to print out a rule set

```



```

*/
public static String printRuleSet(BusinessRule
ruleArtifact)
{
    out.clear();
    out.printlnBold("Rule Set");
    RuleSet ruleSet = (RuleSet) ruleArtifact;
    out.println("Name: " + ruleSet.getName());
    out.println("Namespace: " +
ruleSet.getTargetNameSpace());

    // The rules in a rule set are contained in a
rule block
    RuleBlock ruleBlock =
ruleSet.getFirstRuleBlock();

    Iterator<RuleSetRule> ruleIterator =
ruleBlock.iterator();

    RuleSetRule rule = null;

    // Iterate through the rules in the rule block.
while (ruleIterator.hasNext())
{
    rule = ruleIterator.next();
    out.printBold("Rule: ");
    out.println(rule.getName());
    out.println("Display Name: " +
rule.getDisplayName());
    out.println("Description: " +
rule.getDescription());
    // The expanded user presentation
will automatically populate the
// presentation with the parameter
values and can be used for
// display if the rule was defined
with a template. If no
// template was defined the
expanded user presentation
// is the same as the regular
presentation.
    out.println("Expanded User
Presentation: "
+
rule.getExpandedUserPre
sentation());
    // The regular user presentation
will have placeholders in the
// string where the parameter can
be substituted if the rule
// was defined with a template. If
the rule was not defined with
// a template, the user
presentation will only be a string
// without placeholders. The
placeholders are of a format of {n}
// where n is the index (zerobased)
of the parameter in the
// template. This value can be used
to create an interface for
// editing where there are fields
with the parameter values
// available for editing
    out.println("User Presentation: " +
rule.getUserPresentation());

    // Check if the rule was defined

```

```

with a template
if (rule instanceof
RuleSetTemplateInstanceRule) {
    RuleSetTemplateInstance
    Rule templateInstance =
    (RuleSetTemplateInstanc
    eRule) rule;

    RuleSetRuleTemplate
    template =
    templateInstance
    .getRuleSetRuleTemplate
    ();

    List<Parameter>
    parameters =
    template.getParameters(
    );
    Iterator<Parameter>
    paramIterator =
    parameters.iterator();

    Parameter parameter =
    null;

    // Retrieve all of the
    parameters and output
    the name and value
    while
    (paramIterator.hasNext(
    ))
    {
        parameter =
        paramIterator.next();

        out.println("Parameter
        Name: " +
        parameter.getName());
        out.println("Parameter
        Value: "
        +
        templateInstance.getPar
        ameterValue(
        parameter.getName()).ge
        tValue());
    }
}
out.println("");
return out.toString();
}
}
}

```

Widget programming

Business Space provides widgets that are enabled for various business process management products from IBM. However, you can also create your own widgets, possibly integrating them with widgets from IBM. The following references explain why you would create your widgets, how you would create them and provides references to APIs that you could use in creating your own widgets.

In the Business Space information center, see these topics:

- Overview of developing widgets
- Widget programming guide

Developing client applications for business processes and tasks

You can use a modeling tool to build and deploy business processes and tasks. These processes and tasks are interacted with at runtime, for example, a process is started, or tasks are claimed and completed. You can use Business Process Choreographer Explorer to interact with processes and tasks, or the Business Process Choreographer APIs to develop customized clients for these interactions.

About this task

These clients can be Enterprise JavaBeans (EJB) clients, Web service clients, or Web clients that exploit the Business Process Choreographer Explorer JavaServer Faces (JSF) components. Business Process Choreographer provides Enterprise JavaBeans (EJB) APIs and interfaces for Web services for you to develop these clients. The EJB API can be accessed by any Java application, including another EJB application. The interfaces for Web services can be accessed from either Java environments or Microsoft .Net environments.

Comparison of the programming interfaces for interacting with business processes and human tasks

Enterprise JavaBeans (EJB), Web service, and Java Message Service (JMS), and Representational State Transfer Services (REST) generic programming interfaces are available for building client applications that interact with business processes and human tasks. Each of these interfaces has different characteristics.

The programming interface that you choose depends on several factors, including the functionality that your client application must provide, whether you have an existing end-user client infrastructure, whether you want to handle human workflows. To help you decide which interface to use, the following table compares the characteristics of the EJB, Web service, JMS, and REST programming interfaces.

	EJB interface	Web service interface	JMS message interface	REST interface
Functionality	This interface is available for both business processes and human tasks. Use this interface to build clients that work generically with processes and tasks.	This interface is available for both business processes and human tasks. Use this interface to build clients for a known set of processes and tasks.	This interface is available for business processes only. Use this interface to build messaging clients for a known set of processes.	This interface is available for both business processes and human tasks. Use this interface to build Web 2.0-style clients for a known set of processes and tasks.

	EJB interface	Web service interface	JMS message interface	REST interface
Data handling	<p>Supports remote artifact loading of schemas for accessing business object metadata.</p> <p>If the EJB client application is running in the same cell as the WebSphere Process Server that it connects to, the schemas that are needed for the business objects of the processes and tasks do not have to be available on the client, they can be loaded from the server using the remote artifact loader (RAL).</p> <p>RAL can also be used cross-cell if the client application runs in a full WebSphere Process Server installation. However, RAL cannot be used in a cross-cell setup where the client application runs in a WebSphere Process Server client installation.</p>	<p>Schema artifacts for input data, output data, and variables, must be available in an appropriate format on the client.</p>	<p>Schema artifacts for input data, output data, and variables, must be available in an appropriate format on the client.</p>	<p>Schema artifacts for input data, output data, and variables, must be available in an appropriate format on the client.</p>
Client environment	<p>A WebSphere Process Server installation or a WebSphere Process Server client installation.</p>	<p>Any runtime environment that supports Web service calls, including Microsoft .NET environments.</p>	<p>Any runtime environment that supports JMS clients, including SCA modules that use SCA JMS imports.</p>	<p>Any runtime environment that supports REST clients.</p>
Security	<p>Java Platform, Enterprise Edition (Java EE) security.</p>	<p>Web services security.</p>	<p>Java Platform, Enterprise Edition (Java EE) security for the WebSphere Process Server installation. You can also secure the queues where the JMS client application puts the API messages, for example, using WebSphere MQ security mechanisms.</p>	<p>Client application that call the REST methods must use an appropriate HTTP authentication mechanism.</p>

An operation can be exposed by multiple protocols. Observe the following general considerations if you use the same operation in different protocols.

- In Web service and REST interfaces, all object identifiers, such as PIID, AIID, and TKIID are represented by a string type. Only the EJB API interface expects a type-safe object ID.
- Operation overloading is only used for EJB methods and not for WSDL operations. In some cases, multiple WSDL operations exist, in other cases, only one WSDL operation exists that allows all of the parameter variations either by omission (`minOccurs="0"`), or null values (`nillable="true"`).
- In some EJB methods, XML namespaces and local names are passed as separate parameters. Most WSDL operations use the QName XML schema type to pass these parameters.
- Asynchronous interactions with long-running WSDL request-response operations, such as the `callWithReplyContext` operation in the EJB interface or the `callAsync` operation in the WSDL interface, are represented by the `call` operation in the JMS interface.

- The EJB interface returns a set of API objects, which expose getter and setter methods for the contained fields. Web service and REST interfaces return complex-typed (XML or JSON) documents to the client.
- Some Human Task Manager services operating on human tasks are also available as Business Flow Manager services operating on activities that call a human task.

Related tasks

Developing EJB client applications

The EJB APIs provide a set of generic methods for developing EJB client applications for working with the business processes and human tasks that are installed on a WebSphere Process Server.

Developing Web service API client applications

You can develop client applications that access business process applications and human task applications through the Business Process Choreographer Web services APIs. The client application development process consists of a number of mandatory and optional steps, including generating a Web service proxy and adding security and transaction policies to the client application.

Developing JMS client applications

You can develop client applications that access business process applications asynchronously through the Java Messaging Service (JMS) API.

Queries on business process and task data

Instance data for long-running business processes and human tasks are stored persistently in the database and are accessible by queries. Also, template data for business process templates and human task templates can be accessed using a query interface.

The EJB query interfaces, query API, and query table API, are available with Business Process Choreographer.

Depending on the clients that access process or task related data, one or more of the interfaces can be the right choice. REST and Web services APIs are available in Business Process Choreographer for querying task and process list data. However, for high volume process list and task list queries, use the Business Process Choreographer EJB query table API or REST query table API for performance reasons.

Comparison of the programming interfaces for retrieving process and task data

Business Process Choreographer provides a query table API and a query API for retrieving process and task data. Each of these interfaces has different characteristics.

The query interface that you choose depends on several factors, including the functionality that your client application must provide, whether you have an existing end-user client infrastructure, and performance considerations. To help you decide which interface to use, the following table compares the characteristics of the query table and the query programming interfaces.

Characteristic	query table API	query API
Availability	The query table API is available for the Business Flow Manager EJB interface and the REST programming interface.	The query API is available for EJB, Web service, JMS, and REST programming interfaces.

Characteristic	query table API	query API
Methods for content retrieval	The API provides the following methods: <ul style="list-style-type: none"> • queryEntities • queryEntityCount • queryRows • queryRowCount 	The API provides the following methods: <ul style="list-style-type: none"> • query • queryAll • queryProcessTemplates • queryTaskTemplates
Methods for meta data retrieval	The API provides the following methods: <ul style="list-style-type: none"> • getQueryTableMetaData • findQueryTableMetaData 	The API provides the following methods: <ul style="list-style-type: none"> • QueryResultSet.getColumnType
Query table name	Specifies the query table on which the query table API is run. Only one query table can be queried at any one time. For example, queryEntities("CUST.TASKS", ...).	The SELECT clause specifies the columns and predefined database views on which the query runs. This specification is similar to an SQL select clause. For example, query("TASK.TKIID, TASK.STATE, WORK_ITEM.REASON", ...).
SELECT clause and selected attributes	Use the filter options of the query table API to specify the attributes that the query is to return. Because the query is run against one query table, the attributes are uniquely identifiable by their names.	Use the SELECT clause to specify attributes. The syntax of the attribute name is: <i>view_name.attribute_name</i> . For example, to search for task states, specify TASK.STATE in your query.
WHERE clause and filters	Use the queryCondition property on the query table API to further filter the result of queries. Query tables provide pre-filtered content if primary query table filters, authorization filters, or query table filters have been specified on the query table definition.	Use the WHERE clause to filter the result of the query.
WHERE clause and selection criteria	The WHERE clause of the query API is not needed in this form on the query table API. Use the queryCondition property on the query table API for additional filtering. Selection criteria in the query table definition select a particular property of the attached query table. This is achieved in addition to the filtering by the WHERE clause on the query API.	Selection criteria are not available for the query API. However, selection criteria are similar to the part of the WHERE clause that defines, for example, the name or locale of QUERY_PROPERTY, or TASK_CPROP, or TASK_DESC. For example, a WHERE clause of QUERY_PROPERTY.NAME='xyz' is the same as specifying NAME='xyz' as a selection criterion on the query table definition for the QUERY_PROPERTY attached query table.
Work items and authorization	Use the WORK_ITEM query table to access work items. You can customize the use of work items on the query table definition when the query table is developed and on the query table API, using the AuthorizationOptions object or the AdminAuthorizationOptions object. For example, to exclude everybody work items when querying the TASK query table, specify a queryCondition property WI.EVERYBODY=0 or specify setUseEverybody(Boolean.FALSE) on the AuthorizationOptions property.	Use the WORK_ITEM view to access work items. All four types of work items are considered for the query result: everybody, individual, groups, and inherited work items. To filter the work items for a specific type of work item, customize the WHERE clause. For example, to exclude the everybody work items, specify WORK_ITEM.EVERYBODY=0, in the WHERE clause.
Parameters	You can use parameters in filters and selection criteria for composite query tables.	Parameters are not available for the query API unless stored queries are used.
Stored queries and query tables	The difference between a stored query and a query table is that stored queries are defined for one particular query, while a query table is defined for a particular set of queries. For example, the query table definition does not allow the specification of an order-by clause because this information is typically available only when the query is run.	You can use stored queries to run query that contains a predefined set of options.
Materialized views	Materialized views are not available for the query table API.	Materialized views use database technologies to provide performance improvements for queries.

Characteristic	query table API	query API
Custom tables	Supplemental query tables offer the same functionality as custom tables.	Custom tables are used to include data in queries that is external to the Business Process Choreographer database schema.
queryAll and authorization options	The queryAll functionality is provided by the AdminAuthorizationOptions object, which can be passed to the query table API instead of the AuthorizationOptions object. The caller must be in the BPESystemAdministrator, TaskSystemAdministrator, BPESystemMonitor, or TaskSystemMonitor.	The queryAll method which can be used by users that have the BPESystemAdministrator Java EE role to return all of the objects in the query result without being restricted by work items for a particular user or group.
Internationalization	For attributes of query tables and for the query table, localized display names and descriptions are available when query tables are used.	Names of the columns of the selected views, as they appear in the database or as they are specified in the select clause, are returned.

Query tables in Business Process Choreographer

Query tables support task and process list queries on data that is contained in the Business Process Choreographer database schema. This includes human task data and business process data that is managed by Business Process Choreographer, and external business data. Query tables provide an abstraction on the data of Business Process Choreographer that can be used by client applications. In this way, client applications become independent of the actual implementation of the query table. Query table definitions are deployed on Business Process Choreographer containers, and are accessible using the query table API.

There are three types of query tables:

- Predefined query tables
- Supplemental query tables
- Composite query tables

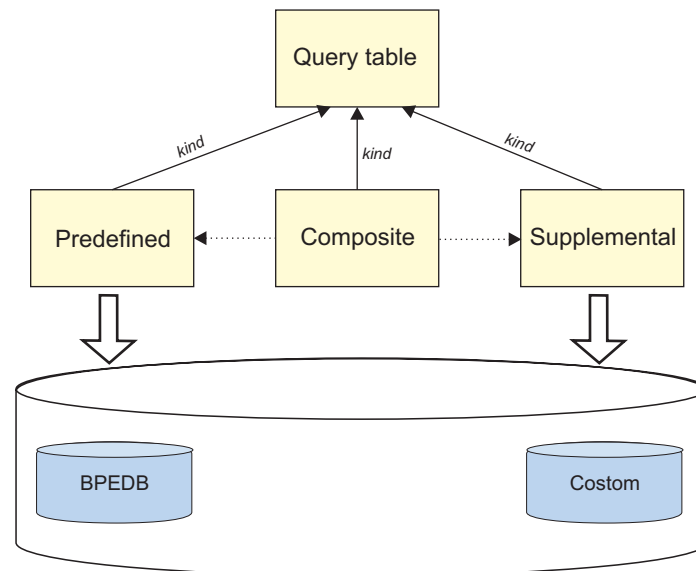


Figure 72. Query tables in Business Process Choreographer

Query tables are represented using similar models in the query table runtime, and you can use the query table API to query them. While predefined and

supplemental query tables point directly to tables or views in the database, composite query tables compose parts of this data, which is represented in a single query table.

Query tables enhance the predefined database views and the existing query interfaces of Business Process Choreographer, and they:

- Are optimized for running process and task list queries, using performance optimized access patterns.
- Simplify and consolidate access to the information needed.
- Allow for the fine-grained configuration of authorization and filter options.

You can customize the query tables, for example, you can configure a query table so that it contains only those tasks or process instances that are relevant in a particular scenario. It is also recommended that you use query tables where performance is important, such as with high volume process list and task list queries.

The Query Table Builder is provided as an Eclipse plug-in to:

- Develop composite and supplemental query tables
- Import and export query table definitions in XML format

You can download the Query Table Builder on the WebSphere Business Process Management SupportPacs site. Look for PA71 WebSphere Process Server - Query Table Builder. To access the link, see the related references section of this topic.

Predefined query tables

Predefined query tables provide access to the data in the Business Process Choreographer database. They are the query table representation of the corresponding predefined Business Process Choreographer database views, such as the TASK view or the PROCESS_INSTANCE view. These predefined query tables enhance the functionality and performance of the predefined database views because they are optimized for running process and task list queries.

The predefined query tables can be queried directly using the query table API. When you access the tables using the query table API, you are offered more options for configuration than when you use the query API.

Properties

Predefined query tables have the following properties:

Table 25. Properties of predefined query tables

Property	Description
Name	The query table name is the name of one of the predefined database views, in uppercase, for example, TASK.

Table 25. Properties of predefined query tables (continued)

Property	Description
Attributes	<p>Attributes of predefined query tables define the pieces of information that are available for queries. These attributes are the names of columns, in uppercase, that are specified by the predefined database views.</p> <p>The attributes are defined with a name and a type. The type is one of the following:</p> <ul style="list-style-type: none"> • Boolean: A boolean value • Decimal: A floating point number • ID: An object ID, such as TKIID of the TASK query table TASK • Number: An integer, short, or long • String: A string • Timestamp: A timestamp
Authorization	<p>Predefined query tables use either instance-based or role-based authorization.</p> <ul style="list-style-type: none"> • Predefined query tables with instance data require instance-based authorization. This means that only objects with a work item for the user who performs the query are returned. However, using the AdminAuthorizationOptions object, this verification can be reduced to a verification of the existence of a work item of any user. The user must have the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used for those queries. • Predefined query tables with template data require role-based authorization, which means that only users in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used, can access the contents of those query tables.

Predefined query tables with instance data

The following table shows the predefined query tables that contain instance data. These query tables:

- Can be used as the primary query of a composite query table.
- Use instance-based authorization if queried directly. This is accomplished with a join (SQL-) with the view that stores authorization information, that is, the predefined WORK_ITEM view or query table.
- Contain instance data, for example data of task instances or process instances.

Table 26. Predefined query tables containing instance data

Instance data	Query table name
Information about activities of a process instance.	ACTIVITY
	ACTIVITY_ATTRIBUTE
	ACTIVITY_SERVICE
Information about escalations belonging to human tasks.	ESCALATION
	ESCALATION_CPROP
	ESCALATION_DESC

Table 26. Predefined query tables containing instance data (continued)

Instance data	Query table name
Information about process instances.	PROCESS_ATTRIBUTE
	PROCESS_INSTANCE
	QUERY_PROPERTY
Information about human tasks.	TASK
	TASK_CPROP
	TASK_DESC

The WORK_ITEM query table also contains instance data, but this is not available as the primary query table or an attached query table. Work item information is available implicitly when querying query tables that use instance-based authorization. That is, attributes of the WORK_ITEM query table can be used when querying a query table with instance-based authorization, even though the attributes are not explicitly specified by the query table.

Predefined query tables with template data

Predefined query tables with template data require role-based authorization. They can be queried only by administrators using the AdminAuthorizationOptions object.

The following table shows the predefined query tables that contain template data. These query tables:

- Can be used as the primary query table of a composite query table.
- Use role-based authorization if queried directly. This means that the caller using the API query method must be in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used, or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used, and AdminAuthorizationOptions must be used.
- Contain template data, for example, the template data of task templates or process templates.

Table 27. Predefined query tables containing template data

Template data	Query table name
Information about application components.	APPLICATION_COMP
Information about escalation templates.	ESC_TEMPL
	ESC_TEMPL_CPROP
	ESC_TEMPL_DESC
Information about process templates.	PROCESS_TEMPLATE
	PROCESS_TEMPL_ATTR
Information about task templates.	TASK_TEMPL
	TASK_TEMPL_CPROP
	TASK_TEMPL_DESC

Related concepts

“Supplemental query tables”

Supplemental query tables in Business Process Choreographer expose to the query table API business data that is not managed by Business Process Choreographer. With supplemental query tables, this external data can be used with data from the predefined query tables when retrieving business process instance information or human task information.

“Composite query tables” on page 311

Composite query tables in Business Process Choreographer do not have a specific representation of data in the database; they comprise of a combination of data from related predefined and supplemental query tables. Use a composite query table to retrieve the information for a process instance list or task list, such as My To Dos.

“Query table development” on page 318

Supplemental and composite query tables in Business Process Choreographer are developed during application development using the Query Table Builder. Predefined query tables cannot be developed or deployed. They are available when Business Process Choreographer is installed and provide a simple view on the artifacts in the Business Process Choreographer database schema.

“Query table queries” on page 337

Queries are run on query tables in Business Process Choreographer using the query table API, which is available on the Business Flow Manager EJB and the REST API.

“Authorization for query tables” on page 327

You can use instance-based authorization, role-based authorization, or no authorization when you run queries on query tables.

Supplemental query tables

Supplemental query tables in Business Process Choreographer expose to the query table API business data that is not managed by Business Process Choreographer. With supplemental query tables, this external data can be used with data from the predefined query tables when retrieving business process instance information or human task information.

Supplemental query tables relate to database tables or database views in the Business Process Choreographer database. They are query tables that contain business data that is maintained by customer applications. Supplemental query tables provide information in a composite query table in addition to information that is contained in a predefined query table.

Supplemental query tables have the following properties:

Table 28. Properties of supplemental query tables

Property	Description
Name	<p>The query table name must be unique in a Business Process Choreographer installation. When the query is run, this name is used to identify the query table that is queried.</p> <p>A query table is uniquely identified using its name, which is defined as <i>prefix.name</i>. The maximum length of <i>prefix.name</i> is 28 characters. The prefix must be different to the reserved prefix 'IBM', for example, 'COMPANY.BUS_DATA'. Do not use a digit at the end of the table name. If a table is used multiple times within a query, the name of the table is extended with a number ranging from 0 to 9. For example, CUSTOM_VIEW0, CUSTOM_VIEW1 and so on. If there is already a digit at the end of your table name, Business Process Choreographer will remove this digit, which causes an QueryUnknownTableException.</p>
Database name	The name of the related table or view in the database. Only uppercase letters may be used.
Database schema	The schema of the related table or view in the database. Only uppercase letters can be used. The database schema must be different to the database schema of the Business Process Choreographer database. Nevertheless, the table or view must be accessible with the same JDBC data source that is used to access the Business Process Choreographer database.
Attributes	<p>Attributes of supplemental query tables define the pieces of information that are available for queries. These attributes must match the related name of the columns in the related database table or view.</p> <p>The attributes are defined with a name and a type. The name is defined in uppercase. The type is one of the following:</p> <ul style="list-style-type: none"> • Boolean: A boolean value • Decimal: A floating point number • ID: An object ID of 16 bytes in length, such as TKIID of the TASK query table • Number: An integer, short, or long • String: A string • Timestamp: A timestamp
Join	Joins must be defined on supplemental query tables if they are attached in composite query tables. A join defines which attributes are used to correlate information in the supplemental query table with the information in the primary query table. When a join is defined, the source attribute and the target attribute must be of the same type.
Authorization	No authorization is specified for supplemental query tables, therefore, all authenticated users can see the contents.

Related concepts

“Predefined query tables” on page 306

Predefined query tables provide access to the data in the Business Process Choreographer database. They are the query table representation of the corresponding predefined Business Process Choreographer database views, such as the TASK view or the PROCESS_INSTANCE view. These predefined query tables enhance the functionality and performance of the predefined database views because they are optimized for running process and task list queries.

“Composite query tables”

Composite query tables in Business Process Choreographer do not have a specific representation of data in the database; they comprise of a combination of data from related predefined and supplemental query tables. Use a composite query table to retrieve the information for a process instance list or task list, such as My To Dos.

“Query table development” on page 318

Supplemental and composite query tables in Business Process Choreographer are developed during application development using the Query Table Builder. Predefined query tables cannot be developed or deployed. They are available when Business Process Choreographer is installed and provide a simple view on the artifacts in the Business Process Choreographer database schema.

“Query table queries” on page 337

Queries are run on query tables in Business Process Choreographer using the query table API, which is available on the Business Flow Manager EJB and the REST API.

“Authorization for query tables” on page 327

You can use instance-based authorization, role-based authorization, or no authorization when you run queries on query tables.

Composite query tables

Composite query tables in Business Process Choreographer do not have a specific representation of data in the database; they comprise of a combination of data from related predefined and supplemental query tables. Use a composite query table to retrieve the information for a process instance list or task list, such as My To Dos.

Composite query tables are designed by client developers and they allow for a fine-grained configuration of filters and authorization options for optimized data access when the query is run. They are realized with SQL, which is optimized for task and process list queries.

It is recommended that you use composite query tables in production scenarios in place of the standard Business Process Choreographer query APIs, because composite query tables provide an abstraction over the actual implementation of the query and thus enable query optimizations.

Furthermore, you can change composite query tables at runtime without redeploying the client that accesses the query table.

The following figure provides an overview of the content of composite query tables:

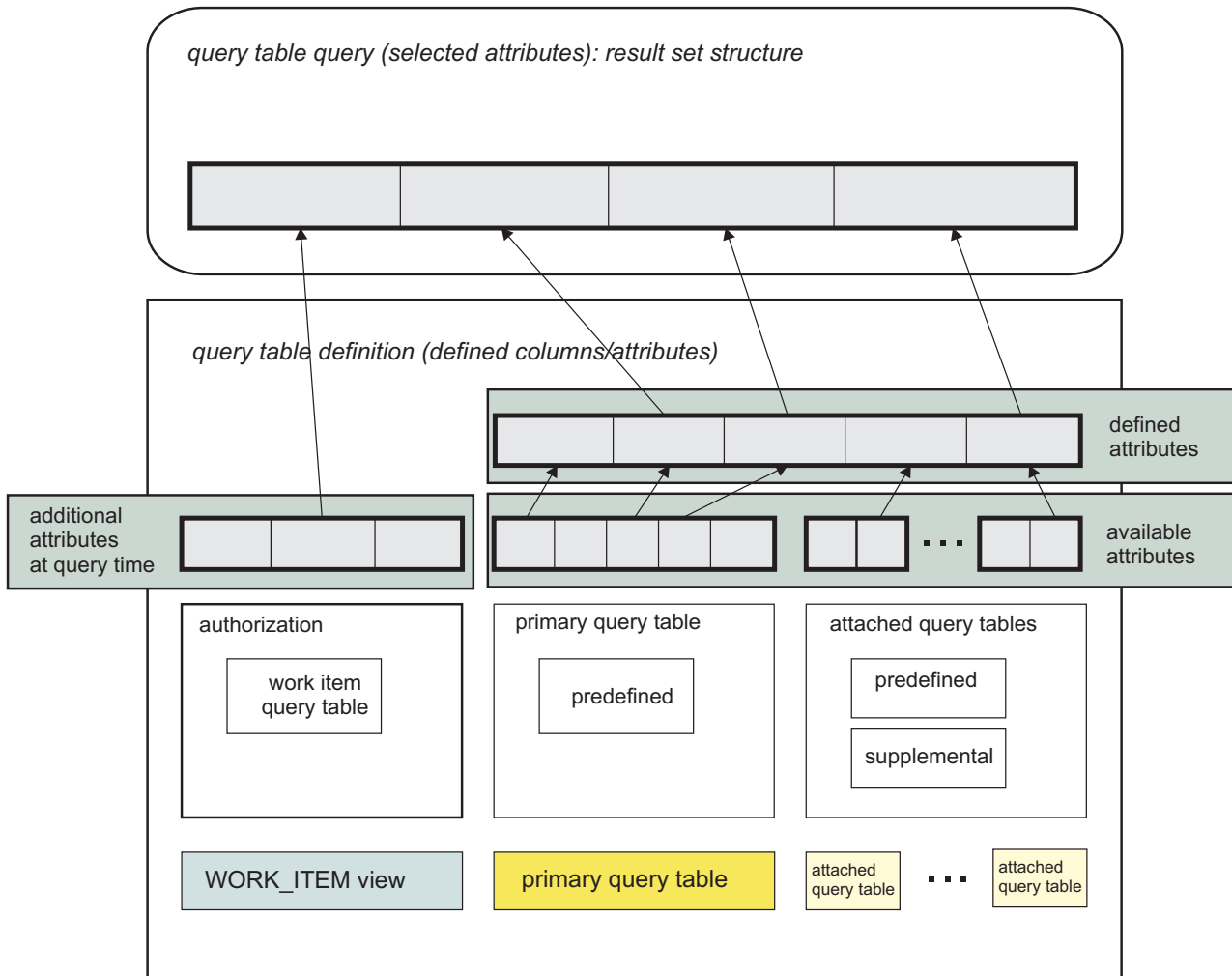


Figure 73. Composite query table content

All composite query tables are defined with one primary query table and zero or more attached query tables.

Primary query tables:

- Constitute the main information that is contained in a composite query table.
- Must be one of the predefined query tables.
- Uniquely identify each object in the composite query table by the primary key. For example, for the TASK predefined query table, this is the task ID TKIID.
- Authorize the contents of a query table using work items which are contained in the WORK_ITEM query table, if instance-based authorization is used.
- Determine the list of objects that are returned as rows of a table when querying the composite query table.

Attached query tables:

- Can be predefined query tables and supplemental query tables, which are already deployed on the system.
- Are available to provide information in addition to the information that is provided by the primary query table. For example, if TASK is the primary query

table, the description of the task provided in the TASK_DESC query table can be added to the contents of the composite query table.

Typically, the primary query table is chosen based on the purpose of the composite query table.

- If the composite query table describes a task list, the TASK query table is the primary query table.
- If the composite query table describes a process list, the PROCESS_INSTANCE query table is the primary query table.
- Lists of activities are retrieved using the ACTIVITY primary query table.
- Lists of human task escalations are retrieved using the ESCALATION primary query table.

The relationship between primary and attached query tables

The attached query table and the primary query table must have a one-to-one or one-to-zero relationship. If the one-to-one or one-to-zero relationship is violated, a runtime exception occurs when the query is run.

Primary query tables and attached query tables are correlated using a join attribute that is defined on the attached query table. This join attribute cannot be changed for predefined query tables, because it describes the relationship between the data in the various query tables of Business Process Choreographer. The join attribute is usually sufficient to maintain the one-to-one or one-to-zero relationship. For example, the CONTAINMENT_CTX_ID attribute is used on the TASK query table to attach the related process instance information that is identified by the PIID attribute on the PROCESS_INSTANCE query table.

When a one to many relationship exists, you must specify an additional criterion, known as selection criterion, in the Query Table Builder when you define the query table. For example, this could be "LOCALE='en_US'". A task can have several descriptions that are identified using different locales for a single task.

Example 1:

The following figure provides a sample visualization of the selection criteria that is specified on attached query tables:

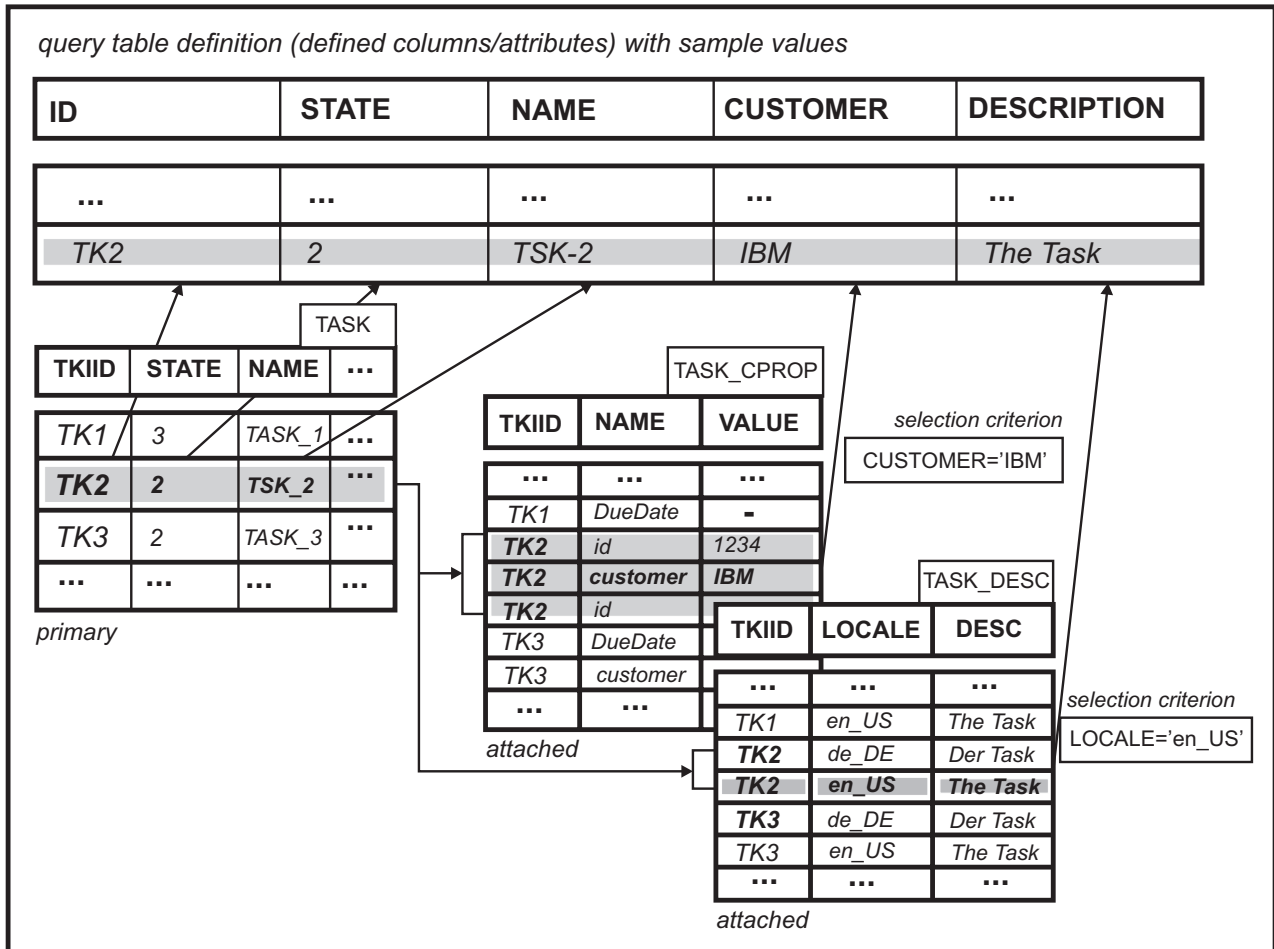


Figure 74. Composite query table with selection criteria

The composite query table contains the ID, STATE, NAME, CUSTOMER, and DESCRIPTION attributes.

- ID, STATE, and NAME are provided by the TASK primary query table.
- CUSTOMER is a custom property on TASK. Custom properties are stored in the TASK_CPROP query table. For a particular task, a custom property is uniquely identified using its name. This is reflected in the selection criterion "CUSTOMER='IBM'".
- DESCRIPTION is the description of the task, which is stored in TASK_DESC query table. For each task instance, the task description for a particular task is uniquely identified by its locale. This is reflected in the selection criterion "LOCALE='en_US'".

Example 2:

The focus of this example is on the relationship between the primary and the attached query tables, using TASK as the primary query table and TASK_DESC as the attached query table. When you define your composite query table, the LOCALE attribute of the TASK_DESC query table must be specified to ensure that there is a one-to-one or one-to-zero relationship between the primary query table and the attached query table. The table shows sample contents of a composite query table with a valid selection criterion for the TASK_DESC attached query table.

Table 29. Valid contents of a composite query table

TASK primary query table information	TASK_DESC attached query table information	
NAME	LOCALE	DESCRIPTION
task_one	en_US	This is a description.
task_two	en_US	This is a description.
...

The following table shows hypothetical invalid contents (in **bold type**) if the selection criterion is set incorrectly, which means that the one-to-one or one-to-zero relationship is violated.

Table 30. Invalid contents of a composite query table

Information from TASK (primary query table)	Information from TASK_DESC (attached query table)	
NAME	LOCALE	DESCRIPTION
task_one	en_US	This is a description.
task_one	de_DE	Das ist eine Beschreibung.
...

Properties

Composite query tables have the following properties:

Table 31. Properties of composite query tables

Property	Description
Name	<p>The query table name must be unique within a Business Process Choreographer installation. When the query is run, this query table name is used to identify the query table that is queried.</p> <p>A query table is uniquely identified using its name, which is defined as <i>prefix.name</i> for composite query tables. The maximum length of the <i>prefix.name</i> is 28 characters. The prefix must be different from the reserved prefix 'IBM', for example, 'COMPANY.TODO_TASK_LIST'. Do not use a digit at the end of the table name. If a table is used multiple times within a query, the name of the table is extended with a number ranging from 0 to 9. For example, CUSTOM_VIEW0, CUSTOM_VIEW1 and so on. If there is already a digit at the end of your table name, Business Process Choreographer will remove this digit, which causes an QueryUnknownTableException.</p>

Table 31. Properties of composite query tables (continued)

Property	Description
Attributes	<p>Attributes of composite query tables define the pieces of information that are available for queries.</p> <p>The attributes are defined with a name, in uppercase. The type is inherited from the referenced attribute, which is one of the following:</p> <ul style="list-style-type: none"> • Boolean: A boolean value • Decimal: A floating point number • ID: An object ID, such as TKIID of query table TASK • Number: An integer, short, or long • String: A string • Timestamp: A timestamp <p>Attributes of composite query tables are defined using a reference to attributes of the primary query table or the attached query tables. The attributes of the composite query tables inherit the types and constants of referenced attributes.</p> <p>In addition to the attributes that are part of the query table definition, work item information can be queried at runtime. This is possible if the primary query table contains instance data, such as TASK or PROCESS_INSTANCE, and if instance-based authorization is used on the composite query table. For example, the query can be defined to return only human tasks of which the user is a potential owner.</p>
Authorization	<p>Each composite query table defines if instance-based, role-based, or no authorization is used when queries are run on it.</p> <p>If instance-based authorization is defined, only objects with a work item for the user who performs the query are returned. However, using AdminAuthorizationOptions this verification can be reduced to a verification of the existence of a work item of any user. The user must be in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used, for those queries, and AdminAuthorizationOptions must be passed to the query table API.</p> <p>If role-based authorization is defined, the user must be in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used, for those queries, and AdminAuthorizationOptions must be passed to the query table API.</p> <p>If no authorization is defined, the query is run without checks against the existence of work items of the related objects in the query table. All authenticated users can see the contents of the query table.</p> <p>Instance-based authorization can be defined if the primary query table contains instance data; role-based authorization can be defined if the primary query table contains template data. No authorization can be defined on composite query tables regardless of which primary query table is used.</p>

Filters

Filters are used to limit the number of objects, or rows, that are contained in a composite query table.

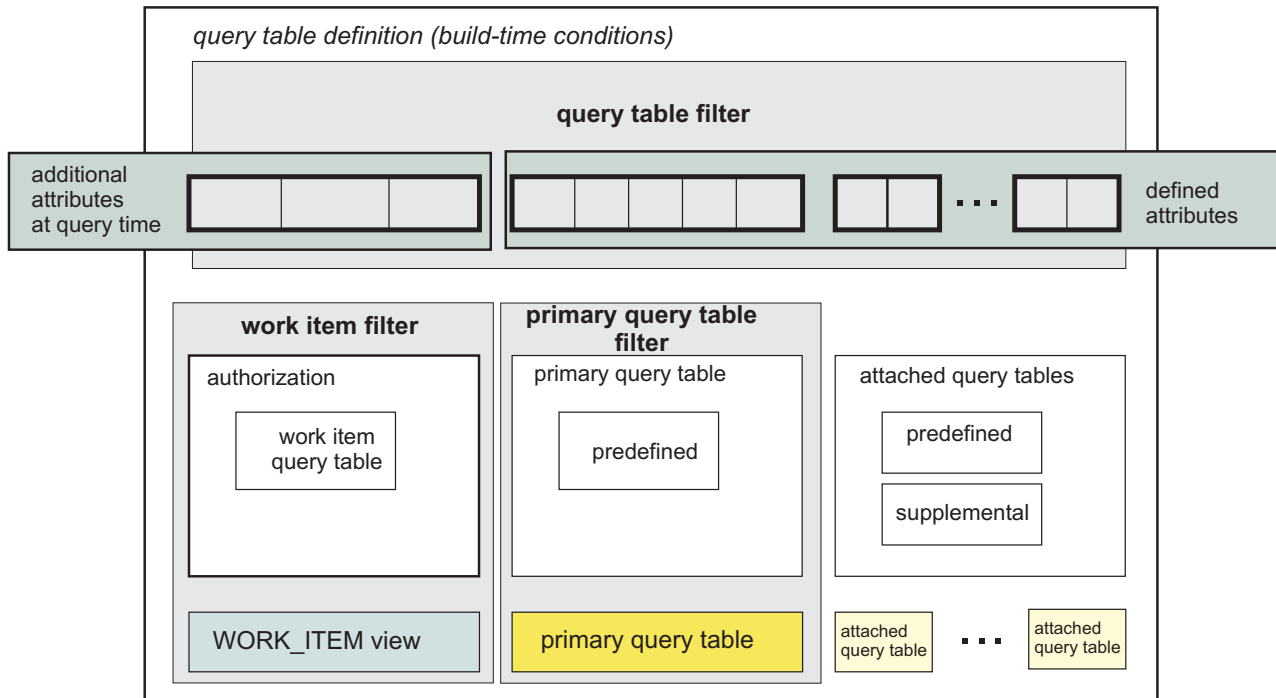


Figure 75. Filters in composite query tables

Filters in composite query tables can be defined during development on the:

- Primary query table, as the primary query table filter.
- Implicitly available WORK_ITEM query table which is responsible for authorization if the primary query table contains instance data. This filter is called the authorization filter, and is available only if the composite query table is configured to use instance-based authorization.
- Composite query table, as the query table filter.

Filters are defined during query table development. For example, a composite query table with the TASK primary query table can filter on tasks that are in the ready state ("STATE=STATE_READY" as the primary query table filter).

Authorization

Authorization for accessing the contents of a composite query table with a primary query table is similar to the authorization that is used to access the primary query table. The difference is that composite query tables can be configured to be more restrictive.

- If instance-based authorization is configured for use, the data contained in the composite query table is verified for existing work items in the WORK_ITEM query table. This verification is made against the primary query table. Everybody, individual, group, and inherited work items are used for the verification, depending on the configuration of the composite query table. If inherited work items are specified, objects that have a process instance as parent, such as a participating human task, with a related everybody, individual, or

group work item as configured, are contained in the composite query table. Typically, inherited work items are useful only for administrators.

- Composite query tables with a primary query table that contains template data must not be set to use instance-based authorization. If role-based authorization is used, queries can be run only by users that are in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used, and the AdminAuthorizationOptions object must be used.

Related concepts

“Predefined query tables” on page 306

Predefined query tables provide access to the data in the Business Process Choreographer database. They are the query table representation of the corresponding predefined Business Process Choreographer database views, such as the TASK view or the PROCESS_INSTANCE view. These predefined query tables enhance the functionality and performance of the predefined database views because they are optimized for running process and task list queries.

“Supplemental query tables” on page 309

Supplemental query tables in Business Process Choreographer expose to the query table API business data that is not managed by Business Process Choreographer. With supplemental query tables, this external data can be used with data from the predefined query tables when retrieving business process instance information or human task information.

Query table development

Supplemental and composite query tables in Business Process Choreographer are developed during application development using the Query Table Builder. Predefined query tables cannot be developed or deployed. They are available when Business Process Choreographer is installed and provide a simple view on the artifacts in the Business Process Choreographer database schema.

The Query Table Builder is available as an Eclipse plug-in and can be downloaded on the WebSphere Business Process Management SupportPacs site. Look for PA71 WebSphere Process Server - Query Table Builder. To access the link, see the related references section of this topic.

Query tables impact on the way applications are developed and deployed. The following steps describe the roles involved when you design and develop a Business Process Choreographer application that uses query tables.

Table 32. Query table development steps

Step	Who	Description
1. Analysis	Business analyst, client developer	Analyze which query tables are needed in the client application. Questions to be answered are: <ul style="list-style-type: none"> • How many task or process lists are provided to the user? Are there task or process lists that can share the same query table? • What kind of authorization is used? Instance-based authorization, role-based authorization, or none? • Are there other query tables already defined in the system that can be reused? • Must the query tables provide the content in multiple languages? If so, the selection criteria on attached query tables should be <code>LOCALE=\$LOCALE</code>.
2. Query table development	Client developer, business analyst	Develop the query tables that are used in the client application. Try to specify the definition of the query tables such that the best performance is achieved with query table queries.
3. Query table deployment	Administrator	Query tables must be deployed to the runtime before they can be used. This step is done using the <code>manageQueryTable.py wsadmin</code> command.
4. Query table queries	Client developer	To run queries against query tables is the last step of query table development. The client developer must know the name of the query table and its attributes.

The following is sample code, which uses the query table API to query a query table. Examples 1 and 2 are provided to query the predefined query table TASK for simplicity reasons. Examples 3 and 4 query a composite query table, which is assumed to be deployed on the system. In application development, you should use composite query tables rather than directly querying the predefined query tables.

Example 1

```
// get the naming context and lookup the Business
// Flow Manager Enterprise JavaBeans home; note that the Business Flow
// Manager Enterprise JavaBeans home should be cached for performance
// reasons; also, it is assumed that there's an Enterprise JavaBeans
// reference to the local business flow manager Enterprise JavaBeans
Context ctx = new InitialContext();
LocalBusinessFlowManagerHome home =
    (LocalBusinessFlowManagerHome)
    ctx.lookup("java:comp/env/ejb/BFM");

// if the human task manager Enterprise JavaBeans is used, do:
// LocalHumanTaskManagerHome home =
//    (LocalHumanTaskManagerHome) ctx.lookup("java:comp/env/ejb/HTM");
// assuming that a EJB reference to the human task manager EJB
// has been defined

// create the business flow manager client-side stub
```

```

LocalBusinessFlowManager bfm = home.create();
// if the human task manager EJB is used, do:
// LocalHumanTaskManager htm = home.create();
// note that the human task manager Enterprise JavaBeans provides the
// same methods as the business flow manager Enterprise JavaBeans
// *****
// ***** example 1 *****
// *****

// execute a query against the TASK predefined query
// table; this relates to a simple My ToDo's task list
EntityResultSet ers = null;
ers = bfm.queryEntities("TASK", null, null, null);

// print the result to STDOUT
EntityInfo entityInfo = ers.getEntityInfo();
List attList = entityInfo.getAttributeInfo();
int attSize = attList.size();

Iterator iter = ers.getEntities().iterator();
while (iter.hasNext()) {
    System.out.print("Entity: ");
    Entity entity = (Entity) iter.next();
    for (int i = attSize - 1; i >= 0; i--) {
        AttributeInfo ai = (AttributeInfo) attList.get(i);
        System.out.print(
            entity.getAttributeValue(ai.getName()));
    }
    System.out.println();
}

```

Example 2

```

// *****
// ***** example 2 *****
// *****

// same example as example 1, but using the row-based
// query approach
RowResultSet rrs = null;
rrs = bfm.queryRows("TASK", null, null, null);

attList = rrs.getAttributeInfo();
attSize = attList.size();

// print the result to STDOUT
while (rrs.next()) {
    System.out.print("Row: ");
    for (int i = attSize - 1; i >= 0; i--) {
        AttributeInfo ai = (AttributeInfo) attList.get(i);
        System.out.print(
            rrs.getAttributeValue(ai.getName()));
    }
    System.out.println();
}

```

Example 3

```

// *****
// ***** example 3 *****
// *****

// execute a query against a composite query table
// that has been deployed on the system before;
// the name is assumed to be COMPANY.TASK_LIST

```

```

    ers = bfm.queryEntities(
        "COMPANY.TASK_LIST", null, null, null);
^
    // print the result to STDOUT ...

```

Example 4

```

// *****
// ***** example 4 *****
// *****

// query against the same query table as in example 3,
// but with customized options
FilterOptions fo = new FilterOptions();

// return only objects which are in state ready
fo.setQueryCondition("STATE=STATE_READY");

// sort by the id of the object
fo.setSortAttributes("ID");

// limit the number of entities to 50
fo.setThreshold(50);

// only get a sub-set of the defined attributes
// on the query table
fo.setSelectedAttributes("ID, STATE, DESCRIPTION");

AuthorizationOptions ao = new AuthorizationOptions();

// do not return objects that everybody is allowed
// to see
ao.setEverybodyUsed(Boolean.FALSE);

ers = bfm.queryEntities(
    "COMPANY.TASK_LIST", fo, ao, null);

// print the result to STDOUT ...

```

Related concepts

“Query table queries” on page 337

Queries are run on query tables in Business Process Choreographer using the query table API, which is available on the Business Flow Manager EJB and the REST API.

“Filters and selection criteria of query tables”

Filters and selection criteria are defined during query table development using the Query Table Builder, which uses a syntax similar to SQL WHERE clauses. Use these clearly defined filters and selection criteria to specify conditions that are based on attributes of query tables.

“Predefined query tables” on page 306

Predefined query tables provide access to the data in the Business Process Choreographer database. They are the query table representation of the corresponding predefined Business Process Choreographer database views, such as the TASK view or the PROCESS_INSTANCE view. These predefined query tables enhance the functionality and performance of the predefined database views because they are optimized for running process and task list queries.

“Supplemental query tables” on page 309

Supplemental query tables in Business Process Choreographer expose to the query table API business data that is not managed by Business Process Choreographer. With supplemental query tables, this external data can be used with data from the predefined query tables when retrieving business process instance information or human task information.

“Composite query tables” on page 311

Composite query tables in Business Process Choreographer do not have a specific representation of data in the database; they comprise of a combination of data from related predefined and supplemental query tables. Use a composite query table to retrieve the information for a process instance list or task list, such as My To Dos.

Filters and selection criteria of query tables

Filters and selection criteria are defined during query table development using the Query Table Builder, which uses a syntax similar to SQL WHERE clauses. Use these clearly defined filters and selection criteria to specify conditions that are based on attributes of query tables.

For information about installing the Query Table Builder, see the WebSphere Business Process Management SupportPacs site. Look for PA71 WebSphere Process Server - Query Table Builder. To access the link, see the related references section of this topic.

Attributes

Attributes used in an expression refer to query table attributes. Depending on the location of the expression, different attributes are available. For the client developer, query filters passed to the query table API are the only location where expressions can be used. For developers of composite query tables, various other locations exist where expressions can be used. The following table describes the attributes that are available at the different locations.

Table 33. Attributes for query table expressions

Where	Expression	Available attributes
Query table API	Query filter	<ul style="list-style-type: none"> All attributes defined on the query table. If instance-based authorization is used, all attributes defined on the WORK_ITEM query tables, prefixed with 'WI.' . <p>Examples:</p> <ul style="list-style-type: none"> STATE=STATE_READY, if the query table contains a STATE attribute and if a STATE_READY constant is defined for this attribute STATE=STATE_READY AND WI.REASON=REASON_POTENTIAL_OWNER, if the query table contains a STATE attribute and the query table uses instance-based authorization
Composite query table	Query table filter	<ul style="list-style-type: none"> STATE=STATE_READY, if the query table contains a STATE attribute and if a STATE_READY constant is defined for this attribute STATE=STATE_READY AND WI.REASON=REASON_POTENTIAL_OWNER, if the query table contains a STATE attribute and the query table uses instance-based authorization
	Primary query table filter	<ul style="list-style-type: none"> All attributes defined for the primary query table. <p>Example:</p> <ul style="list-style-type: none"> STATE=STATE_READY, if the query table contains a STATE attribute and a STATE_READY constant is defined for this attribute
	Authorization filter	<ul style="list-style-type: none"> All attributes defined on the WORK_ITEM predefined query table, prefixed with 'WI.' . <p>Example:</p> <ul style="list-style-type: none"> WI.REASON=REASON_POTENTIAL_OWNER
	Selection criterion	<ul style="list-style-type: none"> All attributes defined on the related attached query table. <p>Example:</p> <ul style="list-style-type: none"> LOCALE='en_US', if the attached query table contains a LOCALE attribute, such as the TASK_DESC query table

The following figure shows the various locations of filters and selection criteria in expressions, and includes examples:

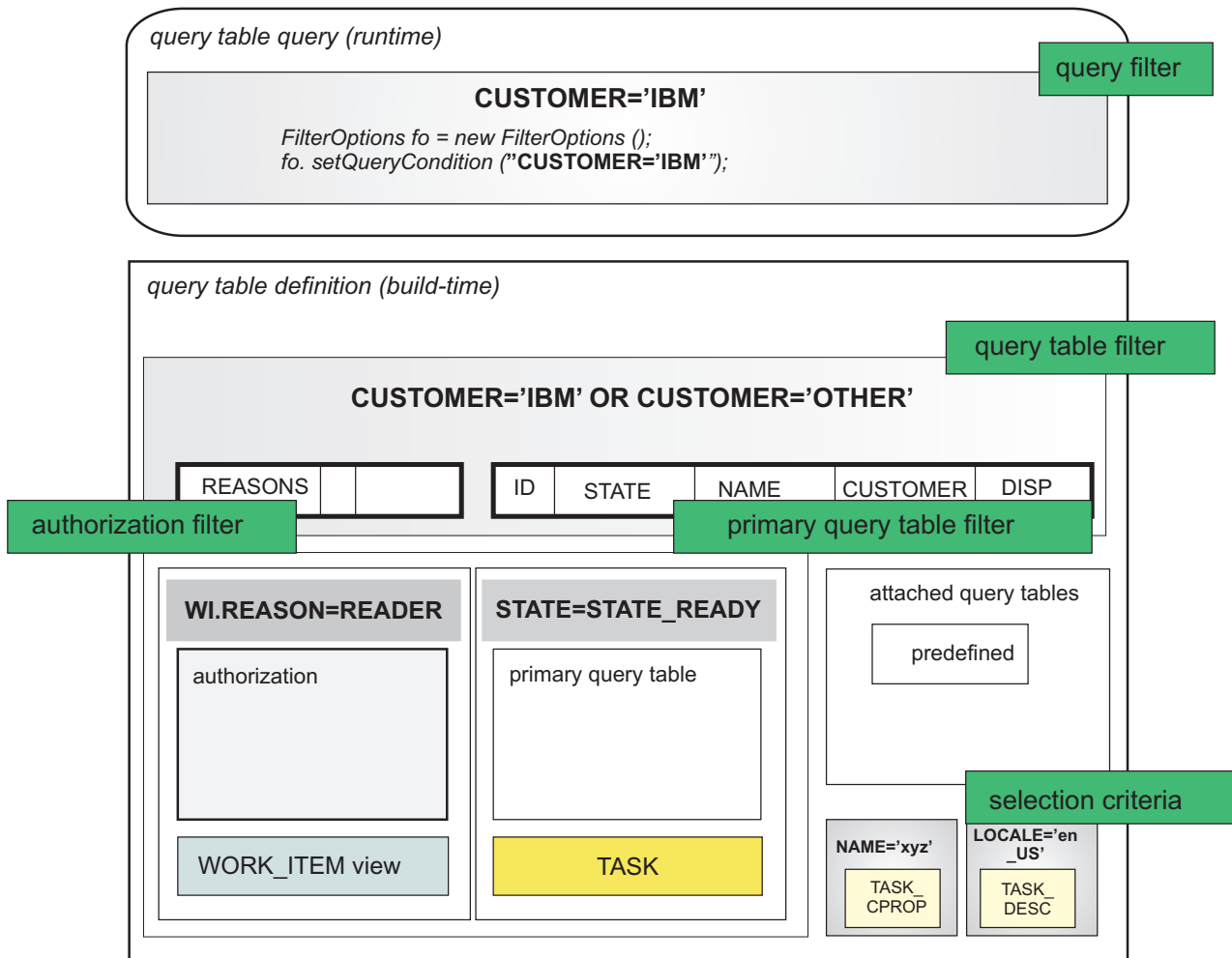


Figure 76. Filters and selection criteria in expressions

Expressions

Expressions have the following syntax:

```
expression ::= attribute binary_op value |
             attribute unary_op |
             attribute list_op list |
             (expression) |
             expression AND expression |
             expression> OR expression
```

The following rules apply:

- AND takes precedence over OR. Subexpressions are connected using AND and OR.
- Brackets can be used to group expressions and must be balanced.

Examples:

- STATE = STATE_READY
- NAME IS NOT NULL
- STATE IN (2, 5, STATE_FINISHED)
- ((PRIORITY=1) OR (WI.REASON=2)) AND (STATE=2)

An expression is executed in a certain scope which determines the attributes that are valid for the expression. Selection criteria, or query filters, are run in the scope of the query table on which the query is run.

The following example is for a query that is run on the predefined TASK query table:

```
'(STATE=STATE_READY AND WI.REASON=REASON_POTENTIAL_OWNER)
OR (WI.REASON=REASON_OWNER)'
```

Binary operators

The following binary operators are available:

```
binary_op ::= = | < | > | <> | <= | >= | LIKE | NOT LIKE
```

The following rules apply:

- The left-side operand of a binary operator must reference an attribute of a query table.
- The right-side operand of a binary operator must be a literal value, constant value, or parameter.
- The LIKE and NOT LIKE operators are only valid for attributes of attribute type STRING.
- The left-side operand and the right-side operand must be of compatible attribute types.
- User parameters must be compatible to the attribute type of the left-side attribute.

Examples:

- STATE > 2
- NAME LIKE 'start%'
- STATE <> PARAM(theState)

Unary operators

The following unary operators are available:

```
unary_op ::= IS NULL | IS NOT NULL
```

The following rules apply:

- The left-side operand of a unary operator must reference an attribute of a query table. Valid attributes depend on the location of the filter or selection criterion.
- All attributes can be checked for null values, for example: CUSTOMER IS NOT NULL.

Example:

```
DESCRIPTION IS NOT NULL
```

List operators

The following list operators are available:

```
list_op ::= IN | NOT IN
```

The following rules apply:

- The right-side of a list operator must not be replaced by a user parameter.

- User parameters can be used within the list on the right-side operand.

Example:

```
STATE IN (STATE_READY, STATE_RUNNING, PARAM(st), 1)
```

Lists are represented as follows:

```
list ::= value [, list]
```

The following rules apply:

- The right-side of a list operator must not be replaced by a user parameter.
- User parameters can be used within the list on the right-side operand.

Examples:

- (2, 5, 8)
- (STATE_READY, STATE_CLAIMED)

Values

In expressions, a value is one of the following:

- **Constant:** A constant value, which is defined for the attribute of a predefined query table. For example, STATE_READY is defined for the STATE attribute of the TASK query table.
- **Literal:** Any hardcoded value.
- **Parameter:** A parameter is replaced when the query is run with a specific value.

Constants are available for some attributes of predefined query tables. For information about constants that are available on attributes of predefined query tables, refer to the information about predefined views. Only constants that define integer values are exposed with query tables. Also, instead of constants, related literal values, or parameters can be used.

Examples:

- STATE_READY on the STATE attribute of the TASK query table can be used in a filter to check whether the task is in the ready state.
- REASON_POTENTIAL_OWNER on the REASON attribute of the WORK_ITEM query table can be used in a filter to check whether the user who runs the query against a query table is a potential owner.
- Query filter STATE=STATE_READY is the same as STATE=2, if the query is run on the TASK query table.

Literals can also be used in expressions. A special syntax must be used for timestamps and for IDs.

Examples:

- STATE=1
- NAME='theName'
- CREATED > TS ('2008-11-26 T12:00:00')
- TKTID=ID('_TKT:801a011e.9d57c52.ab886df6.1fcc0000')

Parameters in expressions allow for a dynamicity of composite query tables. There are user parameters and system parameters:

- User parameters are specified using PARAM (*name*). This parameter must be provided when the query is run. It is passed as an instance of the `com.ibm.bpe.api.Parameter` class into the query table API.
- System parameters are parameters that are provided by the query table runtime, without being specified when the query is run. The system parameters \$USER and \$LOCALE are available.
 - \$USER, which is a string, contains the value of the user who runs the query.
 - \$LOCALE, which is a string, contains the value of the locale that is used when the query is run. An example for the value of \$LOCALE is 'en_US'.

You can specify a parameter in the selection criteria of an attached query table which selects on a specific locale. For example, if the primary query table is TASK in a composite query table and an attached query table is TASK_DESC. The following are examples of parameters:

- STATE=PARAM(theState)
- LOCALE=\$LOCALE
- OWNER=\$USER

Related concepts

“Query table development” on page 318

Supplemental and composite query tables in Business Process Choreographer are developed during application development using the Query Table Builder. Predefined query tables cannot be developed or deployed. They are available when Business Process Choreographer is installed and provide a simple view on the artifacts in the Business Process Choreographer database schema.

“Query table queries” on page 337

Queries are run on query tables in Business Process Choreographer using the query table API, which is available on the Business Flow Manager EJB and the REST API.

Related tasks

“Creating query tables for Business Process Choreographer Explorer” on page 356

You can use query tables instead of the EJB query API to improve the performance of Business Process Choreographer Explorer. To create the query tables, use the Query Table Builder.

Authorization for query tables

You can use instance-based authorization, role-based authorization, or no authorization when you run queries on query tables.

The authorization type is defined on the query table.

- Instance-based authorization indicates that objects in the query table are authorized using a work item. This is done by verifying if a suitable work item exists.
- Role-based authorization is based on Java EE roles. It indicates that the caller using the API query method must be in the `BPSystemAdministrator` Java EE role if the Business Flow Manager EJB is used or the `TaskSystemAdministrator` Java EE role if the Human Task Manager EJB is used to see the contents of the query table. It is available for predefined query tables with template data and for composite query tables with a primary query table that contains template data. Objects in those query tables do not have related work items.
- When no authorization is specified, all authenticated users can see all contents of the query table, after filters are applied.

The type of authorization on predefined query tables and the type of authorization that can be configured on composite and supplemental query tables is outlined in the following table.

Table 34. Types of authorization for query tables

Query table	Instance-based authorization	Role-based authorization	No authorization
Predefined	Required for predefined query tables with instance data.	Required for predefined query tables with template data.	N/A
Composite	<p>Can be turned off which means that no authorization is used and the security constraints are overridden. That is, every authenticated user can use the query table to retrieve data, independently of whether they are authorized for the respective objects.</p> <p>Composite query tables with a primary query table that contains template data must not be set to use instance-based authorization.</p>	<p>Can be turned off, for example for composite query tables with a primary query table that contains template data. This means that no authorization is used and the security constraints are overridden. That is, every authenticated user can use the query table to retrieve data, independently of whether they are authorized for the respective objects.</p> <p>Composite query tables with a primary query table that contains instance data must not be set to use role-based authorization.</p>	All authenticated users can see all contents of the query table, after filters are applied.
Supplemental	Supplemental query tables must not be set to use instance-based authorization because they are not managed by Business Process Choreographer, and therefore it has no authorization information for the contents of these tables.	Supplemental query tables must not be set to use role-based authorization.	All authenticated users can see all contents of the query table, after filters are applied.

The following figure provides an overview of the available options for the authorization types, depending on the type of query table. Also, it outlines the different behaviors and the query table API and its authorization options.

Authorization	Instance-based authorization	None	Role-based authorization
Composite query table	primary query table with instance data	all	primary query table with template data
Predefined query tables	instance data	n/a	template data
Supplemental query tables	n/a	business data	n/a
Query with AuthorizationOptions	(A) Query result contains objects with work items related to the caller.	(C) Query result contains all objects that are in this query table.	n/a
Query with AdminAuthorizationOptions*	(B) Query result contains all objects that are in this query table.	(C) Query result contains all objects that are in this query table.	(D) Query result contains all objects that are in this query table.

Figure 77. Instance-based authorization for query tables

*) If the onBehalfUser is set, (A) applies

Instance-based authorization for objects in the query result using work items depend on the authorization parameter that is passed to the query table API and on the setting of the instance-based authorization flag of the query table.

- (A) Queries on predefined or composite query tables using the AuthorizationOptions object return entities that correlate with a related work item for this particular user. This is also the case if the AdminAuthorizationOptions object is used and onBehalfUser is set. Standard clients which present task or process lists to users usually use this combination of query tables and query table API parameters.

- (B) The full content of a query table consists of the entities that have a related work item, as configured with the instance-based authorization of the query table. Instance-based authorization considers four types of work items: everybody, individual, group, and inherited. The caller using the API query method must be in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used. This combination of query tables and query table API parameters is intended for use in administrative scenarios where the full list of available tasks or processes must be shown or searched.
- (C) Queries on query tables that do not use instance-based or role-based authorization return the same result if AdminAuthorizationOptions or AuthorizationOptions is passed into the query table API. This is available for supplemental and composite query tables. There is no check on work items or Java EE roles, therefore all authenticated users see the full content. Clients that do not want to restrict object visibility by applying the instance-based or role-based authorization constraints that are provided by Business Process Choreographer can turn off authorization checks when query table definitions are developed. When using claim and complete, however, users must have related work items.
- (D) Template data in predefined query tables or composite query tables with role-based authorization configured can be accessed only with role-based authorization. This requires the caller using the API query method to be in the BPESystemAdministrator Java EE role if the Business Flow Manager EJB is used or the TaskSystemAdministrator Java EE role if the Human Task Manager EJB is used. The query table API can be used to access template information instead of the query API.

Work items and instance-based authorization

Instance-based authorization provided by Business Process Choreographer is based on work items. Each work item describes who has which rights on what object. This information is accessible using the WORK_ITEM query table, if instance-based authorization is used.

The table describes the different types of work items that are considered if instance-based authorization is used when a query is run against a query table:

Table 35. Work item types

Work item type	Description
everybody	Allows all users to access a specific object, such as a task or a process instance. In this case, the EVERYBODY attribute of the related work item is set to TRUE.
individual	Work items that are created for particular users. The OWNER_ID attribute of the related work item is set to a specific user. Multiple work items which differ in the OWNER_ID attribute can exist for an object, such as a task.
group	Work items that are created for users of a particular group. The GROUP_NAME attribute of the related work item is set to a specific group.
inherited	Readers and administrators of process instances are also allowed to inherit the access to the human tasks which belong to these process instances, including escalations. Checks for an inherited work item in task queries are performed with complex SQL joins at runtime, which impacts on performance.

Work items are created by Business Process Choreographer in different situations. For example, at task creation, work items are created for the different roles, such as reader and potential owner, if related people assignment criteria were specified.

The following table describes the types of work items that are created, depending on the people assignment criteria that are defined, if instance-based authorization is used when a query is run on a query table. Inherited work items do not appear in the table because they reflect a relationship that is not explicitly modeled during process application development.

Table 36. Work items and people assignment criteria

Work item type	Related people assignment criteria
everybody	Everybody
individual	All people assignment criteria except verbs <i>Nobody</i> , <i>Everybody</i> , and <i>Group</i>
group	Group

Authorization filter on composite query tables

On composite query tables, you can specify an authorization filter if instance-based authorization is used. This filter restricts the work items which are used for authorization, based on certain attributes of work items. For example, the authorization filter "WI.REASON=REASON_POTENTIAL_OWNER" on a composite query table with the TASK primary query table restricts the tasks that are returned when a person runs a query. The result contains only tasks that represent a to-do for that person, that is, the result is restricted to those tasks the person is authorized to claim. This filter can also be specified as the query table filter or as the query filter, but for query performance reasons, it is beneficial to specify these filters as the authorization filter.

Related concepts

“Predefined query tables” on page 306

Predefined query tables provide access to the data in the Business Process Choreographer database. They are the query table representation of the corresponding predefined Business Process Choreographer database views, such as the TASK view or the PROCESS_INSTANCE view. These predefined query tables enhance the functionality and performance of the predefined database views because they are optimized for running process and task list queries.

“Supplemental query tables” on page 309

Supplemental query tables in Business Process Choreographer expose to the query table API business data that is not managed by Business Process Choreographer. With supplemental query tables, this external data can be used with data from the predefined query tables when retrieving business process instance information or human task information.

“Composite query tables” on page 311

Composite query tables in Business Process Choreographer do not have a specific representation of data in the database; they comprise of a combination of data from related predefined and supplemental query tables. Use a composite query table to retrieve the information for a process instance list or task list, such as My To Dos.

“Authorization options for the query table API” on page 342

When you run a query on a query table in Business Process Choreographer, authorization options can be passed as input parameters to the methods of the query table API.

Related tasks

“Creating query tables for Business Process Choreographer Explorer” on page 356

You can use query tables instead of the EJB query API to improve the performance of Business Process Choreographer Explorer. To create the query tables, use the Query Table Builder.

Attribute types for query tables

Attribute types are needed in Business Process Choreographer when query tables are defined, when literal values are used in queries, and when values of a query result are accessed. Rules and mappings are available for each of the attribute types.

A subset of the types that are available in the Java programming language and databases is used to define the type of an attribute of a query table. Attribute types are an abstraction of the concrete Java type or database type. For supplemental query tables, you must use a valid database type to attribute type mapping.

The following table describes the attribute types:

Table 37. Attribute types

Attribute type	Description
ID	The ID which is used to identify a human task (TKIID), a process instance (PIID), or other objects. For example, IDs are used to claim or complete a particular human task, which is identified with the specified TKIID.
STRING	Task descriptions or query properties can be represented as a string.
NUMBER	Numbers are used for attributes, such as the priority on a task.

Table 37. Attribute types (continued)

Attribute type	Description
TIMESTAMP	Timestamps describe a point in time, such as the time when a human task is created, or a process instance is finished.
DECIMAL	Decimals can be used as the type for query properties, for example when defining a query property with a variable of XSD type double.
BOOLEAN	Booleans can have one of two values, true or false. For example, human tasks provide an attribute, autoClaim, which identifies whether the task is claimed automatically if only a single user exists as the potential owner for this task.

Database type to attribute type mapping:

Use attribute types to define query tables in Business Process Choreographer, when you run queries on the query tables, and to access values of a query result.

The following table describes the database types and their mapping to attribute types:

Table 38. Database type to attribute type mapping

Database type	Attribute type
A binary type with 16 bytes. This is the type used for IDs such as TKIID on TASK of the Business Process Choreographer tables.	ID
A character based type. The length depends on the column in the database table that is referenced by the attribute of the query table.	STRING
An integer database type, such as integer, short, or long.	NUMBER
A timestamp database type.	TIMESTAMP
A decimal type, such as float or double.	DECIMAL
A type that is convertible to a Boolean value, such as a number. 1 is interpreted as <i>true</i> , and all other numbers as <i>false</i> .	BOOLEAN

Supplemental query tables typically refer to existing database tables and views, such that table or view creation is not necessary.

Example

Consider a table in a DB2 environment, CUSTOM.ADDITIONAL_INFO, which is to be represented in Business Process Choreographer as a supplemental query table. The following SQL statement creates the database table:

```
CREATE TABLE CUSTOM.ADDITIONAL_INFO
(
  PIID      CHAR(16) FOR BIT DATA,
  INFO     VARCHAR(220),
  COUNT    INTEGER
);
```

The following mapping of database column types to query table attribute types is used for a supplemental query table for the CUSTOM.ADDITIONAL_INFO table.

Table 39. Database types to attribute types mapping example

Database column and type	Query table attribute and type
PIID CHAR(16) FOR BIT DATA	PIID (ID)
INFO VARCHAR(220)	INFO (STRING)
COUNT INTEGER	COUNT (NUMBER)

Attribute type to literal representation mapping:

Attribute types are used when query tables are defined in Business Process Choreographer, when queries are run on the query tables, and when values of a query result are accessed. Use this topic for information on attribute type to literal representation mapping.

Literal values can be used in expressions to define filter and selection criteria, such as in filters of composite query tables, and in filters that are passed to the query table API.

The following table describes the attribute types and their mapping to literal values. Placeholders are marked *italic*. Note that the attribute types ID and TIMESTAMP, which can be passed to the query table API, use a special syntax, which is also used by the query API.

Table 40. Attribute type to literal values mapping

Attribute type	Syntax and usage as literal value in expressions
ID	ID ('string representation of an ID') When developing client applications, IDs are represented either as a string or as an instance of the com.ibm.bpe.api.OID interface. The string representation can be obtained from an instance of the com.ibm.bpe.api.OID interface using the toString method. The string must be enclosed in single quotation marks.
STRING	'the string' The string must be enclosed in quotes.
NUMBER	number The number as text, and no quotation marks. Constants are defined for some number attributes on predefined query tables, and can be used.
TIMESTAMP	TS ('YYYY-MM-DDThh:mm:ss') The timestamp must be specified as: <ul style="list-style-type: none"> • YYYY is the 4-digit year • MM is the 2-digit month of the year • DD is the 2-digit day of the month • hh is the 2-digit hour of the day (24-hour) • mm is the 2-digit minutes of the hour • ss is the 2-digit seconds of the minute The timestamp is interpreted as defined in the user's time zone.
DECIMAL	number.fraction The decimal number as text and no quotation marks; the .fraction part is optional.

Table 40. Attribute type to literal values mapping (continued)

Attribute type	Syntax and usage as literal value in expressions
BOOLEAN	true, false
	The Boolean value as text.

Example

- `filterOptions.setQueryCondition("STATE=2");`
- `filterOptions.setQueryCondition("STATE=STATE_READY");`
- a selection criterion on an attached query table TASK_DESC:
"LOCALE='en_US'"
- `filterOptions.setQueryCondition("PTID=ID('_PT:8001011e.1dee8e51.247d6df6.29a60000'))");`

Attribute type to parameter mapping:

Use attribute types when you define query tables in Business Process Choreographer, when you run queries on the query tables, and to access values of a query result.

The following table describes the attribute types and their mapping to parameter values that can be used in expressions to define filter and selection criteria, such as in filters of composite query tables, and in filters passed to the query table API.

Table 41. Attribute type to user parameter values mapping

Attribute type	Usage as parameter value in expressions
ID	<p><code>PARAM(name)</code></p> <p>When developing client applications, IDs are represented either as a string or as an instance of the <code>com.ibm.bpe.api.OID</code> interface.</p> <p>As a parameter, both representations are valid. An array of bytes reflecting a valid OID can also be used (byte).</p>
STRING	<p><code>PARAM(name)</code></p> <p>The string representation of the object that is passed to the query table API at runtime by the <code>toString</code> method.</p>
NUMBER	<p><code>PARAM(name)</code></p> <p>A <code>java.lang.Long</code>, <code>java.lang.Integer</code>, <code>java.lang.Short</code>, or a <code>java.lang.String</code> representation of the number is passed to the query table API. Names of constants, as defined on some attributes of predefined query tables, can be also passed.</p>
TIMESTAMP	<p><code>PARAM(name)</code></p> <p>The following representations are valid:</p> <ul style="list-style-type: none"> • A <code>java.lang.String</code> representation of the timestamp • Instances of <code>com.ibm.bpe.api.UTCDate</code> • Instances of <code>java.util.Calendar</code>

Table 41. Attribute type to user parameter values mapping (continued)

Attribute type	Usage as parameter value in expressions
DECIMAL	PARAM(<i>name</i>) A java.lang.Long, java.lang.Integer, java.lang.Short, java.lang.Double, java.lang.Float, or a java.lang.String representation of the decimal is passed to the query table API.
BOOLEAN	PARAM(<i>name</i>) Valid values are: <ul style="list-style-type: none"> • A java.lang.String representation of the boolean • A java.lang.Short, java.lang.Integer, java.lang.Long with appropriate values; 0 (for false), or 1 (for true) • A java.lang.Boolean object

Example

```

...
// this example shows a query against a composite query table
// COMP.TASKS with a parameter "customer"
java.util.List params = new java.util.ArrayList();

list.add(new com.ibm.bpe.api.Parameter("customer", "IBM");
// the business flow manager Enterprise JavaBeans or the
// human task manager Enterprise JavaBeans can be used to access query tables
service.bfm.queryEntities("COMP.TASKS", null, null, params);
...

```

Attribute type to Java object type mapping:

Attribute types are used when query tables are defined in Business Process Choreographer, when queries are run on the query tables, and when values of a query result are accessed. Use this topic for information on attribute type to Java object type mapping.

The following table describes the attribute types and their mapping to Java object types in query result sets.

Table 42. Attribute type to Java object type mapping

Attribute type	Related Java object type
ID	com.ibm.bpe.api.OID
STRING	java.lang.String
NUMBER	java.lang.Long
TIMESTAMP	java.util.Calendar
DECIMAL	java.lang.Double
BOOLEAN	java.lang.Boolean

Example

```

...
// the following example shows a query against a composite query table
// COMP.TA; attribute "STATE" is of attribute type NUMBER
...
// run the query
// the business flow manager Enterprise JavaBeans or the

```

```

// human task manager Enterprise JavaBeans can be used to access query tables
EntityResultSet rs = bfm.queryEntities("COMP.TA",null,null,params);

// get the entities and iterate over it
List entities = rs.getEntities();
for (int i = 0 ; i < entities.size(); i++) {

    // work on a particular entity
    Entity en = (Entity) entities.get(i);

    // note that the following code could be written
    // more generalized using the attribute info objects
    // contained in ei.getAttributeInfo()

    // get attribute STATE
    Long state = (Long) en.getAttributeValue("STATE");
    ...
}
...

```

Attribute type compatibility:

Use attribute types when you define query tables in Business Process Choreographer, when you run queries on the query tables, and to access values of a query result.

The following table shows the attribute types and their compatible attribute types, which can be used to define filters and selection criteria in query tables. Compatible attribute types are marked with X.

Table 43. Attribute type compatibility

Attribute type	ID	STRING	NUMBER	TIMESTAMP	DECIMAL	BOOLEAN
ID	X					
STRING		X				
NUMBER			X		X	
TIMESTAMP				X		
DECIMAL			X		X	
BOOLEAN						X

In query table expressions that specify filter and condition criteria, types of attributes or values that are compared must be compatible. For example, `WI.OWNER_ID=1` is an invalid filter because the left-side operand is of type `STRING`, and the right-side operand is of type `NUMBER`.

Query table queries

Queries are run on query tables in Business Process Choreographer using the query table API, which is available on the Business Flow Manager EJB and the REST API.

A query is run on one query table only. Entity-based API methods and row-based API methods are used to retrieve content from query tables. Input parameters are passed into the methods of the query table API.

Related concepts

“Query table development” on page 318

Supplemental and composite query tables in Business Process Choreographer are developed during application development using the Query Table Builder. Predefined query tables cannot be developed or deployed. They are available when Business Process Choreographer is installed and provide a simple view on the artifacts in the Business Process Choreographer database schema.

“Predefined query tables” on page 306

Predefined query tables provide access to the data in the Business Process Choreographer database. They are the query table representation of the corresponding predefined Business Process Choreographer database views, such as the TASK view or the PROCESS_INSTANCE view. These predefined query tables enhance the functionality and performance of the predefined database views because they are optimized for running process and task list queries.

“Supplemental query tables” on page 309

Supplemental query tables in Business Process Choreographer expose to the query table API business data that is not managed by Business Process Choreographer. With supplemental query tables, this external data can be used with data from the predefined query tables when retrieving business process instance information or human task information.

“Composite query tables” on page 311

Composite query tables in Business Process Choreographer do not have a specific representation of data in the database; they comprise of a combination of data from related predefined and supplemental query tables. Use a composite query table to retrieve the information for a process instance list or task list, such as My To Dos.

“Filters and selection criteria of query tables” on page 322

Filters and selection criteria are defined during query table development using the Query Table Builder, which uses a syntax similar to SQL WHERE clauses. Use these clearly defined filters and selection criteria to specify conditions that are based on attributes of query tables.

Query table API methods:

Queries are run on query tables in Business Process Choreographer using the query table API. Entity-based API methods and row-based API methods are available to retrieve content from query tables.

The following entity-based methods and row-based methods are provided to run queries on query tables in Business Process Choreographer using the query table API:

Table 44. Methods for queries run on query tables

Purpose	Methods
Query contents	<ul style="list-style-type: none">• queryEntities• queryRows <p>Both methods return contents of the query table. The queryEntities method returns content based on entities and queryRows returns content based on rows.</p>

Table 44. Methods for queries run on query tables (continued)

Purpose	Methods
Query the number of objects	<ul style="list-style-type: none"> • queryEntityCount • queryRowCount <p>Both methods return the number of objects in the query table, while the actual number can depend on whether the entity-based or the row-based approach is taken.</p>

Entity-based queries, using the queryEntities method and the queryEntityCount method, assume that a query table contains uniquely identifiable entities, as defined by the primary key on the primary query table.

Row-based queries, using the queryRows method and the queryRowCount method, return a result set like JDBC, which is row-based, and provides first and next methods for navigating in it. The result set that is returned when you run a query on a query table using the query table API can be compared to QueryResultSet that is returned by the query API. In general, the number of rows is greater than the number of entities that are contained in a query table. The same entity, for example, a human task which is identified by its task ID, such as TKIID, might occur multiple times in the row result set.

A specific instance that is contained in any predefined query table exists only once in a Business Process Choreographer environment. Examples of instances are human tasks and business processes. Those instances are uniquely identified using an ID or a set of IDs. This is the TKIID for instances of human tasks and the PIID for process instances.

Composite query tables are composed of a primary query table and zero or more attached query tables. Objects that are contained in composite query tables are uniquely identified by the unique ID of the objects that are contained in the primary query table. The primary query table of a composite query table determines its entity type. For example, a composite query table with the TASK primary query table contains entities of the TASK type. The one-to-one or one-to-zero relationship between the primary and attached query tables ensures that the attached query tables do not result in duplicate entities.

Entity-based queries exploit the uniquely identifiable entities of a query table, as defined by the primary key on the primary query table. A client application programmer for user interfaces is typically interested in unique instances without duplicates, for example, to display a human task once only on the user interface. Unique instances are returned if the entity-based query table API is used.

Row-based queries can return duplicate rows of the primary query table if instance-based authorization is used.

- Information from the WORK_ITEM query table is retrieved with the query. For example, if the WI.REASON attribute is retrieved in addition to the attributes that are defined on the query table, multiple rows qualify for the result. This is because there can be multiple reasons why a user can access an entity, such as, a task or a process instance.
- Instance-based authorization is used, and distinct is not specified. Even though work item information is not retrieved, multiple rows may be returned if instance-based authorization is used.

If the entity-based query table API is used:

- Entity-based queries are always run with the SQL distinct operator.
- Entity-based queries return a result which allows array values for work-item-related information.

Query table API parameters:

You use query table API methods to retrieve content when you run queries against a query table in Business Process Choreographer.

The following input parameters are passed to the methods of the query table API:

Table 45. Parameters of the query table API

Parameter	Optional	Type and description
Query table name	No	java.lang.String The unique name of the query table.
Filter options	Yes	com.ibm.bpe.api.FilterOptions if the Business Flow Manager Enterprise JavaBeans is used or com.ibm.task.api.FilterOptions if the Human Task Manager Enterprise JavaBeans is used. Options which can be used to define the query. For example, a query threshold is set on this parameter to limit the number of results returned.
Authorization options	Yes	com.ibm.bpe.api.AuthorizationOptions or com.ibm.bpe.api.AdminAuthorizationOptions if the Business Flow Manager Enterprise JavaBeans is used. com.ibm.task.api.AuthorizationOptions or com.ibm.task.api.AdminAuthorizationOptions if the Human Task Manager Enterprise JavaBeans is used. Authorization can be further constrained if instance-based authorization is used. For query tables which require role-based authorization, an instance of AdminAuthorizationOptions must be passed.
Parameters	Yes	A java.util.List of com.ibm.bpe.api.Parameter if the Business Flow Manager Enterprise JavaBeans is used or com.ibm.task.api.Parameter if the Human Task Manager Enterprise JavaBeans is used. This parameter is used to pass user parameters, which are specified in a filter or selection criterion on a composite query table.

A query is run on one specific query table only. The relationship between multiple query tables is defined with composite query tables. In terms of the query API (as distinct from the query table API), this corresponds to database views.

You specify filters and selection criteria in expressions during query table development using the Query Table Builder. For more information, refer to the information center topic about composite query tables and the topic about filter and search criteria of query tables. For information about the Query Table Builder, see the WebSphere Business Process Management SupportPacs site. Look for PA71 WebSphere Process Server - Query Table Builder. To access the link, see the related references section of this topic.

Query table name:

When you run a query on a query table in Business Process Choreographer, the query table name is passed as an input parameter to the methods of the query table API.

The query table name is the name of the query table on which the query is run.

- For predefined query tables, this is the name of the predefined query table.
- For composite and supplemental query tables, this is the name of the respective query table that is specified while modeling the query table. The name of a composite or supplemental query table follows the *prefix.name* naming convention, and *prefix* may not be 'IBM'.

Both the query table name and prefix must be in uppercase. The maximum length of the query table name is 28 characters.

Filter options for query tables:

When you run a query on a query table in Business Process Choreographer, filter options can be passed as input parameters to the methods of the query table API.

An instance of the `com.ibm.bpe.api.FilterOptions` class if the Business Flow Manager Enterprise JavaBeans is used, or an instance of the `com.ibm.task.api.FilterOptions` if the Human Task Manager Enterprise JavaBeans is used, can be passed to the query table API. The filter options allow a configuration of the query using:

- A threshold and offset (`skipCount`)
- Sort attributes (similar to the ORDER BY clause in an SQL query)
- A user-provided query filter
- The set of attributes returned, including work item information
- Other

The result set that can be obtained from a query table is specified by the definition of the query table. However, you might want to specify additional options when the query is run. The following table describes the options that can be specified as filter options using the `FilterOptions` object.

Table 46. Query table API parameters: Filter options

Option	Type	Description
Selected attributes	<code>java.lang.String</code>	<ul style="list-style-type: none"> • A comma separated list of attributes of the query table that must be returned in the result set. • If instance-based authorization is used, work item information can be retrieved by specifying attributes of the <code>WORK_ITEM</code> query table, prefixed with <code>WI.</code>, for example, <code>WI.REASON</code>. • If null is specified, all attributes of the query table are returned, without work item information.
Sort attributes	<code>java.lang.String</code>	<p>A comma separated list of attributes of the query table, optionally followed by <code>ASC</code> or <code>DESC</code>, for ascending or descending, respectively.</p> <p>This list is similar to the SQL ORDER BY clause: <code>sortAttributes ::= attribute [ASC DESC] [, sortAttributes]</code>. If <code>ASC</code> or <code>DESC</code> is not specified, <code>ASC</code> is assumed. Sorting occurs in the sequence of the sort attributes. This example sorts tasks in query table <code>TASK</code> in descending order by state, and within the groups of the same <code>STATE</code> by <code>NAME</code>, in ascending order: <code>"STATE DESC, NAME ASC"</code>.</p>

Table 46. Query table API parameters: Filter options (continued)

Option	Type	Description
Threshold	java.lang.Integer	<p>Defines the maximum:</p> <ul style="list-style-type: none"> • Number of rows returned if queryRows is used. • Number of entities returned if queryEntities is used. The actual number of available entities in the respective query table may exceed the threshold number of entities for the query even if the entity result set does not contain as many entities as the threshold number. This is due to technical reasons if work item information is selected. • Count returned if queryRowCount or queryEntityCount is used. <p>The default is null which means that no threshold is set.</p>
Skip count	java.lang.Integer	<p>Defines the number of rows (row-based queries) or the number of entities (entity-based queries) that are skipped. As with the threshold parameter, skipCount may not be accurate for entity-based queries.</p> <p>Skip count is used to allow paging over a large result set. The default is null which means that no skipCount is set.</p>
Time zone	java.util.TimeZone	The time zone that is used when converting timestamps. An example is CREATED on the predefined query table TASK. If not specified (null), the time zone on the server is used.
Locale	java.util.Locale	The locale which is used to calculate the value of the \$LOCALE system parameter. An example usage of \$LOCALE in a selection criterion is: 'LOCALE=\$LOCALE'.
Distinct rows	java.lang.Boolean	Used for row-based queries only. If set to true, row-based queries return distinct rows. This does not imply that unique rows are returned due to the possible multiplicity of work item information.
Query condition	java.lang.String	This option performs additional filtering on the result set. All of the attributes that are defined for the query table can be referenced. If authorization is required for the query table, columns that are defined for the WORK_ITEM query table can also be referenced using the WI prefix, for example, WI.REASON=REASON_POTENTIAL_OWNER.

Authorization options for the query table API:

When you run a query on a query table in Business Process Choreographer, authorization options can be passed as input parameters to the methods of the query table API.

Use an instance of the com.ibm.bpe.api.AuthorizationOptions class or the com.ibm.bpe.api.AdminAuthorizationOptions if the Business Flow Manager EJB is used, or an instance of the com.ibm.task.api.AuthorizationOptions class or the com.ibm.task.api.AdminAuthorizationOptions class if the Human Task Manager EJB is used, to specify additional authorization options when the query is run.

If instance-based authorization is used, instances of the AuthorizationOptions class allow the specification of the type of work items used to identify eligible instances that are returned by the query.

An instance of the AuthorizationOptions class can be passed to the query table API if the query is run on a predefined query table that contains instance data. It can also be passed if the query is run on a composite query table with a primary query

table that contains instance data and instance-based authorization is configured to be used. If the query is run on a predefined query table with template data or a composite query table with role-based authorization configured, an `com.ibm.bpe.api.EngineNotAuthorizedException` exception is thrown if the Business Flow Manager EJB is used or `com.ibm.task.api.NotAuthorizedException` is thrown if the Human Task Manager EJB is used. In all other cases, the authorization options passed to the query table API are ignored.

Composite query tables can restrict the types of work items that are considered when identifying objects (or entities) that are contained in it. For example, if the authorization options that are passed to the query table API are configured to use everybody work items, this is only taken into account if everybody work items are defined for use on the definition of the composite query table. As a simple rule, a work item type that is not specified to be considered on the query table definition cannot be overwritten to be considered by the query table API, but a work item type that is specified to be considered on the query table definition can be overwritten not to be used. Also, the authorization type of a composite or predefined query table cannot be overwritten by the query table API.

Depending on the type of query table that is queried, different authorization option defaults apply if the authorization object is not specified or if the related attributes (everybody, individual, group, or inherited) are set to null, which is the default.

The following table shows the authorization option defaults for instance-based authorization for the query table type and work item type used.

Table 47. Query table API parameters: Authorization option defaults for instance-based authorization

Query table type	Everybody work item	Individual work item	Group work item	Inherited work item
Predefined with instance data	TRUE	TRUE	TRUE	FALSE
Predefined with template data	N/A	N/A	N/A	N/A
Composite with a primary query table with instance data	TRUE	TRUE	TRUE	TRUE
Composite with a primary query table with template data	N/A	N/A	N/A	N/A
Supplemental	N/A	N/A	N/A	N/A

N/A means that instance-based authorization is not used and, therefore, any setting on the authorization object with respect to work items is ignored.

If TRUE is specified, the resulting query will only consider the specific work item type if the query table is defined to use this type of work item. This is true for all predefined query tables with instance data, but might not be true for a composite query table. For the group work item, the latter must also be enabled on the

human task container. An example of the inherited work item set to TRUE is that the administrator of a process instance may see participating human task instances that are created for that process instance.

Specify an instance of the AdminAuthorizationOptions class instead of an instance of the AuthorizationOptions class if:

- A query is run on a query table with role-based authorization. Predefined query tables with template data require role-based authorization, and composite query tables with a primary query table with template data can be configured to require role-based authorization.
- A query is run on a query table with instance data or on a composite query table with a primary query table that contains instance data. It should return the content of that query table, regardless of restrictions due to authorization for a particular user. This behavior is equivalent to using the queryAll method on the query API (as distinct from the query table API).
- A query should be executed on behalf of another user.

The following table describes how the various behaviors above are accomplished:

Table 48. Query table API parameters: AdminAuthorizationOptions

Situation	Description
onBehalfUser set to null	<ul style="list-style-type: none"> • If the query is run on a query table with role-based authorization, all contents of that query table are returned. • If the query is run on a query table which uses instance-based authorization, the particular objects contained in the query table are not checked for work items for a particular user. All objects that are contained in the query table are returned.
onBehalfUser set to a particular user	The query is run with the authority of the specified user, and the objects in the query table are checked against the work items for this user, if instance-based authorization is used.

If you specify AdminAuthorizationOptions, the caller must be in the BPESystemAdministrator or BPESystemMonitor Java EE role if the Business Flow Manager EJB is used, or in the TaskSystemAdministrator or TaskSystemMonitor Java EE role if the Human Task Manager EJB is used.

Related concepts

“Authorization for query tables” on page 327

You can use instance-based authorization, role-based authorization, or no authorization when you run queries on query tables.

Parameters:

When you run a query on a query table in Business Process Choreographer, you can pass user parameters as input parameters to the methods of the query table API. In query table definitions, you can specify parameters in filters on the primary query table, on the authorization, and on the query table. Parameters can also be specified in selection criteria on attached query tables.

The system parameters, \$USER and \$LOCALE, are replaced at runtime in filters and selection criteria, and are not required to be passed into the query table API. The input value for the calculation of the \$LOCALE system parameter is provided by setting the locale in the filter options.

User parameters must be passed into the query table API when the query is run. This is accomplished by passing a list of instances of the com.ibm.bpe.api.Parameter class if the Business Flow Manager EJB is used, or an instance of the com.ibm.task.api.Parameter class if the Human Task Manager EJB is used.

The following properties must be specified on a parameter object:

Table 49. User parameters for the query table API

Property	Description
Name	The name of the parameter as used in the query table definition. The name is case sensitive.
Value	The value of the parameter. The type of the parameter must be compatible with the type of the left-hand operand of all filters and selection criteria where this parameter is used. Constants that are defined on some attributes of predefined query tables can be passed as a string, for example STATE_READY.

Example

```
// execute a query against a composite query
// table CUST.CPM with the primary query table filter
// set to 'STATE=PARAM(theState)'
EntityResultSet ers = null;
List parameterList = new ArrayList();
parameterList.add(new Parameter
("theState", new Integer(2)));

// run the query;
// the business flow manager EJB or the
// human task manager EJB can be used to access query tables
ers = bfm.queryEntities
("CUST.CPM", null, null, parameterList);

// work on the result set
// ...
```

Results of query table queries:

You use query table API methods when you run queries on a query table in Business Process Choreographer. The result of a queryEntityCount method or queryRowCount method query is a number. The queryEntities and the queryRows methods return result sets.

EntityResultSet

An instance of the com.ibm.bpe.api.EntityResultSet class is returned by the method queryEntities if the Business Flow Manager Enterprise JavaBeans is used. An instance of the com.ibm.task.api.EntityResultSet class is returned by the method queryEntities if the Human Task Manager Enterprise JavaBeans is used. An entity result set has the following properties:

Table 50. Entity result set properties of a query table API entity

Property	Description
queryTableName	Name of the query table on which the query was run.
entityTypeName	<ul style="list-style-type: none"> • If the query was run on a composite query table, this is the name of the primary query table. • If the query was run on a predefined query table or on a supplemental query table, this is the name of the query table, that is, the same value as the <i>queryTableName</i> property.
EntityInfo	This property contains the meta information of the entities that are contained in the entity result set. A java.util.List list of com.ibm.bpe.api.AttributeInfo objects if the Business Flow Manager EJB is used, or a list of com.ibm.task.api.AttributeInfo objects if the Human Task Manager EJB is used, can be retrieved on this object. This list contains the attribute names and attribute types of the information contained in the entities of this result set. Meta information about the attributes which constitute the key for these entities is also contained.
entities	A java.util.List list of com.ibm.bpe.api.Entity objects if the Business Flow Manager EJB is used, or a list of com.ibm.task.api.Entity objects if the Human Task Manager is used.
locale	The locale that is calculated for the \$LOCALE system parameter.

Instances of the Entity class contain the information that is retrieved from the query table query. An entity represents a uniquely identifiable object such as a task, a process instance, an activity, or an escalation. The following properties are available for entities:

Table 51. Entity properties of a query table API entity

Property	Description
EntityInfo	The EntityInfo object which is also contained in the entity result set. A java.util.List list of com.ibm.bpe.api.AttributeInfo objects if the Business Flow Manager EJB is used, or a list of com.ibm.task.api.AttributeInfo objects if the Human Task Manager EJB is used, can be retrieved on this object. This list contains the attribute names and attribute types of the information contained in the entities of this result set. Meta information about the attributes which constitute the key for these entities is also contained.
attributeValue (<i>attributeName</i>)	The value of the specified attribute that is retrieved for this entity. The type is contained in the related AttributeInfo object of this attribute.
attributeValuesOfArray (<i>attributeName</i>)	An array of values. Use this property if the attribute info property <i>array</i> is set to true which is currently the case only if the attribute refers to work item information.

The number of entities in the entity result set is retrieved using the size method on the list of entities.

Example: Entity-based query table API:

```
...
// the following example shows a query against
// predefined query table TASK, using the entity-based API

...
// run the query
// service is a (Local)BusinessFlowManager object or a
// (Local)HumanTaskManager object
EntityResultSet rs = service.queryEntities("TASK", null, null, null);

// get the entities meta information
EntityInfo ei = rs.getEntityInfo();
List atts = ei.getAttributeInfo();

// get the entities and iterate over it
Iterator entitiesIter = rs.getEntities().iterator();
while (entitiesIter.hasNext()) {

    // work on a particular entity
    Entity en = (Entity) entitiesIter.next();

    for (int i = 0; i < atts.size(); i++) {
        AttributeInfo ai = (AttributeInfo) atts.get(i);
        Serializable value = en.getAttributeValue(ai.getName());

        // process...
    }
}
...

```

RowResultSet

An instance of the `com.ibm.bpe.api.RowResultSet` class is returned by the method `queryRows` if the Business Flow Manager Enterprise JavaBeans is used. An instance of the `com.ibm.task.api.RowResultSet` class is returned by the method `queryRows` if the Human Task Manager Enterprise JavaBeans is used. This type of result set is similar to a JDBC result set. A row result set has the following properties:

Table 52. Row result set properties of a query table API row

Property	Description
<code>primaryQueryTableName</code>	<ul style="list-style-type: none">If the query was run on a composite query table, this is the name of the primary query table.If the query was run on a predefined query table or on a supplemental query table, this is the name of the query table, that is, the same value as property <code>queryTableName</code>.
<code>attributeInfo</code>	This property contains a list of <code>com.ibm.bpe.api.AttributeInfo</code> objects if the Business Flow Manager Enterprise JavaBeans is used or a list of <code>com.ibm.task.api.AttributeInfo</code> objects if the Human Task Manager Enterprise JavaBeans is used. They describe the meta information for this result set. <code>AttributeInfo</code> objects contain the attribute names and attribute types of the information. Meta data about keys is not contained because row result sets do not have a key.
<code>attributeValue</code>	The value of the specified attribute that was retrieved for this row. The type is contained in the related <code>AttributeInfo</code> object of this attribute.

Table 52. Row result set properties of a query table API row (continued)

Property	Description
next, first, last, previous	The row result set is navigated using these methods. Compare its usage to iterators, enumerations, or JDBC result sets.

The number of rows in the row result set is retrieved using the size method on the list of rows.

Example: Row-based query table API

```

...
// the following example shows a query against
// predefined query table TASK, using the entity-based API
...
// run the query
// service is a (Local)BusinessFlowManager object or a
// (Local)HumanTaskManager object
RowResultSet rs = service.queryRows("TASK", null, null, null);

// get the entities meta information
List atts = rs.getAttributeInfo();

// get the entities and iterate over it
while (rs.next()) {

    // work on a particular row
    for (int i = 0; i < atts.size(); i++) {
        AttributeInfo ai = (AttributeInfo) atts.get(i);
        Serializable value = rs.getAttributeValue(ai.getName()) ;

        // process...
    }
}
...

```

Query table queries for meta data retrieval

Queries are run on query tables in Business Process Choreographer using the query table API. Methods are available to retrieve meta data from query tables.

The following methods are provided to retrieve meta data when you run queries on query tables in Business Process Choreographer using the query table API:

Table 53. Methods for meta data retrieval on query tables

Purpose	Method
Return the meta data of a specific query table	getQueryTableMetaData
Return a list of query table meta data with specific properties	findQueryTableMetaData
Return contents of a query table, based on entities, and a subset of the meta data for the selected attributes	queryEntities
Return contents of a query table, based on rows, and a subset of the meta data for the selected attributes	queryRows

Meta data of query tables consists of data that relates to structure and data that relates to internationalization.

The following table shows the meta data that is related to the structure of a query table.

Table 54. Meta data related to query table structure

Meta data	Description	Returned by getQuery- TableMetaData	Returned by findQuery- TableMetaData	Returned by queryEntities	Returned by queryRows
Query table name	The name of the query table	Yes	Yes	Yes	Yes
Primary query table name	For supplemental and predefined query tables, name of the query table; for composite query tables the name of the primary query table	Yes	Yes	Yes	Yes
Kind	The type of query table: predefined, composite, or supplemental	Yes	Yes	No	No
Authorization	The authorization that is defined on the query table: <ul style="list-style-type: none"> • Use of work items • Instance-based, role-based, or no authorization 	Yes	Yes	No	No
Defined attributes	Meta data of the attributes that are defined on the query table	Yes	Yes	No. Meta data of the selected attributes is returned.	No. Meta data of the selected attributes is returned.
Key attributes	Key attributes of the query table	Yes	Yes	Yes	No. Not applicable to row-based queries.

The following table shows the meta data that is related to the internationalization of a query table.

Table 55. Meta data related to query table internationalization

Meta data	Description	Returned by getQuery- TableMetaData	Returned by findQuery- TableMetaData	Returned by queryEntities	Returned by queryRows
locales[]	Locales for which display names and descriptions of the query table and attributes are defined.	Yes	Yes	No	No
Locale	Value of the \$LOCALE system parameter which results from the locale that is passed to the API.	Yes	Yes	Yes	Yes
Display name and description of the query table	Display names and descriptions for the query table, which are provided for all defined locales.	Yes	Yes	No	No

Table 55. Meta data related to query table internationalization (continued)

Meta data	Description	Returned by getQuery- TableMetaData	Returned by findQuery- TableMetaData	Returned by queryEntities	Returned by queryRows
Display names and descriptions of the attributes	Display names and descriptions for the attributes, which are provided for all defined locales.	Yes	Yes	No	No

All EJB query table API methods which return query table meta data accept a locale parameter, such as `FilterOptions.setLocale` and `MetaDataOptions.setLocale`. This parameter should be set to the Java locale that the client uses to present information to the user. This locale parameter is used to calculate the value of the `$LOCALE` system parameter, which can be used in filters and selection criteria. The locale that is returned contains the actual Java locale that is used for `$LOCALE`.

If the display names and descriptions of a specific query table are retrieved, pass `getLocale` to the related methods to get the display names and descriptions in the same locale as the descriptions of the tasks. For example, these descriptions are attached using a selection criterion of `'LOCALE=$LOCALE'`.

Example

```
// the following example shows how meta data for a particular
// composite query table can be retrieved

...
// run the query
MetaDataOptions mdo = new MetaDataOptions("TASK", null, false, new Locale("en_US"));
List list = bfm.findQueryTableMetaData(mdo);

// to get the meta data of a specific query table
// use bfm.getQueryTableMetaData(...)

// iterate through the list of query tables that have TASK as primary query table
// => at least one query table is returned: the predefined query table TASK

Iterator iter = list.iterator();
while (iter.hasNext()) {
    QueryTableMetaData md = (QueryTableMetaData) iter.next();
    Locale effectiveLocale = md.getLocale();
    String queryTableDisplayName = md.getDisplayName(effectiveLocale);
    System.out.println("found query table: " + queryTableDisplayName);
    List attributesList = md.getAttributeMetaData();
    Iterator attrIter = attributesList.iterator();
    while (attrIter.hasNext()) {
        AttributeMetaData amd = (AttributeMetaData) attrIter.next();
        String attributeDisplayName = amd.getDisplayName(effectiveLocale);
        System.out.println("\tattribute:" + attributeDisplayName);
    }
}
```

Best match locale

When specifying the conditions on an attached query table, using the value `$LOCALE` can return unexpected results if the specified locale does not match the metadata exactly. For example if you pass a locale `en_US` with a query against a query table that has metadata specifying the language as `en`, the result returned will be `null`.

To avoid such cases, you can use `LOCALE=$LOCALE_BEST_MATCH`, which applies a best-match algorithm to calculate the actual locale used in the query. For example, a query with locale `en_US` against a query table in language `en` is performed using `en`.

You cannot specify any other logical or comparison operators in the condition `LOCALE=$LOCALE_BEST_MATCH`. You can only use the best-match locale condition on attached query tables, specifying it as a condition on other queries results in an error.

Internationalization for query table meta data

Internationalization is supported for query table meta data.

Display names and descriptions can be provided for composite query tables in different locales. For example, a composite query table can define a display name for the query table in the `en_US` locale, the `de` locale, and in the default locale. This is done when the query table is developed using the Query Table Builder. To deploy query tables with localized display names and descriptions, the `-deploy jarFile` option must be used when the query table is deployed on the Business Process Choreographer container.

In terms of locale handling, the behavior of the query table API methods, `queryEntities` and `queryRows`, and the meta data methods of the query table API, `getQueryTableMetaData` and `findQueryTableMetaData`, is similar to that provided by Java resource bundles.

To make the display names and descriptions of the query table meta data consistent with the contents of the query table, the value of the `$LOCALE` system parameter depends on the locales for which display names and descriptions are specified on the query table.

Example

Consider the following scenario of a client which displays task lists or process lists and creates a request to query a query table.

- The client did not specify the locale it uses to present information to the user. It is likely that the application is not enabled for different languages.
 - A default locale is specified on the query table for display names and descriptions. This is the case for all composite and supplemental query tables that are built with the current version of the Query Table Builder. Therefore, the value of `$LOCALE` is set to `default`.
 - The query table does not specify display names or descriptions on the query table for the default locale. This is the case for all predefined query tables and for all query tables that are deployed using the `-deploy qtdFile` option. The value of `$LOCALE` is based on the Java resource bundle method.
- The client specified the locale to use to present information to the user. For example, this is the case when the REST API for query tables is used.
 - Display names and descriptions are specified on the query table. The Java resource bundle method is used to calculate the value of `$LOCALE`, based on the locale that is passed in by the client.
 - Display names and descriptions are not specified on the query table. The value of `$LOCALE` is set to the value that is passed in by the client.

Best match locale

When specifying the conditions on an attached query table, using the value `$LOCALE` can return unexpected results if the specified locale does not match the metadata exactly. For example if you pass a locale `en_US` with a query against a query table that has metadata specifying the language as `en`, the result returned will be `null`.

To avoid such cases, you can use `LOCALE=$LOCALE_BEST_MATCH`, which applies a best-match algorithm to calculate the actual locale used in the query. For example, a query with locale `en_US` against a query table in language `en` is performed using `en`.

You cannot specify any other logical or comparison operators in the condition `LOCALE=$LOCALE_BEST_MATCH`. You can only use the best-match locale condition on attached query tables, specifying it as a condition on other queries results in an error.

Related tasks

“Creating query tables for Business Process Choreographer Explorer” on page 356
You can use query tables instead of the EJB query API to improve the performance of Business Process Choreographer Explorer. To create the query tables, use the Query Table Builder.

Query tables and query performance

Query tables introduce a clean programming model for developing client applications that retrieve lists of human tasks and business processes in Business Process Choreographer. Using query tables improves the performance. Information is provided about the query table API parameters and other factors that affect the performance.

Query response times on query tables depend mainly on the authorization options, filters, and selection criteria that are used. The following are some general performance tips to consider.

- Authorization options have considerable performance impact. Enable authorization using as few options as is possible, such as individual and group work items. Avoid using inherited work items. The authorization options can be further restricted when the query is run. Also, if not needed, specify that authorization using work items is not required.
- If authorization using work items is required, specify an authorization filter. For example, to allow only objects in the query table with a potential owner work item, use `WI.REASON=REASON_POTENTIAL_OWNER`.
- Filtering on the primary query table is efficient, for example, to allow only tasks in the ready state in the query table where `TASK` is the primary query table.
- Filters on the query table, as well as query filters, which are filters that are passed when the query is run, are less efficient as primary filters in terms of performance.
- Avoid, where possible, using parameters in filters and selection criteria.
- Avoid using `LIKE` operators in filters and selection criteria.

Composite query table definition

The following table provides information about the query performance impact of options that are defined on composite query tables. It also provides information

other topics related to composite query table definitions. The impact given in column Performance Impact is an average performance impact, actual impact observations may vary.

Table 56. Query performance impact of composite query table options

Object or topic	Performance impact	Description
Query table filter	Negative	Filters on query tables are the filters with the highest negative impact on query performance. These filters typically cannot use any defined indexes in the database.
Primary query table filter	Positive	A filter on the primary query table provides high performance filtering at a very early stage of the query result set calculation. It is suggested to restrict the contents of the query table using a primary query table filter.
Authorization filter	Positive	A filter on authorization can improve the performance of the query, such as how the primary query table filter improves it. If possible, an authorization filter should be applied. For example, if reader work items should not be considered, specify <code>WI.REASON=REASON_READER</code> .
Selection criteria	None	Some primary query table to attached query table relationships require the definition of a selection criterion in order to meet the one-to-one or one-to-zero relationship. A selection criterion typically has low performance impact because it is evaluated for a small numbers of rows only.
Parameters	None	Currently, using parameters in query tables has no negative performance impact. Nevertheless, parameters should be used only if needed.
Instance-based authorization	Negative	If instance-based authorization is used, each object in the query table must be checked against the existence of a work item. Work items are represented as entries in the <code>WORK_ITEM</code> query table. This verification affects performance.
Instance-based authorization: <ul style="list-style-type: none"> • everybody • individuals • groups • inherited 	Negative	Each type of work item that is specified for use in the query table has a performance impact. Applications with high volume queries should only use individual and group work items, or only one of those. Inherited work items are usually not required, in particular when defining task lists that return human tasks representing to-dos. They should be used only when it is clear that they are needed, for example, to return lists of tasks that belong to a business process where a person might have read access based on the authorization for the enclosing business process.
Role-based authorization or no authorization	None	If role-based authorization or no authorization is used, checks against work items are not made.
Number of defined attributes	Currently none	Currently, the number of attributes contained in a query table has no impact on performance. Nevertheless, only those attributes that are needed should be part of a query table.

Query table API

The following table provides information about the query performance impact of options that are specified on the query table API. The impact given in the Performance impact column is an average performance impact; actual impact observations may vary.

Table 57. Query performance impact of query table API options

Option	Performance impact	Description
Selected attributes	Negative (less is better)	The number of attributes that are selected when a query is run on a query table impacts on the number that need to be processed both by the database and by the Business Process Choreographer query table runtime. Also, for composite query tables, information from attached query tables need be retrieved only if those are either specified by the selected attributes or referenced by the query table filter or by the query filter.
Query filter	Negative	If specified, the query filter currently has the same performance impact as the query table filter. However, it is a good practice if filters are specified on query tables rather than passed into the query table API.
Sort attributes	Negative	The sorting of query result sets is an expensive operation, and database optimizations are restricted if sorting is used. If not needed, sorting should be avoided. Most applications require sorting, however.
Threshold	Positive	The specification of a threshold can greatly improve the performance of queries. It is a best practice to always specify a threshold.
Skip count	Negative	Skipping a particular number of objects in the query result set is expensive and should be done only if required, for example when paging over a query result.
Time zone	None	The time zone setting has no performance impact.
Locale	None	The locale setting has no performance impact.
Distinct rows	Negative	Using distinct in queries has some performance impact but might be necessary in order to retrieve non-duplicate rows. This option impacts only on row based queries and is ignored otherwise.
Count queries	Positive	If only the total number of entities or the number of rows for a particular query is needed, that is, the contents are not needed for all entries of the query table, the method <code>queryEntityCount</code> or <code>queryRowCount</code> should be used. The Business Process Choreographer runtime can apply optimizations that are valid only for count queries.

Other considerations

Other factors to consider with regard to performance are:

Table 58. Query table performance: Other considerations

Item	Description
Number of query tables on the system	The number of query tables which are deployed on a Business Process Choreographer container does not influence the performance of query table queries. Also, currently, it does not influence the navigation of business process instances, nor does it have impact on claim or complete operations on human tasks. Due to maintainability, keep the number of query tables at a reasonable level. Typically, one query table represents one task list or process list which is displayed on the user interface.
Database tuning	Although optimized SQL is used to access the contents of a query table, database tuning needs to be implemented on a Business Process Choreographer database: <ul style="list-style-type: none">• Database memory should be set to a maximum, taking into account other processes that are running on the database server, as well as hardware constraints.• Statistics on the database must be up-to-date, and should be updated on a regular basis. Typically, those procedures are already implemented in large topologies. For example, collect database statistics for the optimizer once per week in order to reflect changes of the data in the database.• Database systems provide tools to reorganize (or defragment) the data containers. The physical layout of the data in a database can also influence query performance and access paths of queries.• Optimal indexes are the key for good query performance. Business Process Choreographer comes with predefined indexes which are optimized for both process navigation and query performance of typical scenarios. In customized environments, additional indexes may be necessary in order to support high volume task or process list queries. Use tools provided by the database in order to support the queries which are run on a query table.

Creating query tables for Business Space

In the Query Table Builder, you can use the composite query table definition that has predefined properties to create query tables for Business Space.

Before you begin

The Query Table Builder is available as an Eclipse plug-in and can be downloaded from the WebSphere Business Process Management SupportPacs site. Look for PA71 WebSphere Process Server - Query Table Builder. To access the link, see the related references section of this topic.

Procedure

1. In the Query Table Builder, right click your project, then select **New** → **Composite Query Definition for Business Space**. Follow the wizard instructions to create a query table definition. The new query table definition consists of the predefined properties. If required, add more properties to the query table definition and deploy the query table definition file to the WebSphere Process Server.

Note: The names you give the properties in the Query Table Builder are used as the names for the task properties in Business Space for Choreographer.

2. After the query table definition files have been created and deployed you can configure them in Business Space. For example, if you have deployed a query table definition file for the Tasks List widget:
 - a. Open the widget menu and select **Configure**, then the **Content** tab.
 - b. On the **Content** tab, open the **Select task list to display** drop down list to display the lists that you can make available to the user of the widget. Select **Add task lists**. The query table definition you deployed should be available in this list for selection.

If the query table definition is not available you need to go back to the Query Table Builder and check whether the definition file was correctly defined and deployed.

Creating query tables for Business Process Choreographer Explorer

You can use query tables instead of the EJB query API to improve the performance of Business Process Choreographer Explorer. To create the query tables, use the Query Table Builder.

Before you begin

The Query Table Builder is available as an Eclipse plug-in and can be downloaded from the WebSphere Business Process Management SupportPacs site. Look for PA71 WebSphere Process Server - Query Table Builder. To access the link, see the related references section of this topic.

Procedure

1. In the Query Table Builder, right click your project, then select **New** → **Composite Query Definition for Business Space**. This option ensures that all of the columns that are required for Business Process Choreographer Explorer are preselected.
2. Follow the wizard instructions to create a query table definition. If required, add more properties to the query table definition. Consider the following aspects when you define your query table:

Filter criteria

When you create views in Business Process Choreographer Explorer based on query tables, you cannot specify additional filter or variables for your search criteria. You must specify these filter criteria and the parameters for the variables when you create the query table.

You can use a query table for more than one view in Business Process Choreographer Explorer by using parameters in the query table definition. For more flexibility, you can also specify whether the default values of the parameters can be overwritten when the query for the custom view is run.

Authorization

When you create views in Business Process Choreographer Explorer based on query tables, you cannot filter your search criteria based on the user role. You must set the filter criteria for user roles when you define the query table. For primary query tables based on template information, use instance-based authorization as the authorization type

and not role-based authorization. For primary query tables based on instance information, specify the appropriate instance-based authorization filter.

Internationalization

When you define properties in the Query Table Builder, you can also specify the names and descriptions of these properties in different languages. When the query for the customized view is run, Business Process Choreographer Explorer uses the translation that is appropriate for the language setting of your browser.

Display name and description for the query table definition

In the Query Table Builder, you can provide a display name and description for all of the languages that are supported by the view.

Display name and description for the columns

At runtime, Business Process Choreographer Explorer retrieves the appropriate internationalized column names that are displayed in a result list. For columns that come from your primary query table, such as PIID, Business Process Choreographer Explorer uses the translations that are already available for all of the supported languages.

For columns that come from an attached query table, such as QUERY_PROPERTY, you need to provide display names and descriptions in the Query Table Builder in all of the languages that are supported by your business.

Task names and description

If you have internationalized task names and descriptions in WebSphere Integration Developer, they are displayed in Business Process Choreographer Explorer according to the language and country settings of your browser. If your browser settings do not match the settings that are defined in the process model, the translation of the default language is used.

Sort criteria

When you define sort criteria for a query table definition, be aware that several properties, for example, process state, are stored as integer values, while Business Process Choreographer Explorer displays them as translated strings in the resulting list. This might produce unexpected sorting results in some languages.

The new query table definition consists of the predefined properties and any additional properties that you define.

What to do next

Deploy and test the query definition in the Query Table Builder on an application server. If this is the server to which Business Process Choreographer Explorer is connected, you can now use the query table when you customize Business Process Choreographer Explorer for your own use, or for different user groups. If Business Process Choreographer Explorer is connected to a different server, you must deploy the query table on the appropriate server before you can use it to create customized views.

Related concepts

“Authorization for query tables” on page 327

You can use instance-based authorization, role-based authorization, or no authorization when you run queries on query tables.

“Filters and selection criteria of query tables” on page 322

Filters and selection criteria are defined during query table development using the Query Table Builder, which uses a syntax similar to SQL WHERE clauses. Use these clearly defined filters and selection criteria to specify conditions that are based on attributes of query tables.

“Internationalization for query table meta data” on page 351

Internationalization is supported for query table meta data.

Business Process Choreographer EJB query API

Use the query method or the queryAll method of the service API to retrieve stored information about business processes and tasks.

The query method can be called by all users, and it returns the properties of the objects for which work items exist. The queryAll method can be called only by users who have one of the following Java EE roles: BPSystemAdministrator, TaskSystemAdministrator, BPSystemMonitor, or TaskSystemMonitor. This method returns the properties of all the objects that are stored in the database.

All API queries are mapped to SQL queries. The form of the resulting SQL query depends on the following aspects:

- Whether the query was invoked by someone with one of the Java EE roles.
- The objects that are queried. Predefined database views are provided for you to query the object properties.
- The insertion of a from clause, join conditions, and user-specific conditions for access control.

You can include both custom properties and variable properties in queries. If you include several custom properties or variable properties in your query, this results in self-joins on the corresponding database table. Depending on your database system, these query() calls might have performance implications.

You can also store queries in the Business Process Choreographer database using the createStoredQuery method. You provide the query criteria when you define the stored query. The criteria are applied dynamically when the stored query runs, that is, the data is assembled at runtime. If the stored query contains parameters, these are also resolved when the query runs.

For more information on the Business Process Choreographer APIs, see the Javadoc in the com.ibm.bpe.api package for process-related methods and in the com.ibm.task.api package for task-related methods.

Syntax of the API query method

The syntax of the Business Process Choreographer API queries is similar to SQL queries. A query can include a select clause, a where clause, an order-by clause, a skip-tuples parameter, a threshold parameter and a time-zone parameter.

The syntax of the query depends on the object type. The following table shows the syntax for each of the different object types.

Table 59. Query syntax for different object types

Object	Syntax
Process template	ProcessTemplateData[] queryProcessTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);
Task template	TaskTemplate[] queryTaskTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);
Business-process and task-related data	QueryResultSet query (java.lang.String selectClause, java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer skipTuples java.lang.Integer threshold, java.util.TimeZone timezone);

Select clause:

The select clause in the query function identifies the object properties that are to be returned by a query.

The select clause describes the query result. It specifies a list of names that identify the object properties (columns of the result) to return. Its syntax is similar to the syntax of an SQL SELECT clause; use commas to separate parts of the clause. Each part of the clause must specify a column from one of the predefined views. The columns must be fully specified by view name and column name. The columns returned in the QueryResultSet object appear in the same order as the columns specified in the select clause.

The select clause does not support SQL aggregation functions, such as AVG(), SUM(), MIN(), or MAX().

To select the properties of multiple name-value pairs, such as custom properties and properties of variables that can be queried, add a one-digit counter to the view name. This counter can take the values 1 through 9.

Examples of select clauses

- "WORK_ITEM.OBJECT_TYPE, WORK_ITEM.REASON"
Gets the object types of the associated objects and the assignment reasons for the work items.
- "DISTINCT WORK_ITEM.OBJECT_ID"
Gets all of the IDs of objects, without duplicates, for which the caller has a work item.
- "ACTIVITY.TEMPLATE_NAME, WORK_ITEM.REASON"
Gets the names of the activities the caller has work items for and their assignment reasons.
- "ACTIVITY.STATE, PROCESS_INSTANCE.STARTER"
Gets the states of the activities and the starters of their associated process instances.
- "DISTINCT TASK.TKIID, TASK.NAME"

Gets all of the IDs and names of tasks, without duplicates, for which the caller has a work item.

- "TASK_CPROP1.STRING_VALUE, TASK_CPROP2.STRING_VALUE"
Gets the values of the custom properties that are specified further in the where clause.
- "QUERY_PROPERTY1.STRING_VALUE, QUERY_PROPERTY2.INT_VALUE"
Gets the values of the properties of variables that can be queried. These parts are specified further in the where clause.
- "COUNT(DISTINCT TASK.TKIID)"
Counts the number of work items for unique tasks that satisfy the where clause.

Where clause:

The where clause in the query function describes the filter criteria to apply to the query domain.

The syntax of a where clause is similar to the syntax of an SQL WHERE clause. You do not need to explicitly add an SQL from clause or join predicates to the API where clause, these constructs are added automatically when the query runs. If you do not want to apply filter criteria, you must specify null for the where clause.

The where-clause syntax supports:

- Keywords: AND, OR, NOT
- Comparison operators: =, <=, <, <>, >, >=, LIKE
The LIKE operation supports the wildcard characters that are defined for the queried database.
- Set operation: IN

The following rules also apply:

- Specify object ID constants as ID('string-rep-of-oid').
- Specify binary constants as BIN('UTF-8 string').
- Use symbolic constants instead of integer enumerations. For example, instead of specifying an activity state expression ACTIVITY.STATE=2, specify ACTIVITY.STATE=ACTIVITY.STATE.STATE_READY.
- If the value of the property in the comparison statement contains single quotation marks ('), double the quotation marks, for example, "TASK_CPROP.STRING_VALUE='d'automatisation".
- Refer to properties of multiple name-value pairs, such as custom properties, by adding a one-digit suffix to the view name. For example: "TASK_CPROP1.NAME='prop1' AND "TASK_CPROP2.NAME='prop2'"
- Specify time-stamp constants as TS('yyyy-mm-ddThh:mm:ss'). To refer to the current date, specify CURRENT_DATE as the timestamp.

You must specify at least a date or a time value in the timestamp:

- If you specify a date only, the time value is set to zero.
- If you specify a time only, the date is set to the current date.
- If you specify a date, the year must consist of four digits; the month and day values are optional. Missing month and day values are set to 01. For example, TS('2003') is the same as TS('2003-01-01T00:00:00').

- If you specify a time, these values are expressed in the 24-hour system. For example, if the current date is 1 January 2003, TS('T16:04') or TS('16:04') is the same as TS('2003-01-01T16:04:00').

Examples of where clauses

- Comparing an object ID with an existing ID
`"WORK_ITEM.WIID = ID('_WI:800c00ed.df8d7e7c.feffff80.38')"`

This type of where clause is usually created dynamically with an existing object ID from a previous call. If this object ID is stored in a *wiid1* variable, the clause can be constructed as:

```
"WORK_ITEM.WIID = ID('" + wiid1.toString() + "')"
```

- Using time stamps
`"ACTIVITY.STARTED >= TS('2002-06-1T16.00.00')"`
- Using symbolic constants
`"WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER"`
- Using Boolean values true and false
`"ACTIVITY.BUSINESS_RELEVANCE = TRUE"`
- Using custom properties
`"TASK_CPROP1.NAME = 'prop1' AND " TASK_CPROP1.STRING_VALUE = 'v1' AND
TASK_CPROP2.NAME = 'prop2' AND " TASK_CPROP2.STRING_VALUE = 'v2'"`

Order-by clause:

The order-by clause in the query function specifies the sort criteria for the query result set.

You can specify a list of columns from the views by which the result is sorted. These columns must be fully qualified by the name of the view and the column.

The order-by clause syntax is similar to the syntax of an SQL order-by clause; use commas to separate each part of the clause. You can also specify ASC to sort the columns in ascending order, and DESC to sort the columns in descending order. If you do not want to sort the query result set, you must specify null for the order-by clause.

Sort criteria are applied on the server, that is, the locale of the server is used for sorting. If you specify more than one column, the query result set is ordered by the values of the first column, then by the values of the second column, and so on. You cannot specify the columns in the order-by clause by position as you can with an SQL query.

Examples of order-by clauses

- "PROCESS_TEMPLATE.NAME"
Sorts the query result alphabetically by the process-template name.
- "PROCESS_INSTANCE.CREATED, PROCESS_INSTANCE.NAME DESC"
Sorts the query result by the creation date and, for a specific date, sorts the results alphabetically by the process-instance name in reverse order.
- "ACTIVITY.OWNER, ACTIVITY.TEMPLATE_NAME, ACTIVITY.STATE"
Sorts the query result by the activity owner, then the activity-template name, and then the state of the activity.

Skip-tuples parameter:

The skip-tuples parameter specifies the number of query-result-set tuples from the beginning of the query result set that are to be ignored and not to be returned to the caller in the query result set.

Use this parameter with the threshold parameter to implement paging in a client application, for example, to retrieve the first 20 items, then the next 20 items, and so on.

If this parameter is set to `null` and the threshold parameter is not set, all of the qualifying tuples are returned.

Example of a skip-tuples parameter

- `new Integer(5)`
Specifies that the first five qualifying tuples are not to be returned.

Threshold parameter:

The threshold parameter in the query function restricts the number of objects returned from the server to the client in the query result set.

Because query result sets in production scenarios can contain thousands or even millions of items, specify a value for the threshold parameter. If you set the threshold parameter accordingly, the database query is faster and less data needs to transfer from the server to the client. The threshold parameter can be useful, for example, in a graphical user interface where only a small number of items should be displayed at one time.

If this parameter is set to `null` and the skip-tuples parameter is not set, all of the qualifying objects are returned.

Example of a threshold parameter

- `new Integer(50)`
Specifies that 50 qualifying tuples are to be returned.

Timezone parameter:

The time-zone parameter in the query function defines the time zone for time-stamp constants in the query.

Time zones can differ between the client that starts the query and the server that processes the query. Use the time-zone parameter to specify the time zone of the time-stamp constants used in the where clause, for example, to specify local times. The dates returned in the query result set have the same time zone that is specified in the query.

If the parameter is set to `null`, the timestamp constants are assumed to be Coordinated Universal Time (UTC) times.

Examples of time-zone parameters

- ```
process.query("ACTIVITY.AIID",
 "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
 (String)null,
 (Integer)null,
 java.util.TimeZone.getDefault());
```

Returns object IDs for activities that started later than 17:40 local time on 1 January 2005.



```

• process.query("ACTIVITY.AIID",
 "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
 (String)null, (Integer)null, (TimeZone)null);

```

Return object IDs for activities that started later than 17:40 UTC on 1 January 2005. This specification is, for example, 6 hours earlier in Eastern Standard Time.

### Filtering data using variables in queries:

A query result returns the objects that match the query criteria. You might want to filter these results on the values of variables.

### About this task

You can define variables that are used by a process at runtime in its process model. For these variables, you declare which parts can be queried.

For example, John Smith, calls his insurance company's service number to find out the progress of his insurance claim for his damaged car. The claims administrator uses the customer ID to the find the claim.

### Procedure

1. Optional: List the properties of the variables in a process that can be queried.

Use the process template ID to identify the process. You can skip this step if you know which variables can be queried.

```

List variableProperties = process.getQueryProperties(ptid);
for (int i = 0; i < variableProperties.size(); i++)
{
 QueryProperty queryData = (QueryProperty)variableProperties.get(i);
 String variableName = queryData.getVariableName();
 String name = queryData.getName();
 int mappedType = queryData.getMappedType();
 ...
}

```

2. List the process instances with variables that match the filter criteria.

For this process, the customer ID is modeled as part of the variable customerClaim that can be queried. You can therefore use the customer's ID to find the claim.

```

QueryResultSet result = process.query
("PROCESS_INSTANCE.NAME, QUERY_PROPERTY.STRING_VALUE",
 "QUERY_PROPERTY.VARIABLE_NAME = 'customerClaim' AND " +
 "QUERY_PROPERTY.NAME = 'customerID' AND " +
 "QUERY_PROPERTY.STRING_VALUE like 'Smith%'",
 (String)null, (Integer)null,
 (Integer)null, (TimeZone)null);

```

This action returns a query result set that contains the process instance names and the values of the customer IDs for customers whose IDs start with Smith.

### Query results:

A query result set contains the results of a Business Process Choreographer API query.

The elements of the result set are properties of the objects that satisfy the where clause given by the caller, and that the caller is authorized to see. You can read elements in a relative fashion using the API next method or in an absolute fashion using the first and last methods. Because the implicit cursor of a query result set is

initially positioned before the first element, you must call either the first or next methods before reading an element. You can use the size method to determine the number of elements in the set.

An element of the query result set comprises the selected attributes of work items and their associated referenced objects, such as activity instances and process instances. The first attribute (column) of a QueryResultSet element specifies the value of the first attribute specified in the select clause of the query request. The second attribute (column) of a QueryResultSet element specifies the value of the second attribute specified in the select clause of the query request, and so on.

You can retrieve the values of the attributes by calling a method that is compatible with the attribute type and by specifying the appropriate column index. The numbering of the column indexes starts with 1.

| Attribute type | Method                                                       |
|----------------|--------------------------------------------------------------|
| String         | getString                                                    |
| OID            | getOID                                                       |
| Timestamp      | getTimestamp<br>getString<br>getTimestampAsLong              |
| Integer        | getInteger<br>getShort<br>getLong<br>getString<br>getBoolean |
| Boolean        | getBoolean<br>getShort<br>getInteger<br>getLong<br>getString |
| byte[]         | getBinary                                                    |

### Example:

The following query is run:

```
QueryResultSet resultSet = process.query("ACTIVITY.STARTED,
 ACTIVITY.TEMPLATE_NAME AS NAME,
 WORK_ITEM.WIID, WORK_ITEM.REASON",
 (String)null, (String)null,
 (Integer)null, (TimeZone)null);
```

The returned query result set has four columns:

- Column 1 is a time stamp
- Column 2 is a string
- Column 3 is an object ID
- Column 4 is an integer

You can use the following methods to retrieve the attribute values:

```
while (resultSet.next())
{
 java.util.Calendar activityStarted = resultSet.getTimestamp(1);
```

```
String templateName = resultSet.getString(2);
WIID wiid = (WIID) resultSet.getOID(3);
Integer reason = resultSet.getInteger(4);
}
```

You can use the display names of the result set, for example, as headings for a printed table. These names are the column names of the view or the name defined by the AS clause in the query. You can use the following method to retrieve the display names in the example:

```
resultSet.getColumnDisplayName(1) returns "STARTED"
resultSet.getColumnDisplayName(2) returns "NAME"
resultSet.getColumnDisplayName(3) returns "WIID"
resultSet.getColumnDisplayName(4) returns "REASON"
```

## User-specific access conditions

User-specific access conditions are added when the SQL SELECT statement is generated from the API query. These conditions guarantee that only those objects are returned to the caller that satisfy the condition specified by the caller and to which the caller is authorized.

The access condition that is added depends on whether the user is a system administrator.

## Queries invoked by users who are not system administrators

The generated SQL WHERE clause combines the API where clause with an access control condition that is specific to the user. The query retrieves only those objects that the user is authorized to access, that is, only those objects for which the user has a work item. A work item represents the assignment of a user or user group to an authorization role of a business object, such as a task or process. If, for example, the user, John Smith, is a member of the potential owners role of a given task, a work item object exists that represents this relationship.

For example, if a user, who is not a system administrator, queries tasks, the following access condition is added to the WHERE clause if group work items are not enabled:

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND (WI.OWNER_ID = 'user'
 OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
```

So, if John Smith wants to get a list of tasks for which he is the potential owner, the API where clause might look as follows:

```
"WORK_ITEM.REASON == WORK_ITEM.REASON.REASON_POTENTIAL_OWNER"
```

This API where clause results in the following access condition in the SQL statement:

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND (WI.OWNER_ID = 'JohnSmith'
 OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
AND WI.REASON = 1
```

This also means that if John Smith wants to see the activities and tasks for which he is a process reader or a process administrator and for which he does not have a work item, then a property from the PROCESS\_INSTANCE view must be added to the select, where, or order-by clause of the query, for example, PROCESS\_INSTANCE.PIID.

If group work items are enabled, an additional access condition is added to the WHERE clause that allows a user to access objects that the group has access to.

### Queries invoked by system administrators

System administrators can invoke the query method to retrieve objects that have associated work items. In this case, a join with the WORK\_ITEM view is added to the generated SQL query, but no access control condition for the WORK\_ITEM.OWNER\_ID.

In this case, the SQL query for tasks contains the following:

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
```

### queryAll queries

This type of query can be invoked only by system administrators or system monitors. Neither conditions for access control nor a join to the WORK\_ITEM view are added. This type of query returns all of the data for all of the objects.

### Examples of the query and queryAll methods

These examples show the syntax of various typical API queries and the associated SQL statements that are generated when the query is processed.

#### Example: Querying tasks in the ready state:

This example shows how to use the query method to retrieve tasks that the logged-on user can work with.

John Smith wants to get a list of the tasks that have been assigned to him. For a user to be able to work on a task, the task must be in the ready state. The logged-on user must also have a potential owner work item for the task. The following code snippet shows the query method call for this query:

```
query("DISTINCT TASK.TKIID",
 "TASK.KIND IN (TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING)
 AND " +
 "TASK.STATE = TASK.STATE.STATE_READY AND " +
 "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
 (String)null, (String)null, (Integer)null, (TimeZone)null)
```

The following actions are taken when the SQL SELECT statement is generated:

- A condition for access control is added to the where clause. This example assumes that group work items are not enabled.
- Constants, such as TASK.STATE.STATE\_READY, are replaced by their numeric values.
- A FROM clause and join conditions are added.

The following code snippet shows the SQL statement that is generated from the API query:

```
SELECT DISTINCT TASK.TKIID
FROM TASK TA, WORK_ITEM WI,
WHERE WI.OBJECT_ID = TA.TKIID
AND TA.KIND IN (101, 105)
AND TA.STATE = 2
AND WI.REASON = 1
AND (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
```

To restrict the API query to tasks for a specific process, for example, `sampleProcess`, the query looks as follows:

```
query("DISTINCT TASK.TKIID",
 "PROCESS_TEMPLATE.NAME = 'sampleProcess' AND "+
 "TASK.KIND IN (TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING)
 AND " +
 "TASK.STATE = TASK.STATE.STATE_READY AND " +
 "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
 (String)null, (String)null, (Integer)null, (TimeZone)null)
```

#### Example: Querying tasks in the claimed state:

This example shows how to use the query method to retrieve tasks that the logged-on user has claimed.

The user, John Smith, wants to search for tasks that he has claimed and are still in the claimed state. The condition that specifies "claimed by John Smith" is `TASK.OWNER = 'JohnSmith'`. The following code snippet shows the query method call for the query:

```
query("DISTINCT TASK.TKIID",
 "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
 "TASK.OWNER = 'JohnSmith'",
 (String)null, (String)null, (Integer)null, (TimeZone)null)
```

The following code snippet shows the SQL statement that is generated from the API query:

```
SELECT DISTINCT TASK.TKIID
 FROM TASK TA, WORK_ITEM WI,
 WHERE WI.OBJECT_ID = TA.TKIID
 AND TA.STATE = 8
 AND TA.OWNER = 'JohnSmith'
 AND (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
```

When a task is claimed, work items are created for the owner of the task. So, an alternative way of forming the query for John Smith's claimed tasks is to add the following condition to the query instead of using `TASK.OWNER = 'JohnSmith'`:

```
WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER
```

The query then looks like the following code snippet:

```
query("DISTINCT TASK.TKIID",
 "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
 "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER",
 (String)null, (String)null, (Integer)null, (TimeZone)null)
```

The following actions are taken when the SQL SELECT statement is generated:

- A condition for access control is added to the where clause. This example assumes that group work items are not enabled.
- Constants, such as `TASK.STATE.STATE_READY`, are replaced by their numeric values.
- A FROM clause and join conditions are added.

The following code snippet shows the SQL statement that is generated from the API query:

```
SELECT DISTINCT TASK.TKIID
 FROM TASK TA, WORK_ITEM WI,
 WHERE WI.OBJECT_ID = TA.TKIID
```

```

AND TA.STATE = 8
AND WI.REASON = 4
AND (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true)

```

John is about to go on vacation so his team lead, Anne Grant, wants to check on his current work load. Anne has system administrator rights. The query she invokes is the same as the one John invoked. However, the SQL statement that is generated is different because Anne is an administrator. The following code snippet shows the generated SQL statement:

```

SELECT DISTINCT TASK.TKIID
FROM TASK TA, WORK_ITEM WI,
WHERE TA.TKIID = WI.OBJECT_ID =
AND TA.STATE = 8
AND TA.OWNER = 'JohnSmith')

```

Because Anne is an administrator, an access control condition is not added to the WHERE clause.

#### Example: Querying escalations:

This example shows how to use the query method to retrieve escalations for the logged-on user.

When a task is escalated, and escalation receiver work item is created. The user, Mary Jones wants to see a list of tasks that have been escalated to her. The following code snippet shows the query method call for the query:

```

query("DISTINCT ESCALATION.ESIID, ESCALATION.TKIID",
 "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_ESCALATION_RECEIVER",
 (String)null, (String)null, (Integer)null, (TimeZone)null)

```

The following actions are taken when the SQL SELECT statement is generated:

- A condition for access control is added to the where clause. This example assumes that group work items are not enabled.
- Constants, such as TASK.STATE.STATE\_READY, are replaced by their numeric values.
- A FROM clause and join conditions are added.

The following code snippet shows the SQL statement that is generated from the API query:

```

SELECT DISTINCT ESCALATION.ESIID, ESCALATION.TKIID
FROM ESCALATION ESC, WORK_ITEM WI
WHERE ESC.ESIID = WI.OBJECT_ID
AND WI.REASON = 10
AND
(WI.OWNER_ID = 'MaryJones' OR WI.OWNER_ID = null AND WI.EVERYBODY = true)

```

#### Example: Using the queryAll method:

This example shows how to use the queryAll method to retrieve all of the activities that belong to a process template.

The queryAll method is available only to users with system administrator or system monitor rights. The following code snippet shows the queryAll method call for the query to retrieve all of the activities that belong to the process template, sampleProcess:

```
queryAll("DISTINCT ACTIVITY.AIID",
 "PROCESS_TEMPLATE.NAME = 'sampleProcess'",
 (String)null, (String)null, (Integer)null, (TimeZone)null)
```

The following code snippet shows the SQL query that is generated from the API query:

```
SELECT DISTINCT ACTIVITY.AIID
FROM ACTIVITY AI, PROCESS_TEMPLATE PT
WHERE AI.PTID = PT.PTID
AND PT.NAME = 'sampleProcess'
```

Because the call is invoked by an administrator, an access control condition is not added to the generated SQL statement. A join with the WORK\_ITEM view is also not added. This means that the query retrieves all of the activities for the process template, including those activities without work items.

#### Example: Including query properties in a query:

This example shows how to use the query method to retrieve tasks that belong to a business process. The process has query properties defined for it that you want to include in the search.

For example, you want to search for all of the human tasks in the ready state that belong to a business process. The process has a query property, **customerID**, with the value CID\_12345, and a namespace. The following code snippet shows the query method call for the query:

```
query (" DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
 PROCESS_INSTANCE.NAME",
 " QUERY_PROPERTY.NAME = 'customerID' AND " +
 " QUERY_PROPERTY.STRING_VALUE = 'CID_12345' AND " +
 " QUERY_PROPERTY.NAMESPACE =
 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
 " TASK.KIND IN
 (TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING) AND " +
 " TASK.STATE = TASK.STATE.STATE_READY ",
 (String)null, (String)null, (Integer)null, (TimeZone)null);
```

If you now want to add a second query property to the query, for example, **Priority**, with a given namespace, the query method call for the query looks as follows:

```
query (" DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
 PROCESS_INSTANCE.NAME",
 " QUERY_PROPERTY1.NAME = 'customerID' AND " +
 " QUERY_PROPERTY1.STRING_VALUE = 'CID_12345' AND " +
 " QUERY_PROPERTY1.NAMESPACE =
 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
 " QUERY_PROPERTY2.NAME = 'Priority' AND " +
 " QUERY_PROPERTY2.NAMESPACE =
 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
 " TASK.KIND IN
 (TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING) AND " +
 " TASK.STATE = TASK.STATE.STATE_READY ",
 (String)null, (String)null, (Integer)null, (TimeZone)null);
```

If you add more than one query property to the query, you must number each of the properties that you add as shown in the code snippet. However, querying custom properties affects performance; performance decreases with the number of custom properties in the query.

#### Example: Including custom properties in a query:

This example shows how to use the query method to retrieve tasks that have custom properties.

For example, you want to search for all of the human tasks in the ready state that have a custom property, **customerID**, with the value CID\_12345. The following code snippet shows the query method call for the query:

```
query (" DISTINCT TASK.TKIID ",
 " TASK_CPROP.NAME = 'customerID' AND " +
 " TASK_CPROP.STRING_VALUE = 'CID_12345' AND " +
 " TASK.KIND IN
 (TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING) AND " +
 " TASK.STATE = TASK.STATE.STATE_READY ",
 (String)null, (String)null, (Integer)null, (TimeZone)null);
```

If you now want to retrieve the tasks and their custom properties, the query method call for the query looks as follows:

```
query (" DISTINCT TASK.TKIID, TASK_CPROP.NAME, TASK_CPROP.STRING_VALUE",
 " TASK.KIND IN
 (TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING) AND " +
 " TASK.STATE = TASK.STATE.STATE_READY ",
 (String)null, (String)null, (Integer)null, (TimeZone)null);
```

The SQL statement that is generated from this API query is shown in the following code snippet:

```
SELECT DISTINCT TA.TKIID , TACP.NAME , TACP.STRING VALUE
FROM TASK TA LEFT JOIN TASK_CPROP TACP ON (TA.TKIID = TACP.TKIID),
WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND TA.KIND IN (101, 105)
AND TA.STATE = 2
AND (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID IS NULL AND WI.EVERYBODY = 1)
```

This SQL statement contains an outer join between the TASK view and the TASK\_CPROP view. This means that tasks that satisfy the WHERE clause are retrieved even if they do not have any custom properties.

## Managing stored queries

Stored queries provide a way to save queries that are run often. The stored query can be either a query that is available to all users (public query), or a query that belongs to a specific user (private query).

### About this task

A stored query is a query that is stored in the database and identified by a name. A private and a public stored query can have the same name; private stored queries from different owners can also have the same name.

You can have stored queries for business process objects, task objects, or a combination of these two object types.

### Related concepts

“Parameters in stored queries”

A stored query is a query that is stored in the database and identified by a name. The qualifying tuples are assembled dynamically when the query is run. To make stored queries reusable, you can use parameters in the query definition that are resolved at runtime.

### Parameters in stored queries:



A stored query is a query that is stored in the database and identified by a name. The qualifying tuples are assembled dynamically when the query is run. To make stored queries reusable, you can use parameters in the query definition that are resolved at runtime.

For example, you have defined custom properties to store customer names. You can define queries to return the tasks that are associated with a particular customer, ACME Co. To query this information, the where clause in your query might look similar to the following example:

```
String whereClause =
 "TASK.STATE = TASK.STATE.STATE_READY
 AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
 AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = 'ACME Co.'";
```

To make this query reusable so that you can also search for the customer, BCME Ltd, you can use parameters for the values of the custom property. If you add parameters to the task query, it might look similar to the following example:

```
String whereClause =
 "TASK.STATE = TASK.STATE.STATE_READY
 AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
 AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = '@param1'";
```

The @param1 parameter is resolved at runtime from the list of parameters that is passed to the query method. The following rules apply to the use of parameters in queries:

- Parameters can only be used in the where clause.
- Parameters are strings.
- Parameters are replaced at runtime using string replacement. If you need special characters you must specify these in the where clause or passed-in at runtime as part of the parameter.
- Parameter names consist of the string @param concatenated with an integer number. The lowest number is 1, which points to the first item in the list of parameters that is passed to the query API at runtime.
- A parameter can be used multiple times within a where clause; all occurrences of the parameter are replaced by the same value.

#### **Related tasks**

“Managing stored queries” on page 370

Stored queries provide a way to save queries that are run often. The stored query can be either a query that is available to all users (public query), or a query that belongs to a specific user (private query).

#### **Managing public stored queries:**

Public stored queries are created by the system administrator. These queries are available to all users.

#### **About this task**

As the system administrator, you can create, view, and delete public stored queries. If you do not specify a user ID in the API call, it is assumed that the stored query is a public stored query.

#### **Procedure**

1. Create a public stored query.

For example, the following code snippet creates a stored query for process instances and saves it with the name `CustomerOrdersStartingWithA`.

```
process.createStoredQuery("CustomerOrdersStartingWithA",
 "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
 "PROCESS_INSTANCE.NAME LIKE 'A%'",
 "PROCESS_INSTANCE.NAME",
 (Integer)null, (TimeZone)null);
```

The result of the stored query is a sorted list of all the process-instance names that begin with the letter A and their associated process instance IDs (PIID).

2. Run the query defined by the stored query.

```
QueryResultSet result = process.query("CustomerOrdersStartingWithA",
 new Integer(0), null);
```

This action returns the objects that fulfill the criteria. In this case, all of the customer orders that begin with A.

3. List the names of the available public stored queries.

The following code snippet shows how to limit the list of returned queries to just the public queries.

```
String[] storedQuery = process.getStoredQueryNames(StoredQueryData.KIND_PUBLIC);
```

4. Optional: Check the query that is defined by a specific stored query.

A stored private query can have the same name as a stored public query. If these names are the same, the private stored query is returned. The following code snippet shows how to return only the public query with the specified name. If you use the Human Task Manager API to retrieve information about a stored query, use `StoredQuery` for the returned object instead of `StoredQueryData`.

```
StoredQueryData storedQuery = process.getStoredQuery
 (StoredQueryData.KIND_PUBLIC, "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

5. Delete a public stored query.

The following code snippet shows how to delete the stored query that you created in step 1.

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

### Managing private stored queries for other users:

Private queries can be created by any user. These queries are available only to the owner of the query and the system administrator.

#### About this task

As the system administrator, you can manage private stored queries that belong to a specific user.

#### Procedure

1. Create a private stored query for the user ID Smith.

For example, the following code snippet creates a stored query for process instances and saves it with the name `CustomerOrdersStartingWithA` for the user ID Smith.

```
process.createStoredQuery("Smith", "CustomerOrdersStartingWithA",
 "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
 "PROCESS_INSTANCE.NAME LIKE 'A%'",
 "PROCESS_INSTANCE.NAME",
 (Integer)null, (TimeZone)null,
 (List)null, (String)null);
```

The result of the stored query is a sorted list of all the process-instance names that begin with the letter A and their associated process instance IDs (PIID).

2. Run the query defined by the stored query.

```
QueryResultSet result = process.query
 ("Smith", "CustomerOrdersStartingWithA",
 (Integer)null, (Integer)null, (List)null);
new Integer(0));
```

This action returns the objects that fulfill the criteria. In this case, all of the customer orders that begin with A.

3. Get a list of the names of the private queries that belong to a specific user.

For example, the following code snippet shows how to get a list of private queries that belongs to the user Smith.

```
String[] storedQuery = process.getStoredQueryNames("Smith");
```

4. View the details of a specific query.

The following code snippet shows how to view the details of the CustomerOrdersStartingWithA query that is owned by the user Smith.

```
StoredQueryData storedQuery = process.getStoredQuery
 ("Smith", "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

If you use the Human Task Manager API to retrieve information about a stored query, use `StoredQuery` for the returned object instead of `StoredQueryData`.

5. Delete a private stored query.

The following code snippet shows how to delete a private query that is owned by the user Smith.

```
process.deleteStoredQuery("Smith", "CustomerOrdersStartingWithA");
```

### Working with your private stored queries:

If you are not a system administrator, you can create, run, and delete your own private stored queries. You can also use the public stored queries that the system administrator created.

#### Procedure

1. Create a private stored query.

For example, the following code snippet creates a stored query for process instances and saves it with a specific name. If a user ID is not specified, it is assumed that the stored query is a private stored query for the logged-on user.

```
process.createStoredQuery("CustomerOrdersStartingWithA",
 "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
 "PROCESS_INSTANCE.NAME LIKE 'A%'",
 "PROCESS_INSTANCE.NAME",
 (Integer)null, (TimeZone)null);
```

This query returns a sorted list of all the process-instance names that begin with the letter A and their associated process instance IDs (PIID).

2. Run the query defined by the stored query.

```
QueryResultSet result = process.query("CustomerOrdersStartingWithA",
 new Integer(0));
```

This action returns the objects that fulfill the criteria. In this case, all of the customer orders that begin with A.

3. Get a list of the names of the stored queries that the logged-on user can access.

The following code snippet shows how to get both the public and the private stored queries that the user can access.

```
String[] storedQuery = process.getStoredQueryNames();
```

4. View the details of a specific query.

The following code snippet shows how to view the details of the CustomerOrdersStartingWithA query that is owned by the user Smith.

```
StoredQueryData storedQuery = process.getStoredQuery
 ("CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

If you use the Human Task Manager API to retrieve information about a stored query, use StoredQuery for the returned object instead of StoredQueryData.

5. Delete a private stored query.

The following code snippet shows how to delete a private stored query.

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

---

## Developing EJB client applications for business processes and human tasks

The EJB APIs provide a set of generic methods for developing EJB client applications for working with the business processes and human tasks that are installed on a WebSphere Process Server.

### About this task

With these Enterprise JavaBeans (EJB) APIs, you can create client applications to do the following:

- Manage the life cycle of processes and tasks from starting them through to deleting them when they complete
- Repair activities and processes
- Manage and distribute the workload over members of a work group

The EJB APIs are provided as two stateless session enterprise beans:

- BusinessFlowManagerService interface provides the methods for business process applications
- HumanTaskManagerService interface provides the methods for task-based applications

For more information on the EJB APIs, see the Javadoc in the com.ibm.bpe.api package and the com.ibm.task.api package.

The following steps provide an overview of the actions you need to take to develop an EJB client application.

## Procedure

1. Decide on the functionality that the application is to provide.
2. Decide which of the session beans that you are going to use.  
Depending on the scenarios that you want to implement with your application, you can use one, or both, of the session beans.
3. Determine the authorization authorities needed by users of the application.  
The users of your application must be assigned the appropriate authorization roles to call the methods that you include in your application, and to view the objects and the attributes of these objects that these methods return. When an instance of the appropriate session bean is created, WebSphere Application Server associates a context with the instance. The context contains information about the caller's principal ID, group membership list, and roles. This information is used to check the caller's authorization for each call.  
The Javadoc contains authorization information for each of the methods.
4. Decide how to render the application.  
The EJB APIs can be called locally or remotely.
5. Develop the application.
  - a. Access the EJB API.
  - b. Use the EJB API to interact with processes or tasks.
    - Query the data.
    - Work with the data.

### Related concepts

Alternate administration authorization mode

### Related reference

“BusinessFlowManagerService interface” on page 401

The BusinessFlowManagerService interface exposes business-process functions that can be called by a client application.

“HumanTaskManagerService interface” on page 419

The HumanTaskManagerService interface exposes task-related functions that can be called by a local or a remote client.

## Accessing the EJB APIs

The Enterprise JavaBeans (EJB) APIs are provided as two stateless session enterprise beans. Business process applications and task applications access the appropriate session enterprise bean through the home interface of the bean.

### About this task

The BusinessFlowManagerService interface provides the methods for business process applications, and the HumanTaskManagerService interface provides the methods for task-based applications. The application can be any Java application, including another Enterprise JavaBeans (EJB) application.

### Accessing the remote interface of the session bean

An EJB client application for business processes or human tasks accesses the remote interface of the session bean through the remote home interface of the bean.

### About this task

The session bean can be either the BusinessFlowManager session bean for process applications or the HumanTaskManager session bean for task applications.

## Procedure

1. Add a reference to the remote interface of the session bean to the application deployment descriptor. Add the reference to one of the following files:
  - The `application-client.xml` file, for a Java Platform, Enterprise Edition (Java EE) client application
  - The `web.xml` file, for a Web application
  - The `ejb-jar.xml` file, for an Enterprise JavaBeans (EJB) application

The reference to the remote home interface for process applications is shown in the following example:

```
<ejb-ref>
 <ejb-ref-name>ejb/BusinessFlowManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
 <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
```

The reference to the remote home interface for task applications is shown in the following example:

```
<ejb-ref>
 <ejb-ref-name>ejb/HumanTaskManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.task.api.HumanTaskManagerHome</home>
 <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
```

If you use WebSphere Integration Developer to add the EJB reference to the deployment descriptor, the binding for the EJB reference is automatically created when the application is deployed. For more information on adding EJB references, refer to the WebSphere Integration Developer documentation.

2. Decide on how you are going to provide definitions of business objects.

To work with business objects in a remote client application, you need to have access to the corresponding schemas for the business objects (XSD or WSDL files) that are used to interact with a process or task. Access to these files can be provided in one of the following ways:

- If the client application does not run in a Java EE managed environment, package the files with the client application's EAR file.
- If the client application is a Web application or an EJB client in a managed Java EE environment, either package the files with the client application's EAR file or leverage remote artifact loading.
  - a. Use the Business Process Choreographer EJB API `createMessage` and the `ClientObjectWrapper.getObject` methods to load the remote business object definitions from the corresponding application on the server transparently.
  - b. Use the Service Data Object Programming API to create or read a business object as part of an already instantiated business object. Do this by using the `commonj.sdo.DataObject.createDataObject` or `getDataObject` methods on the `DataObject` interface.
  - c. When you want to create a business object as the value for a business object's property that is typed using the XML schema `any` or `anyType`, use the Business Object services to create or read your business object. To do this, you must set the remote artifact loader context to point to the application that the schemas will be loaded from. Then you can use the appropriate Business Object services.

For example, create a business object, where "ApplicationName" is the name of the application that contains your business object definitions.

```

BOFactory bofactory = (BOFactory) new
 ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");

com.ibm.wsspi.al.ALContext.setContext
 ("RALTemplateName", "ApplicationName");
try {
 DataObject dataObject = bofactory.create("uriName", "typeName");
} finally {
 com.ibm.wsspi.al.ALContext.unset();
}

```

For example, read XML input, where "ApplicationName" is the name of the application that contains your business object definitions.

```

BOXMLSerializer serializerService =
 (BOXMLSerializer) new ServiceManager().locateService
 ("com/ibm/websphere/bo/BOXMLSerializer");
ByteArrayInputStream input = new ByteArrayInputStream("<?xml?>..");

com.ibm.wsspi.al.ALContext.setContext
 ("RALTemplateName", "ApplicationName");
try {
 BOXMLDocument document = serializerService.readXMLDocument(input);
 DataObject dataObject = document.getDataObject();
} finally {
 com.ibm.wsspi.al.ALContext.unset();
}

```

3. Locate the remote home interface of the session bean through the Java Naming and Directory Interface (JNDI).

The following example shows this step for a process application:

```

// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

// Lookup the remote home interface of the BusinessFlowManager bean
Object result =
 initialContext.lookup("java:comp/env/ejb/BusinessFlowManagerHome");

// Convert the lookup result to the proper type
BusinessFlowManagerHome processHome =
 (BusinessFlowManagerHome) javax.rmi.PortableRemoteObject.narrow
 (result, BusinessFlowManagerHome.class);

```

The remote home interface of the session bean contains a create method for EJB objects. The method returns the remote interface of the session bean.

4. Access the remote interface of the session bean.

The following example shows this step for a process application:

```
BusinessFlowManager process = processHome.create();
```

Access to the session bean does not guarantee that the caller can perform all of the actions provided by the bean; the caller must also be authorized for these actions. When an instance of the session bean is created, a context is associated with the instance of the session bean. The context contains the caller's principal ID, group membership list, and indicates whether the caller has one of the Business Process Choreographer Java EE roles. The context is used to check the caller's authorization for each call, even when administrative security is not set. If administrative security is not set, the caller's principal ID has the value UNAUTHENTICATED.

5. Call the business functions exposed by the service interface.

The following example shows this step for a process application:

```
process.initiate("MyProcessModel", input);
```

Calls from applications are run as transactions. A transaction is established and ended in one of the following ways:

- Automatically by WebSphere Application Server (the deployment descriptor specifies TX\_REQUIRED).
- Explicitly by the application. You can bundle application calls into one transaction:

```
// Obtain user transaction interface
UserTransaction transaction=
 (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

// On successful return, commit the transaction
transaction.commit();
```

**Tip:** To prevent database lock conflicts, avoid running statements similar to the following in parallel:

```
// Obtain user transaction interface
UserTransaction transaction=
 (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

The `getActivityInstance` method and other read operations set a read lock. In this example, a read lock on the activity instance is upgraded to an update lock on the activity instance. This can result in a database deadlock when these transactions are run in parallel.

## Example

Here is an example of how steps 3 through 5 might look for a task application.

```
//Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

//Lookup the remote home interface of the HumanTaskManager bean
Object result =
 initialContext.lookup("java:comp/env/ejb/HumanTaskManagerHome");

//Convert the lookup result to the proper type
HumanTaskManagerHome taskHome =
 (HumanTaskManagerHome)javax.rmi.PortableRemoteObject.narrow
 (result,HumanTaskManagerHome.class);

...
//Access the remote interface of the session bean.
HumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```



## Accessing the local interface of the session bean

An EJB client application for business processes or human tasks accesses the local interface of the session bean through the local home interface of the bean.

### About this task

The session bean can be either the `BusinessFlowManager` session bean for process applications or the `HumanTaskManager` session bean for human task applications.

### Procedure

1. Add a reference to the local interface of the session bean to the application deployment descriptor. Add the reference to one of the following files:
  - The `application-client.xml` file, for a Java Platform, Enterprise Edition (Java EE) client application
  - The `web.xml` file, for a Web application
  - The `ejb-jar.xml` file, for an Enterprise JavaBeans (EJB) application

The reference to the local home interface for process applications is shown in the following example:

```
<ejb-local-ref>
 <ejb-ref-name>ejb/LocalBusinessFlowManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
 <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
</ejb-local-ref>
```

The reference to the local home interface for task applications is shown in the following example:

```
<ejb-local-ref>
 <ejb-ref-name>ejb/LocalHumanTaskManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
 <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>
```

If you use WebSphere Integration Developer to add the EJB reference to the deployment descriptor, the binding for the EJB reference is automatically created when the application is deployed. For more information on adding EJB references, refer to the WebSphere Integration Developer documentation.

2. Locate the local home interface of the session bean through the Java Naming and Directory Interface (JNDI).

The following example shows this step for a process application:

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

// Lookup the local home interface of the BusinessFlowManager bean

LocalBusinessFlowManagerHome processHome =
 (LocalBusinessFlowManagerHome) initialContext.lookup
 ("java:comp/env/ejb/LocalBusinessFlowManagerHome");
```

The local home interface of the session bean contains a `create` method for EJB objects. The method returns the local interface of the session bean.

3. Access the local interface of the session bean.

The following example shows this step for a process application:

```
LocalBusinessFlowManager process = processHome.create();
```

Access to the session bean does not guarantee that the caller can perform all of the actions provided by the bean; the caller must also be authorized for these actions. When an instance of the session bean is created, a context is associated

with the instance of the session bean. The context contains the caller's principal ID, group membership list, and indicates whether the caller has one of the Business Process Choreographer Java EE roles. The context is used to check the caller's authorization for each call, even when administrative security is not set. If administrative security is not set, the caller's principal ID has the value UNAUTHENTICATED.

4. Call the business functions exposed by the service interface.

The following example shows this step for a process application:

```
process.initiate("MyProcessModel",input);
```

Calls from applications are run as transactions. A transaction is established and ended in one of the following ways:

- Automatically by WebSphere Application Server (the deployment descriptor specifies TX\_REQUIRED).
- Explicitly by the application. You can bundle application calls into one transaction:

```
// Obtain user transaction interface
UserTransaction transaction=
 (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

// On successful return, commit the transaction
transaction.commit();
```

**Tip:** To prevent database deadlocks, avoid running statements similar to the following in parallel:

```
// Obtain user transaction interface
UserTransaction transaction=
 (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

The `getActivityInstance` method and other read operations set a read lock. In this example, a read lock on the activity instance is upgraded to an update lock on the activity instance. This can result in a database deadlock when these transactions are run in parallel

## Example

Here is an example of how steps 2 through 4 might look for a task application.

```
//Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

//Lookup the local home interface of the HumanTaskManager bean
LocalHumanTaskManagerHome taskHome =
 (LocalHumanTaskManagerHome)initialContext.lookup
 ("java:comp/env/ejb/LocalHumanTaskManagerHome");

...
//Access the local interface of the session bean
```

```

LocalHumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkid,input);

```

## Developing applications for business processes

A business process is a set of business-related activities that are invoked in a specific sequence to achieve a business goal. Examples are provided that show how you might develop applications for typical actions on processes.

### About this task

A business process can be either a microflow or a long-running process:

- Microflows are short running business processes that are executed synchronously. After a very short time, the result is returned to the caller.
- Long-running, interruptible processes are executed as a sequence of activities that are chained together. The use of certain constructs in a process causes interruptions in the process flow, for example, invoking a human task, invoking a service using an synchronous binding, or using timer-driven activities.

Parallel branches of the process are usually navigated asynchronously, that is, activities in parallel branches are executed concurrently. Depending on the type and the transaction setting of the activity, an activity can be run in its own transaction.

### Required roles for actions on process instances

Access to the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on a process. The caller must be logged on to the client application with a role that is authorized to perform the action.

The following table shows the actions on a process instance that a specific role can take.

Action	Caller's principal role		
	Reader	Starter	Administrator
createMessage	x	x	x
createWorkItem			x
delete			x
deleteWorkItem			x
forceTerminate			x
getActiveEventHandlers	x		x
getActivityInstance	x		x
getAllActivities	x		x
getAllWorkItems	x		x
getClientUISettings	x	x	x
getCustomProperties	x	x	x
getCustomProperty	x	x	x
getCustomPropertyNames	x	x	x
getFaultMessage	x	x	x
getInputClientUISettings	x	x	x

Action	Caller's principal role		
	Reader	Starter	Administrator
getInputMessage	x	x	x
getOutputClientUISettings	x	x	x
getOutputMessage	x	x	x
getProcessInstance	x	x	x
getVariable	x	x	x
getWaitingActivities	x	x	x
getWorkItems	x		x
restart			x
resume			x
setCustomProperty		x	x
setVariable			x
suspend			x
transferWorkItem			x

**Note:** If process administration is restricted to system administrators, then instance-based administration is disabled. This means that administrative actions on processes, scopes, and activities are limited to users in the BPESystemAdministrator role. In addition, reading, viewing, and monitoring a process instance or parts of it can only be performed by users in the BPESystemAdministrator or BPESystemMonitor roles. For more information about this administration mode, see [doc/bpc/cnot\\_instance\\_based\\_admin.dita](#).

### Required roles for actions on business-process activities

Access to the BusinessFlowManager interface does not guarantee that the caller can perform all of the actions on an activity. The caller must be logged on to the client application with a role that is authorized to perform the action.

The following table shows the actions on an activity instance that a specific role can take.

Action	Caller's principal role				
	Reader	Editor	Potential owner	Owner	Administrator
cancelClaim				x	x
claim			x		x
complete				x	x
createMessage	x	x	x	x	x
createWorkItem					x
deleteWorkItem					x
forceComplete					x
forceRetry					x
getActivityInstance	x	x	x	x	x
getAllWorkItems	x	x	x	x	x
getClientUISettings	x	x	x	x	x
getCustomProperties	x	x	x	x	x

Action	Caller's principal role				
	Reader	Editor	Potential owner	Owner	Administrator
getCustomProperty	x	x	x	x	x
getCustomPropertyNames	x	x	x	x	x
getFaultMessage	x	x	x	x	x
getFaultNames	x	x	x	x	x
getInputMessage	x	x	x	x	x
getOutputMessage	x	x	x	x	x
getVariable	x	x	x	x	x
getVariableNames	x	x	x	x	x
getInputVariableNames	x	x	x	x	x
getOutputVariableNames	x	x	x	x	x
getWorkItems	x	x	x	x	x
setCustomProperty		x		x	x
setFaultMessage		x		x	x
setOutputMessage		x		x	x
setVariable					x
transferWorkItem				x To potential owners or administrators only	x

**Note:** If process administration is restricted to system administrators, then instance-based administration is disabled. This means that administrative actions on processes, scopes, and activities are limited to users in the BPESystemAdministrator role. In addition, reading, viewing, and monitoring a process instance or parts of it can only be performed by users in the BPESystemAdministrator or BPESystemMonitor roles. For more information about this administration mode, see [doc/bpc/cnot\\_instance\\_based\\_admin.dita](#).

## Managing the life cycle of a business process

A process instance comes into existence when a Business Process Choreographer API method that can start a process is invoked. The navigation of the process instance continues until all of its activities are in an end state. Various actions can be taken on the process instance to manage its life cycle.

### About this task

Examples are provided that show how you might develop applications for the following typical life-cycle actions on processes.

#### Starting business processes:

The way in which a business process is started depends on whether the process is a microflow or a long-running process. The service that starts the process is also important to the way in which a process is started; the process can have either a unique starting service or several starting services.

## About this task

Examples are provided that show how you might develop applications for typical scenarios for starting microflows and long-running processes.

*Running a microflow that contains a unique starting service:*

A microflow can be started by a receive activity or a pick activity. The starting service is unique if the microflow starts with a receive activity or when the pick activity has only one onMessage definition.

## About this task

If the microflow implements a request-response operation, that is, the process contains a reply, you can use the call method to run the process passing the process template name as a parameter in the call.

If the microflow is a one-way operation, use the sendMessage method to run the process. This method is not covered in this example.

## Procedure

1. Optional: List the process templates to find the name of the process you want to run.

This step is optional if you already know the name of the process.

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
PROCESS_TEMPLATE.EXECUTION_MODE.EXECUTION_MODE_MICROFLOW",
"PROCESS_TEMPLATE.NAME",
new Integer(50),
(TimeZone)null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started by the call method.

2. Start the process with an input message of the appropriate type.

When you create the message, you must specify its message type name so that the message definition is contained.

```
ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
(template.getID(),
template.getInputMessageType());
DataObject myMessage = null;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
myMessage = (DataObject)input.getObject();
//set the strings in the message, for example, a customer name
myMessage.setString("CustomerName", "Smith");
}

//run the process
ClientObjectWrapper output = process.call(template.getName(), input);
DataObject myOutput = null;
if (output.getObject() != null && output.getObject() instanceof DataObject)
{
myOutput = (DataObject)output.getObject();
int order = myOutput.getInt("OrderNo");
}
```

This action creates an instance of the process template, CustomerTemplate, and passes some customer data. The operation returns only when the process is complete. The result of the process, OrderNo, is returned to the caller.

*Running a microflow that contains a non-unique starting service:*

A microflow can be started by a receive activity or a pick activity. The starting service is not unique if the microflow starts with a pick activity that has multiple onMessage definitions.

### About this task

If the microflow implements a request-response operation, that is, the process contains a reply, you can use the call method to run the process passing the ID of the starting service in the call.

If the microflow is a one-way operation, use the sendMessage method to run the process. This method is not covered in this example.

### Procedure

1. Optional: List the process templates to find the name of the process you want to run.

This step is optional if you already know the name of the process.

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
 PROCESS_TEMPLATE.EXECUTION_MODE.EXECUTION_MODE_MICROFLOW",
"PROCESS_TEMPLATE.NAME",
 new Integer(50),
 (TimeZone)null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started as microflows.

2. Determine the starting service to be called.

This example uses the first template that is found.

```
ProcessTemplateData template = processTemplates[0];
ActivityServiceTemplateData[] startActivities =
 process.getStartActivities(template.getID());
```

3. Start the process with an input message of the appropriate type.

When you create the message, you must specify its message type name so that the message definition is contained.

```
ActivityServiceTemplateData activity = startActivities[0];
//create a message for the service to be called
ClientObjectWrapper input =
 process.createMessage(activity.getServiceTemplateID(),
 activity.getActivityTemplateID(),
 activity.getInputMessageType());

DataObject myMessage = null;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)input.getObject();
 //set the strings in the message, for example, a customer name
 myMessage.setString("CustomerName", "Smith");
}
//run the process
ClientObjectWrapper output = process.call(activity.getServiceTemplateID(),
 activity.getActivityTemplateID(),
 input);
//check the output of the process, for example, an order number
DataObject myOutput = null;
```

```

if (output.getObject() != null && output.getObject() instanceof DataObject)
{
 myOutput = (DataObject)output.getObject();
 int order = myOutput.getInt("OrderNo");
}

```

This action creates an instance of the process template, `CustomerTemplate`, and passes some customer data. The operation returns only when the process is complete. The result of the process, `OrderNo`, is returned to the caller.

*Starting a long-running process that contains a unique starting service:*

If the starting service is unique, you can use the `initiate` method and pass the process template name as a parameter. This is the case when the long-running process starts with either a single receive or pick activity and when the single pick activity has only one `onMessage` definition.

### Procedure

1. Optional: List the process templates to find the name of the process you want to start.

This step is optional if you already know the name of the process.

```

ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
 PROCESS_TEMPLATE.EXECUTION_MODE.EXECUTION_MODE_LONG_RUNNING",
 "PROCESS_TEMPLATE.NAME",
 new Integer(50),
 (TimeZone)null);

```

The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started by the `initiate` method.

2. Start the process with an input message of the appropriate type.

When you create the message, you must specify its message type name so that the message definition is contained. If you specify a process-instance name, it must not start with an underscore. If a process-instance name is not specified, the process instance ID (PIID) in String format is used as the name.

```

ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
 (template.getID(),
 template.getInputMessageType());
DataObject myMessage = null;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)input.getObject();
 //set the strings in the message, for example, a customer name
 myMessage.setString("CustomerName", "Smith");
}
//start the process
PIID piid = process.initiate(template.getName(), "CustomerOrder", input);

```

This action creates an instance, `CustomerOrder`, and passes some customer data. When the process starts, the operation returns the object ID of the new process instance to the caller.

The starter of the process instance is set to the caller of the request. This person receives a work item for the process instance. The process administrators, readers, and editors of the process instance are determined and receive work items for the process instance. The follow-on activity instances are determined. These are started automatically or, if they are human task, receive, or pick activities, work items are created for the potential owners.

*Starting a long-running process that contains a non-unique starting service:*



A long-running process can be started through multiple initiating receive or pick activities. You can use the initiate method to start the process. If the starting service is not unique, for example, the process starts with multiple receive or pick activities, or a pick activity that has multiple onMessage definitions, then you must identify the service to be called.

### Procedure

1. Optional: List the process templates to find the name of the process you want to start.

This step is optional if you already know the name of the process.

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
PROCESS_TEMPLATE.EXECUTION_MODE.EXECUTION_MODE_LONG_RUNNING",
"PROCESS_TEMPLATE.NAME",
new Integer(50),
(TimeZone)null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted templates that can be started as long-running processes.

2. Determine the starting service to be called.

```
ProcessTemplateData template = processTemplates[0];
ActivityServiceTemplateData[] startActivities =
process.getStartActivities(template.getID());
```

3. Start the process with an input message of the appropriate type.

When you create the message, you must specify its message type name so that the message definition is contained. If you specify a process-instance name, it must not start with an underscore. If a process-instance name is not specified, the process instance ID (PIID) in String format is used as the name.

```
ActivityServiceTemplateData activity = startActivities[0];
//create a message for the service to be called
ClientObjectWrapper input = process.createMessage
(activity.getServiceTemplateID(),
activity.getActivityTemplateID(),
activity.getInputMessageType());

DataObject myMessage = null;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
myMessage = (DataObject)input.getObject();
//set the strings in the message, for example, a customer name
myMessage.setString("CustomerName", "Smith");
}
//start the process
PIID piid = process.sendMessage(activity.getServiceTemplateID(),
activity.getActivityTemplateID(),
input);
```

This action creates an instance and passes some customer data. When the process starts, the operation returns the object ID of the new process instance to the caller.

The starter of the process instance is set to the caller of the request and receives a work item for the process instance. The process administrators, readers, and editors of the process instance are determined and receive work items for the process instance. The follow-on activity instances are determined. These are started automatically or, if they are human task, receive, or pick activities, work items are created for the potential owners.

### Suspending and resuming a business process:

You can suspend long-running, top-level process instance while it is running and resume it again to complete it.

## Before you begin

### About this task

You might want to suspend a process instance, for example, so that you can configure access to a back-end system that is used later in the process. When the prerequisites for the process are met, you can resume the process instance. You might also want to suspend a process to fix a problem that is causing the process instance to fail, and then resume it again when the problem is fixed.

To suspend a process instance, it must be in the running or failing state. The caller must be a process administrator or a system administrator. However, if Business Flow Manager is using the alternate process administration authorization mode, which restricts process administration to system administrators, then only callers in the BPESystemAdministrator role can perform this action.

### Procedure

1. Get the running process, CustomerOrder, that you want to suspend.

```
ProcessInstanceData processInstance =
 process.getProcessInstance("CustomerOrder");
```

2. Suspend the process instance.

```
PIID piid = processInstance.getID();
process.suspend(piid);
```

This action suspends the specified top-level process instance. The process instance is put into the suspended state. In this state, activities that are started can still be finished but no new activities are activated. Subprocesses with the autonomy attribute set to child are also suspended if they are in the running, failing, terminating, or compensating state. Inline tasks and stand-alone tasks that are associated with this process instance are not suspended.

3. Resume the process instance.

```
process.resume(piid);
```

This action puts the process instance and its subprocesses into the states they had before they were suspended.

### Restarting a business process:

You can restart a process instance that is in the finished, terminated, failed, or compensated state.

### About this task

Restarting a process instance is similar to starting a process instance for the first time. However, when a process instance is restarted, the process instance ID is known and the input message for the instance is available.

If the process has more than one receive activity or pick activity (also known as a receive choice activity) that can create the process instance, all of the messages that belong to these activities are used to restart the process instance. If any of these activities implement a request-response operation, the response is sent again when the associated reply activity is navigated.

The caller must be a process administrator or a system administrator. However, if Business Flow Manager is using the alternate process administration authorization mode, which restricts process administration to system administrators, then only callers in the BPESystemAdministrator role can perform this action.

### Procedure

1. Get the process that you want to restart.

```
ProcessInstanceData processInstance =
 process.getProcessInstance("CustomerOrder");
```

2. Restart the process instance.

```
PIID piid = processInstance.getID();
process.restart(piid);
```

This action restarts the specified process instance.

### Terminating a process instance:

Sometimes, a top-level process instance that in an unrecoverable state must be terminated.

### About this task

To perform this action, the caller must be a process administrator or a system administrator. However, if Business Flow Manager is using the alternate process administration authorization mode, which restricts process administration to system administrators, then only callers in the BPESystemAdministrator role can perform this action.

Because a process instance terminates immediately, without waiting for any outstanding subprocesses or activities, you should only take this action in exceptional situations.

### Procedure

1. Retrieve the process instance that is to be terminated.

```
ProcessInstanceData processInstance =
 process.getProcessInstance("CustomerOrder");
```

2. Terminate the process instance.

If you terminate a process instance, you can terminate the process instance with or without compensation.

To terminate the process instance with compensation:

```
PIID piid = processInstance.getID();
process.forceTerminate(piid, CompensationBehaviour.INVOKE_COMPENSATION);
```

To terminate the process instance without compensation:

```
PIID piid = processInstance.getID();
process.forceTerminate(piid);
```

If you terminate the process instance with compensation, the compensation of the process is run as if a fault had occurred on the top-level scope. If you terminate the process instance without compensation, the process instance is terminated immediately without waiting for activities, to-do tasks, or inline invocation tasks to end normally.

Applications that are started by the process and standalone tasks that are related to the process are not terminated by the force terminate request. If these applications are to be terminated, you must add statements to your process application that explicitly terminate the applications started by the process.

## Deleting process instances:

Completed process instances are automatically deleted from the Business Process Choreographer database if the corresponding property is set for the process template in the process model. You might want to keep process instances in your database, for example, to query data from process instances that are not written to the audit log. However, stored process instance data does not only impact disk space and performance but also prevents process instances that use the same correlation set values from being created. Therefore, you should regularly delete process instance data from the database.

### About this task

To delete a process instance, you need process administrator rights and the process instance must be a top-level process instance.

The following example shows how to delete all of the finished process instances.

### Procedure

1. List the process instances that are finished.

```
QueryResultSet result =
 process.query("DISTINCT PROCESS_INSTANCE.PIID",
 "PROCESS_INSTANCE.STATE =
 PROCESS_INSTANCE.STATE.STATE_FINISHED",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns a query result set that lists process instances that are finished.

2. Delete the process instances that are finished.

```
while (result.next())
{
 PIID piid = (PIID) result.getOID(1);
 process.delete(piid);
}
```

This action deletes the selected process instance and its inline tasks from the database.

## Processing human task activities

Human task activities in business processes are assigned to various people in your organization through work items. When a process is started, work items are created for the potential owners.

### About this task

When a human task activity is activated, both an activity instance and an associated to-do task are created. Handling of the human task activity and the work item management is delegated to Human Task Manager. Any state change of the activity instance is reflected in the task instance and vice versa.

A potential owner claims the activity. This person is responsible for providing the relevant information and completing the activity.

### Procedure

1. List the activities belonging to a logged-on person that are ready to be worked on:

```

QueryResultSet result =
 process.query("ACTIVITY.AIID",
 "ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
 ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
 WORK_ITEM.REASON =
 WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
 (String)null, (Integer)null, (TimeZone)null);

```

This action returns a query result set that contains the activities that can be worked on by the logged-on person.

2. Claim the activity to be worked on:

```

if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);
 ClientObjectWrapper input = process.claim(aaid);
 DataObject activityInput = null ;
 if (input.getObject() != null && input.getObject() instanceof DataObject)
 {
 activityInput = (DataObject)input.getObject();
 // read the values
 ...
 }
}

```

When the activity is claimed, the input message of the activity is returned.

3. When work on the activity is finished, complete the activity. The activity can be completed either successfully or with a fault message. If the activity is successful, an output message is passed. If the activity is unsuccessful, the activity is put into the failed or stopped state and a fault message is passed. You must create the appropriate messages for these actions. When you create the message, you must specify the message type name so that the message definition is contained.

a. To complete the activity successfully, create an output message.

```

ActivityInstanceData activity = process.getActivityInstance(aaid);
ClientObjectWrapper output =
 process.createMessage(aaid, activity.getOutputMessageType());
DataObject myMessage = null ;
if (output.getObject() != null && output.getObject() instanceof DataObject)
{
 myMessage = (DataObject)output.getObject();
 //set the parts in your message, for example, an order number
 myMessage.setInt("OrderNo", 4711);
}

//complete the activity
process.complete(aaid, output);

```

This action sets an output message that contains the order number.

b. To complete the activity when a fault occurs, create a fault message.

```

//retrieve the faults modeled for the human task activity
List faultNames = process.getFaultNames(aaid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
 process.createMessage(aaid, faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if (myFault.getObject() != null && input.getObject() instanceof DataObject)
{

```

```

myMessage = (DataObject)myFault.getObject();
//set the parts in the message, for example, a customer name
myMessage.setInt("error",1304);
}

process.complete(aiid, myFault,(String) faultNames.get(0));

```

This action sets the activity in either the failed or the stopped state. If the **continueOnError** parameter for the activity in the process model is set to true, the activity is put into the failed state and the navigation continues. If the **continueOnError** parameter is set to false and the fault is not caught on the surrounding scope, the activity is put into the stopped state. In this state the activity can be repaired using force complete or force retry.

### Related concepts



Continue-on-error behavior of activities and business processes

When you define a business process, you can specify what should happen if an unexpected fault is raised and a fault handler is not defined for that fault. You can use the **Continue On Error** setting when you define your process to specify that it is to stop where the fault occurs.

### Processing a single person workflow

Some workflows are performed by only one person, for example, ordering books from an online bookstore. This type of workflow has no parallel paths. The `initiateAndClaimFirst` and `completeAndClaimSuccessor` APIs support the processing of this type of workflow. This example shows the implementation of a single person workflow using a client-side page flow.

### About this task

A single person workflow is also referred to as a *page flow* or a *screen flow*. There are two kinds of page flows:

- Client-side page flows, where the navigation between the different pages is realized using client-side technology, such as a multi-page Lotus® Forms form.
- Server-side page flows are realized using a business process and a set of human tasks that are modeled so that subsequent tasks are assigned to the same person.

Server-side page flows are more powerful than client-side page flows, but they consume more server resources to process them. Therefore, consider using this type of workflow primarily in the following situations:

- You need to invoke services between steps carried out in a user interface, for example, to retrieve or update data.
- You have auditing requirements that require CEI events to be written after a user interface interaction completes.

A typical example of a single person workflow is the ordering process in an online bookstore, where the purchaser completes a sequence of actions to order a book. This sequence of actions can be implemented as a series of human task activities (to-do tasks). If the purchaser decides to order several books, this is equivalent to starting an order process, and claiming the next human task activity.

The `initiateAndClaimFirst` API starts the page flow, that is, it starts the specified process and claims the first human task activity in the sequence of activities. It returns information about the claimed activity, including the input message to be worked on.

The `completeAndClaimSuccessor` API then completes the human task activity and claims the next one in the same process instance for the logged-on person. It returns information about the next claimed activity, including the input message to be worked on. Because the next activity is made available within the same transaction of the activity that completed, the transactional behavior of all the human task activities in the process model must be set to `participates`.

Compare this example with the example that uses both the Business Flow Manager API and the Human Task Manager API.

## Procedure

1. Start the book ordering process and claim the first activity in the sequence of activities. Start the process with an input message of the appropriate type. When you create the message, you must specify its message type name so that the message definition is contained. If you specify a process instance name, it must not start with an underscore. If a process instance name is not specified, the process instance ID (PIID) in String format is used as the name.

- a. Retrieve the process template to create an input message of the appropriate type.

```
ProcessTemplateData template = process.getProcessTemplate("CustomerOrder");
ClientObjectWrapper input = process.createMessage(template.getID(),
 template.getInputMessageType());
DataObject myMessage = null;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)input.getObject();
 //set the strings in the message, for example, a customer name
 myMessage.setString("CustomerName", "Smith");
}
```

- b. Start the process and claim the first human task activity.

```
InitiateAndClaimFirstResult result =
 process.initiateAndClaimFirst("CustomerOrder", "MyOrderProcess", input);
AIID aaid = result.getAIID();
ClientObjectWrapper input = result.getInputMessage();
```

When the first activity is claimed, the input message and the ID of the claimed activity is returned.

2. When work on the activity is finished, complete the activity, and claim the next activity.

To complete the activity, an output message is passed. When you create the output message, you must specify the message type name so that the message definition is contained.

```
ActivityInstanceData activity = process.getActivityInstance(aaid);
ClientObjectWrapper output =
 process.createMessage(aaid, activity.getOutputMessageType());
DataObject myMessage = null ;
if (output.getObject() != null && output.getObject() instanceof DataObject)
{
 myMessage = (DataObject)output.getObject();
 //set the parts in your message, for example, an order number
 myMessage.setInt("OrderNo", 4711);
}

//complete the activity and claim the next one
CompleteAndClaimSuccessorResult successor =
 process.completeAndClaimSuccessor(aaid, output);
```

This action sets an output message that contains the order number and claims the next activity in the sequence. If `AutoClaim` is set for successor activities and

if there are multiple paths that can be followed, all of the successor activities are claimed and a random activity is returned as the next activity. If there are no more successor activities that can be assigned to this user, `Null` is returned.

If the process contains parallel paths that can be followed and these paths contain human task activities for which the logged-on user is a potential owner of more than one of these activities, a random activity is claimed automatically and returned as the next activity.

3. Work on the next activity.

```
String name = successor.getActivityName();

ClientObjectWrapper nextInput = successor.getInputMessage();
if (nextInput.getObject() !=
 null && nextInput.getObject() instanceof DataObject)
{
 activityInput = (DataObject)input.getObject();
 // read the values
 ...
}

aiid = successor.getAIID();
```

4. Continue with step 2 to complete the activity.

### Related tasks

“Processing a single person workflow that includes human tasks” on page 424  
Some workflows are performed by only one person, for example, ordering books from an online bookstore. This example shows how to implement a single person workflow using a server-side page flow. Both the Business Flow Manager and the Human Task Manager APIs are used to process the workflow.

### Sending a message to a waiting activity

You can use inbound message activities (receive activities, `onMessage` in pick activities, `onEvent` in event handlers) to synchronize a running process with events from the “outside world”. For example, the receipt of an e-mail from a customer in response to a request for information might be such an event.

### About this task

You can use originating tasks to send the message to the activity.

### Procedure

1. List the activity service templates that are waiting for a message from the logged-on user in a process instance with a specific process instance ID.

```
ActivityServiceTemplateData[] services = process.getWaitingActivities(piid);
```

2. Send a message to the first waiting service.

It is assumed that the first service is the one that you want serve. The caller must be a potential starter of the activity that receives the message, or an administrator of the process instance.

```
VTID vtid = services[0].getServiceTemplateID();
ATID atid = services[0].getActivityTemplateID();
String inputType = services[0].getInputMessageType();

// create a message for the service to be called
ClientObjectWrapper message =
 process.createMessage(vtid,atid,inputMessageType);
DataObject myMessage = null;
if (message.getObject() != null && message.getObject() instanceof DataObject)
{
 myMessage = (DataObject)message.getObject();
}
```



```

 //set the strings in the message, for example, chocolate is to be ordered
 myMessage.setString("Order", "chocolate");
 }

 // send the message to the waiting activity
 process.sendMessage(vtid, atid, message);
}

```

This action sends the specified message to the waiting activity service and passes some order data.

You can also specify the process instance ID to ensure that the message is sent to the specified process instance. If the process instance ID is not specified, the message is sent to the activity service, and the process instance that is identified by the correlation values in the message. If the process instance ID is specified, the process instance that is found using the correlation values is checked to ensure that it has the specified process instance ID.

## Handling events

An entire business process and each of its scopes can be associated with event handlers that are invoked if the associated event occurs. Event handlers are similar to receive or pick activities in that a process can provide Web service operations using event handlers.

### About this task

You can invoke an event handler any number of times as long as the corresponding scope is running. In addition, multiple instances of an event handler can be activated concurrently.

The following code snippet shows how to get the active event handlers for a given process instance and how to send an input message.

### Procedure

1. Determine the data of the process instance ID and list the active event handlers for the process.

```

ProcessInstanceData processInstance =
 process.getProcessInstance("CustomerOrder2711");
EventHandlerTemplateData[] events = process.getActiveEventHandlers(
 processInstance.getID());

```

2. Send the input message.

This example uses the first event handler that is found.

```

EventHandlerTemplateData event = null;
if (events.length > 0)
{
 event = events[0];

 // create a message for the service to be called
 ClientObjectWrapper input = process.createMessage(
 event.getID(), event.getInputMessageType());

 if (input.getObject() != null && input.getObject() instanceof DataObject)
 {
 DataObject inputMessage = (DataObject)input.getObject();
 // set content of the message, for example, a customer name, order number
 inputMessage.setString("CustomerName", "Smith");
 inputMessage.setString("OrderNo", "2711");

 // send the message
 process.sendMessage(event.getProcessTemplateName(),
 event.getPortTypeNamespace(),

```

```

 event.getPortTypeName(),
 event.getOperationName(),

 input);
 }
}

```

This action sends the specified message to the active event handler for the process.

## Analyzing the results of a process

A process can expose Web services operations that are modeled as Web Services Description Language (WSDL) one-way or request-response operations. The results of long-running processes with one-way interfaces cannot be retrieved using the `getOutputMessage` method, because the process has no output. However, you can query the contents of variables, instead.

### About this task

The results of the process are stored in the database only if the process template from which the process instance was derived does not specify automatic deletion of the derived process instances.

### Procedure

Analyze the results of the process, for example, check the order number.

```

QueryResultSet result = process.query
 ("PROCESS_INSTANCE.PIID",
 "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
 PROCESS_INSTANCE.STATE =
 PROCESS_INSTANCE.STATE.STATE_FINISHED",
 (String)null, (Integer)null, (TimeZone)null);
if (result.size() > 0)
{
 result.first();
 PIID piid = (PIID) result.getOID(1);
 ClientObjectWrapper output = process.getOutputMessage(piid);
 DataObject myOutput = null;
 if (output.getObject() != null && output.getObject() instanceof DataObject)
 {
 myOutput = (DataObject)output.getObject();
 int order = myOutput.getInt("OrderNo");
 }
}

```

## Repairing activities

A long-running process can contain activities that are also long running. These activities might encounter uncaught errors and go into the stopped state. An activity in the running state might also appear to be not responding. In both of these cases, a process administrator can act on the activity in a number of ways so that the navigation of the process can continue.

### About this task

The Business Process Choreographer API provides the `forceRetry` and `forceComplete` methods for repairing activities. Examples are provided that show how you might add repair actions for activities to your applications.

#### Forcing the completion of an activity:

Activities in long-running processes can sometimes encounter faults. If these faults are not caught by a fault handler in the enclosing scope and the associated activity template specifies that the activity stops when an error occurs, the activity is put into the stopped state so that it can be repaired. In this state, you can force the completion of the activity.

### About this task

You can also force the completion of activities in the running state if, for example, an activity is not responding.

Additional requirements exist for certain types of activities.

### Human task activities

You can pass parameters in the force-complete call, such as the message that should have been sent or the fault that should have been raised.

### Script activities

You cannot pass parameters in the force-complete call. However, you must set the variables that need to be repaired.

### Invoke activities

You can also force the completion of invoke activities that call an asynchronous service that is not a subprocess if these activities are in the running state. You might want to do this, for example, if the asynchronous service is called and it does not respond.

### Procedure

1. List the stopped activities in the stopped state.

```
QueryResultSet result =
 process.query("DISTINCT ACTIVITY.AIID",
 "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
 PROCESS_INSTANCE.NAME='CustomerOrder'",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns the stopped activities for the CustomerOrder process instance.

2. Complete the activity, for example, a stopped human task activity.

In this example, an output message is passed.

```
if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);
 ActivityInstanceData activity = process.getActivityInstance(aaid);
 ClientObjectWrapper output =
 process.createMessage(aaid, activity.getOutputMessageType());
 DataObject myMessage = null;
 if (output.getObject() != null && output.getObject() instanceof DataObject)
 {
 myMessage = (DataObject)output.getObject();
 //set the parts in your message, for example, an order number
 myMessage.setInt("OrderNo", 4711);
 }

 boolean continueOnError = true;
 process.forceComplete(aaid, output, continueOnError);
}
```

This action completes the activity. If an error occurs, the **continueOnError** parameter determines the action to be taken if a fault is provided with the forceComplete request.

In the example, **continueOnError** is true. This value means that if a fault is provided, the activity is put into the failed state. The fault is propagated to the enclosing scopes of the activity until it is either handled or the process scope is reached. The process is then put into the failing state and it eventually reaches the failed state.

### Related concepts



Continue-on-error behavior of activities and business processes

When you define a business process, you can specify what should happen if an unexpected fault is raised and a fault handler is not defined for that fault. You can use the **Continue On Error** setting when you define your process to specify that it is to stop where the fault occurs.

### Retrying the execution of a stopped activity:

If an activity in a long-running process encounters an uncaught fault in the enclosing scope and if the associated activity template specifies that the activity stops when an error occurs, the activity is put into the stopped state so that it can be repaired. You can retry the execution of the activity.

### About this task

You can set variables that are used by the activity. With the exception of script activities, you can also pass parameters in the force-retry call, such as the message that was expected by the activity.

### Procedure

1. List the stopped activities.

```
QueryResultSet result =
 process.query("DISTINCT ACTIVITY.AIID",
 "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
 PROCESS_INSTANCE.NAME='CustomerOrder'",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns the stopped activities for the CustomerOrder process instance.

2. Retry the execution of the activity, for example, a stopped human task activity.

```
if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);
 ActivityInstanceData activity = process.getActivityInstance(aaid);
 ClientObjectWrapper input =
 process.createMessage(aaid, activity.getOutputMessageType());
 DataObject myMessage = null;
 if (input.getObject() != null && input.getObject() instanceof DataObject)
 {
 myMessage = (DataObject)input.getObject();
 //set the strings in your message, for example, chocolate is to be ordered
 myMessage.setString("OrderNo", "chocolate");
 }

 boolean continueOnError = true;
 process.forceRetry(aaid, input, continueOnError);
}
```

This action retries the activity. If an error occurs, the **continueOnError** parameter determines the action to be taken if an error occurs during processing of the forceRetry request.

In the example, **continueOnError** is true. This means that if an error occurs during processing of the forceRetry request, the activity is put into the failed state. The fault is propagated to the enclosing scopes of the activity until it is either handled or the process scope is reached. The process is then put into the failing state and a fault handler on the process level is run before the process state ends in the failed state.

## Related concepts



Continue-on-error behavior of activities and business processes

When you define a business process, you can specify what should happen if an unexpected fault is raised and a fault handler is not defined for that fault. You can use the **Continue On Error** setting when you define your process to specify that it is to stop where the fault occurs.

## Repairing activities that stopped because a join, loop, or counter evaluation failed:

Activities can stop because an exception occurred when a join or loop condition, or a forEach counter value was evaluated. The administrator decides not to retry the execution of the activity, for example, because the evaluation might fail again. In such cases, the correct values for the expression can be supplied using the Business Process Choreographer EJB API so that the navigation of the process can continue.

## About this task

You can set the value of a join condition for any type of activity, the value of a loop condition of a while or repeat-until activity. You can also set the values of start and final counters, and the maximum number of completed branches for a forEach activity. The value that you set for the completed branches depends on the definition of the forEach activity in the process model. If an early exit condition is specified in the model, set a value for the maximum completed branches. If an early exit condition is not specified, set the value of the maximum completed branches to null.

The following sample shows how to set the value of a loop condition.

## Procedure

1. List the activities that stopped because the evaluation of a loop condition failed.

```
QueryResultSet result = process.query(
 "DISTINCT ACTIVITY.AIID",
 "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
 ACTIVITY.STOP_REASON = ACTIVITY.STOP_REASON.STOP_REASON_IMPLEMENTATION_FAILED AND
 (ACTIVITY.KIND = ACTIVITY.KIND.KIND_WHILE OR
 ACTIVITY.KIND = ACTIVITY.KIND.KIND_REPEAT_UNTIL) AND
 PROCESS_INSTANCE.NAME='CustomerOrder'",
 (String)null, (Integer)null, (TimeZone)null);
```

Similarly, you can list the activities that stopped because the evaluation of a join condition or a forEach counter failed.

- For a failed join condition, use the following expression:

```
ACTIVITY.STOP_REASON.STOP_REASON_ACTIVATION_FAILED
```

- For a failed forEach counter, use the following expression:

```
ACTIVITY.STOP_REASON.STOP_REASON_IMPLEMENTATION_FAILED AND
(ACTIVITY.KIND = ACTIVITY.KIND.KIND_FOR_EACH_SERIAL OR
ACTIVITY.KIND = ACTIVITY.KIND.KIND_FOR_EACH_PARALLEL)
```

This action returns the activities for the CustomerOrder process instance that stopped because the evaluation of a loop condition failed.

2. Provide the value of the loop condition, for example, true.

```
if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);

 process.forceLoopCondition(aaid, true);
}
```

This action sets the value of the loop condition for the activity to true and the navigation of the process instance continues.

Similarly, you can set the value of a join condition (`process.forceJoinCondition(aaid, true);`) or the values of forEach activity counters (`process.forceForEachCounterValues(aaid, 1, 5, new Integer(2));`).

### Updating correlation sets associated with stopped activities:

Correlation sets are used to support stateful collaboration between Web services. In such cases, the correct values for the expression can be supplied using the Business Process Choreographer EJB API so that the navigation of the process can continue.

#### About this task

An activity that is in a stopped state can require an update of its associated correlation set for one of the following reasons:

- An exception occurred when the correlation set was evaluated. The correlation set is to be initialized but it is already initialized.
- An exception occurred when the correlation set was evaluated. The correlation set is to not be initialized but its values are not set. This can occur, for example, because an initializing activity was skipped.
- The activity needs to be retried. If the correlation set is initialized by the activity, then it can be uninitialized or changed before the `forceRetry` method is called.
- The activity needs to be completed. If the correlation set is initialized by the activity, then it can be uninitialized or changed before the `forceComplete` method is called.

You can retrieve the correlation set instances of a process or activity instance. The following example shows how to initialize or uninitialize correlation set instances.

#### Procedure

1. List the stopped activities in the stopped state.

```
QueryResultSet result =
 process.query("DISTINCT ACTIVITY.AIID",
 "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
 PROCESS_INSTANCE.NAME='CustomerOrder'",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns the stopped activities for the CustomerOrder process instance.

2. Retrieve the correlation set instances that are defined for the activity.

```
AIID aaid = null;

List correlationSet = null;

if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);

 ActivityInstanceData activity = process.getActivityInstance(aaid);
```

```

 correlationSet = process.getCorrelationSetInstances
 (aaid, activity.getInputMessageType());
 }

```

3. Uninitialize the correlation set, for example, MyCorrelationSet.

```

for (int i=0; i<correlationSet.size(); i++)
{
 CorrelationSetInstanceData correlationSetInstance =
 (CorrelationSetInstanceData)correlationSet.get(i);

 if (correlationSetInstance.isInitialized() &&
 correlationSetInstance.getCorrelationSetName().equals("MyCorrelationSet"))
 {
 process.uninitializeCorrelationSet
 (activity.getProcessInstanceID(), correlationSetInstance.getCorrelationSetName());
 }
}

```

This action uninitializes the correlation set MyCorrelationSet.

4. Initialize the correlation set, for example, MyCorrelationSet. In this example, a string-valued property of the correlation set is set.

```

for (int i=0; i<correlationSet.size(); i++)
{
 CorrelationSetInstanceData correlationSetInstance =
 (CorrelationSetInstanceData)correlationSet.get(i);

 if (correlationSetInstance.getCorrelationSetName().equals("MyCorrelationSet"))
 {
 List correlationSetProperties =
 correlationSetInstance.getCorrelationSetProperties();
 for (int j=0; j<correlationSetProperties.size(); j++)
 {
 CorrelationPropertyInstanceData property =
 (CorrelationPropertyInstanceData)correlationSetProperties.get(j);

 if (property.getPropertyName().equals("MyProperty"))
 {
 property.setValue("NewValue");

 process.initializeCorrelationSet
 (activity.getProcessInstanceID(), correlationSetInstance);
 }
 }
 }
}

```

This action initializes the string-valued property MyProperty in the correlation set MyCorrelationSet.

## BusinessFlowManagerService interface

The BusinessFlowManagerService interface exposes business-process functions that can be called by a client application.

The methods that can be called by the BusinessFlowManagerService interface depend on the state of the process or the activity and the authorization of the person that uses the application containing the method. The main methods for manipulating business process objects are listed here. For more information about these methods and the other methods that are available in the BusinessFlowManagerService interface, see the Javadoc in the com.ibm.bpe.api package.

## Process templates

A process template is a versioned, deployed, and installed process model that contains the specification of a business process. It can be instantiated and started

by issuing appropriate requests, for example, `sendMessage()`. The execution of the process instance is driven automatically by the server.

*Table 60. API methods for process templates*

Method	Description
<code>getProcessTemplate</code>	Retrieves the specified process template.
<code>queryProcessTemplates</code>	Retrieves process templates that are stored in the database.

## Process instances

The following API methods are related to starting process instances.

*Table 61. API methods that are related to starting process instances*

Method	Description
<code>call</code>	Creates and runs a microflow.
<code>callWithReplyContext</code>	Creates and runs a microflow with a unique starting service or a long-running process with a unique starting service from the specified process template. The call waits asynchronously for the result.
<code>callWithUISettings</code>	Creates and runs a microflow and returns the output message and the client user interface (UI) settings.
<code>initiate</code>	Creates a process instance and initiates processing of the process instance. Use this method for long-running processes. You can also use this method for microflows that you want to fire and forget.
<code>initiateAndSuspend</code>	Creates a process instance but immediately suspends the further processing of the process instance.
<code>initiateAndClaimFirst</code>	Creates a process instance and claims the first inline human task.
<code>sendMessage</code>	Sends the specified message to the specified activity service and process instance. If a process instance with the same correlation set values does not exist, it is created. The process can have either unique or non-unique starting services.
<code>getStartActivities</code>	Returns information about the activities that can start a process instance from the specified process template.
<code>getActivityServiceTemplate</code>	Retrieves the specified activity service template.

*Table 62. API methods for controlling the life cycle of process instances*

Method	Description
<code>suspend</code>	Suspends the execution of a long-running, top-level process instance that is in the running or failing state.



Table 62. API methods for controlling the life cycle of process instances (continued)

Method	Description
resume	Resumes the execution of a long-running, top-level process instance that is in the suspended state.
restart	Restarts a long-running, top-level process instance that is in the finished, failed, or terminated state.
forceTerminate	Terminates the specified top-level process instance, its subprocesses with child autonomy, and its running, claimed, or waiting activities.
delete	Deletes the specified top-level process instance and its subprocesses with child autonomy.
query	Retrieves the properties from the database that match the search criteria.
queryEntities	Uses query tables to retrieve the properties from the database that match the search criteria.
getWaitingActivities	Returns information about the activities that are waiting for a message so that the processing of these activities can continue.
migrate	Migrates a process instance to the specified newer version of its process model.

## Activities

For invoke activities, you can specify in the process model that these activities continue in error situations. If the `continueOnError` flag is set to `false` and an unhandled error occurs, the activity is put into the stopped state. A process administrator can then repair the activity. The `continueOnError` flag and the associated repair functions can, for example, be used in a long-running process where an invoke activity fails occasionally, but the effort required to model compensation and fault handling is too high.

The following methods are available for working with and repairing activities.

Table 63. API methods for controlling the life cycle of activity instances

Method	Description
claim	Claims a ready activity instance for a user to work on the activity.
cancelClaim	Cancels the claim of the activity instance.
complete	Completes the activity instance.
completeAndClaimSuccessor	Completes the activity instance and claims the next one in the same process instance for the logged-on person.

Table 63. API methods for controlling the life cycle of activity instances (continued)

Method	Description
forceComplete	Forces the completion of the following: <ul style="list-style-type: none"> <li>• An activity instance that is in the running or stopped state.</li> <li>• A human task activity that is in the state ready or claimed.</li> <li>• A wait activity in state waiting.</li> </ul>
forceRetry	Forces the repetition of the following: <ul style="list-style-type: none"> <li>• An activity instance that is in the running or stopped state.</li> <li>• A human task activity that is in the state ready or claimed.</li> </ul>
forceNavigate, forceForEach, forceLoop, forceJoin	These methods force the navigation of a stopped activity.
skip	Skips processing of the activity.
jump	Jumps from one activity to the other.
query	Retrieves the properties from the database that match the search criteria.
queryEntities	Uses query tables to retrieve the properties from the database that match the search criteria.

## Variables and custom properties

The interface provides a get and a set method to retrieve and set values for variables. You can also associate named properties with, and retrieve named properties from, process and activity instances. Custom property names and values must be of the `java.lang.String` type.

Table 64. API methods for variables and custom properties

Method	Description
getVariable	Retrieves the specified variable.
setVariable	Sets the specified variable.
getCustomProperty	Retrieves the named custom property of the specified activity or process instance.
getCustomProperties	Retrieves the custom properties of the specified activity or process instance.
getCustomPropertyNames	Retrieves the names of the custom properties for the specified activity or process instance.
setCustomProperty	Stores custom-specific values for the specified activity or process instance.

## Developing applications for human tasks

A task is the means by which components invoke humans as services or by which humans invoke services. Examples of typical applications for human tasks are provided.

## About this task

For more information on the Human Task Manager API, see the Javadoc in the `com.ibm.task.api` package.

### Starting an invocation task that invokes a synchronous interface

An invocation task is associated with a Service Component Architecture (SCA) component. When the task is started, it invokes the SCA component. Start an invocation task synchronously only if the associated SCA component can be called synchronously.

## About this task

Such an SCA component can, for example, be implemented as a microflow or as a simple Java class.

This scenario creates an instance of a task template and passes some customer data. The task remains in the running state until the two-way operation returns. The result of the task, `OrderNo`, is returned to the caller.

## Procedure

1. Optional: List the task templates to find the name of the invocation task you want to run.

This step is optional if you already know the name of the task.

```
TaskTemplate[] taskTemplates = task.queryTaskTemplates(
 "TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
 "TASK_TEMPL.NAME",
 new Integer(50),
 (TimeZone)null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted originating templates.

2. Create an input message of the appropriate type.

```
TaskTemplate template = taskTemplates[0];

// create a message for the selected task
ClientObjectWrapper input = task.createInputMessage(template.getID());
DataObject myMessage = null ;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)input.getObject();
 //set the parts in the message, for example, a customer name
 myMessage.setString("CustomerName", "Smith");
}
```

3. Create the task and run the task synchronously.

For a task to run synchronously, it must be a two-way operation. The example uses the `createAndCallTask` method to create and run the task.

```
ClientObjectWrapper output = task.createAndCallTask(template.getName(),
 template.getNamespace(),
 input);
```

4. Analyze the result of the task.

```
DataObject myOutput = null;
if (output.getObject() != null && output.getObject() instanceof DataObject)
{
 myOutput = (DataObject)output.getObject();
 int order = myOutput.getInt("OrderNo");
}
```

## Starting an invocation task that invokes an asynchronous interface

An invocation task is associated with a Service Component Architecture (SCA) component. When the task is started, it invokes the SCA component. Start an invocation task asynchronously only if the associated SCA component can be called asynchronously.

### About this task

Such an SCA component can, for example, be implemented as a long-running process or a one-way operation.

This scenario creates an instance of a task template and passes some customer data.

### Procedure

1. Optional: List the task templates to find the name of the invocation task you want to run.

This step is optional if you already know the name of the task.

```
TaskTemplate[] taskTemplates = task.queryTaskTemplates
("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
 "TASK_TEMPL.NAME",
 new Integer(50),
 (TimeZone)null);
```

The results are sorted by name. The query returns an array containing the first 50 sorted originating templates.

2. Create an input message of the appropriate type.

```
TaskTemplate template = taskTemplates[0];

// create a message for the selected task
ClientObjectWrapper input = task.createInputMessage(template.getID());
DataObject myMessage = null ;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)input.getObject();
 //set the parts in the message, for example, a customer name
 myMessage.setString("CustomerName", "Smith");
}
```

3. Create the task and run it asynchronously.

The example uses the `createAndStartTask` method to create and run the task.

```
task.createAndStartTask(template.getName(),
 template.getNamespace(),
 input,
 (ReplyHandlerWrapper)null);
```

## Creating and starting a task instance

This scenario shows how to create an instance of a task template that defines a collaboration task (also known as a *human task* in the API) and start the task instance.

### Procedure

1. Optional: List the task templates to find the task template ID (TKTID) of the collaboration task you want to run.

This step is optional if you already know the task template ID.

```

TaskTemplate[] taskTemplates = task.queryTaskTemplates
("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_HUMAN",
 "TASK_TEMPL.NAME",
 new Integer(50),
 (TimeZone)null);

```

The results are sorted by name. The query returns an array containing the first 50 sorted task templates.

2. Create an input message of the appropriate type.

```

TaskTemplate template = taskTemplates[0];

// create a message for the selected task
ClientObjectWrapper input = task.createInputMessage(template.getID());
DataObject myMessage = null ;
if (input.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)input.getObject();
 //set the parts in the message, for example, a customer name
 myMessage.setString("CustomerName", "Smith");
}

```

3. Create and start the collaboration task; a reply handler is not specified in this example.

The example uses the `createAndStartTask` method to create and start the task.

```

TKIID tkiid = task.createAndStartTask(template.getName(),
 template.getNamespace(),
 input,
 (ReplyHandlerWrapper)null);

```

Work items are created for the people concerned with the task instance. For example, a potential owner can claim the new task instance.

4. Claim the task instance.

```

ClientObjectWrapper input2 = task.claim(tkiid);
DataObject taskInput = null ;
if (input2.getObject() != null && input2.getObject() instanceof DataObject)
{
 taskInput = (DataObject)input2.getObject();
 // read the values
 ...
}

```

When the task instance is claimed, the input message of the task is returned.

## Processing to-do tasks or collaboration tasks

To-do tasks (also known as *participating tasks* in the API) or collaboration tasks (also known as *human tasks* in the API) are assigned to various people in your organization through work items. To-do tasks and their associated work items are created, for example, when a process navigates to a human task activity.

### About this task

One of the potential owners claims the task associated with the work item. This person is responsible for providing the relevant information and completing the task.

### Procedure

1. List the tasks belonging to a logged-on person that are ready to be worked on.

```

QueryResultSet result =
 task.query("TASK.TKIID",
 "TASK.STATE = TASK.STATE.STATE_READY AND
 (TASK.KIND = TASK.KIND.KIND_PARTICIPATING OR
 TASK.KIND = TASK.KIND.KIND_HUMAN)AND

```

```

WORK_ITEM.REASON =
 WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
 (String)null, (Integer)null, (TimeZone)null);

```

This action returns a query result set that contains the tasks that can be worked on by the logged-on person.

2. Claim the task to be worked on.

```

if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 ClientObjectWrapper input = task.claim(tkiid);
 DataObject taskInput = null ;
 if (input.getObject() != null && input.getObject() instanceof DataObject)
 {
 taskInput = (DataObject)input.getObject();
 // read the values
 ...
 }
}

```

When the task is claimed, the input message of the task is returned.

3. When work on the task is finished, complete the task.

The task can be completed either successfully or with a fault message. If the task is successful, an output message is passed. If the task is unsuccessful, a fault message is passed. You must create the appropriate messages for these actions.

- a. To complete the task successfully, create an output message.

```

ClientObjectWrapper output =
 task.createOutputMessage(tkiid);
DataObject myMessage = null ;
if (output.getObject() != null && output.getObject() instanceof DataObject)
{
 myMessage = (DataObject)output.getObject();
 //set the parts in your message, for example, an order number
 myMessage.setInt("OrderNo", 4711);
}

//complete the task
task.complete(tkiid, output);

```

This action sets an output message that contains the order number. The task is put into the finished state.

- b. To complete the task when a fault occurs, create a fault message.

```

//retrieve the faults modeled for the task
List faultNames = task.getFaultNames(tkiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
 task.createFaultMessage(tkiid, (String)faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if (myFault.getObject() != null && input.getObject() instanceof DataObject)
{
 myMessage = (DataObject)myFault.getObject();
 //set the parts in the message, for example, a customer name
 myMessage.setInt("error",1304);
}

task.complete(tkiid, (String)faultNames.get(0), myFault);

```

This action sets a fault message that contains the error code. The task is put into the failed state.

### Related concepts



State transition diagrams for collaboration tasks

Collaboration tasks support people when they perform work for other people. During the life cycle of a collaboration task, certain interactions are possible only in certain task states, and these interactions, in turn, influence the state of the task.

### Suspending and resuming a task instance

You can suspend collaboration task instances (also known as *human tasks* in the API) or to-do task instances (also known as *participating tasks* in the API).

### Before you begin

The task instance can be in the ready or claimed state. It can be escalated. The caller must be the owner, originator, or administrator of the task instance.

### About this task

You can suspend a task instance while it is running. You might want to do this, for example, so that you can gather information that is needed to complete the task. When the information is available, you can resume the task instance.

### Procedure

1. Get a list of tasks that are claimed by the logged-on user.

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
 "TASK.STATE = TASK.STATE.STATE_CLAIMED",
 (String)null,
 (Integer)null,
 (TimeZone)null);
```

This action returns a query result set that contains a list of the tasks that are claimed by the logged-on user.

2. Suspend the task instance.

```
if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 task.suspend(tkiid);
}
```

This action suspends the specified task instance. The task instance is put into the suspended state.

3. Resume the process instance.

```
task.resume(tkiid);
```

This action puts the task instance into the state it had before it was suspended.

### Analyzing the results of a task

A to-do task (also known as a *participating* task in the API) or a collaboration task (also known as a *human task* in the API) runs asynchronously. If a reply handler is specified when the task starts, the output message is automatically returned when the task completes. If a reply handler is not specified, the message must be retrieved explicitly.

## About this task

The results of the task are stored in the database only if the task template from which the task instance was derived does not specify automatic deletion of the derived task instances.

## Procedure

Analyze the results of the task.

The example shows how to check the order number of a successfully completed task.

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
 "TASK.NAME = 'CustomerOrder' AND
 TASK.STATE = TASK.STATE.STATE_FINISHED",
 (String)null, (Integer)null, (TimeZone)null);

if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 ClientObjectWrapper output = task.getOutputMessage(tkiid);
 DataObject myOutput = null;
 if (output.getObject() != null && output.getObject() instanceof DataObject)
 {
 myOutput = (DataObject)output.getObject();
 int order = myOutput.getInt("OrderNo");
 }
}
```

## Terminating a task instance

Sometimes it is necessary for someone with administrator rights to terminate a task instance that is known to be in an unrecoverable state. Because the task instance is terminated immediately, you should terminate a task instance only in exceptional situations.

## Procedure

1. Retrieve the task instance to be terminated.

```
Task taskInstance = task.getTask(tkiid);
```

2. Terminate the task instance.

```
TKIID tkiid = taskInstance.getID();
task.terminate(tkiid);
```

The task instance is terminated immediately without waiting for any outstanding tasks.

## Deleting task instances

Task instances are only automatically deleted when they complete if this is specified in the associated task template from which the instances are derived. This example shows how to delete all of the task instances that are finished and are not automatically deleted.

## Procedure

1. List the task instances that are finished.

```
QueryResultSet result =
 task.query("DISTINCT TASK.TKIID",
 "TASK.STATE = TASK.STATE.STATE_FINISHED",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns a query result set that lists task instances that are finished.



2. Delete the task instances that are finished.

```
while (result.next())
{
 TKIID tkiid = (TKIID) result.getOID(1);
 task.delete(tkiid);
}
```

## Releasing a claimed task

When a potential owner claims a task, this person is responsible for completing the task. However, sometimes the claimed task must be released so that another potential owner can claim it.

### About this task

Sometimes it is necessary for someone with administrator rights to release a claimed task. This situation might occur, for example, when a task must be completed but the owner of the task is absent. The owner of the task can also release a claimed task.

### Procedure

1. List the claimed tasks owned by a specific person, for example, Smith.

```
QueryResultSet result =
 task.query("DISTINCT TASK.TKIID",
 "TASK.STATE = TASK.STATE.STATE_CLAIMED AND
 TASK.OWNER = 'Smith'",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns a query result set that lists the tasks claimed by the specified person, Smith.

2. Release the claimed task.

```
if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 task.cancelClaim(tkiid, true);
}
```

This action returns the task to the ready state so that it can be claimed by one of the other potential owners. Any output or fault data that is set by the original owner is kept.

## Managing work items

During the lifetime of an activity instance or a task instance, the set of people associated with the object can change, for example, because a person is on vacation, new people are hired, or the workload needs to be distributed differently. To allow for these changes, you can develop applications to create, delete, or transfer work items.

### About this task

A work item represents the assignment of an object to a user or group of users for a particular reason. The object is typically a human task activity instance, a process instance, or a task instance. The reasons are derived from the role that the user has for the object. An object can have multiple work items because a user can have different roles in association with the object, and a work item is created for each of these roles. For example, a to-do task instance can have an administrator, reader, editor, and owner work item at the same time.

The actions that can be taken to manage work items depend on the role that the user has, for example, an administrator can create, delete and transfer work items, but the task owner can transfer work items only.

## Procedure

- Create a work item.

```
// query the task instance for which an additional
// administrator is to be specified
QueryResultSet result = task.query("TASK.TKIID",
 "TASK.NAME='CustomerOrder'",
 (String)null, (Integer)null,
 (TimeZone)null);

if (result.size() > 0)
{
 result.first();
 // create the work item
 task.createWorkItem((TKIID)(result.getOID(1)),
 WorkItem.REASON_ADMINISTRATOR,"Smith");
}
```

This action creates a work item for the user Smith who has the administrator role.

- Delete a work item.

```
// query the task instance for which a work item is to be deleted
QueryResultSet result = task.query("TASK.TKIID",
 "TASK.NAME='CustomerOrder'",
 (String)null, (Integer)null,
 (TimeZone)null);

if (result.size() > 0)
{
 result.first();
 // delete the work item
 task.deleteWorkItem((TKIID)(result.getOID(1)),
 WorkItem.REASON_READER,"Smith");
}
```

This action deletes the work item for the user Smith who has the reader role.

- Transfer a work item.

```
// query the task that is to be rescheduled
QueryResultSet result =
 task.query("DISTINCT TASK.TKIID",
 "TASK.NAME='CustomerOrder' AND
 TASK.STATE=TASK.STATE.STATE_READY AND
 WORK_ITEM.REASON=WORK_ITEM.REASON.POTENTIAL_OWNER AND
 WORK_ITEM.OWNER_ID='Miller'",
 (String)null, (Integer)null, (TimeZone)null);
if (result.size() > 0)
{
 result.first();
 // transfer the work item from user Miller to user Smith
 // so that Smith can work on the task
 task.transferWorkItem((TKIID)(result.getOID(1)),
 WorkItem.REASON_POTENTIAL_OWNER,"Miller","Smith");
}
```

This action transfers the work item to the user Smith so that he can work on it.

## Creating task templates and task instances at runtime

You usually use a modeling tool, such as WebSphere Integration Developer to build task templates. You then install the task templates in WebSphere Process Server and create instances from these templates, for example, using Business Process Choreographer Explorer. However, you can also create human or participating task instances or templates at runtime.

## About this task

You might want to do this, for example, when the task definition is not available when the application is deployed, the tasks that are part of a workflow are not yet known, or you need a task to cover some ad hoc collaboration between a group of people.

You can model ad hoc To-do or Collaboration tasks by creating instances of the `com.ibm.task.api.TaskModel` class, and using them to either create a reusable task template, or directly create a run-once task instance. To create an instance of the `TaskModel` class, a set of factory methods is available in the `com.ibm.task.api.ClientTaskFactory` factory class. Modeling human tasks at runtime is based on the Eclipse Modeling Framework (EMF).

## Procedure

1. Create an `org.eclipse.emf.ecore.resource.ResourceSet` using the `createResourceSet` factory method.
2. Optional: If you intend to use complex message types, you can either define them using the `org.eclipse.xsd.XSDFactory` that you can get using the factory method `getXSDFactory()`, or directly import an existing XML schema using the `loadXSDSchema` factory method .  
To make the complex types available to the WebSphere Process Server, deploy them as part of an enterprise application.
3. Create or import a Web Services Definition Language (WSDL) definition of the type `javax.wsdl.Definition`.  
You can create a new WSDL definition using the `createWSDLDefinition` method. Then you can add it a port type and operation. You can also directly import an existing WSDL definition using the `loadWSDLDefinition` factory method.
4. Create the task definition using the `createTTask` factory method.  
If you want to add or manipulate more complex task elements, you can use the `com.ibm.wbit.tel.TaskFactory` class that you can retrieve using the `getTaskFactory` factory method .
5. Create the task model using the `createTaskModel` factory method, and pass it the resource bundle that you created in the step 1 and which aggregates all other artifacts you created in the meantime.
6. Optional: Validate the model using the `TaskModel` `validate` method.

## Results

Use one of the Human Task Manager EJB API create methods that have a **TaskModel** parameter to either create a reusable task template, or a run-once task instance.

### Creating runtime tasks that use simple Java types:

This example creates a runtime task that uses only simple Java types in its interface, for example, a `String` object.

## About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

## Procedure

1. Access the ClientTaskFactory and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. Create the WSDL definition and add the descriptions of your operations.

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
 (resourceSet, new QName("http://www.ibm.com/task/test/", "test"));
```

```
// create a port type
PortType portType = factory.createPortType(definition, "doItPT");
```

```
// create an operation; the input and output messages are of type String:
// a fault message is not specified
```

```
Operation operation = factory.createOperation
 (definition, portType, "doIt",
 new QName("http://www.w3.org/2001/XMLSchema", "string"),
 new QName("http://www.w3.org/2001/XMLSchema", "string"),
 (Map)null);
```

3. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```
TTask humanTask = factory.createTTask(resourceSet,
 TTaskKinds.HTASK_LITERAL,
 "TestTask",
 new UTCDate("2005-01-01T00:00:00"),
 "http://www.ibm.com/task/test/",
 portType,
 operation);
```

This step initializes the properties of the task model with default values.

4. Modify the properties of your human task model.

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance(TBoolean, YES_LITERAL);
```

```
// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();
```

```
// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");
```

```
// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
 taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);
```

```
// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel(resourceSet);
```

6. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. Create the runtime task instance or template.

Use the HumanTaskManagerService interface to create the task instance or the task template. Because the application uses simple Java types only, you do not need to specify an application name.

- The following snippet creates a task instance:

```
atask.createTask(taskModel, (String)null, "HTM");
```

- The following snippet creates a task template:

```
task.createTaskTemplate(taskModel, (String)null);
```

## Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

## Creating runtime tasks that use complex types:

This example creates a runtime task that uses complex types in its interface. The complex types are already defined, that is, the local file system on the client has XSD files that contain the description of the complex types.

## About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

## Procedure

1. Access the ClientTaskFactory and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. Add the XSD definitions of your complex types to the resource set so that they are available when you define your operations.

The files are located relative to the location where the code is executed.

```
factory.loadXSDSchema(resourceSet, "InputBO.xsd");
factory.loadXSDSchema(resourceSet, "OutputBO.xsd");
```

3. Create the WSDL definition and add the descriptions of your operations.

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
 (resourceSet, new QName("http://www.ibm.com/task/test/", "test"));
```

```
// create a port type
PortType portType = factory.createPortType(definition, "doItPT");
```

```
// create an operation; the input message is an InputBO and
// the output message an OutputBO;
// a fault message is not specified
```

```
Operation operation = factory.createOperation
 (definition, portType, "doIt",
 new QName("http://Input", "InputBO"),
 new QName("http://Output", "OutputBO"),
 (Map)null);
```

4. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```
TTask humanTask = factory.createTTask(resourceSet,
 TTaskKinds.HTASK_LITERAL,
 "TestTask",
 new UTCDate("2005-01-01T00:00:00"),
 "http://www.ibm.com/task/test/",
 portType,
 operation);
```

This step initializes the properties of the task model with default values.

5. Modify the properties of your human task model.

```

// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance(TBoolean, YES_LITERAL);

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
 taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);

```

6. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel(resourceSet);
```

7. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

8. Create the runtime task instance or template.

Use the `HumanTaskManagerService` interface to create the task instance or the task template. You must provide an application name that contains the data type definitions so that they can be accessed. The application must also contain a dummy task or process so that the application is loaded by Business Process Choreographer.

- The following snippet creates a task instance:

```
task.createTask(taskModel, "B0application", "HTM");
```

- The following snippet creates a task template:

```
task.createTaskTemplate(taskModel, "B0application");
```

## Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

### Creating runtime tasks that use an existing interface:

This example creates a runtime task that uses an interface that is already defined, that is, the local file system on the client has a file that contains the description of the interface.

### About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

### Procedure

1. Access the `ClientTaskFactory` and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. Access the WSDL definition and the descriptions of your operations.

The interface description is located relative to the location where the code is executed.

```
Definition definition = factory.loadWSDLDefinition(
 resourceSet, "interface.wsdl");
PortType portType = definition.getPortType(
 new QName(definition.getTargetNamespace(), "doItPT"));
Operation operation = portType.getOperation
 ("doIt", (String)null, (String)null);
```

3. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```
TTask humanTask = factory.createTTask(resourceSet,
 TTaskKinds.HTASK_LITERAL,
 "TestTask",
 new UTCDate("2005-01-01T00:00:00"),
 "http://www.ibm.com/task/test/",
 portType,
 operation);
```

This step initializes the properties of the task model with default values.

4. Modify the properties of your human task model.

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance(TBoolean, YES_LITERAL);

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
 taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel(resourceSet);
```

6. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. Create the runtime task instance or template.

Use the HumanTaskManagerService interface to create the task instance or the task template. You must provide an application name that contains the data type definitions so that they can be accessed. The application must also contain a dummy task or process so that the application is loaded by Business Process Choreographer.

- The following snippet creates a task instance:  
task.createTask( taskModel, "B0application", "HTM" );
- The following snippet creates a task template:  
task.createTaskTemplate( taskModel, "B0application" );

## Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

## Creating runtime tasks that use an interface from the calling application:

This example creates a runtime task that uses an interface that is part of the calling application. For example, the runtime task is created in a Java snippet of a business process and uses an interface from the process application.

### About this task

The example runs only inside the context of the calling enterprise application, for which the resources are loaded.

### Procedure

1. Access the ClientTaskFactory and create a resource set to contain the definitions of the new task model.

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();

// specify the context class loader so that following resources are found
ResourceSet resourceSet = factory.createResourceSet
 (Thread.currentThread().getContextClassLoader());
```

2. Access the WSDL definition and the descriptions of your operations.

Specify the path within the containing package JAR file.

```
Definition definition = factory.loadWSDLDefinition(resourceSet,
 "com/ibm/workflow/metaflow/interface.wsdl");
PortType portType = definition.getPortType(
 new QName(definition.getTargetNamespace(), "doItPT"));
Operation operation = portType.getOperation
 ("doIt", (String)null, (String)null);
```

3. Create the EMF model of your new human task.

If you are creating a task instance, a valid-from date (UTCDate) is not required.

```
TTask humanTask = factory.createTTask(resourceSet,
 TTaskKinds.HTASK_LITERAL,
 "TestTask",
 new UTCDate("2005-01-01T00:00:00"),
 "http://www.ibm.com/task/test/",
 portType,
 operation);
```

This step initializes the properties of the task model with default values.

4. Modify the properties of your human task model.

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance(TBoolean, YES_LITERAL);

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
 taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. Create the task model that contains all the resource definitions.

```
TaskModel taskModel = ClientTaskFactory.createTaskModel(resourceSet);
```



6. Validate the task model and correct any validation problems that are found.

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. Create the runtime task instance or template.

Use the `HumanTaskManagerService` interface to create the task instance or the task template. You must provide an application name that contains the data type definitions so that they can be accessed.

- The following snippet creates a task instance:

```
task.createTask(taskModel, "WorkflowApplication", "HTM");
```

- The following snippet creates a task template:

```
task.createTaskTemplate(taskModel, "WorkflowApplication");
```

## Results

If a runtime task instance is created, it can now be started. If a runtime task template is created, you can now create task instances from the template.

## HumanTaskManagerService interface

The `HumanTaskManagerService` interface exposes task-related functions that can be called by a local or a remote client.

The methods that can be called depend on the state of the task and the authorization of the person that uses the application containing the method. The main methods for manipulating task objects are listed here. For more information about these methods and the other methods that are available in the `HumanTaskManagerService` interface, see the Javadoc in the `com.ibm.task.api` package.

## Task templates

The following methods are available to work with task templates.

*Table 65. API methods for task templates*

Method	Description
<code>getTaskTemplate</code>	Retrieves the specified task template.
<code>createTask</code>	Creates a task instance from the specified task template.
<code>createAndCallTask</code>	Creates and runs a task instance from the specified task template and waits synchronously for the result.
<code>createAndStartTask</code>	Creates and starts a task instance from the specified task template.
<code>createAndStartTaskAsSubtask</code>	Creates and starts a task instance as a subtask of the specified task.
<code>createInputMessage</code>	Creates an input message for the specified task template. For example, create a message that can be used to start a task.
<code>queryTaskTemplates</code>	Retrieves task templates that are stored in the database.

## Task instances

The following methods are available to work with task instances.

Table 66. API methods for task instances

Method	Description
getTask	Retrieves a task instance; the task instance can be in any state.
callTask	Starts an invocation task synchronously.
startTask	Starts a task that has already been created.
startTaskAsSubtask	Starts a task as a subtask of the task instance.
suspend	Suspends the collaboration or to-do task.
resume	Resumes the collaboration or to-do task.
restart	Restarts the task instance.
terminate	Terminates the specified task instance. If an invocation task is terminated, this action has no impact on the invoked service.
delete	Deletes the specified task instance.
claim	Claims the task for processing.
update	Updates the task instance.
complete	Completes the task instance.
completeWithFollowOnTask	Completes the task instance and starts a follow-on task.
cancelClaim	Releases a claimed task instance so that it can be worked on by another potential owner.
createWorkItem	Creates a work item for the task instance.
transferWorkItem	Transfers the work item to a specified owner.
deleteWorkItem	Deletes the work item.

## Escalations

The following methods are available to work with escalations.

Table 67. API methods for working with escalations

Method	Description
getEscalation	Retrieves the specified escalation instance.
triggerEscalation	Manually triggers an escalation.

## Custom properties

Tasks, task templates, and escalations can all have custom properties. The interface provides a get and a set method to retrieve and set values for custom properties. You can also associate named properties with, and retrieve named properties from task instances. Custom property names and values must be of the `java.lang.String` type. The following methods are valid for tasks, task templates, and escalations.

Table 68. API methods for variables and custom properties

Method	Description
getCustomProperty	Retrieves the named custom property of the specified task instance.
getCustomProperties	Retrieves the custom properties of the specified task instance.
getCustomPropertyNames	Retrieves the names of the custom properties for the task instance.
setCustomProperty	Stores custom-specific values for the specified task instance.

## Developing applications for business processes and human tasks

People are involved in most business process scenarios. For example, a business process requires people interaction when the process is started or administered, or when human task activities are performed. To support these scenarios, you need to use both the Business Flow Manager API and the Human Task Manager API.

### About this task

To involve people in business process scenarios, you can include the following task kinds in the business process:

- An inline invocation task (also known as an *originating task* in the API).  
You can provide an invocation task for every receive activity, for each onMessage element of a pick activity, and for each onEvent element of an event handler. This task then controls who is authorized to start a process or communicate with a running process instance.
- An administration task.  
You can provide an administration task to specify who is authorized to administer the process or perform administrative operations on the failed activities of the process.
- A to-do task (also known as a *participating task* in the API).  
A to-do task implements a human task activity. This type of activity allows you to involve people in the process.

Human task activities in the business process represent the to-do tasks that people perform in the business process scenario. You can use both the Business Flow Manager API and the Human Task Manager API to realize these scenarios:

- The business process is the container for all of the activities that belong to the process, including the human task activities that are represented by to-do tasks. When a process instance is created, a unique object ID (PIID) is assigned.
- When a human task activity is activated during the execution of the process instance, an activity instance is created, which is identified by its unique object ID (AIID). At the same time, an inline to-do task instance is also created, which is identified by its object ID (TKIID). The relationship of the human task activity to the task instance is achieved by using the object IDs:
  - The to-do task ID of the activity instance is set to the TKIID of the associated to-do task.
  - The containment context ID of the task instance is set to the PIID of the process instance that contains the associated activity instance.

- The parent context ID of the task instance is set to the AIID of the associated activity instance.
- The life cycles of all inline to-do task instances are managed by the process instance. When the process instance is deleted, then the task instances are also deleted. For example, all of the tasks that have the containment context ID set to the PIID of the process instance are automatically deleted.

## Determining the process templates or activities that can be started

A business process can be started by invoking the `call`, `initiate`, or `sendMessage` methods of the Business Flow Manager API. If the process has only one starting activity, you can use the method signature that requires a process template name as a parameter. If the process has more than one starting activity, you must explicitly identify the starting activity.

### About this task

When a business process is modeled, the modeler can decide that only a subset of users can create a process instance from the process template. This is done by associating an inline invocation task to a starting activity of the process and by specifying authorization restrictions on that task. Only the people that are potential starters or administrators of the task are allowed to create an instance of the task, and thus an instance of the process template.

If an inline invocation task is not associated with the starting activity, or if authorization restrictions are not specified for the task, everybody can create a process instance using the starting activity.

A process can have more than one starting activity, each with different people queries for potential starters or administrators. This means that a user can be authorized to start a process using activity A but not using activity B.

### Procedure

1. Use the Business Flow Manager API to create a list of the current versions of process templates that are in the started state.

**Tip:** The `queryProcessTemplates` method excludes only those process templates that are part of applications that are not yet started. So, if you use this method without filtering the results, the method returns all of the versions of the process templates regardless of which state they are in.

```
// current timestamp in UTC format, converted to yyyy-mm-ddThh:mm:ss
String now = (new UTCDate()).toXsdString();
String whereClause = "PROCESS_TEMPLATE.STATE =
PROCESS_TEMPLATE.STATE.STATE_STARTED AND
PROCESS_TEMPLATE.VALID_FROM =
(SELECT MAX(VALID_FROM) FROM PROCESS_TEMPLATE
WHERE NAME=PROCESS_TEMPLATE.NAME AND
VALID_FROM <= TS('" + now + "'))";
```

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
(whereClause,
"PROCESS_TEMPLATE.NAME",
(Integer)null, (TimeZone)null);
```

The results are sorted by process template name.

2. Create the list of process templates and the list of starting activities for which the user is authorized.

The list of process templates contains those process templates that have a single starting activity. These activities are either not secured or the logged-on user is allowed to start them. Alternatively, you might want to gather the process templates that can be started by at least one of the starting activities.

**Tip:** A process administrator can also start a process instance. However, if Business Flow Manager is using the alternate process administration authorization mode, which restricts process administration to system administrators, then only users in the BPESystemAdministrator role can perform this action. Therefore, to get a complete list of templates, you also need to check whether the logged-on user is an administrator.

```
List authorizedProcessTemplates = new ArrayList();
List authorizedActivityServiceTemplates = new ArrayList();
```

- Determine the starting activities for each of the process templates.

```
for(int i=0; i<processTemplates.length; i++)
{
 ProcessTemplateData template = processTemplates[i];
 ActivityServiceTemplateData[] startActivities =
 process.getStartActivities(template.getID());
```

- For each starting activity, retrieve the ID of the associated inline invocation task template.

```
for(int j=0; j<startActivities.length; j++)
{
 ActivityServiceTemplateData activity = startActivities[j];
 TKID tktid = activity.getTaskTemplateID();
```

- If an invocation task template does not exist, the process template is not secured by this starting activity.

In this case, everybody can create a process instance using this start activity.

```
boolean isAuthorized = false;
 if (tktid == null)
 {
 isAuthorized = true;
 authorizedActivityServiceTemplates.add(activity);
 }
```

- If an invocation task template exists, use the Human Task Manager API to check the authorization for the logged-on user.

In the example, the logged-on user is Smith. The logged-on user must be a potential starter of the invocation task or an administrator.

```
if (tktid != null)
{
 isAuthorized =
 task.isUserInRole
 (tkid, "Smith", WorkItem.REASON_POTENTIAL_STARTER) ||
 task.isUserInRole(tktid, "Smith", WorkItem.REASON_ADMINISTRATOR);

 if (isAuthorized)
 {
 authorizedActivityServiceTemplates.add(activity);
 }
}
```

If the user has the specified role, or if people assignment criteria for the role are not specified, the `isUserInRole` method returns the value `true`.

- Check whether the process can be started using only the process template name.

```
if (isAuthorized && startActivities.length == 1)
{
 authorizedProcessTemplates.add(template);
}
```

6. End the loops.

```
 } // end of loop for each activity service template
 } // end of loop for each process template
```

## Processing a single person workflow that includes human tasks

Some workflows are performed by only one person, for example, ordering books from an online bookstore. This example shows how to implement a single person workflow using a server-side page flow. Both the Business Flow Manager and the Human Task Manager APIs are used to process the workflow.

### About this task

A single person workflow is also referred to as a *page flow* or a *screen flow*. There are two kinds of page flows:

- Client-side page flows, where the navigation between the different pages is realized using client-side technology, such as a multi-page Lotus Forms form.
- Server-side page flows are realized using a business process and a set of human tasks that are modeled so that subsequent tasks are assigned to the same person.

Server-side page flows are more powerful than client-side page flows, but they consume more server resources to process them. Therefore, consider using this type of workflow primarily in the following situations:

- You need to invoke services between steps carried out in a user interface, for example, to retrieve or update data.
- You have auditing requirements that require CEI events to be written after a user interface interaction completes.

In an online bookstore, the purchaser completes a sequence of actions to order a book. This sequence of actions can be implemented as a series of human task activities (to-do tasks). If the purchaser decides to order several books, this is equivalent to claiming the next human task activity. Information about the sequence of tasks is maintained by Business Flow Manager, while the tasks themselves are maintained by Human Task Manager.

Compare this example with the example that uses only the Business Flow Manager API.

### Procedure

1. Use the Business Flow Manager API to get the process instance that you want to work on.

In this example, an instance of the CustomerOrder process.

```
ProcessInstanceData processInstance =
 process.getProcessInstance("CustomerOrder");
String piid = processInstance.getID().toString();
```

2. Use the Human Task Manager API to query the ready to-do tasks (kind participating) that are part of the specified process instance.

Use the containment context ID of the task to specify the containing process instance. For a single person workflow, the query returns the to-do task that is associated with the first human task activity in the sequence of human task activities.

```
//
// Query the list of to-do tasks that can be claimed by the logged-on user
// for the specified process instance
//
QueryResultSet result =
```

```

task.query("DISTINCT TASK.TKIID",
 "TASK.CONTAINMENT_CTX_ID = ID('" + piid + "') AND
 TASK.STATE = TASK.STATE.STATE_READY AND
 TASK.KIND = TASK.KIND.KIND_PARTICIPATING AND
 WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
 (String)null, (Integer)null, (TimeZone)null);

```

3. Claim the to-do task that is returned.

```

if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 ClientObjectWrapper input = task.claim(tkiid);
 DataObject activityInput = null ;
 if (input.getObject() != null && input.getObject() instanceof DataObject)
 {
 taskInput = (DataObject)input.getObject();
 // read the values
 ...
 }
}

```

When the task is claimed, the input message of the task is returned.

4. Determine the human task activity that is associated with the to-do task.  
You can use one of the following methods to correlate activities to their tasks.

- The task.getActivityID method:

```
AIID aiid = task.getActivityID(tkiid);
```

- The parent context ID that is part of the task object:

```

AIID aiid = null;
Task taskInstance = task.getTask(tkiid);

OID oid = taskInstance.getParentContextID();
if (oid != null and oid instanceof AIID)
{
 aiid = (AIID)oid;
}

```

5. When work on the task is finished, use the Business Flow Manager API to complete the task and its associated human task activity, and claim the next human task activity in the process instance.

To complete the human task activity, an output message is passed. When you create the output message, you must specify the message type name so that the message definition is contained.

```

ActivityInstanceData activity = process.getActivityInstance(aiid);
ClientObjectWrapper output =
 process.createMessage(aiid, activity.getOutputMessageType());
DataObject myMessage = null ;
if (output.getObject() != null && output.getObject() instanceof DataObject)
{
 myMessage = (DataObject)output.getObject();
 //set the parts in your message, for example, an order number
 myMessage.setInt("OrderNo", 4711);
}

//complete the human task activity and its associated to-do task,
// and claim the next human task activity
CompleteAndClaimSuccessorResult successor =
 process.completeAndClaimSuccessor(aiid, output);

```

This action sets an output message that contains the order number and claims the next human task activity in the sequence. If AutoClaim is set for successor activities and if there are multiple paths that can be followed, all of the

successor activities are claimed and a random activity is returned as the next activity. If there are no more successor activities that can be assigned to this user, `Null` is returned.

If the process contains parallel paths that can be followed and these paths contain human task activities for which the logged-on user is a potential owner of more than one of these activities, a random activity is claimed automatically and returned as the next activity.

6. Work on the next human task activity.

```
ClientObjectWrapper nextInput = successor.getInputMessage();
if (nextInput.getObject() !=
 null && nextInput.getObject() instanceof DataObject)
{
 activityInput = (DataObject)input.getObject();
 // read the values
 ...
}

aiid = successor.getAIID();
```

7. Continue with step 5 to complete the human task activity and to retrieve the next human task activity.

### Related tasks

“Processing a single person workflow” on page 392

Some workflows are performed by only one person, for example, ordering books from an online bookstore. This type of workflow has no parallel paths. The `initiateAndClaimFirst` and `completeAndClaimSuccessor` APIs support the processing of this type of workflow. This example shows the implementation of a single person workflow using a client-side page flow.

## Handling exceptions and faults

A BPEL process might encounter a fault at different points in the process.

### About this task

Business Process Execution Language (BPEL) faults originate from:

- Web service invocations (Web Services Description Language (WSDL) faults)
- Throw activities
- BPEL standard faults that are recognized by Business Process Choreographer

Mechanisms exist to handle these faults. Use one of the following mechanisms to handle faults that are generated by a process instance:

- Pass control to the corresponding fault handlers
- Compensate previous work in the process
- Stop the process and let someone repair the situation (force-retry, force-complete)

A BPEL process can also return faults to a caller of an operation provided by the process. You can model the fault in the process as a reply activity with a fault name and fault data. These faults are returned to the API caller as checked exceptions.

If a BPEL process does not handle a BPEL fault or if an API exception occurs, a runtime exception is returned to the API caller. An example for an API exception is when the process model from which an instance is to be created does not exist.

The handling of faults and exceptions is described in the following tasks.



## Handling Business Process Choreographer EJB API exceptions

If a method in the `BusinessFlowManagerService` interface or the `HumanTaskManagerService` interface does not complete successfully, an exception is thrown that denotes the cause of the error. You can handle this exception specifically to provide guidance to the caller.

### About this task

However, it is common practice to handle only a subset of the exceptions specifically and to provide general guidance for the other potential exceptions. All specific exceptions inherit from a generic `ProcessException` or `TaskException`. Catch generic exceptions with a final `catch(ProcessException)` or `catch(TaskException)` statement. This statement helps to ensure the upward compatibility of your application program because it takes account of all of the other exceptions that can occur.

### Checking which fault is set for a human task activity

When a human task activity is processed, it can complete successfully. In this case, you can pass an output message. If the human task activity does not complete successfully, you can pass a fault message.

### About this task

You can read the fault message to determine the cause of the error.

### Procedure

1. List the task activities that are in a failed or stopped state.

```
QueryResultSet result =
 process.query("ACTIVITY.AIID",
 "(ACTIVITY.STATE = ACTIVITY.STATE.STATE_FAILED OR
 ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED) AND
 ACTIVITY.KIND=ACTIVITY.KIND.KIND_STAFF",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns a query result set that contains failed or stopped activities.

2. Read the name of the fault.

```
if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);
 ClientObjectWrapper faultMessage = process.getFaultMessage(aaid);
 DataObject fault = null ;
 if (faultMessage.getObject() != null && faultMessage.getObject()
 instanceof DataObject)
 {
 fault = (DataObject)faultMessage.getObject();
 Type type = fault.getType();
 String name = type.getName();
 String uri = type.getURI();
 }
}
```

This returns the fault name. You can also analyze the unhandled exception for a stopped activity instead of retrieving the fault name.

### Checking which fault occurred for a stopped invoke activity

In a well-designed process, exceptions and faults are usually handled by fault handlers. You can retrieve information about the exception or fault that occurred for an invoke activity from the activity instance.

## About this task

If an activity causes a fault to occur, the fault type determines the actions that you can take to repair the activity.

### Procedure

1. List the human task activities that are in a stopped state.

```
QueryResultSet result =
 process.query("ACTIVITY.AIID",
 "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
 ACTIVITY.KIND=ACTIVITY.KIND.KIND_INVOKE",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns a query result set that contains stopped invoke activities.

2. Read the name of the fault.

```
if (result.size() > 0)
{
 result.first();
 AIID aaid = (AIID) result.getOID(1);
 ActivityInstanceData activity = process.getActivityInstance(aaid);

 ProcessException excp = activity.getUnhandledException();
 if (excp instanceof ApplicationFaultException)
 {
 ApplicationFaultException fault = (ApplicationFaultException)excp;
 String faultName = fault.getFaultName();
 }
}
```

## Checking which unhandled exception or fault occurred for a failed process instance

In a well-designed process, exceptions and faults are usually handled by a fault handler. If the process implements a two-way operation, you can retrieve information about a fault or handled exception from the fault name property of the process instance object. For faults, you can also retrieve the corresponding fault message using the `getFaultMessage` API.

## About this task

If a process instance fails because of an exception that is not handled by any fault handler, you can retrieve information about the unhandled exception from the process instance object. By contrast, if a fault is caught by a fault handler, then information about the fault is not available. You can, however, retrieve the fault name and message and return to the caller by using a `FaultReplyException` exception.

### Procedure

1. List the process instances that are in the failed state.

```
QueryResultSet result =
 process.query("PROCESS_INSTANCE.PIID",
 "PROCESS_INSTANCE.STATE =
 PROCESS_INSTANCE.STATE.STATE_FAILED",
 (String)null, (Integer)null, (TimeZone)null);
```

This action returns a query result set that contains the failed process instances.

2. Read the information for the unhandled exception.

```

if (result.size() > 0)
{
 result.first();
 PIID piid = (PIID) result.getOID(1);
 ProcessInstanceData pInstance = process.getProcessInstance(piid);

 ProcessException excp = pInstance.getUnhandledException();
 if (excp instanceof RuntimeException)
 {
 RuntimeException xcp = (RuntimeException)excp;
 Throwable cause = xcp.getRootCause();
 }
 else if (excp instanceof StandardFaultException)
 {
 StandardFaultException xcp = (StandardFaultException)excp;
 String faultName = xcp.getFaultName();
 }
 else if (excp instanceof ApplicationFaultException)
 {
 ApplicationFaultException xcp = (ApplicationFaultException)excp;
 String faultName = xcp.getFaultName();
 }
}
}

```

## Results

Use this information to look up the fault name or the root cause of the problem.

---

## Developing Web services API client applications for business processes and human tasks

You can develop client applications that access business process applications and human task applications through the Business Process Choreographer Web services APIs. The client application development process consists of a number of mandatory and optional steps, including generating a Web service proxy and adding security and transaction policies to the client application.

### About this task

Beginning with Version 7, the JAX-WS-based Web services API replaces the JAX-RPC-based Business Process Choreographer Web services API in Version 6 (first published in release 6.0.2). The JAX-RPC-based Business Process Choreographer Web services API will be deprecated, such that new Web service client applications should be implemented using the JAX-WS-based API.

**Note:** The Business Process Choreographer Java Message Service (JMS) API is still using the WSDL and XML Schema definitions for Version 6.

You can develop client applications in any Web services client environment. The following steps provide an overview of the actions you need to take to develop such an application.

### Procedure

1. Decide which Web services API your client application needs to use: the Business Flow Manager API, Human Task Manager API, or both.
2. Export the necessary files from the WebSphere Process Server environment.
3. In your client application development environment, generate a *Web service proxy* using the exported artifacts.

4. Develop the code for your client application.
5. Add any necessary security or transaction policies to your client application.

## Web service components and sequence of control

In Web services applications, a number of client-side and server-side components participate in the sequence of control that represents a Web service request and response.

A typical sequence of control is as follows.

1. On the client side:
  - a. A client application (provided by the user) issues a request for a Web service.
  - b. A Web service proxy (also provided by the user, but which can be automatically generated using client-side utilities) wraps the service request in a SOAP request envelope and forwards the request to a URL defined as the Web service's endpoint.
2. The network transmits the request to the Web service endpoint using HTTP or HTTPS.
3. On the server side:
  - a. The generic Web services API receives and decodes the request.
  - b. The request is either handled directly by the generic Business Flow Manager or Human Task Manager component, or forwarded to the specified business process or human task.
  - c. The returned data is wrapped in a SOAP response envelope.
4. The network transmits the response to the client-side environment using HTTP or HTTPS.
5. Back on the client side:
  - a. The client-side development infrastructure unwraps the SOAP response envelope.
  - b. The Web service proxy extracts the data from the SOAP response and passes it to the client application.
  - c. The client application processes the returned data as necessary.

### Example

The following is a possible outline for a client application that accesses the Human Task Manager Web services API to process a to-do task:

1. The client application issues a query Web service call to the WebSphere Process Server requesting a list of to-do tasks to be worked on by a user.
2. The WebSphere Process Server returns the list of to-do tasks.
3. The client application then issues a claim Web service call to claim one of the to-do tasks.
4. The WebSphere Process Server returns the input message for the task.
5. The client application issues a complete Web service call to complete the task with an output or fault message.

## Web service API requirements for business processes and human tasks

Business processes and human tasks developed with the WebSphere Integration Developer to run on the Business Process Choreographer must conform to specific rules to be accessible through the Web services APIs.

The requirements are:

- The interfaces of business processes and human tasks must be defined using the "document/literal wrapped" style defined in the Java API for XML-based Web Services (JAX-WS 2.0) specification. This is the default style for all business processes and human tasks developed with WebSphere Integration Developer.
- Do not use the maxOccurs attribute in parameter elements of the operations, or ensure that the value of this attribute is set to the default value, maxOccurs="1".
- Fault messages that are exposed by business processes and human tasks for Web service operations must comprise a single WSDL message part defined with an XML Schema element. For example:

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

### Related information

 [Java API for XML-based Web Services \(JAX-WS 2.0\) downloads page](#)

 [Which style of WSDL should I use?](#)

## JAX-WS-based Business Process Choreographer Web services APIs

Beginning with Version 7, the JAX-WS-based Web services API replaces the JAX-RPC-based Business Process Choreographer Web services API in Version 6 (first published in release 6.0.2). Two Business Process Choreographer Web services interfaces are provided, one for business processes and one for human tasks, each with their own file artifacts and XML definition namespaces.

The following table provides an overview of the file artifacts and XML definition namespaces for the JAX-WS-based Web services.

*Table 69. File artifacts and XML definition namespaces for the JAX-WS-based Web services*

Business Process Choreographer Web services interface	JAX-WS Web services file artifact	JAX-WS Web services XML namespace
Business Flow Manager Web Service	BFMJAXWSService.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0/Binding">http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0/Binding</a>
Business Flow Manager Web Service Interface	BFMJAXWSInterface.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0">http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0</a>
Business Flow Manager Web Service Data Types	BFMDataTypes.xsd	<a href="http://www.ibm.com/xmlns/prod/websphere/business-process/types/7.0">http://www.ibm.com/xmlns/prod/websphere/business-process/types/7.0</a>
Business Flow Manager Callback Web Service	BFMJAXWSCallbackService.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/business-process/callback-services/7.0/Binding">http://www.ibm.com/xmlns/prod/websphere/business-process/callback-services/7.0/Binding</a>
Business Flow Manager Callback Web Service Interface	BFMJAXWSCallbackInterface.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/business-process/callback-services/7.0">http://www.ibm.com/xmlns/prod/websphere/business-process/callback-services/7.0</a>
Human Task Manager Web Service	HTMJAXWSService.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/human-task/services/7.0/Binding">http://www.ibm.com/xmlns/prod/websphere/human-task/services/7.0/Binding</a>
Human Task Manager Web Service Interface	HTMJAXWSInterface.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/human-task/services/7.0">http://www.ibm.com/xmlns/prod/websphere/human-task/services/7.0</a>
Human Task Manager Web Service Data Types	HTMDataTypes.xsd	<a href="http://www.ibm.com/xmlns/prod/websphere/human-task/types/7.0">http://www.ibm.com/xmlns/prod/websphere/human-task/types/7.0</a>
Human Task Manager Callback Web Service	HTMJAXWSCallbackService.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/human-task/callback-services/7.0/Binding">http://www.ibm.com/xmlns/prod/websphere/human-task/callback-services/7.0/Binding</a>

Table 69. File artifacts and XML definition namespaces for the JAX-WS-based Web services (continued)

Business Process Choreographer Web services interface	JAX-WS Web services file artifact	JAX-WS Web services XML namespace
Human Task Manager Callback Web Service Interface	HTMJAXWSCallbackInterface.wsdl	<a href="http://www.ibm.com/xmlns/prod/websphere/human-task/callback-services/7.0">http://www.ibm.com/xmlns/prod/websphere/human-task/callback-services/7.0</a>
Common Business Process Choreographer Data Types	BPCDataTypes.xsd	<a href="http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/7.0">http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/7.0</a>

## Business Process Choreographer Web services API: Standards

Use the following links to find relevant supplemental information about the standards that apply to Web applications. The information resides on non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to IBM WebSphere Process Server, but is useful for understanding Web services in general.

- Java API for XML-based Web Services (JAX-WS 2.0) (JSR-224; Java Community Process)
- Java Architecture for XML Binding (JAXB) 2.0 (JSR-222; Java Community Process)
- Web Services Description Language (WSDL) 1.1 (W3C)
- XML Schema Part 0: Primer Second Edition (W3C)
- XML Schema Part 1: Structures Second Edition (W3C)
- XML Schema Part 2: Datatypes Second Edition (W3C)
- Simple Object Access Protocol (SOAP) 1.1 (W3C)
- Web Services Policy Framework (WS-Policy) 1.5 (W3C)
- WS-Security 1.1 (OASIS)
- WS-Security UserName Token Profile 1.1 (OASIS)
- WS-AtomicTransaction 1.2 (OASIS)
- WS-Interoperability Basic Profile 1.1 (WS-Interoperability Organization)

## Publishing and exporting artifacts from the server environment for Web services client applications

Before you can develop client applications to access the Business Process Choreographer Web services APIs, you must publish and export a number of artifacts from the WebSphere server environment.

### About this task

The artifacts to be exported are:

- Web Service Definition Language (WSDL) files describing the Web service endpoint, the port types and operations that make up the Business Process Choreographer Web services API (always required for the Web service proxy generation).
- XML Schema Definition (XSD) files containing definitions of data types referenced by services in the Business Process Choreographer WSDL files (always required for the Web service proxy generation).
- Your own WSDL and XSD files describing interfaces and data types for your business processes or human tasks running on the WebSphere server. These

additional files are only required if your client application needs to interact directly with your business processes or human tasks through the Web services APIs. They are not necessary if your client application is only going to invoke operations that can be fulfilled by Business Process Choreographer without direct interaction with your process or task instances, such as issuing queries.

- Web Service Policy (WS-Policy) files describing the quality of service attributes for the Web services API. They may be exported in order to serve as a base for creating client-side Web service policies.

#### **WS-Security**

The request message must contain either a Username token or an LPTA token.

#### **WS-Transaction**

The request message can contain a WS-AtomicTransaction context. If this context is present, the request is processed in the transaction scope of the caller.

After these artifacts are published, you need to copy them to your client programming environment, where they are used to generate a Web service proxy and helper classes.

### **Publishing Business Process Choreographer WSDL files**

A Web Service Definition Language (WSDL) file contains a detailed description of all the operations available with a Web services API. Separate WSDL files are available for the Business Flow Manager and Human Task Manager Web services APIs. These files are used to generate a Web service proxy for your application.

#### **Before you begin**

Before publishing the WSDL files, be sure to specify the correct Web services endpoint address. This is the URL that your client application uses to access the Web services APIs.

#### **About this task**

You must publish these WSDL files, and any XSD files referenced by the WSDL files. You can then copy them from the WebSphere environment to your development environment, where they are used to generate a Web service proxy and helper classes. You only need to publish the Business Process Choreographer WSDL files once.

#### **Publishing the business process WSDL file for Web services applications:**

Use the administrative console to publish the WSDL file.

#### **Procedure**

1. Log on to the administrative console with a user ID with administrator rights.
2. Click **Applications** → **SCA modules**.

**Note:** You can also click **Applications** → **Application Types** → **WebSphere enterprise applications** to display a list of all available enterprise applications.

3. Choose the **BPEContainer** application from the list of SCA modules or applications.
4. Select **Publish WSDL files** from the list of **Additional properties**
5. Click the .zip file in the list.

6. On the File Download window that appears, click **Save**.
7. Browse to a local folder and click **Save**.

### Results

The exported .zip file is named BPEContainer\_nodename\_servername\_WSDLFiles.zip. The .zip file contains a WSDL file that describes the Web services, and any XSD files that are referenced by the WSDL file.

**Note:** The exported .zip file contains WSDL and XSD artifacts of both the JAX-WS Web service introduced in Version 7 and the JAX-RPC Web service used in Version 6. When you generate a the Web service proxy using the wsimport tool, you select the JAX-WS Web service artifacts and the JAX-RPC artifacts are ignored.

### Publishing the human task WSDL file for Web services applications:

Use the administrative console to publish the WSDL file.

### Procedure

1. Log on to the administrative console with a user ID with administrator rights.
2. Click **Applications** → **SCA modules**.

**Note:** You can also click **Applications** → **Application Types** → **WebSphere enterprise applications** to display a list of all available enterprise applications.

3. Choose the **TaskContainer** application from the list of SCA modules or applications.
4. Select **Publish WSDL files** from the list of **Additional properties**
5. Click the .zip file in the list.
6. On the File Download window that appears, click **Save**.
7. Browse to a local folder and click **Save**.

### Results

The exported .zip file is named TaskContainer\_nodename\_servername\_WSDLFiles.zip. The .zip file contains a WSDL file that describes the Web services, and any XSD files that are referenced by the WSDL file.

**Note:** The exported .zip file contains WSDL and XSD artifacts of both the JAX-WS Web service introduced in Version 7 and the JAX-RPC Web service used in Version 6. When you generate a the Web service proxy using wsimport tool, you select the JAX-WS Web service artifacts and the JAX-RPC artifacts are ignored.

### Exporting WSDL and XSD files for business process and human task Web services applications

Business processes and human tasks have well-defined interfaces that allow them to be accessed externally as Web services. You need to export the WSDL interface definitions and the XML Schema data type definitions to your client programming environment.

### About this task

This procedure must be repeated for each business process or human task that your client application needs to interact with.



For example, to create and start a human task, the following items of information must be passed to the task interface:

- The task template name
- The task template namespace
- An input message, containing formatted business data
- A response wrapper for returning the response message
- A fault message for returning faults and exceptions

These items are encapsulated within a single business object. All operations of the Web service interface are modeled as a document/literal wrapped operation. Input and output parameters for these operations are encapsulated in wrapper documents. Other business objects define the corresponding response and fault message formats.

In order to create and start the business process or human task through a Web service, these wrapper objects must be made available to the client application on the client side.

This is achieved by exporting the business objects from the WebSphere environment as Web Service Definition Language (WSDL) and XML Schema Definition (XSD) files, and importing the data type definitions into your client programming environment.

### Procedure

1. Launch the WebSphere Integration Developer Workspace if it is not already running.
2. Select the Library module containing the business objects to be exported. A Library module is a compressed file that contains the necessary business objects.
3. Export the Library module.
4. Copy the exported files to your client application development environment.

### Example

Assume a business process exposes the following Web service operation:

```
<wsdl:operation name="updateCustomer">
 <wsdl:input message="tns:updateCustomerRequestMsg"
 name="updateCustomerRequest"/>
 <wsdl:output message="tns:updateCustomerResponseMsg"
 name="updateCustomerResponse"/>
 <wsdl:fault message="tns:updateCustomerFaultMsg"
 name="updateCustomerFault"/>
</wsdl:operation>
```

with the WSDL messages defined as:

```
<wsdl:message name="updateCustomerRequestMsg">
 <wsdl:part element="types:updateCustomer"
 name="updateCustomerParameters"/>
</wsdl:message>
<wsdl:message name="updateCustomerResponseMsg">
 <wsdl:part element="types:updateCustomerResponse"
 name="updateCustomerResult"/>
</wsdl:message>
```

```

<wsdl:message name="updateCustomerFaultMsg">
 <wsdl:part element="types:updateCustomerFault"
 name="updateCustomerFault"/>
</wsdl:message>

```

The *concrete* customer-defined elements `types:updateCustomer`, `types:updateCustomerResponse`, and `types:updateCustomerFault` must be passed to and received from the Web services APIs using `UserData` parameters in all *generic* operations (`call`, `sendMessage`, and so on) performed by the client application.

The customer-defined elements are created, serialized and deserialized on the client application side using classes generated using the exported XSD files. The generation of these classes is part of the Web service proxy generation where the exported WSDL and XSD files are included.

The generic operations of the Web service interface propagate the document wrapper element to and from the operation that is implemented by the business process or human task. For the sample operation in the previous example, a Web service SOAP message might look as follows:

```

<soapenv:Envelope xmlns:soapenv="..." ...>
 <soapenv:Header>
 ...
 </soapenv:Header>
 <soapenv:Body>
 <bfm:sendMessage
 xmlns:bfm="http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0">
 <processTemplateName>customerProcessTemplate</processTemplateName>
 <portType xmlns:cns="http://example.com/customerProcess">cns:customerProcessPortType</portType>
 <operation>updateCustomer</operation>
 <input>
 <cns:updateCustomer xmlns:cns="http://example.com/customerProcess">
 <street>1600 Pennsylvania Avenue Northwest</street>
 <city>Washington, DC 20006</city>
 </cns:updateCustomer>
 </input>
 </bfm:sendMessage>
 </soapenv:Body>
</soapenv:Envelope>

```

## Developing client applications in the Java Web services environment

You can use any Java-based development environment compatible with Java Web services to develop client applications for the Business Process Choreographer Web services APIs.

### Generating a Web service proxy (Java Web services)

Java Web services client applications use a *Web service proxy* to interact with the Business Process Choreographer Web services APIs.

#### About this task

A Web service proxy for Java Web services contains a number of JavaBeans classes that the client application calls to perform Web service requests. The Web service proxy handles the assembly of service parameters into SOAP messages, sends SOAP messages to the Web service over HTTP, receives responses from the Web service, and passes any returned data to the client application.

Basically, therefore, a Web service proxy allows a client application to call a Web service as if it were a local function.

**Note:** You only need to generate a Web service proxy once. All client applications accessing the same Web services API can then use the same Web service proxy.

In the IBM Web services environment, you can generate a Web service proxy in one of the following ways.

- Use Rational Application Developer or WebSphere Integration Developer integrated development environments.
- Use the `wsimport` command-line tool.

Other Java Web services development environments usually include either the `wsimport` tool or proprietary client application generation facilities.

### **Using Rational Application Developer to generate a Web service proxy for a Web services application:**

You can use the Rational Application Developer integrated development environment to generate a Web service proxy for your Web services client application. The following sequence of steps applies to Rational Application Developer Version 7.5.3.

#### **Before you begin**

Before generating a Web service proxy, you must have previously exported the WSDL and XSD files that describe the business process or human task Web services interfaces from the WebSphere environment and copied them to your client programming environment.

#### **Procedure**

1. Add the appropriate WSDL file to your project:

- For business processes:
  - a. Unzip the exported file `BPEContainer_nodename_servername_WSDLFiles.zip` to a temporary directory. Do not change the contents of this directory, and be aware that only the following WSDL and XSD files are used for the Web service proxy generation for interactions with business processes:
    - `BFMJAXWSService.wsdl`
    - `BFMJAXWSInterface.wsdl`
    - `BFMJAXWSCallbackService.wsdl`
    - `BFMJAXWSCallbackInterface.wsdl`
    - `BFMDataTypes.xsd`
    - `BPCDataTypes.xsd`
    - `wsa.xsd`
  - b. Import the subdirectory `META-INF` from the unzipped directory `BPEContainer_nodename_servername.ear/bfmjaxws.jar`.
- For human tasks:
  - a. Unzip the exported file `TaskContainer_nodename_servername_WSDLFiles.zip` to a temporary directory. Do not change the contents of this directory, and be aware that only the following WSDL and XSD files are used for the Web service proxy generation for interactions with human tasks:
    - `HTMJAXWSService.wsdl`
    - `HTMJAXWSInterface.wsdl`

- HTMJAXWSCallbackService.wsdl
- HTMJAXWSCallbackInterface.wsdl
- HTMDataTypes.xsd
- BPCDataTypes.xsd
- wsdl.xsd

- b. Import the subdirectory META-INF from the unzipped directory TaskContainer\_*nodename\_servername*.ear/htmjaxws.jar.

A new wsdl directory and subdirectory structure are created in your project.

2. Select the BFMJAXWSService.wsdl file located in the newly-created wsdl directory.
3. Right-click and choose **Web services** → **Generate client**.  
Before continuing with the remaining steps, ensure that the server has started.
4. On the Web Services window, click **Next** to accept all defaults.
5. On the Web Service JAX-WS Web Service Client Configuration window, change the Version of JAX-WS code to be generated to 2.0 and click **Finish** to accept all other defaults
6. Redo steps 2 to 5 of this procedure with HTMJAXWSService.wsdl and overwrite all files if you are prompted.

## Results

A Web service proxy, made up of a number of proxy, locator, and JAXB classes, is generated and added to your project.

## Using the wsimport command-line tool to generate a Web service proxy for a Web services application:

You can use the wsimport command-line tool to generate a Web service proxy for a Web services application.

### Before you begin

Before generating a Web service proxy, you must have previously exported the WSDL files that describe the business process or human task Web services APIs from the WebSphere environment, and copied them to your client programming environment.

### Procedure

1. Generate a Web service proxy for the Business Process Choreographer Web services API:

**Note:** For a detailed description of the wsimport command-line tool for JAX-WS applications, see the WebSphere Application Server wsimport command-line tool documentation.

```
wsimport.bat BFMJAXWSService.wsdl myService1.wsdl myService2.wsdl
-d proxy-bfm
-wsdllocation <bfm_location>
```

```
wsimport.bat HTMJAXWSService.wsdl myService1.wsdl myService2.wsdl
-d proxy-htm
-wsdllocation <htm_location>
```

In this example, `myService1.wsdl` and `myService2.wsdl` contain interface definitions of custom business processes, or human tasks, or both. In addition, `<bfm_location>` and `<htm_location>` can be obtained from the WSDL `<port>` element in `BFMJAXWSService.wsdl` and `HTMJAXWSService.wsdl`, respectively.

You can merge both proxies into one common directory (for example, `proxy-bpc`) and overwrite existing files if you are prompted.

2. Include the generated class files in your project.

### Related tasks

“Creating a client application for business processes and human tasks (Java Web services)”

A client application sends requests to and receives responses from the Business Process Choreographer Web services APIs. By using a Web service proxy to manage communications and helper classes to format complex data types, a client application can invoke Web service methods as if they were local functions.

### Creating a client application for business processes and human tasks (Java Web services)

A client application sends requests to and receives responses from the Business Process Choreographer Web services APIs. By using a Web service proxy to manage communications and helper classes to format complex data types, a client application can invoke Web service methods as if they were local functions.

### Before you begin

Before starting to create a client application, generate the Web service proxy.

### About this task

You can develop client applications using any Web services-compatible development tool, for example IBM Rational Application Developer. You can build any type of Web services application to call the Web services APIs.

### Procedure

1. Create a new client application project.
2. Generate the Web service proxy.
3. Code your client application.
4. Build the project.
5. Run the client application.

### Example

The following example shows how to use the Business Flow Manager Web service API.

```
try {
 // create bfm proxy
 BFMJAXWSPortType bfm = new BFMJAXWSService().getBFMJAXWSPort();

 // call getProcessTemplate
 ProcessTemplateType ptt =
 bfm.getProcessTemplate("MY_PROCESS_TEMPLATE_NAME");

 // handle return value
 System.out.println("Process template '" + ptt.getName() +
 "' found, details following:");
 System.out.println("Execution mode: " +
 ptt.getExecutionMode());
 System.out.println("Schema version: " +
 ptt.getSchemaVersion());
}
```

```

} catch (Exception e) {
 if (e instanceof ProcessFaultMsg)
 {
 ProcessFaultMsg pfm = (ProcessFaultMsg) e;
 List<FaultStackType> list =
 (pfm.getFaultInfo()).getFaultStack();
 FaultStackType fault = list.get(0);
 System.out.println("ProcessFaultMessage: " +
 fault.getMessage());
 }
 else
 {
 e.printStackTrace(System.out);
 }
}

```

### Related tasks

“Using the wsimport command-line tool to generate a Web service proxy for a Web services application” on page 438

You can use the wsimport command-line tool to generate a Web service proxy for a Web services application.

“Generating a Web service proxy (Java Web services)” on page 436

Java Web services client applications use a *Web service proxy* to interact with the Business Process Choreographer Web services APIs.

## Adding security

The Business Process Choreographer Web service requires that you configure your client application for an authentication mechanism.

### About this task

By default, Business Process Choreographer supports the following authentication mechanisms:

#### Username Token

A web service consumer supplies a Username token as a means of identifying the requestor by "username", and optionally using a password to authenticate that identity to the web service provider.

#### Binary Security Token – Lightweight Third-Party Authentication (LTPA) Token

A web service consumer supplies an LTPA token as a means of authenticating the requestor to the web service provider

You can replace the Business Process Choreographer Web service security policy by an alternative authentication mechanism. However, it is not possible to invoke Business Process Choreographer Web service operations as an unauthenticated user, so one authentication mechanism is always required.

## Adding transaction support

Web service client applications can be configured to allow server-side request processing to participate in the client's transaction, by passing a client application context as part of the service request. This atomic transaction support is defined in the Web Services-Atomic Transaction (WS-AT) specification.

### About this task

Business Process Choreographer runs each Web service operation request as a separate global transaction. Client applications can be configured to use transaction support in one of the following ways:

- Propagate the client's transaction context. Server-side request processing is performed within the client application transaction context and therefore committed (or backed out) together with the client's transaction. Conversely, if the server encounters a problem while the Web service operation is running and requests a rollback, the client application's transaction is also rolled back.
- Not use transaction support. Business Process Choreographer creates a new global transaction in which to run the request, but server-side request processing is not performed with the client application transaction context

The Web service policy attached to the Business Process Choreographer Web service allows that each request message may contain a WS-AT transaction context as described above. If you choose to invoke the Web service operations without passing a client transaction context, it is safe to ignore the provider-side transaction policy and configure the Web service client without a transaction policy.

---

## Developing client applications using the Business Process Choreographer JMS API

You can develop client applications that access business process applications asynchronously through the Java Messaging Service (JMS) API.

### About this task

JMS client applications exchange request and response messages with the JMS API. To create a request message, the client application fills a JMS TextMessage message body with an XML element representing the document/literal wrapper of the corresponding operation.

## Requirements for business processes

Business processes developed with the WebSphere Integration Developer to run on the Business Process Choreographer must conform to specific rules to be accessible through the JMS API.

The requirements are:

1. The interfaces of business processes must be defined using the "document/literal wrapped" style defined in the Java API for XML-based RPC (JAX-RPC 1.1) specification. This is the default style for all business processes and human tasks developed with the WebSphere Integration Developer.
2. Fault messages exposed by business processes and human tasks for Web service operations must comprise a single WSDL message part defined with an XML Schema element. For example:

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

### Related information

 [Java API for XML based RPC \(JAX-RPC\) downloads page](#)

 [Which style of WSDL should I use?](#)

## Authorization for JMS renderings

To authorize use of the JMS interface, security settings must be enabled in WebSphere Application Server.

When the business process container is installed, the role **JMSAPIUser** must be mapped to a user ID. This user ID is used to issue all JMS API requests. For example, if **JMSAPIUser** is mapped to "User A", all JMS API requests appear to the process engine to originate from "User A".

The **JMSAPIUser** role must be assigned the following authorities:

Request	Required authorization
forceTerminate	Process administrator
sendEvent	Potential activity owner or process administrator

**Note:** For all other requests, no special authorizations are required.

Special authority is granted to a person with the role of business process administrator. A business process administrator is a special role; it is different from the process administrator of a process instance. A business process administrator has all privileges.

You cannot delete the user ID of the process starter from your user registry while the process instance exists. If you delete this user ID, the navigation of this process cannot continue. You receive the following exception in the system log file:

```
no unique ID for: <user ID>
```

## Accessing the JMS interface

To send and receive messages through the JMS interface, an application must first create a connection to the `BPC.cellname.Bus`, create a session, then generate message producers and consumers.

### About this task

The process server accepts Java Message Service (JMS) messages that follow the point-to-point paradigm. An application that sends or receives JMS messages must perform the following actions.

The following example assumes that the JMS client is executed in a managed environment (EJB, application client, or Web client container).

### Procedure

1. Create a connection to the `BPC.cellname.Bus`. No preconfigured connection factory exists for a client application's requests: a client application can either use the JMS API's `ReplyConnectionFactory` or create its own connection factory, in which case it can use Java Naming and Directory Interface (JNDI) lookup to retrieve the connection factory. The JNDI-lookup name must be the same as the name specified when configuring the Business Process Choreographer's external request queue. The following example assumes the client application creates its own connection factory named "jms/clientCF".

```
//Obtain the default initial JNDI context.
Context initialContext = new InitialContext();

// Look up the connection factory.
// Create a connection factory that connects to the BPC bus.
// Call it, for example, "jms/clientCF".
// Also configure an appropriate authentication alias.
ConnectionFactory connectionFactory =
```



```

 (ConnectionFactory)initialcontext.lookup("jms/clientCF");

 // Create the connection.
 Connection connection = connectionFactory.createConnection();
2. Create a session so that message producers and consumers can be created.
 // Create a transaction session using auto-acknowledgment.
 Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
3. Create a message producer to send messages. The JNDI-lookup name must be
 the same as the name specified when configuring the Business Process
 Choreographer's external request queue.
 // Look up the destination of the Business Process Choreographer input queue to
 // send messages to.
 Queue sendQueue = (Queue) initialcontext.lookup("jms/BFMJMSAPIQueue");

 // Create a message producer.
 MessageProducer producer = session.createProducer(sendQueue);
4. Create a message consumer to receive replies. The JNDI-lookup name of the
 reply destination can specify a user-defined destination, but it can also specify
 the default (Business Process Choreographer-defined) reply destination
 jms/BFMJMSReplyQueue. In both cases, the reply destination must lie on the
 BPC.<cellname>.Bus.
 // Look up the destination of the reply queue.
 Queue replyQueue = (Queue) initialcontext.lookup("jms/BFMJMSReplyQueue");

 // Create a message consumer.
 MessageConsumer consumer = session.createConsumer(replyQueue);
5. Send a message.
 // Start the connection.
 connection.start();

 // Create a message - see the task descriptions for examples - and send it.
 // This method is defined elsewhere ...
 String payload = createXMLDocumentForRequest();
 TextMessage requestMessage = session.createTextMessage(payload);

 // Set mandatory JMS header.
 // targetFunctionName is the operation name of JMS API
 // (for example, getProcessTemplate, sendMessage)
 requestMessage.setStringProperty("TargetFunctionName", targetFunctionName);

 // Set the reply queue; this is mandatory if the replyQueue
 // is not the default queue (as it is in this example).
 requestMessage.setJMSReplyTo(replyQueue);

 // Send the message.
 producer.send(requestMessage);

 // Get the message ID.
 String jmsMessageID = requestMessage.getJMSMessageID();

 session.commit();
6. Receive the reply.
 // Receive the reply message and analyse the reply.
 TextMessage replyMessage = (TextMessage) consumer.receive();

 // Get the payload.
 String payload = replyMessage.getText();

 session.commit();
7. Close the connection and free the resources.

```

```
// Final housekeeping; free the resources.
session.close();
connection.close();
```

**Note:** It is not necessary to close the connection after each transaction. Once a connection has been started, any number of request and response messages can be exchanged before the connection is closed. The example shows a simple case with a single call within a single business method.

## Structure of a Business Process Choreographer JMS message

The header and body of each JMS message must have a predefined structure.

A Java Message Service (JMS) message consists of:

- A message header for message identification and routing information.
- The body (payload) of the message that holds the content.

The Business Process Choreographer supports text message formats only.

### Message header

JMS allows clients to access a number of message header fields.

The following header fields can be set by a Business Process Choreographer JMS client:

#### JMSReplyTo

The destination to send a reply to the request. If this field is not specified in the request message, the reply is sent to the Export interface's default reply destination (an Export is a client interface rendering of a business process component). This destination can be obtained using `initialContext.lookup("jms/BFMJMSReplyQueue");`

#### TargetFunctionName

The name of the WSDL operation, for example, "queryProcessTemplates". This field must always be set. Note that the TargetFunctionName specifies the operation of the generic JMS message interface described here. This should not be confused with operations provided by concrete processes or tasks that can be invoked indirectly, for example, using the call or sendMessage operations.

A Business Process Choreographer client can also access the following header fields:

#### JMSMessageID

Uniquely identifies a message. Set by the JMS provider when the message is sent. If the client sets the JMSMessageID before sending the message, it is overwritten by the JMS provider. If the ID of the message is required for authentication purposes, the client can retrieve the JMSMessageID after sending the message.

#### JMSCorrelationID

Links messages. Do not set this field. A Business Process Choreographer reply message contains the JMSMessageID of the request message.

Each response message contains the following JMS header fields:

- **IsBusinessException**  
"False" for WSDL output messages, or "true" for WSDL fault messages.

ServiceRuntimeExceptions are not returned to asynchronous client applications. When a severe exception occurs during the processing of a JMS request message, it results in a runtime failure, causing the transaction that is processing this request message to roll back. The JMS request message is then delivered again. If the failure occurs early, during processing of the message as part of the SCA Export (for example, while deserializing the message), retries are attempted up to the maximum number of failed deliveries specified by the SCA Export's receive destination. After the maximum number of failed deliveries is reached, the request message is added to the system exception destination of the Business Process Choreographer bus. If, however, the failure occurs during actual processing of the request by the Business Flow Manager's SCA component, the failed request message is handled by the WebSphere Process Server's failed event management infrastructure, that is, it may end up in the failed event management database if retries do not resolve the exceptional situation.

### Message body

Operations exposed by business processes or human tasks must comply with the document/literal wrapper style. The JMS message body is a String containing an XML document that represents the document/literal wrapper element of the operation. The generic operations of the JMS message interface propagate the document-wrapper element to and from the operation that is implemented by the business process or human task.

The following example shows a simple valid request message body:

```
<bfm:queryProcessTemplates
 xmlns:bfm="http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0">
 <whereClause>PROCESS_TEMPLATE.STATE IN (1)</whereClause>
</bfm:queryProcessTemplates>
```

The following example shows a more complex, valid request message body. The client application has a sendMessage API operation for submitting a message to a specific process. The process input message is one of the API parameters; this message is the input message of a business operation exposed by a customer process. The process contains a receive activity that consumes the message.

The bfm:sendMessage element is the document wrapper element of the JMS API operation. It includes the cns:updateCustomer element, which is the document wrapper element for the operation that is implemented by the process. This process has, for example, a bpel:receive activity that references the cns:customerProcessPortType WSDL port type, and the updateCustomer WSDL operation.

```
<bfm:sendMessage
 xmlns:bfm="http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0">
 <processTemplateName>customerProcessTemplate</processTemplateName>
 <portType xmlns:cns="http://example.com/customerProcess">cns:customerProcessPortType</portType>
 <operation>updateCustomer</operation>
 <cns:updateCustomer xmlns:cns="http://example.com/customerProcess">
 <street>1600 Pennsylvania Avenue Northwest</street>
 <city>Washington, DC 20006</city>
 </cns:updateCustomer>
</bfm:sendMessage>
```

### Related tasks

“Checking the response message for business exceptions”  
JMS client applications must check the message header of all response messages for business exceptions.

## Copying artifacts for JMS client applications

A number of artifacts can be copied from the WebSphere Process Server environment to help in the creation of JMS client applications.

### About this task

These artifacts are mandatory only if you use the BOXMLSerializer to create the JMS message body. For the JMS API, these artifacts are:

- BFMIF.wsdl
- BFMIF.xsd
- BPCGen.xsd
- wsa.xsd

You must publish and export these files from the WebSphere Process Server environment to your development environment.

### Publishing the business process WSDL file for JMS applications

Use the administrative console to publish the WSDL file.

#### Procedure

1. Log on to the administrative console with a user ID with administrator rights.
2. Click **Applications** → **SCA modules**.

**Note:** You can also click **Applications** → **Application Types** → **WebSphere enterprise applications** to display a list of all available enterprise applications.

3. Choose the **BPEContainer** application from the list of SCA modules or applications.
4. Select **Publish WSDL files** from the list of **Additional properties**
5. Click the .zip file in the list.
6. On the File Download window that appears, click **Save**.
7. Browse to a local folder and click **Save**.

#### Results

The exported .zip file is named BPEContainer\_WSDLFiles.zip. The .zip file contains a WSDL file, and any XSD files that are referenced by the WSDL file.

## Checking the response message for business exceptions

JMS client applications must check the message header of all response messages for business exceptions.

### About this task

A JMS client application must first check the **IsBusinessException** property in the response message's header.

For example:

## Example

```
// receive response message
Message receivedMessage = ((JmsProxy) getToBeInvokedUponObject()).receiveMessage();
String strResponse = ((TextMessage) receivedMessage).getText();

if (receivedMessage.getStringProperty("IsBusinessException") {
 // strResponse is a bussiness fault
 // any api can end w/a processFaultMsg
 // the call api also w/a businessFaultMsg
}
else {
 // strResponse is the output message
}
```

### Related concepts

“Structure of a Business Process Choreographer JMS message” on page 444  
The header and body of each JMS message must have a predefined structure.

## Example: executing a long running process using the Business Process Choreographer JMS API

This example shows how to create a generic client application that uses the JMS API to work with long-running processes.

### Procedure

1. Set up the JMS environment, as described in “Accessing the JMS interface” on page 442.
2. Obtain a list of installed process definitions.
  - Send `queryProcessTemplates`.
  - This returns a list of `ProcessTemplate` objects.
3. Obtain a list of start activities (receive or pick with `createInstance="yes"`).
  - Send `getStartActivities`.
  - This returns a list of `InboundOperationTemplate` objects.
4. Create an input message. This is environment-specific, and might require the use of predeployed, process-specific artifacts.
5. Create a process instance.
  - Issue a `sendMessage`.

With the JMS API, you can also use the `call` operation for interacting with long-running, request-response operations provided by a business process. This operation returns the operation result or fault to the specified reply-to destination, even after a long period of time. Therefore, if you use the `call` operation, you do not need to use the `query` and `getOutputMessage` operations to obtain the process’ output or fault message.

6. Optional: Obtain output messages from the process instances by repeating the following steps:
  - a. Issue `query` to obtain the finished state of the process instance.
  - b. Issue `getOutputMessage`.
7. Optional: Work with additional operations exposed by the process:
  - a. Issue `getWaitingActivities` or `getActiveEventHandlers` to obtain a list of `InboundOperationTemplate` objects.
  - b. Create input messages.
  - c. Send messages with `sendMessage`.
8. Optional: Get and set custom properties that are defined on the process or contained activities with `getCustomProperties` and `setCustomProperties`.

9. Finish working with a process instance:
  - a. Send delete and terminate to finish working with the long-running process.

---

## Developing Web applications for business processes and human tasks, using JSF components

Business Process Choreographer provides several JavaServer Faces (JSF) components. You can extend and integrate these components to add business-process and human-task functionality to Web applications.

### About this task

You can use WebSphere Integration Developer to build your Web application. For applications that include human tasks, you can generate a JSF custom client. For more information on generating a JSF client, go to the information center for WebSphere Integration Developer.

You can also develop your Web client using the JSF components provided by Business Process Choreographer.

### Procedure

1. Create a dynamic project and change the Web Project Features properties to include the JSF base components.

For more information on creating a Web project, go to the information center for WebSphere Integration Developer.

2. Add the prerequisite Business Process Choreographer Explorer Java archive (JAR files).

Add the following files to the WEB-INF/lib directory of your project:

- bpcclientcore.jar
- bfmclientmodel.jar
- htmclientmodel.jar
- bpcjsfcomponents.jar

These files are in the *install\_root/ProcessChoreographer/client* directory.

3. Add the EJB references that you need to the Web application deployment descriptor, web.xml file.

```
<ejb-ref id="EjbRef_1">
 <ejb-ref-name>ejb/BusinessProcessHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
 <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
<ejb-ref id="EjbRef_2">
 <ejb-ref-name>ejb/HumanTaskManagerEJB</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.task.api.HumanTaskManagerHome</home>
 <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
<ejb-local-ref id="EjbLocalRef_1">
 <ejb-ref-name>ejb/LocalBusinessProcessHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
 <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
</ejb-local-ref>
<ejb-local-ref id="EjbLocalRef_2">
 <ejb-ref-name>ejb/LocalHumanTaskManagerEJB</ejb-ref-name>
```

```

 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
 <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>

```

4. Add the Business Process Choreographer Explorer JSF components to the JSF application.

a. Add the tag library references that you need for your applications to the JavaServer Pages (JSP) files. Typically, you need the JSF and HTML tag libraries, and the tag library required by the JSF components.

- <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
- <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
- <%@ taglib uri="http://com.ibm.bpe.jsf/taglib" prefix="bpe" %>

b. Add an <f:view> tag to the body of the JSP page, and an <h:form> tag to the <f:view> tag.

c. Add the JSF components to the JSP files.

Depending on your application, add the List component, the Details component, the CommandBar component, or the Message component to the JSP files. You can add multiple instances of each component.

d. Configure the managed beans in the JSF configuration file.

By default, the configuration file is the faces-config.xml file. This file is in the WEB-INF directory of the Web application.

Depending on the component that you add to your JSP file, you also need to add the references to the query and other wrapper objects to the JSF configuration file. To ensure correct error handling, you also need to define both an error bean and a navigation target for the error page in the JSF configuration file. Ensure that you use BPCError for the name of the error bean and error for the name of the navigation target of the error page.

```

<faces-config>
...
<managed-bean>
 <managed-bean-name>BPCError</managed-bean-name>
 <managed-bean-class>com.ibm.bpc.clientcore.util.ErrorBeanImpl
 </managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

...
<navigation-rule>
...
<navigation-case>
<description>
The general error page.
</description>
<from-outcome>error</from-outcome>
<to-view-id>/Error.jsp</to-view-id>
</navigation-case>
...
</navigation-rule>
</faces-config>

```

In error situations that trigger the error page, the exception is set on the error bean.

e. Implement the custom code that you need to support the JSF components.

5. Deploy the application.

If you are deploying the application in a network deployment environment, change the target resource Java Naming and Directory Interface (JNDI) names to values where the Business Flow Manager and Human Task Manager APIs can be found in your cell.

- If your business process containers are configured on another server in the same managed cell, the names have the following structure:

```
cell/nodes/nodename/servers/servername/com/ibm/bpe/api/BusinessManagerHome
cell/nodes/nodename/servers/servername/com/ibm/task/api/HumanTaskManagerHome
```

- If your business process containers are configured on a cluster in the same cell, the names have the following structure:

```
cell/clusters/clustername/com/ibm/bpe/api/BusinessFlowManagerHome
cell/clusters/clustername/com/ibm/task/api/HumanTaskManagerHome
```

Map the EJB references to the JNDI names or manually add the references to the `ibm-web-bnd.xmi` file.

The following table lists the reference bindings and their default mappings.

Table 70. Mapping of the reference bindings to JNDI names

Reference binding	JNDI name	Comments
ejb/BusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	Remote session bean
ejb/LocalBusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	Local session bean
ejb/HumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	Remote session bean
ejb/LocalHumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	Local session bean

## Results

Your deployed Web application contains the functionality provided by the Business Process Choreographer Explorer components.

## What to do next

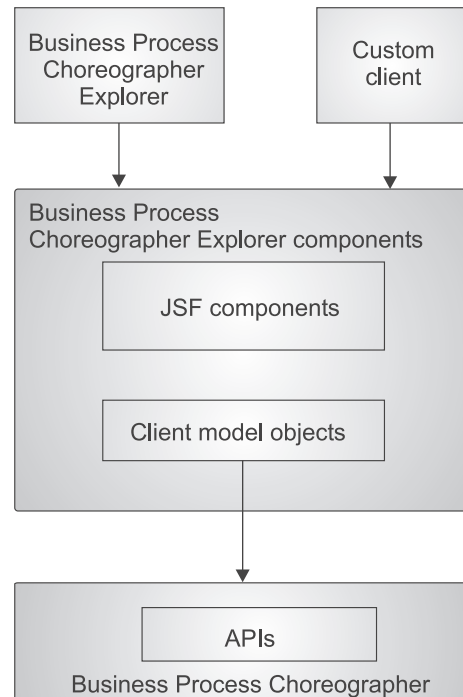
If you are using custom JSPs for the process and task messages, you must map the Web modules that are used to deploy the JSPs to the same servers that the custom JSF client is mapped to.

## Business Process Choreographer Explorer components

The Business Process Choreographer Explorer components are a set of configurable, reusable elements that are based on the JavaServer Faces (JSF) technology. You can imbed these elements in Web applications. The Web applications can then access installed business process and human task applications.

The components consist of a set of JSF components and a set of client model objects. The relationship of the components to Business Process Choreographer, Business Process Choreographer Explorer, and other custom clients is shown in the following figure.





## JSF components

The Business Process Choreographer Explorer components include the following JSF components. You embed these JSF components in your JavaServer Pages (JSP) files when you build Web applications for working with business processes and human tasks.

- List component
 

The List component displays a list of application objects in a table, for example, tasks, activities, process instances, process templates, work items, or escalations. This component has an associated list handler.
- Details component
 

The Details component displays the properties of tasks, work items, activities, process instances, and process templates. This component has an associated details handler.
- CommandBar component
 

The CommandBar component displays a bar with buttons. These buttons represent commands that operate on either the object in a details view or the selected objects in a list. These objects are provided by a list handler or a details handler.
- Message component
 

The Message component displays a message that can contain either a Service Data Object (SDO) or a simple type.

## Client model objects

The client model objects are used with the JSF components. The objects implement some of the interfaces of the underlying Business Process Choreographer API and wrap the original object. The client model objects provide national language support for labels and converters for some properties.

## Error handling in JSF components

The JavaServer Faces (JSF) components exploit a predefined managed bean, `BPCError`, for error handling. In error situations that trigger the error page, the exception is set on the error bean.

This bean implements the `com.ibm.bpc.clientcore.util.ErrorBean` interface. The error page is displayed in the following situations:

- If an error occurs during the execution of a query that is defined for a list handler, and the error is generated as a `ClientException` error by the `execute` method of a command
- If a `ClientException` error is generated by the `execute` method of a command and this error is not an `ErrorsInCommandException` error nor does it implement the `CommandBarMessage` interface
- If an error message is displayed in the component, and you follow the hyperlink for the message

A default implementation of the `com.ibm.bpc.clientcore.util.ErrorBeanImpl` interface is available.

The interface is defined as follows:

```
public interface ErrorBean {

 public void setException(Exception ex);

 /*
 * This setter method call allows a locale and
 * the exception to be passed. This allows the
 * getExceptionMessage methods to return localized Strings
 */
 public void setException(Exception ex, Locale locale);

 public Exception getException();
 public String getStack();
 public String getNestedExceptionMessage();
 public String getNestedExceptionStack();
 public String getRootExceptionMessage();
 public String getRootExceptionStack();

 /*
 * This method returns the exception message
 * concatenated recursively with the messages of all
 * the nested exceptions.
 */
 public String getAllExceptionMessages();

 /*
 * This method is returns the exception stack
 * concatenated recursively with the stacks of all
 * the nested exceptions.
 */
 public String getAllExceptionStacks();
}
```

## Related concepts

“Error handling in the List component” on page 457

When you use the List component to display lists in your JSF application, you can take advantage of the error handling functions provided by the `com.ibm.bpe.jsf.handler.BPCListHandler` class.

## Default converters and labels for client model objects

The client model objects implement the corresponding interfaces of the Business Process Choreographer API.

The List component and the Details component operate on any bean. You can display all of the properties of a bean. However, if you want to set the converters and labels that are used for the properties of a bean, you must use either the `column` tag for the List component, or the `property` tag for the Details component. Instead of setting the converters and labels, you can define default converter and labels for the properties by defining the following static methods. You can define the following static methods:

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
 getConverter(String property);
```

The following table shows the client model objects that implement the corresponding Business Flow Manager and Human Task Manager API classes and provide default labels and converter for their properties. This wrapping of the interfaces provides locale-sensitive labels and converters for a set of properties. The following table shows the mapping of the Business Process Choreographer interfaces to the corresponding client model objects.

Table 71. How Business Process Choreographer interfaces are mapped to client model objects

Business Process Choreographer interface	Client model object class
<code>com.ibm.bpe.api.ActivityInstanceData</code>	<code>com.ibm.bpe.clientmodel.bean.ActivityInstanceBean</code>
<code>com.ibm.bpe.api.ActivityServiceTemplateData</code>	<code>com.ibm.bpe.clientmodel.bean.ActivityServiceTemplateBean</code>
<code>com.ibm.bpe.api.ProcessInstanceData</code>	<code>com.ibm.bpe.clientmodel.bean.ProcessInstanceBean</code>
<code>com.ibm.bpe.api.ProcessTemplateData</code>	<code>com.ibm.bpe.clientmodel.bean.ProcessTemplateBean</code>
<code>com.ibm.task.api.Escalation</code>	<code>com.ibm.task.clientmodel.bean.EscalationBean</code>
<code>com.ibm.task.api.Task</code>	<code>com.ibm.task.clientmodel.bean.TaskInstanceBean</code>
<code>com.ibm.task.api.TaskTemplate</code>	<code>com.ibm.task.clientmodel.bean.TaskTemplateBean</code>

## Adding the List component to a JSF application

Use the Business Process Choreographer Explorer List component to display a list of client model objects, for example, business process instances or task instances.

### Procedure

1. Add the List component to the JavaServer Pages (JSP) file.

Add the `bpe:list` tag to the `h:form` tag. The `bpe:list` tag must include a `model` attribute. Add `bpe:column` tags to the `bpe:list` tag to add the properties of the objects that are to appear in each of the rows in the list.

The following example shows how to add a List component to display task instances.

```

<h:form>

 <bpe:list model="#{TaskPool}">
 <bpe:column name="name" action="taskInstanceDetails" />
 <bpe:column name="state" />
 <bpe:column name="kind" />
 <bpe:column name="owner" />
 <bpe:column name="originator" />
 </bpe:list>

</h:form>

```

The model attribute refers to a managed bean, TaskPool. The managed bean provides the list of Java objects over which the list iterates and then displays in individual rows.

## 2. Configure the managed bean referred to in the bpe:list tag.

For the List component, this managed bean must be an instance of the com.ibm.bpe.jsf.handler.BPCListHandler class.

The following example shows how to add the TaskPool managed bean to the configuration file.

```

<managed-bean>
 <managed-bean-name>TaskPool</managed-bean-name>
 <managed-bean-class>com.ibm.bpe.jsf.handler.BPCListHandler</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
 <managed-property>
 <property-name>query</property-name>
 <value>#{TaskPoolQuery}</value>
 </managed-property>
 <managed-property>
 <property-name>type</property-name>
 <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
 </managed-property>
</managed-bean>

<managed-bean>
 <managed-bean-name>TaskPoolQuery</managed-bean-name>
 <managed-bean-class>sample.TaskPoolQuery</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
 <managed-property>
 <property-name>type</property-name>
 <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
 </managed-property>
</managed-bean>

<managed-bean>
 <managed-bean-name>htmConnection</managed-bean-name>
 <managed-bean-class>com.ibm.task.clientmodel.HTMConnection</managed-bean-class>
 <managed-bean-scope>application</managed-bean-scope>
 <managed-property>
 <property-name>jndiName</property-name>
 <value>java:comp/env/ejb/LocalHumanTaskManagerEJB</value>
 </managed-property>
</managed-bean>

```

The example shows that TaskPool has two configurable properties: query and type. The value of the query property refers to another managed bean, TaskPoolQuery. The value of the type property specifies the bean class, the properties of which are shown in the columns of the displayed list. The associated query instance can also have a property type. If a property type is specified, it must be the same as the type specified for the list handler.

You can add any type of query logic to the JSF application as long as the result of the query can be represented as list of strongly-typed beans. For example,

the TaskPoolQuery is implemented using a list of com.ibm.task.clientmodel.bean.TaskInstanceBean objects.

3. Add the custom code for the managed bean that is referred to by the list handler.

The following example shows how to add custom code for the TaskPool managed bean.

```
public class TaskPoolQuery implements Query {

 public List execute throws ClientException {

 // Examine the faces-config file for a managed bean "htmConnection".
 //
 FacesContext ctx = FacesContext.getCurrentInstance();
 Application app = ctx.getApplication();
 ValueBinding htmVb = app.createValueBinding("#{htmConnection}");
 htmConnection = (HTMConnection) htmVb.getValue(ctx);
 HumanTaskManagerService taskService =
 htmConnection.getHumanTaskManagerService();

 // Then call the actual query method on the Human Task Manager service.
 //
 // Add the database columns for all of the properties you want to show
 // in your list to the select statement
 //
 QueryResultSet queryResult = taskService.query(
 "DISTINCT TASK.TKIID, TASK.NAME, TASK.KIND, TASK.STATE, TASK.TYPE,"
 + "TASK.STARTER, TASK.OWNER, TASK.STARTED, TASK.ACTIVATED, TASK.DUE,"
 + "TASK.EXPIRES, TASK.PRIORITY",
 "TASK.KIND IN(101,102,105) AND TASK.STATE IN(2)
 AND WORK_ITEM.REASON IN (1)",
 (String)null,
 (Integer)null,
 (TimeZone)null);
 List applicationObjects = transformToTaskList (queryResult);
 return applicationObjects ;
 }

 private List transformToTaskList(QueryResultSet result) {

 ArrayList array = null;
 int entries = result.size();
 array = new ArrayList(entries);

 // Transforms each row in the QueryResultSet to a task instance beans.
 for (int i = 0; i < entries; i++) {
 result.next();
 array.add(new TaskInstanceBean(result, connection));
 }
 return array ;
 }
}
```

The TaskPoolQuery bean queries the properties of the Java objects. This bean must implement the com.ibm.bpc.clientcore.Query interface. When the list handler refreshes its contents, it calls the execute method of the query. The call returns a list of Java objects. The getType method must return the class name of the returned Java objects.

## Results

Your JSF application now contains a JavaServer page that displays the properties of the requested list of objects, for example, the state, kind, owner, and originator of the task instances that are available to you.

## How lists are processed

Every instance of the List component is associated with an instance of the `com.ibm.bpe.jsf.handler.BPCListHandler` class.

This list handler tracks the selected items in the associated list and it provides a notification mechanism to associate the list entries with the details pages for the different kinds of items. The list handler is bound to the List component through the **model** attribute of the `bpe:list` tag.

The notification mechanism of the list handler is implemented using the `com.ibm.bpe.jsf.handler.ItemListener` interface. You can register implementations of this interface in the configuration file of your JavaServer Faces (JSF) application.

The notification is triggered when a link in the list is clicked. Links are rendered for all of the columns for which the **action** attribute is set. The value of the **action** attribute is either a JSF navigation target, or a JSF action method that returns a JSF navigation target.

The `BPCListHandler` class also provides a `refreshList` method. You can use this method in JSF method bindings to implement a user interface control for running the query again.

## Query implementations

You can use the list handler to display all kinds of objects and their properties. The content of the list that is displayed depends on the list of objects that is returned by the implementation of the `com.ibm.bpc.clientcore.Query` interface that is configured for the list handler. You can set the query either programmatically using the `setQuery` method of the `BPCListHandler` class, or you can configure it in the JSF configuration files of the application.

You can run queries not only against the Business Process Choreographer APIs, but also against any other source of information that is accessible from your application, for example, a content management system or a database. The only requirement is that the result of the query is returned as a `java.util.List` of objects by the `execute` method.

The type of the objects returned must guarantee that the appropriate getter methods are available for all of the properties that are displayed in the columns of the list for which the query is defined. To ensure that the type of the object that is returned fits the list definitions, you can set the value of the `type` property on the `BPCListHandler` instance that is defined in the faces configuration file to the fully qualified class name of the returned objects. You can return this name in the `getType` call of the query implementation. At runtime, the list handler checks that the object types conform to the definitions.

To map error messages to specific entries in a list, the objects returned by the query must implement a method with the signature `public Object getID()`.

## Default converters and labels

The items returned by a query must be beans and their class must match the class specified as the `type` in the definition of the `BPCListHandler` class or `com.ibm.bpc.clientcore.Query` interface. In addition, the List component checks whether the item class or a superclass implements the following methods:

```

static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
 getConverter(String property);

```

If these methods are defined for the beans, the List component uses the label as the default label for the list and the SimpleConverter as the default converter for the property. You can overwrite these settings with the **label** and **converterID** attributes of the `bpe:list` tag. For more information, see the Javadoc for the SimpleConverter interface and the ColumnTag class.

## User-specific time zone information

The JavaServer Faces (JSF) components provide a utility for handling user-specific time zone information in the List component.

The BPCListHandler class uses the `com.ibm.bpc.clientcore.util.User` interface to get information about the time zone and locale of each user. The List component expects the implementation of the interface to be configured with **user** as the managed-bean name in your JavaServer Faces (JSF) configuration file. If this entry is missing from the configuration file, the time zone in which WebSphere Process Server is running is returned.

The `com.ibm.bpc.clientcore.util.User` interface is defined as follows:

```

public interface User {

 /**
 * The locale used by the client of the user.
 * @return Locale.
 */
 public Locale getLocale();

 /**
 * The time zone used by the client of the user.
 * @return TimeZone.
 */
 public TimeZone getTimeZone();

 /**
 * The name of the user.
 * @return name of the user.
 */
 public String getName();
}

```

## Error handling in the List component

When you use the List component to display lists in your JSF application, you can take advantage of the error handling functions provided by the `com.ibm.bpc.jsf.handler.BPCListHandler` class.

### Errors that occur when queries are run or commands are run

If an error occurs during the execution of a query, the BPCListHandler class distinguishes between errors that were caused by insufficient access rights and other exceptions. To catch errors due to insufficient access rights, the **rootCause** parameter of the ClientException that is thrown by the execute method of the query must be a `com.ibm.bpc.api.EngineNotAuthorizedException` or a `com.ibm.task.api.NotAuthorizedException` exception. The List component displays the error message instead of the result of the query.

If the error is not caused by insufficient access rights, the BPCListHandler class passes the exception object to the implementation of the

com.ibm.bpc.clientcore.util.ErrorBean interface that is defined by the BPCError key in your JSF application configuration file. When the exception is set, the error navigation target is called.

## Errors that occur when working with items that are displayed in a list

The BPCListHandler class implements the com.ibm.bpe.jsf.handler.ErrorHandler interface. You can provide information about these errors with the map parameter of type java.util.Map in the setErrors method. This map contains identifiers as keys and the exceptions as values. The identifiers must be the values returned by the getID method of the object that caused the error. If the map is set and any of the IDs match any of the items displayed in the list, the list handler automatically adds a column containing the error message to the list.

To avoid outdated error messages in the list, reset the errors map. In the following situations, the map is reset automatically:

- The refreshList method BPCListHandler class is called.
- A new query is set on the BPCListHandler class.
- The CommandBar component is used to trigger actions on items of the list. The CommandBar component uses this mechanism as one of the methods for error handling.

### Related concepts

“Error handling in JSF components” on page 452

The JavaServer Faces (JSF) components exploit a predefined managed bean, BPCError, for error handling. In error situations that trigger the error page, the exception is set on the error bean.

## List component: Tag definitions

The Business Process Choreographer Explorer List component displays a list of objects in a table, for example, tasks, activities, process instances, process templates, work items, and escalations.

The List component consists of the JSF component tags: bpe:list and bpe:column. The bpe:column tag is a subelement of the bpe:list tag.

## Component class

com.ibm.bpe.jsf.component.ListComponent

### Example syntax

```
<bpe:list model="#{ProcessTemplateList}">
 rows="20"
 styleClass="list"
 headerStyleClass="listHeader"
 rowClasses="normal">

 <bpe:column name="name" action="processTemplateDetails"/>
 <bpe:column name="validFromTime"/>
 <bpe:column name="executionMode" label="Execution mode"/>
 <bpe:column name="state" converterID="my.state.converter"/>
 <bpe:column name="autoDelete"/>
 <bpe:column name="description"/>

</bpe:list>
```



## Tag attributes

The body of the `bpe:list` tag can contain only `bpe:column` tags. When the table is rendered, the List component iterates over the list of application objects and renders all of the columns for each of the objects.

Table 72. *bpe:list* attributes

Attribute	Required	Description
<code>buttonStyleClass</code>	no	The cascading style sheet (CSS) style class for rendering the buttons in the footer area.
<code>cellStyleClass</code>	no	The CSS style class for rendering individual table cells.
<code>checkbox</code>	no	Determines whether the check box for selecting multiple items is rendered. The attribute has a value of either <code>true</code> or <code>false</code> . If the value is set to <code>true</code> , the check box column is rendered.
<code>headerStyleClass</code>	no	The CSS style class for rendering the table header.
<code>model</code>	yes	A value binding for a managed bean of the <code>com.ibm.bpe.jsf.handler.BPCListHandler</code> class.
<code>rows</code>	no	The number of rows that are shown on a page. If the number of items exceeds the number of rows, paging buttons are displayed at the end of the table. Value expressions are not supported for this attribute.
<code>rowClasses</code>	no	The CSS style class for rendering the rows in the table.
<code>selectAll</code>	no	If this attribute is set to <code>true</code> , all of the items in the list are selected by default.
<code>styleClass</code>	no	The CSS style class for rendering the overall table containing titles, rows, and paging buttons.

Table 73. *bpe:column* attributes

Attribute	Required	Description
<code>action</code>	no	If this attribute is specified, a link is rendered in the column. Either a JavaServer Faces action method or the Faces navigation target is triggered when this link is clicked. A JavaServer Faces action method has the following signature: <code>String method()</code> .
<code>converterID</code>	no	The Faces converter ID that is used for converting the property value. If this attribute is not set, any Faces converter ID that is provided by the model for this property is used.

Table 73. *bpe:column* attributes (continued)

Attribute	Required	Description
label	no	A literal or value binding expression that is used as a label for the header of the column or the cell of the table header row. If this attribute is not set, any label that is provided by the model for this property is used.
name	yes	The name of the property that is displayed in this column.

## Adding the Details component to a JSF application

Use the Business Process Choreographer Explorer Details component to display the properties of tasks, work items, activities, process instances, and process templates.

### Procedure

1. Add the Details component to the JavaServer Pages (JSP) file.

Add the `bpe:details` tag to the `<h:form>` tag. The `bpe:details` tag must contain a **model** attribute. You can add properties to the Details component with the `bpe:property` tag.

The following example shows how to add a Details component to display some of the properties for a task instance.

```
<h:form>

 <bpe:details model="#{TaskInstanceDetails}">
 <bpe:property name="displayName" />
 <bpe:property name="owner" />
 <bpe:property name="kind" />
 <bpe:property name="state" />
 <bpe:property name="escalated" />
 <bpe:property name="suspended" />
 <bpe:property name="originator" />
 <bpe:property name="activationTime" />
 <bpe:property name="expirationTime" />
 </bpe:details>

</h:form>
```

The **model** attribute refers to a managed bean, `TaskInstanceDetails`. The bean provides the properties of the Java object.

2. Configure the managed bean referred to in the `bpe:details` tag.

For the Details component, this managed bean must be an instance of the `com.ibm.bpe.jsf.handler.BPCDetailsHandler` class. This handler class wraps a Java object and exposes its public properties to the details component.

The following example shows how to add the `TaskInstanceDetails` managed bean to the configuration file.

```
<managed-bean>
 <managed-bean-name>TaskInstanceDetails</managed-bean-name>
 <managed-bean-class>com.ibm.bpe.jsf.handler.BPCDetailsHandler</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
 <managed-property>
 <property-name>type</property-name>
 <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
 </managed-property>
</managed-bean>
```

The example shows that the `TaskInstanceDetails` bean has a configurable `type` property. The value of the `type` property specifies the bean class

(`com.ibm.task.clientmodel.bean.TaskInstanceBean`), the properties of which are shown in the rows of the displayed details. The bean class can be any JavaBeans class. If the bean provides default converter and property labels, the converter and the label are used for the rendering in the same way as for the List component.

## Results

Your JSF application now contains a JavaServer page that displays the details of the specified object, for example, the details of a task instance.

### Details component: Tag definitions

The Business Process Choreographer Explorer Details component displays the properties of tasks, work items, activities, process instances, and process templates.

The Details component consists of the JSF component tags: `bpe:details` and `bpe:property`. The `bpe:property` tag is a subelement of the `bpe:details` tag.

### Component class

`com.ibm.bpe.jsf.component.DetailsComponent`

### Example syntax

```
<bpe:details model="#{MyActivityDetails}">
 <bpe:property name="name"/>
 <bpe:property name="owner"/>
 <bpe:property name="activated"/>
</bpe:details>

<bpe:details model="#{MyActivityDetails}" style="style" styleClass="cssStyle">
 style="style"
 styleClass="cssStyle"
</bpe:details>
```

### Tag attributes

Use `bpe:property` tags to specify both the subset of attributes that are shown and the order in which these attributes are shown. If the details tag does not contain any attribute tags, it renders all of the available attributes of the model object.

Table 74. *bpe:details* attributes

Attribute	Required	Description
<code>columnClasses</code>	no	A list of cascading style sheet style (CSS) style classes, separated by commas, for rendering columns.
<code>id</code>	no	The JavaServer Faces ID of the component.
<code>model</code>	yes	A value binding for a managed bean of the <code>com.ibm.bpe.jsf.handler.BPCDetailsHandler</code> class.
<code>rowClasses</code>	no	A list of CSS style classes, separated by commas, for rendering rows.
<code>styleClass</code>	no	The CSS class that is used for rendering the HTML element.

Table 75. *bpe:property* attributes

Attribute	Required	Description
converterID	no	The ID used to register the converter in the JavaServer Faces (JSF) configuration file.
label	no	The label for the property. If this attribute is not set, a default label is provided by the client model class.
name	yes	The name of the property to be displayed. This name must correspond to a named property as defined in the corresponding client model class.

## Adding the CommandBar component to a JSF application

Use the Business Process Choreographer Explorer CommandBar component to display a bar with buttons. These buttons represent commands that operate on the details view of an object or the selected objects in a list.

### About this task

When the user clicks a button in the user interface, the corresponding command is run on the selected objects. You can add and extend the CommandBar component in your JavaServer Faces (JSF) application.

### Procedure

1. Add the CommandBar component to the JavaServer Pages (JSP) file.

Add the `bpe:commandbar` tag to the `<h:form>` tag. The `bpe:commandbar` tag must contain a model attribute.

The following example shows how to add a CommandBar component that provides refresh and claim commands for a task instance list.

```
<h:form>

 <bpe:commandbar model="#{TaskInstanceList}">
 <bpe:command commandID="Refresh" >
 action="#{TaskInstanceList.refreshList}"
 label="Refresh"/>

 <bpe:command commandID="MyClaimCommand" >
 label="Claim" >
 commandClass="<customcode>"/>
 </bpe:commandbar>

</h:form>
```

The **model** attribute refers to a managed bean. This bean must implement the `ItemProvider` interface and provide the selected Java objects. The CommandBar component is usually used with either the List component or the Details component in the same JSP file. Generally, the model that is specified in the tag is the same as the model that is specified in the List component or Details component on the same page. So for the List component, for example, the command acts on the selected items in the list.

In this example, the **model** attribute refers to the `TaskInstanceList` managed bean. This bean provides the selected objects in the task instance list. The bean must implement the `ItemProvider` interface. This interface is implemented by the `BPCListHandler` class and the `BPCDetailsHandler` class.

- Optional: Configure the managed bean that is referred to in the `bpe:commandbar` tag.

If the `CommandBar` **model** attribute refers to a managed bean that is already configured, for example, for a list or details handler, no further configuration is required. If you use neither the `BPCListHandler` class nor the `BPCDetailsHandler` class for the model, you must refer to another object that has a class that implements the `ItemProvider` interface.

- Add the code that implements the custom commands to the JSF application.

The following code snippet shows how to write a command class that implements the `Command` interface. This command class (`MyClaimCommand`) is referred to by the `bpe:command` tag in the JSP file.

```
public class MyClaimCommand implements Command {

 public String execute(List selectedObjects) throws ClientException {
 if(selectedObjects != null && selectedObjects.size() > 0) {
 try {
 // Determine HumanTaskManagerService from an HTMConnection bean.
 // Configure the bean in the faces-config.xml for easy access
 // in the JSF application.
 FacesContext ctx = FacesContext.getCurrentInstance();
 ValueBinding vb =
 ctx.getApplication().createValueBinding("{htmConnection}");
 HTMConnection htmConnection = (HTMConnection) htmVB.getValue(ctx);
 HumanTaskManagerService htm =
 htmConnection.getHumanTaskManagerService();

 Iterator iter = selectedObjects.iterator() ;
 while(iter.hasNext()) {
 try {
 TaskInstanceBean task = (TaskInstanceBean) iter.next() ;
 TKIID tiid = task.getID() ;

 htm.claim(tiid) ;
 task.setState(new Integer(TaskInstanceBean.STATE_CLAIMED)) ;

 }
 catch(Exception e) {
 ; // Error while iterating or claiming task instance.
 // Ignore for better understanding of the sample.
 }
 }
 }
 catch(Exception e) {
 ; // Configuration or communication error.
 // Ignore for better understanding of the sample
 }
 }
 return null;
 }

 // Default implementations
 public boolean isMultiSelectEnabled() { return false; }
 public boolean[] isApplicable(List itemsOnList) {return null; }
 public void setContext(Object targetModel) {}; // Not used here }
}
```

The command is processed in the following way:

- A command is invoked when a user clicks the corresponding button in the command bar. The `CommandBar` component retrieves the selected items from the item provider that is specified in the **model** attribute and passes the list of selected objects to the `execute` method of the `commandClass` instance.

- b. Optional: The **commandClass** attribute refers to a custom command implementation that implements the Command interface. This means that the command must implement the public String execute(List selectedObjects) throws ClientException method. The command returns a result that is used to determine the next navigation rule for the JSF application.
- c. Optional: After the command completes, the CommandBar component evaluates the **action** attribute. The **action** attribute can be a static string or a method binding to a JSF action method with the public String Method() signature. Use the **action** attribute to override the outcome of a command class or to explicitly specify an outcome for the navigation rules. The **action** attribute is not processed if the command generates an exception other than an ErrorsInCommandException exception.
- d. If the **commandClass** attribute does not have a command class specified, the action is immediately called. For example, for the refresh command in the example, the JSF value expression #{TaskInstanceList.refreshList} is called instead of a command.

## Results

Your JSF application now contains a JavaServer page that implements a customized command bar.

### How commands are processed

Use the CommandBar component to add action buttons to your application. The component creates the buttons for the actions in the user interface and handles the events that are created when a button is clicked.

These buttons trigger functions that act on the objects that are returned by a com.ibm.bpe.jsf.handler.ItemProvider interface, such as the BPCListHandler class, or the BPCDetailsHandler class. The CommandBar component uses the item provider that is defined by the value of the **model** attribute in the bpe:commandbar tag.

When a button in the command-bar section of the application's user interface is clicked, the associated event is handled by the CommandBar component in the following way.

1. The CommandBar component identifies the implementation of the com.ibm.bpc.clientcore.Command interface that is specified for the button that generated the event.
2. If the model associated with the CommandBar component implements the com.ibm.bpe.jsf.handler.ErrorHandler interface, the clearErrorMap method is invoked to remove error messages from previous events.
3. The getSelectedItems method of the ItemProvider interface is called. The list of items that is returned is passed to the execute method of the command, and the command is invoked.
4. The CommandBar component determines the JavaServer Faces (JSF) navigation target. If an **action** attribute is not specified in the bpe:commandbar tag, the return value of the execute method specifies the navigation target. If the **action** attribute is set to a JSF method binding, the string returned by the method is interpreted as the navigation target. The **action** attribute can also specify an explicit navigation target.

## CommandBar component: Tag definitions

The Business Process Choreographer Explorer CommandBar component displays a bar with buttons. These buttons operate on the object in a details view or the selected objects in a list.

The CommandBar component consists of the JSF component tags: `bpe:commandbar` and `bpe:command`. The `bpe:command` tag is a subelement of the `bpe:commandbar` tag.

### Component class

`com.ibm.bpe.jsf.component.CommandBarComponent`

### Example syntax

```
<bpe:commandbar model="#{TaskInstanceList}">

 <bpe:command
 commandID="Work on"
 label="Work on..."
 commandClass="com.ibm.bpc.explorer.command.WorkOnTaskCommand"
 context="#{TaskInstanceDetailsBean}" />

 <bpe:command
 commandID="Cancel"
 label="Cancel"
 commandClass="com.ibm.task.clientmodel.command.CancelClaimTaskCommand"
 context="#{TaskInstanceList}" />

</bpe:commandbar>
```

### Tag attributes

Table 76. `bpe:commandbar` attributes

Attribute	Required	Description
<code>buttonStyleClass</code>	no	The cascading style sheet (CSS) style class that is used for rendering the buttons in the command bar.
<code>id</code>	no	The JavaServer Faces ID of the component.
<code>model</code>	yes	A value binding expression to a managed bean that implements the <code>ItemProvider</code> interface. This managed bean is usually the <code>com.ibm.bpe.jsf.handler.BPCListHandler</code> class or the <code>com.ibm.bpe.jsf.handler.BPCDetailsHandler</code> class that is used by the List component or Details component in the same JavaServer Pages (JSP) file as the CommandBar component.
<code>styleClass</code>	no	The CSS style class that is used for rendering the command bar.

Table 77. *bpe:command* attributes

Attribute	Required	Description
action	no	A JavaServer Faces action method or the Faces navigation target that is to be triggered by the command button. The navigation target that is returned by the action overwrites all other navigation rules. The action is called when either an exception is not thrown or an <code>ErrorsInCommandException</code> exception is thrown by the command.
commandClass	no	The name of the command class. An instance of the class is created by the <code>CommandBar</code> component and run if the command button is selected.
commandID	yes	The ID of the command.
context	no	An object that provides context for commands that are specified using the <code>commandClass</code> attribute. The context object is retrieved when the command bar is first accessed.
immediate	no	Specifies when the command is triggered. If the value of this attribute is <code>true</code> , the command is triggered before the input of the page is processed. The default is <code>false</code> .
label	yes	The label of the button that is rendered in the command bar.
rendered	no	Determines whether a button is rendered. The value of the attribute can be either a Boolean value or a value expression.
styleClass	no	The CSS style class that is used for rendering the button. This style overrides the button style defined for the command bar.

## Adding the Message component to a JSF application

Use the Business Process Choreographer Explorer Message component to render data objects and primitive types in a JavaServer Faces (JSF) application.

### About this task

If the message type is a primitive type, a label and an input field are rendered. If the message type is a data object, the component traverses the object and renders the elements within the object.

### Procedure

1. Add the Message component to the JavaServer Pages (JSP) file.

Add the `bpe:form` tag to the `<h:form>` tag. The `bpe:form` tag must include a `model` attribute.

The following example shows how to add a Message component.

```
<h:form>

 <h:outputText value="Input Message" />
 <bpe:form model="{MyHandler.inputMessage}" readOnly="true" />

</h:form>
```



```

 <h:outputText value="Output Message" />
 <bpe:form model="#{MyHandler.outputMessage}" />

</h:form>

```

The **model** attribute of the Message component refers to a `com.ibm.bpc.clientcore.MessageWrapper` object. This wrapper object wraps either a Service Data Object (SDO) object or a Java primitive type, for example, `int` or `boolean`. In the example, the message is provided by a property of the `MyHandler` managed bean.

2. Configure the managed bean referred to in the `bpe:form` tag.

The following example shows how to add the `MyHandler` managed bean to the configuration file.

```

<managed-bean>
 <managed-bean-name>MyHandler</managed-bean-name>
 <managed-bean-class>com.ibm.bpe.sample.jsf.MyHandler</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>

 <managed-property>
 <property-name>type</property-name>
 <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
 </managed-property>
</managed-bean>

```

3. Add the custom code to the JSF application.

The following example shows how to implement input and output messages.

```

public class MyHandler implements ItemListener {

 private TaskInstanceBean taskBean;
 private MessageWrapper inputMessage, outputMessage

 /* Listener method, e.g. when a task instance was selected in a list handler.
 * Ensure that the handler is registered in the faces-config.xml or manually.
 */
 public void itemChanged(Object item) {
 if(item instanceof TaskInstanceBean) {
 taskBean = (TaskInstanceBean) item ;
 }
 }

 /* Get the input message wrapper
 */
 public MessageWrapper getInputMessage() {
 try{
 inputMessage = taskBean.getInputMessageWrapper() ;
 }
 catch(Exception e) {
 ; //...ignore errors for simplicity
 }
 return inputMessage;
 }

 /* Get the output message wrapper
 */
 public MessageWrapper getOutputMessage() {
 // Retrieve the message from the bean. If there is no message, create
 // one if the task has been claimed by the user. Ensure that only
 // potential owners or owners can manipulate the output message.
 try{
 outputMessage = taskBean.getOutputMessageWrapper();
 if(outputMessage == null
 && taskBean.getState() == TaskInstanceBean.STATE_CLAIMED) {
 HumanTaskManagerService htm = getHumanTaskManagerService();
 outputMessage = new MessageWrapperImpl();
 }
 }
 }
}

```

```

 outputMessage.setMessage(
 htm.createOutputMessage(taskBean.getID()).getObject()
);
 }
}
catch(Exception e) {
 ; //...ignore errors for simplicity
}
return outputMessage
}
}
}

```

The MyHandler managed bean implements the `com.ibm.jsf.handler.ItemListener` interface so that it can register itself as an item listener to list handlers. When the user clicks an item in the list, the MyHandler bean is notified in its `itemChanged( Object item )` method about the selected item. The handler checks the item type and then stores a reference to the associated `TaskInstanceBean` object. To use this interface, add an entry to the `itemListener` list in the appropriate list handler in the `faces-config.xml` file.

The MyHandler bean provides the `getInputMessage` and `getOutputMessage` methods. Both of these methods return a `MessageWrapper` object. The methods delegate the calls to the referenced task instance bean. If the task instance bean returns null, for example, because a message is not set, the handler creates and stores a new, empty message. The Message component displays the messages provided by the MyHandler bean.

## Results

Your JSF application now contains a JavaServer page that can render data objects and primitive types.

### Message component: Tag definitions

The Business Process Choreographer Explorer Message component renders `commonj.sdo.DataObject` objects and primitive types, such as integers and strings, in a JavaServer Faces (JSF) application.

The Message component consists of the JSF component tag: `bpe:form`.

### Component class

`com.ibm.bpe.jsf.component.MessageComponent`

### Example syntax

```

<bpe:form model="#{TaskInstanceDetailsBean.inputMessageWrapper}"
 simplification="true" readOnly="true"
 styleClass4table="messageData"
 styleClass4output="messageDataOutput">
</bpe:form>

```

### Tag attributes

Table 78. `bpe:form` attributes

Attribute	Required	Description
id	no	The JavaServer Faces ID of the component.

Table 78. *bpe:form* attributes (continued)

Attribute	Required	Description
model	yes	A value binding expression that refers to either a <code>commonj.sdo.DataObject</code> object or a <code>com.ibm.bpc.clientcore.MessageWrapper</code> object.
readOnly	no	If this attribute is set to true, a read-only form is rendered. By default, this attribute is set to false.
simplification	no	If this attribute is set to true, properties that contain simple types and have a cardinality of zero or one are shown. By default, this attribute is set to true.
style4validinput	no	The cascading style sheet (CSS) style for rendering input that is valid.
style4invalidinput	no	The CSS style for rendering input that is not valid.
styleClass4invalidInput	no	The CSS style class name for rendering input that is not valid.
styleClass4output	no	The CSS style class name for rendering the output elements.
styleClass4table	no	The class name of the CSS table style for rendering the tables rendered by the message component.
styleClass4validInput	no	The CSS style class name for rendering input that is valid.

---

## Developing JSP pages for task and process messages

The Business Process Choreographer Explorer interface provides default input and output forms for displaying and entering business data. You can use JSP pages to provide customized input and output forms.

### About this task

To include user-defined JavaServer Pages (JSP) pages in the Web client, you must specify them when you model a human task in WebSphere Integration Developer. For example, you can provide JSP pages for a specific task and its input and output messages, and for a specific user role or all user roles. At runtime, the user-defined JSP pages are included in the user interface to display output data and collect input data.

The customized forms are not self-contained Web pages; they are HTML fragments that Business Process Choreographer Explorer imbeds in an HTML form, for example, fragments for all of the labels and input fields of a message.

When a button is clicked on the page that contains the customized forms, the input is submitted and validated in Business Process Choreographer Explorer. The validation is based on the type of the properties provided and the locale used in the browser. If the input cannot be validated, the same page is shown again and information about the validation errors is provided in the `messageValidationErrors`

request attribute. The information is provided as a map that maps the XML Path Expression (XPath) of the properties that are not valid to the validation exceptions that occurred.

To add customized forms to Business Process Choreographer Explorer, complete the following steps using WebSphere Integration Developer.

## Procedure

1. Create the customized forms.

The user-defined JSP pages for the input and output forms used in the Web interface need access to the message data. Use Java snippets in a JSP or the JSP execution language to access the message data. Data in the forms is available through the request context.

2. Assign the JSP pages to a task.

Open the human task in the human task editor. In the client settings, specify the location of the user-defined JSP pages and the role to which the customized form applies, for example, administrator. The client settings for Business Process Choreographer Explorer are stored in the task template. At runtime these settings are retrieved with the task template.

3. Package the user-defined JSP pages in a Web archive (WAR file).

You can either include the WAR file in the enterprise archive with the module that contains the tasks or deploy the WAR file separately. If the JSPs are deployed separately, make the JSPs available on the server where the Business Process Choreographer Explorer or the custom client is deployed.

If you are using custom JSPs for the process and task messages, you must map the Web modules that are used to deploy the JSPs to the same servers that the custom JSF client is mapped to.

## Results

The customized forms are rendered in Business Process Choreographer Explorer at runtime.

## User-defined JSP fragments

The user-defined JavaServer Pages (JSP) fragments are imbedded in an HTML form tag. At runtime, Business Process Choreographer Explorer includes these fragments in the rendered page.

The user-defined JSP fragment for the input message is imbedded before the JSP fragment for the output message.

```
<html...>
...
<form...>
 Input JSP (display task input message)

 Output JSP (display task output message)

</form>
...
</html>
```

Because the user-defined JSP fragments are embedded in an HTML form tag, you can add input elements. The name of the input element must match the XML Path Language (XPath) expression of the data element. It is important to prefix the name of the input element with the provided prefix value:

```

<input id="address"
 type="text"
 name="{prefix}/selectPromotionalGiftResponse/address"
 value="{messageMap['/selectPromotionalGiftResponse/address']}"
 size="60"
 align="left" />

```

The prefix value is provided as a request attribute. The attribute ensures that the input name is unique in the enclosing form. The prefix is generated by Business Process Choreographer Explorer and it should not be changed:

```
String prefix = (String)request.getAttribute("prefix");
```

The prefix element is set only if the message can be edited in the given context. Output data can be displayed in different ways depending on the state of the human task. For example, if the task is in the claimed state, the output data can be modified. However, if the task is in the finished state, the data can be displayed only. In your JSP fragment, you can test whether the prefix element exists and render the message accordingly. The following JSTL statement shows how you might test whether the prefix element is set.

```

...
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...
<c:choose>
 <c:when test="{not empty prefix}">
 <!--Read/write mode-->
 </c:when>
 <c:otherwise>
 <!--Read-only mode-->
 </c:otherwise>
</c:choose>

```

---

## Creating plug-ins to customize human task functionality

Business Process Choreographer provides an event handling infrastructure for events that occur during the processing of human tasks. Plug-in points are also provided so that you can adapt the functionality to your needs. You can use the service provider interfaces (SPIs) to create customized plug-ins for handling events and the post processing of people query results.

### About this task

You can create plug-ins for human task API events and escalation notification events. You can also create a plug-in that processes the results that are returned from people resolution. For example, at peak periods you might want to add users to the result list to help balance the workload.

Before you can use the plug-ins, you must install and register them. You can register the plug-in to post process people query results with the TaskContainer application. The plug-in is then available for all tasks.

## Creating API event handlers for Business Process Choreographer

An API event occurs when an API method manipulates a human task. Use the API event handler plug-in service provider interface (SPI) to create plug-ins to handle the task events sent by the API or the internal events that have equivalent API events.

## About this task

Complete the following steps to create an API event handler.

### Procedure

1. Write a class that implements the `APIEventHandlerPlugin5` interface or extends the `APIEventHandler` implementation class. This class can invoke the methods of other classes.
  - If you use the `APIEventHandlerPlugin5` interface, you must implement all of the methods of the `APIEventHandlerPlugin5` interface and the `APIEventHandlerPlugin` interface.
  - If you extend the `APIEventHandler` implementation class, overwrite the methods that you need.

This class runs in the context of a Java Platform, Enterprise Edition (Java EE) Enterprise application. Ensure that this class and its helper classes follow the EJB specification.

**Note:** If you want to call the `HumanTaskManagerService` interface from this class, do not call a method that updates the task that produced the event. This action might result in inconsistent task data in the database.

2. Assemble the plug-in class and its helper classes into a JAR file.

You can make the JAR file available in one of the following ways:

- As a utility JAR file in the application EAR file.
- As a shared library that is installed with the application EAR file.
- As a shared library that is installed with the TaskContainer application. In this case, the plug-in is available for all tasks.

3. Create a service provider configuration file for the plug-in in the `META-INF/services/` directory of your JAR file.

The configuration file provides the mechanism for identifying and loading the plug-in. This file conforms to the Java EE service provider interface specification.

- a. Create a file with the name `com.ibm.task.spi.plug-in_nameAPIEventHandlerPlugin`, where *plug-in\_name* is the name of the plug-in.

For example, if your plug-in is called `Customer` and it implements the `com.ibm.task.spi.APIEventHandlerPlugin5` interface, the name of the configuration file is `com.ibm.task.spi.CustomerAPIEventHandlerPlugin`.

- b. In the first line of the file that is neither a comment line (a line that starts with a number sign (#)) nor a blank line, specify the fully qualified name of the plug-in class that you created in step 1.

For example, if your plug-in class is called `MyAPIEventHandler` and it is in the `com.customer.plugins` package, then the first line of the configuration file must contain the following entry:

```
com.customer.plugins.MyAPIEventHandler.
```

### Results

You have an installable JAR file that contains a plug-in that handles API events and a service provider configuration file that can be used to load the plug-in.

**Notes:** You only have one `eventHandlerName` property available to register both API event handlers and notification event handlers. If you want to use both an API

event handler and a notification event handler, the plug-in implementations must have the same name, for example, `Customer` as the event handler name for the SPI implementation.

You can implement both plug-ins using a single class, or two separate classes. In both cases, you need to create two files in the `META-INF/services/` directory of your JAR file, for example, `com.ibm.task.spi.CustomerNotificationEventHandlerPlugin` and `com.ibm.task.spi.CustomerAPIEventHandlerPlugin`.

Package the plug-in implementation and the helper classes in a single JAR file.

To make a change to an implementation effective, replace the JAR file in the shared library, deploy the associated EAR file again, and restart the server.

## What to do next

You now need to install and register the plug-in so that it is available to the human task container at runtime. You can register API event handlers with a task instance, a task template, or an application component.

### API event handlers

API events occur when a human task is modified or it changes state. To handle these API events, the event handler is invoked directly before the task is modified (pre-event method) and just before the API call returns (post-event method).

If the pre-event method throws an `ApplicationVetoException` exception, the API action is not performed, the exception is returned to the API caller, and the transaction associated with the event is rolled back. If the pre-event method was triggered by an internal event and an `ApplicationVetoException` exception is thrown, the internal event, such as an automatic claim, is not performed but an exception is not returned to the client application. In this case, an information message is written to the `SystemOut.log` file. If the API method throws an exception during processing, the exception is caught and passed to the post-event method. The exception is passed again to the caller after the post-event method returns.

The following rules apply to pre-event methods:

- Pre-event methods receive the parameters of the associated API method or internal event.
- Pre-event methods can throw an `ApplicationVetoException` exception to prevent processing from continuing.

The following rules apply to post-event methods:

- Post-event methods receive the parameters that were supplied to the API call, and the return value. If an exception is thrown by the API method implementation, the post-event method also receives the exception.
- Post-event methods cannot modify return values.
- Post-event methods cannot throw exceptions; runtime exceptions are logged and prevent processing from continuing.

To implement API event handlers, you can implement either the `APIEventHandlerPlugin3` interface, which extends the `APIEventHandlerPlugin` interface, or extend the default `com.ibm.task.spi.APIEventHandler` SPI implementation class. If your event handler inherits from the default

implementation class, it always implements the most recent version of the SPI. If you upgrade to a newer version of Business Process Choreographer, fewer changes are necessary if you want to exploit new SPI methods.

If you have both a notification event handler and an API event handler, both of these handlers must have the same name because you can register only one event handler name.

## Creating notification event handlers for Business Process Choreographer

Notification events are produced when human tasks are escalated. Business Process Choreographer provides functionality for handling escalations, such as creating escalation work items or sending e-mails. You can create notification event handlers to customize the way in which escalations are handled.

### About this task

To implement notification event handlers, you can implement the `NotificationEventHandlerPlugin` interface, or you can extend the default `com.ibm.task.spi.NotificationEventHandler` service provider interface (SPI) implementation class.

Complete the following steps to create a notification event handler.

### Procedure

1. Write a class that implements the `NotificationEventHandlerPlugin` interface or extends the `NotificationEventHandler` implementation class. This class can invoke the methods of other classes.

If you use the `NotificationEventHandlerPlugin` interface, you must implement all of the interface methods. If you extend the SPI implementation class, overwrite the methods that you need.

This class runs in the context of a Java Platform, Enterprise Edition (Java EE) Enterprise application. Ensure that this class and its helper classes follow the EJB specification.

The plug-in is invoked with the authority of the `EscalationUser` role. This role is defined when the human task container is configured.

**Note:** If you want to call the `HumanTaskManagerService` interface from this class, do not call a method that updates the task that produced the event. This action might result in inconsistent task data in the database.

2. Assemble the plug-in class and its helper classes into a JAR file.  
You can make the JAR file available in one of the following ways:
  - As a utility JAR file in the application EAR file.
  - As a shared library that is installed with the application EAR file.
  - As a shared library that is installed with the TaskContainer application. In this case, the plug-in is available for all tasks.
3. Assemble the plug-in class and its helper classes into a JAR file.  
If the helper classes are used by several Java EE applications, you can package these classes in a separate JAR file that you register as a shared library.
4. Create a service provider configuration file for the plug-in in the `META-INF/services/` directory of your JAR file.



The configuration file provides the mechanism for identifying and loading the plug-in. This file conforms to the Java EE service provider interface specification.

- a. Create a file with the name `com.ibm.task.spi.plug-in_nameNotificationEventHandlerPlugin`, where *plug-in\_name* is the name of the plug-in.

For example, if your plug-in is called `HelpDeskRequest` (event handler name) and it implements the `com.ibm.task.spi.NotificationEventHandlerPlugin` interface, the name of the configuration file is `com.ibm.task.spi.HelpDeskRequestNotificationEventHandlerPlugin`.

- b. In the first line of the file that is neither a comment line (a line that starts with a number sign (#)) nor a blank line, specify the fully qualified name of the plug-in class that you created in step 1.

For example, if your plug-in class is called `MyEventHandler` and it is in the `com.customer.plugins` package, then the first line of the configuration file must contain the following entry: `com.customer.plugins.MyEventHandler`.

## Results

You have an installable JAR file that contains a plug-in that handles notification events and a service provider configuration file that can be used to load the plug-in. You can register API event handlers with a task instance, a task template, or an application component.

**Notes:** You only have one `eventHandlerName` property available to register both API event handlers and notification event handlers. If you want to use both an API event handler and a notification event handler, the plug-in implementations must have the same name, for example, `Customer` as the event handler name for the SPI implementation.

You can implement both plug-ins using a single class, or two separate classes. In both cases, you need to create two files in the `META-INF/services/` directory of your JAR file, for example, `com.ibm.task.spi.CustomerNotificationEventHandlerPlugin` and `com.ibm.task.spi.CustomerAPIEventHandlerPlugin`.

Package the plug-in implementation and the helper classes in a single JAR file.

To make a change to an implementation effective, replace the JAR file in the shared library, deploy the associated EAR file again, and restart the server.

## What to do next

You now need to install and register the plug-in so that it is available to the human task container at runtime. You can register notification event handlers with a task instance, a task template, or an application component.

## Installing API event handler and notification event handler plug-ins for human tasks

To use API event handler or notification event handler plug-ins, you must install the plug-in so that it can be accessed by the human task container.

## About this task

The way in which you install the plug-in depends on whether the plug-in is to be used by only one Java Platform, Enterprise Edition (Java EE) application, or several applications.

Complete one of the following steps to install a plug-in.

### Procedure

- Install a plug-in for use by a single Java EE application.  
Add your plug-in JAR file to the application EAR file. In the deployment descriptor editor in WebSphere Integration Developer, install the JAR file for your plug-in as a project utility JAR file for the Java EE application of the main enterprise JavaBeans (EJB) module.
- Install a plug-in for use by several Java EE applications.  
Put the JAR file in a WebSphere Application Server shared library and associate the library with the applications that need access to the plug-in. To make the JAR file available in a network deployment environment, manually distribute the JAR file on each node that hosts a server or cluster member on which any of your applications is deployed. You can use the deployment target scope of your applications, that is the server or cluster on which the applications are deployed, or the cell scope. Be aware that the plug-in classes are then visible throughout the selected deployment scope.

### What to do next

You can now register the plug-in.

## Registering API event handler and notification event handler plug-ins with task templates, task models, and tasks

You can register plug-ins for API event handlers and notification event handlers with tasks, task templates, and task models at various times: when you create an ad hoc task, update an existing task, create an ad hoc task model, or define a task template.

### About this task

You can register plug-ins for API event handlers and notification event handlers with tasks on the following levels:

#### Task template

All of the tasks that are created using the template use the same handlers

#### Ad hoc task model

The tasks that are created using the model use the same handlers

#### Ad hoc task

The task that is created uses the specified handlers

#### Existing task

The task uses the specified handlers

You can register a plug-in in one of the following ways.

## Procedure

- For task templates modeled in WebSphere Integration Developer, specify the plug-in in the task model.
- For ad hoc tasks or ad hoc task models, specify the plug-in when you create the task or task model.  
Use the `setEventHandlerName` method of the `TTask` class to register the name of the event handler.
- Change the event handler for a task instance at runtime.  
Use the `update(Task task)` method to use a different event handler for a task instance at runtime. The caller must have task administrator authority to update this property.

## Using a plug-in to post-process people query results

People resolution in Business Process Choreographer returns a list of the users that are assigned to a specific role, for example, potential owners of a task. You can create a plug-in that changes the results of people queries that are returned by people resolution. For example, to improve workload balancing, you could remove users from the query result who already have a high workload.

### About this task

To modify the results that are returned by people assignment and people substitution, you must write a class that implements the plug-in interface, assemble a JAR file for the plug-in, then install and activate it.

Complete the following steps to create a plug-in to post-process people query results.

### Procedure

1. Implement your people query result post-processing plug-in. Write a class that implements either the `StaffQueryResultPostProcessorPlugin` interface or the `StaffQueryResultPostProcessorPlugin2` interface.
2. Create an installable JAR file.
  - a. Assemble your plug-in class and its helper classes into a JAR file.
  - b. Create a service provider configuration file for the plug-in in the `META-INF/services/` directory of your JAR file. The configuration file provides the mechanism for identifying and loading the plug-in. This file must conform to the Java EE service provider interface specification.
    - 1) In a text editor, create a service provider configuration file with the name `com.ibm.task.spi.plug-in_nameStaffQueryResultPostProcessorPlugin`, where *plug-in\_name* is the name of the plug-in. The name of the configuration file does not depend on the name of the interface that you implemented. For example, if your plug-in is called `MyHandler` and it implements the `com.ibm.task.spi.StaffQueryResultPostProcessorPlugin2` interface, the name of the configuration file is `com.ibm.task.spi.MyHandlerStaffQueryResultPostProcessorPlugin`.
    - 2) In the first line of the file that is neither a comment line (a line that starts with a number sign (#)) nor a blank line, specify the fully qualified name of the plug-in class that you created in step 1. For example, if your plug-in class is called `StaffPostProcessor` and it is in the `com.customer.plugins` package, then the first line of the

configuration file must contain the following entry:  
`com.customer.plugins.StaffPostProcessor.`

You have an installable JAR file that contains a plug-in that post-processes people query results and a service provider configuration file that can be used to load the plug-in.

3. Install the JAR file in a shared library in the application server and associate it with the Human Task Manager application.
  - a. Define a WebSphere Application Server shared library for the plug-in on the scope of the server or cluster where Business Process Choreographer is configured.
  - b. Associate the shared library with the TaskContainer application.
  - c. Make the plug-in JAR file available to each affected WebSphere Process Server installation that hosts a server or a cluster member.
4. Configure the Human Task Manager to use the plug-in.
  - a. In the administrative console, go to the Custom Properties page of the Human Task Manager.

Click either **Servers** → **Clusters** → **WebSphere application server clusters** → *cluster\_name* or **Servers** → **Server Types** → **WebSphere application servers** → *server\_name*, then on the **Configuration** tab, in the **Business Integration** section, expand **Business Process Choreographer**, and click **Human Task Manager**. Under **Additional Properties**, select **Custom Properties**.
  - b. Add a custom property with the name **Staff.PostProcessorPlugin**, and a value of the name that you gave to your plug-in, for example, MyHandler.

The plug-in is now available for post processing people query results.

5. Restart the server to activate the plug-in. The post processing plug-in is invoked after both the people assignment and people substitution have run.

**Note:** If you modify the plug-in, you must replace the JAR file in the shared library, and restart the server.

---

## Part 2. Deploying applications



---

## Overview of preparing and installing modules

Installing modules (also known as deploying) activates the modules in either a test environment or a production environment. This overview briefly describes the test and production environments and some of the steps involved in installing modules.

**Note:** The process for installing applications in a production environment is similar to the process described in “Developing and deploying applications” in the WebSphere Application Server for z/OS information center. If you are unfamiliar with those topics, review those first.

Before installing a module to a production environment, always verify changes in a test environment. To install modules to a test environment, use WebSphere Integration Developer (see the WebSphere Integration Developer information center for more information). To install modules to a production environment, use WebSphere Process Server.

This topic describes the concepts and tasks needed to prepare and install modules to a production environment. Other topics describe the files that house the objects that your module uses and help you move your module from your test environment into your production environment. It is important to understand these files and what they contain so you can be sure that you have correctly installed your modules.

---

## Libraries and JAR files overview

Modules often use artifacts that are located in libraries, which are special projects in WebSphere Integration Developer used for storing shared resources. At deployment time, WebSphere Integration Developer libraries are transformed into utility JAR files and packaged in the applications to be run.

While developing a module, you might identify certain resources or components that could be used by other modules. These artifacts can be shared by using a library.

### What is a library?

A library is a special project in WebSphere Integration Developer that is used for the development, version management, and organization of shared resources, such as those resources that are typically shared between modules. Only a subset of artifact types can be created and stored in a library, including:

- Interfaces or Web services descriptors (files with a .wsdl extension)
- Business object XML schema definitions (files with an .xsd extension)
- Business object maps (files with a .map extension)
- Relationship and role definitions (files with a .rel and .rol extension)

At deployment time, these WebSphere Integration Developer libraries are transformed into utility JAR files in the applications to be run.

When a module needs an artifact, the server locates the artifact from the EAR class path and loads the artifact, if it is not already loaded, into memory. Figure 78 on page 482

page 482 shows how an application contains components and related libraries.

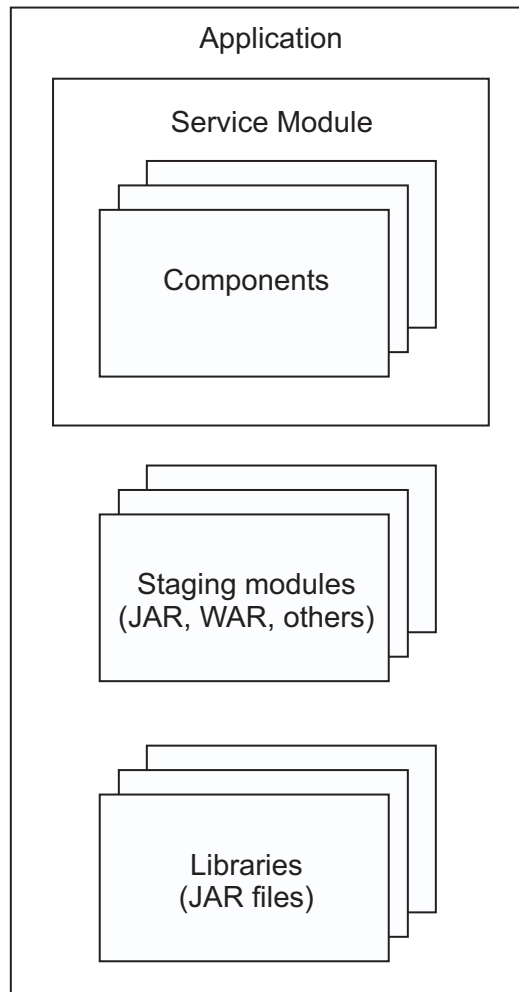


Figure 78. Relationships among module, component, and library

## What are JAR, RAR, and WAR files?

There are a number of files that can contain components of a module. These files are fully described in the Java Platform, Enterprise Edition specification. Details about JAR files can be found in the JAR specification.

In WebSphere Process Server, a JAR file also contains an application, which is the assembled version of the module with all the supporting references and interfaces to any other service components used by the module. To completely install the application, you need this JAR file, any other dependent JAR, Web services archive (WAR), resource archive (RAR), staging libraries (Enterprise Java Beans - EJB) JAR files, and any other archives. You then create an installable EAR file using the `serviceDeploy` command (see ).

## Naming conventions for staging modules

Within the library, there are requirements for the names of the staging modules. These names are unique for a specific module. Name any other modules required to deploy the application so that conflicts with the staging module names do not occur. For a module named *myService*, the staging module names are:



- *myServiceApp*
- *myServiceWeb*

**Note:** The *myServiceEJB* and *myServiceEJBClient* staging modules no longer get created by `serviceDeploy`. However, those file names should not be used, because they could still be deleted by the `serviceDeploy` command.

## Considerations when using libraries

Using libraries provides consistency of business objects and consistency of processing amongst modules because each calling module has its own copy of a specific component. To prevent inconsistencies and failures it is important to make sure that changes to components and business objects used by calling modules are coordinated with all of the calling modules. Update the calling modules by:

1. Copying the module and the latest copy of the libraries to the production server
2. Rebuilding the installable EAR file using the `serviceDeploy` command
3. Stopping the running application containing the calling module and reinstalling it
4. Restarting the application containing the calling module

### Related reference

 [serviceDeploy command-line utility](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

---

## EAR file overview

An EAR file is a critical piece in deploying a service application to a production server.

An enterprise archive (EAR) file is a compressed file that contains the libraries, enterprise beans, and JAR files that the application requires for deployment.

You create a JAR file when you export your application modules from WebSphere Integration Developer. Use this JAR file and any other artifact libraries or objects as input to the installation process. The `serviceDeploy` command creates an EAR file from the input files that contain the component descriptions and Java code that make up the application.

### Related reference

 [serviceDeploy command-line utility](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

---

## Preparing to deploy to a server

After developing and testing a module, you must export the module from a test system and bring it into a production environment for deployment. To install an application you also should be aware of the paths needed when exporting the module and any libraries the module requires.

## Before you begin

Before beginning this task, you should have developed and tested your modules on a test server and resolved problems and performance issues.

**Important:** To prevent replacing an application or module already running in a deployment environment make sure the name of the module or application is unique from any already installed.

## About this task

This task verifies that all of the necessary pieces of an application are available and packaged into the correct files to bring to the production server.

**Note:** You can also export an enterprise archive (EAR) file from WebSphere Integration Developer and install that file directly into WebSphere Process Server.

**Important:** If the services within a component use a database, install the application on a server directly connected to the database.

## Procedure

1. Locate the folder that contains the components for the module you are to deploy.

The component folder should be named *module-name* with a file in it named *module.module*, the base module.

2. Verify that all components contained in the module are in component subfolders beneath the module folder.

For ease of use, name the subfolder similar to *module/component*.

3. Verify that all files that comprise each component are contained in the appropriate component subfolder and have a name similar to *component-file-name.component*.

The component files contain the definitions for each individual component within the module.

4. Verify that all other components and artifacts are in the subfolders of the component that requires them.

In this step you ensure that any references to artifacts required by a component are available. Names for components should not conflict with the names the serviceDeploy command uses for staging modules. See Naming conventions for staging modules.

5. Verify that a references file, *module.references*, exists in the module folder of step 1.

The references file defines the references and the interfaces within the module.

6. Verify that a wires file, *module.wires*, exists in the component folder.

The wires file completes the connections between the references and the interfaces within the module.

7. Verify that a manifest file, *module.manifest*, exists in the component folder.

The manifest lists the module and all the components that comprise the module. It also contains a class path statement so that the serviceDeploy command can locate any other modules needed by the module.

8. Create a compressed file or a JAR file of the module as input to the serviceDeploy command that you will use to prepare the module for installation to the production server.

## Example folder structure for MyValue module prior to deployment

The following example illustrates the directory structure for the module MyValueModule, which is made up of the components MyValue, CustomerInfo, and StockQuote.

```
MyValueModule
 MyValueModule.manifest
 MyValueModule.references
 MyValueModule.wiring
 MyValueClient.jsp
process/myvalue
 MyValue.component
 MyValue.java
 MyValueImpl.java
service/customerinfo
 CustomerInfo.component
 CustomerInfo.java
 Customer.java
 CustomerInfoImpl.java
service/stockquote
 StockQuote.component
 StockQuote.java
 StockQuoteAsynch.java
 StockQuoteCallback.java
 StockQuoteImpl.java
```

### What to do next

Install the module onto the production systems as described in [Installing a module on a production server](#).

#### Related reference



[serviceDeploy](#) command-line utility

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

---

## Considerations for installing service applications on clusters

Installing a service application on a cluster places additional requirements on you. It is important that you keep these considerations in mind as you install any service applications on a cluster.

Clusters can provide many benefits to your processing environment by providing economies of scale to help you balance request workload across servers and provide a level of availability for clients of the applications. Consider the following before installing an application that contains services on a cluster:

- Will users of the application require the processing power and availability provided by clustering?  
If so, clustering is the correct solution. Clustering will increase the availability and capacity of your applications.
- Is the cluster correctly prepared for service applications?  
You must configure the cluster correctly before installing and starting the first application that contains a service. Failure to configure the cluster correctly prevents the applications from processing requests correctly.
- Does the cluster have a backup?

You must install the application on the backup cluster also.

### **Cross-cluster modules**

JNDI resources must not be shared across clusters. A cross-cluster module requires that each cluster have different JNDI resources. A scenario matching the following criteria will result in the log file indicating a `NameNotFoundException`:

- One module has a configured binding that generates JNDI resources.
- Another module is configured to use those generated JNDI resources.
- The modules are deployed in different clusters.

To resolve the problem, modify the module properties so that each module uses the JNDI resources in the same cluster as it.

---

## Deploying a module

You can deploy a module or a mediation module, as generated by WebSphere® Integration Developer, into a production WebSphere Process Server environment using these steps.

### Before you begin

Before deploying a service application to a production server, assemble and test the application on a test server. After testing, export the relevant files as described in *Preparing to deploy to a server* in the Developing and Deploying Modules PDF and bring the files to the production system to deploy. See the information centers for WebSphere Integration Developer and WebSphere Application Server for z/OS for more information.

### Procedure

1. Copy the module and other files onto the production server.  
The modules and resources (EAR, JAR, RAR, and WAR files) needed by the application are moved to your production environment.
2. Run the serviceDeploy command to create an installable EAR file.  
This step defines the module to the server in preparation for installing the application into production.
  - a. Locate the JAR file that contains the module to deploy.
  - b. Issue the serviceDeploy command using the JAR file from the previous step as input.
3. Install the EAR file from step 2. How you install the applications depends on whether you are installing the application on a stand alone server or a server in a cell.

**Note:** You can either use the administrative console or a script to install the application. See the WebSphere Application Server information center for additional information.

4. Save the configuration. The module is now installed as an application.
5. Start the application.

### Results

The application is now active and work should flow through the module.

### What to do next

Monitor the application to make sure the server is processing requests correctly.

## Related reference

 [serviceDeploy command-line utility](#)

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

---

## Installing versioned SCA modules in a production environment

You can deploy versioned Service Component Architecture (SCA) modules into the run time. Each version of a module exists alongside any other versions currently installed on the server or in the cell.

### Before you begin

Make sure you perform the following tasks before installing a versioned SCA module into your production environment:

- In WebSphere Integration Developer, specify that the module is versioned and export it for command-line deployment. See [Creating versioned modules and libraries](#) for more information.
- Determine whether you want to co-deploy different versions of the module on a single server or whether you need to co-deploy multiple instances of the same versioned module on more than one cluster in the same cell.

### About this task

To install versioned modules, perform the following steps.

### Procedure

1. Run `serviceDeploy` against the versioned module you exported to generate an installable EAR file.

```
serviceDeploy moduleName.zip
```

The `serviceDeploy` command returns an installable EAR file whose name contains the version and, optionally, cell ID information.

2. Install the module using one of the following methods:
  - From within the administrative console, click **SCA Modules** and use the **Install** button on the **SCA Modules** page.
  - From within the administrative console, click **Applications > Install New Application**.
  - Use the `AdminApp.install wsadmin` command.
3. If you want to install a versioned module on multiple servers or clusters in a cell, do the following for each module instance you require:
  - a. Use the `createVersionedSCAModule` command to create an instance of the module.

```
createVersionedSCAModule -archiveAbsolutePath input_archive_dir
-workingDirectory working_dir -uniqueCellID cell_ID
```
  - b. Install the resulting EAR file as described in Step 2.
4. Optional: Use the `validateSCAImportExportInformation` command to validate that all SCA bindings and selector export bindings in the specified EAR file exist on the bus.

## Results

You now have one or more versioned applications in your production environment. They can all be administered through the administrative console or through corresponding administrative commands.

**Note:** To preserve versioning information, the installation process automatically modifies the module name to ensure it is unique within the server or cell through the use of either the `serviceDeploy` or `createVersionedSCAModule` command. These commands add the version number, a unique cell ID, or both to the original module name.

*moduleName\_vversionValue\_uniqueCellID*

For example, if you followed the steps in this topic, deploying version 1.0.1 of the module `billingProcess` results in a module called `billingProcess_v1_0_1` and an installed service application called `billingProcess_v1_0_1App`. If you also specify a unique cell ID (for example, `Cell5`), then the module is called `billingProcess_v1_0_1_Cell5` and the installed service application is called `billingProcess_v1_0_1_Cell5App`.

---

## Installing an SCA module with the console

Before you can start running a module or a mediation module, you must deploy it to a server or cluster. Deployment involves creating an installable enterprise archive (EAR) file and installing the EAR file onto the server or cluster.

### Before you begin

If you have exported either a module or a mediation module to a JAR file, use the `serviceDeploy` command to create an installable EAR file from the JAR file. For more information, see “Deploying a module” on page 487.

**Note:** Versioned libraries are shared between applications for run time and management. A versioned library can be associated with an application or module using an installed optional package that declares the shared library in the application's manifest file. Modules that use a library have a dependency on a specific version of that library. An artifact in a versioned library that is used by multiple applications is managed as a single artifact. If the artifact is changed, it affects all dependents.

See the WebSphere Integration Developer information center for details about versioned libraries.

### About this task

You must install the EAR file onto a server or cluster before you can start running the module or mediation module.

Instead of using the administrative console, you can use other methods to install the EAR file, such as the `AdminApp.install` or `AdminApp.installinteractive` command with the `wsadmin` tool.

**Important:** If, after you start performing the steps, you decide not to install the application you must click **Cancel**: do not simply move to another administrative console page.

## Procedure

1. From the administrative console, click **Applications** → **New Application** in the console navigation pane. The first of two **Preparing for application installation** pages is displayed.
2. On the first **Preparing for application installation** page:
  - a. Specify the full path name of the EAR file. For more information, see *Installing applications with the console*.
  - b. Select whether to use default values or specify some of the values yourself:
    - Prompt me only when additional information is required**  
Displays only the module mapping step and other steps where you must specify information.
    - Show me all installation options and parameters**  
Displays all installation steps. To use **Generate default bindings**, which supplies default values for incomplete bindings, select this option.
  - c. Click **Next**.
3. Installing an EAR file containing a mediation flow is like installing any other enterprise application EAR file into WebSphere Application Server. For detailed information about completing the second **Preparing for application installation** page and specifying the options in the remaining wizard steps, see *Installing applications with the console*.
4. When you are installing a mediation module, or a module containing a mediation flow, there is one additional optional step you can perform. On the **Edit module properties** panel, you can edit the values of the properties in the module. If properties belong to a group they are displayed inside an expandable section; if they do not belong to a group you can view them immediately.

## Results

You can now start the module or mediation module.

---

## Creating an installable EAR file using serviceDeploy

To install an application in the production environment, take the files copied to the production server and create an installable EAR file.

### Before you begin

Before starting this task, you must have a JAR file that contains the module and services you are deploying to the server. See “Preparing to deploy to a server” for more information.

### About this task

The serviceDeploy command takes a JAR file, any other dependent EAR, JAR, RAR, WAR and ZIP files and builds an EAR file that you can install on a server.

## Procedure


1. Locate the JAR file that contains the module to deploy.
2. Issue the serviceDeploy command using the JAR file from the previous step as input.

This step creates an EAR file.



- Note:** Perform the following steps at an administrative console.
3. Select the EAR file to install in the administrative console of the server.
  4. Click **Save** to install the EAR file.

#### Related reference

 [serviceDeploy command-line utility](#)  
Use the serviceDeploy command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through wsadmin.

---

## Deploying applications using Apache Ant tasks

ANT tasks allow you to define the deployment of multiple applications to WebSphere Process Server and have them run unattended on a server.

### Before you begin

This task assumes the following:

- The applications being deployed have already been developed and tested.
- The applications are to be installed on the same server or servers.
- You have some knowledge of Apache Ant tasks.
- You understand the deployment process.

Information about developing and testing applications is located in the WebSphere Integration Developer information center.

The Generated API and SPI documentation reference section provides details of application programming interfaces. Apache Ant tasks are described in the package `com.ibm.websphere.ant.tasks`. For the purpose of this topic, the tasks of interest are ServiceDeploy and InstallApplication.

### About this task

If you need to install multiple applications concurrently, develop an Apache Ant task before deployment. The Apache Ant task can then deploy and install the applications on the servers without your involvement in the process.

### Procedure

1. Identify the applications to deploy.
2. Create a JAR file for each application.
3. Copy the JAR files to the target servers.
4. Create an Apache Ant task to run the ServiceDeploy command to create the EAR file for each server.
5. Create an Apache Ant task to run the InstallApplication command for each EAR file from step 4 on the applicable servers.
6. Run the ServiceDeploy Apache Ant task to create the EAR file for the applications.
7. Run the InstallApplication Apache Ant task to install the EAR files from step 6.

### Results

The applications are correctly deployed on the target servers.

## Example of deploying an application unattended

This example of deploying an application unattended shows an Apache ANT task contained in a file `myBuildScript.xml`.

```
<?xml version="1.0">

<project name="OwnTaskExample" default="main" basedir=". ">
 <taskdef name="servicedeploy"
 classname="com.ibm.websphere.ant.tasks.ServiceDeployTask" />
 <target name="main" depends="main2">
 <servicedeploy scaModule="c:/synctest/SyncTargetJAR"
 ignoreErrors="true"
 outputApplication="c:/synctest/SyncTargetEAREAR"
 workingDirectory="c:/synctest"
 cleanStagingModules="true"/>
 </target>
</project>
```

This statement shows how to invoke the Apache Ant task.

```
${WAS}/bin/ws_ant -f myBuildScript.xml
```

**Tip:** Multiple applications can be deployed unattended by adding additional project statements into the file.

## What to do next

Use the administrative console to verify that the newly-installed applications are started and processing the workflow correctly.

### Related reference



serviceDeploy command-line utility

Use the `serviceDeploy` command to package Service Component Architecture (SCA) compliant modules as Java applications that can be installed on a server. The command is useful when performing batch installs through `wsadmin`.

---

## Installing business process and human task applications

You can distribute Service Component Architecture (SCA) modules that contain business processes or human tasks, or both, to deployment targets. A deployment target can be a server or a cluster.

### Before you begin

Verify that Business Flow Manager and Human Task Manager are installed and configured for each application server or cluster on which you want to install your application.

### About this task

You can install business process and task applications from the administrative console, from the command line, or by running an administrative script.

### Results

After a business process or human task application is installed, all of the business process templates and human task templates are put into the start state. You can create process instances and task instances from these templates.

### What to do next

Before you can create process instances or task instances, you must start the application.

---

## How business process and human task applications are installed in a network deployment environment

When process templates or human task templates are installed in a network deployment environment, the following actions are performed automatically by the application installation.

The application is installed in stages. Each stage must complete successfully before the following stage can begin.

1. The application installation starts on the deployment manager.

During this stage, the business process templates and human task templates are configured in the WebSphere configuration repository. The application is also validated. If errors occur, they are reported in the System.out file, in the System.err file, or as FFDC entries on the deployment manager.

2. The application installation continues on the node agent.

During this stage, the installation of the application on one application server instance is triggered. This application server instance is either part of, or is, the deployment target. If the deployment target is a cluster with multiple cluster members, the server instance is chosen arbitrarily from the cluster members of this cluster. If errors occur during this stage, they are reported in the SystemOut.log file, in the SystemErr.log file, or as FFDC entries on the node agent.

3. The application runs on the server instance.

During this stage, the process templates and human templates are deployed to the Business Process Choreographer database on the deployment target. If errors occur, they are reported in the SDSF job data sets for the deployment manager.

---

## Deployment of business processes and human tasks

Use WebSphere Integration Developer or serviceDeploy to package process components or task components in an enterprise application (EAR) file. Each new version of a model that is to be deployed must be packaged in a new enterprise application.

When you install an enterprise application that contains business processes or human tasks, then these are stored as business process templates or human task templates, as appropriate, in the Business Process Choreographer database. Newly installed templates are, by default, in the started state. However, the newly installed enterprise application is in the stopped state. Each installed enterprise application can be started and stopped individually.

You can deploy many different versions of a process template or task template, each in a different enterprise application. The versions are differentiated by their valid-from dates. When you install a new enterprise application, the version of the template that is installed is determined as follows:

- If the name of the template and the target namespace do not already exist, a new template is installed
- If the template name and target namespace are the same as those of an existing template, but the valid-from date is different, a new version of an existing template is installed

**Note:** The template name is derived from the name of the component and not from the business process or human task.

If you do not specify a valid-from date, the date is determined as follows:

- If you use WebSphere Integration Developer, the valid-from date is the date on which the human task or the business process was modeled.
- If you use service deployment, the valid-from date is the date on which the serviceDeploy command was run. Only collaboration tasks get the date on which the application was installed as the valid-from date.

---

## Installing business process and human task applications interactively

You can install an application interactively at runtime using the wsadmin tool and the installInteractive script. You can use this script to change settings that cannot be changed if you use the administrative console to install the application.

### About this task

Perform the following steps to install business process applications interactively.

### Procedure

1. Start the wsadmin tool.

In the *profile\_root/bin* directory, enter wsadmin.

2. Install the application.

At the wsadmin command-line prompt, enter the following command:

```
$AdminApp installInteractive application.ear
```

where *application.ear* is the qualified name of the enterprise archive file that contains your process application. You are prompted through a series of tasks where you can change values for the application.

3. Save the configuration changes.

At the wsadmin command-line prompt, enter the following command:

```
$AdminConfig save
```

You must save your changes to transfer the updates to the master configuration repository. If a scripting process ends and you have not saved your changes, the changes are discarded.

## Configuring process application data source and set reference settings

You might need to configure process applications that run SQL statements for the specific database infrastructure. These SQL statements can come from information service activities or they can be statements that you run during process installation or instance startup.

### About this task

When you install the application, you can specify the following types of data sources:

- Data sources to run SQL statements during process installation
- Data sources to run SQL statements during the startup of a process instance
- Data sources to run SQL snippet activities

The data source required to run an SQL snippet activity is defined in a BPEL variable of type `tDataSource`. The database schema and table names that are required by an SQL snippet activity are defined in BPEL variables of type `tSetReference`. You can configure the initial values of both of these variables.

You can use the wsadmin tool to specify the data sources.

### Procedure

1. Install the process application interactively using the wsadmin tool.
2. Step through the tasks until you come to the tasks for updating data sources and set references.

Configure these settings for your environment. The following example shows the settings that you can change for each of these tasks.

3. Save your changes.

### Example: Updating data sources and set references, using the wsadmin tool

In the **Updating data sources** task, you can change data source values for initial variable values and statements that are used during installation of the process or when the process starts. In the **Updating set references** task, you can configure the settings related to the database schema and the table names.

Task [24]: Updating data sources

```
//Change data source values for initial variable values at process start
```

```

Process name: Test
// Name of the process template
Process start or installation time: Process start
// Indicates whether the specified value is evaluated
//at process startup or process installation
Statement or variable: Variable
// Indicates that a data source variable is to be changed
Data source name: MyDataSource
// Name of the variable
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name to jdbc/newName
Task [25]: Updating set references

// Change set reference values that are used as initial values for BPEL variables

Process name: Test
// Name of the process template
Variable: SetRef
// The BPEL variable name
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name of the data source of the set reference to jdbc/newName
Schema name: [IISAMPLE]
// The name of the database schema
Schema prefix: []:
// The schema name prefix.
// This setting applies only if the schema name is generated.
Table name: [SETREFTAB]: NEWTABLE
// Sets the name of the database table to NEWTABLE
Table prefix: []:
// The table name prefix.
// This setting applies only if the prefix name is generated.

```

---

## Uninstalling business process and human task applications, using the administrative console

You can use the administrative console to uninstall applications that contain business processes or human tasks.

### Before you begin

To uninstall an application that contains business processes or human tasks, the following conditions must apply:

- If the application is installed on a stand-alone server, the server must be running and have access to the Business Process Choreographer database.
- If the application is installed on a cluster, the deployment manager and at least one cluster member must be running. The cluster member must have access to the Business Process Choreographer database.
- If the application is installed on a managed server, the deployment manager and the managed server must be running. The server must have access to the Business Process Choreographer database.
- There are no instances of business process or human task templates present in any state.
- If a process instance was migrated to a newer version of the process but it is waiting for a service invocation to reply, the application that contains the previous version cannot be uninstalled until the reply is received. In all other cases, instances that have been migrated are considered to be instances of the new version, and the application that contains the older version of the process can be uninstalled.

## About this task

To uninstall an enterprise application that contains business processes or human tasks, perform the following actions:

### Procedure

1. In the administrative console, click **Applications** → **Application Types** → **WebSphere enterprise applications**.
2. Select the application that you want to uninstall and click **Stop**.  
This step fails if any process instances or task instances still exist in the application. You can either use the Business Process Choreographer Explorer to delete the instances, or the **-force** option of the `bpcTemplates.jacl` administrative script to stop and delete these instances before the application is uninstalled.
3. Select the application that you want to uninstall, and click **Uninstall**.
4. Click **Save** to save your changes.

### Results

The application is uninstalled.

#### Related tasks

“Uninstalling business process and human task applications, using an administrative command”

Using the `bpcTemplates.jacl` script provides an alternative to the administrative console for uninstalling applications that contain business processes or human tasks.

---

## Uninstalling business process and human task applications, using an administrative command

Using the `bpcTemplates.jacl` script provides an alternative to the administrative console for uninstalling applications that contain business processes or human tasks.

### Before you begin

To uninstall an application that contains business processes or human tasks, the following conditions must apply:

- If the application is installed on a stand-alone server, the server must be running and have access to the Business Process Choreographer database.
- If the application is installed on a cluster, the deployment manager and at least one cluster member must be running. The cluster member must have access to the Business Process Choreographer database.
- If the application is installed on a managed server, the deployment manager and the managed server must be running. The server must have access to the Business Process Choreographer database.
- Ensure that the server process to which the administrative client connects is running. To ensure that the administrative client automatically connects to the server process, do not use the `-conntype NONE` option as a command option.
- If WebSphere administrative security is enabled, and your user ID does not have operator or administrator authority, include the `wsadmin -user` and `-password`

options to specify a user ID that has operator or administrator authority. The `-uninstall` option requires operator authority and the `-force` option requires administrator authority.

- One or more of the following is true:
  - There are no instances of business process or human task templates present in any state.
  - You intend to use the **-force** option.
- If a process instance was migrated to a newer version of the process but it is waiting for a service invocation to reply, the application that contains the previous version cannot be uninstalled until the reply is received. In all other cases, instances that have been migrated are considered to be instances of the new version, and the application that contains the older version of the process can be uninstalled.

## About this task

The following steps describe how to use the `bpcTemplates.jacl` script to uninstall applications that contain business process templates or human task templates.

## Procedure

1. If there are still process instances or task instances associated with the templates in the application that you want to uninstall perform one or both of the following:
  - Use the Business Process Choreographer Explorer to delete the instances.
  - In cases where you are sure that no other business processes depend on the process templates that are defined in the application you want to uninstall, you can use the **-force** option.

### CAUTION:

**If you use the script with this option, it deletes any instances that are associated with the templates, all of the data that is associated with any running instances, stops the templates, and uninstalls the application in one step. Use this option with extreme care.**

2. Change to the Business Process Choreographer subdirectory where the administrative scripts are located. Enter the following command:

```
cd install_root/ProcessChoreographer/admin
```

On Linux<sup>®</sup> and UNIX<sup>®</sup> platforms, enter the following command:

```
cd install_root/ProcessChoreographer/admin
```

On the i5/OS<sup>®</sup> platform, enter the following command:

```
cd install_root/ProcessChoreographer/admin
```

On Windows platforms, enter the following command:

```
cd install_root\ProcessChoreographer\admin
```

3. Stop the templates and uninstall the corresponding application.

```
install_root/bin/wsadmin.sh -f bpcTemplates.jacl
 -uninstall application_name
 [-force]
```

Where:

**-uninstall** *application\_name*

This specifies the name of the application to be uninstalled.

**-force**

This option causes any running instances to be stopped and deleted before



the application is uninstalled. Use this option with care because it also deletes all of the data associated with the running instances.

## **Results**

The application is uninstalled.

### **Related tasks**

“Uninstalling business process and human task applications, using the administrative console” on page 496

You can use the administrative console to uninstall applications that contain business processes or human tasks.



---

## Adapters and their installation

Adapters allow your application to communicate with other components of your enterprise information system.

The process you use to install adapters is described in [Configuring and using adapters](#) in the [WebSphere Integration Developer information center](#).



---

## Troubleshooting a failed deployment

This topic describes the steps to take to determine the cause of a problem when deploying an application. It also presents some possible solutions.

### Before you begin

This topic assumes the following things:

- You have a basic understanding of debugging a module.
- Logging and tracing is active while the module is being deployed.

### About this task

The task of troubleshooting a deployment begins after you receive notification of an error. There are various symptoms of a failed deployment that you have to inspect before taking action.

### Procedure

1. Determine if the application installation failed.

Examine the `SystemOut.log` file for messages that specify the cause of failure. Some of the reasons an application might not install include the following:

- You are attempting to install an application on multiple servers in the same Network Deployment cell.
- An application has the same name as an existing module on the Network Deployment cell to which you are installing the application.
- You are attempting to deploy Java EE modules within an EAR file to different target servers.

**Important:** If the installation has failed and the application contains services, you must remove any SIBus destinations or JCA activation specifications created before the failure before attempting to reinstall the application. The simplest way to remove these artifacts is to click **Save > Discard all** after the failure. If you inadvertently save the changes, you must manually remove the SIBus destinations and JCA activation specifications (see [Deleting SIBus destinations](#) and [Deleting JCA activation specifications](#) in the [Administering](#) section).

2. If the application is installed correctly, examine it to determine if it started successfully.

If the application did not start successfully, the failure occurred when the server attempted to initiate the resources for the application.

- a. Examine the `SystemOut.log` file for messages that will direct you on how to proceed.
- b. Determine if resources required by the application are available and/or have started successfully.

Resources that are not started prevent an application from running. This protects against lost information. The reasons for a resource not starting include:

- Bindings are specified incorrectly
- Resources are not configured correctly
- Resources are not included in the resource archive (RAR) file

- Web resources not included in the Web services archive (WAR) file
- c. Determine if any components are missing.
 

The reason for missing a component is an incorrectly built enterprise archive (EAR) file. Make sure that all of the components required by the module are in the correct folders on the test system on which you built the Java archive (JAR) file. “Preparing to deploy to a server” contains additional information.
  3. Examine the application to see if there is information flowing through it.
 

Even a running application can fail to process information. Reasons for this are similar to those mentioned in step 2b on page 503.

    - a. Determine if the application uses any services contained in another application. Make sure that the other application is installed and has started successfully.
    - b. Determine if the import and export bindings for devices contained in other applications used by the failing application are configured correctly. Use the administrative console to examine and correct the bindings.
  4. Correct the problem and restart the application.

---

## Deleting JCA activation specifications

The system builds JCA application specifications when installing an application that contains services. There are occasions when you must delete these specifications before reinstalling the application.

### Before you begin

If you are deleting the specification because of a failed application installation, make sure the module in the Java Naming and Directory Interface (JNDI) name matches the name of the module that failed to install. The second part of the JNDI name is the name of the module that implemented the destination. For example in `sca/SimpleBOCrsmA/ActivationSpec`, **SimpleBOCrsmA** is the module name.

**Required security role for this task:** When security and role-based authorization are enabled, you must be logged in as administrator or configurator to perform this task.

### About this task

Delete JCA activation specifications when you inadvertently saved a configuration after installing an application that contains services and do not require the specifications.

### Procedure

1. Locate the activation specification to delete.
 

The specifications are contained in the resource adapter panel. Navigate to this panel by clicking **Resources > Resource adapters**.

  - a. Locate the **Platform Messaging Component SPI Resource Adapter**.
 

To locate this adapter, you must be at the **node** scope for a standalone server or at the **server** scope in a deployment environment.
2. Display the JCA activation specifications associated with the Platform Messaging Component SPI Resource Adapter.
 

Click on the resource adapter name and the next panel displays the associated specifications.

3. Delete all of the specifications with a **JNDI Name** that matches the module name that you are deleting.
  - a. Click the check box next to the appropriate specifications.
  - b. Click **Delete**.

## Results

The system removes selected specifications from the display.

## What to do next

Save the changes.

---

## Deleting SIBus destinations

Service integration bus (SIBus) destinations are used to hold messages being processed by SCA modules. If a problem occurs, you might have to remove bus destinations to resolve the problem.

### Before you begin

If you are deleting the destination because of a failed application installation, make sure the module in the destination name matches the name of the module that failed to install. The second part of the destination is the name of the module that implemented the destination. For example in `sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer`, **SimpleBOCrsmA** is the module name.

**Required security role for this task:** When security and role-based authorization are enabled, you must be logged in as administrator or configurator to perform this task.

### About this task

Delete SIBus destinations when you inadvertently saved a configuration after installing an application that contains services or you no longer need the destinations.

**Note:** This task deletes the destination from the SCA system bus only. You must remove the entries from the application bus also before reinstalling an application that contains services (see Deleting JCA activation specifications in the Administering section of this information center).

### Procedure

1. Log into the administrative console.
2. Display the destinations on the SCA system bus.
  - a. In the navigation pane, click **Service integration** → **buses**
  - b. In the content pane, click **SCA.SYSTEM.cell\_name.Bus**
  - c. Under Destination resources, click **Destinations**
3. Select the check box next to each destination with a module name that matches the module that you are removing.
4. Click **Delete**.

## **Results**

The panel displays only the remaining destinations.

## **What to do next**

Delete the JCA activation specifications related to the module that created these destinations.



---

## Part 3. Appendixes







Printed in USA