IBM WebSphere Process Server for Multiplatforms

Technical Overviews

Version 7.0.0

30 April 2010

This edition applies to version 7, release 0, modification 0 of WebSphere Process Server for Multiplatforms (product number 5724-L01) 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.

Contents

Technical overviews

Technical overview information introduces the standards and technical aspects related to the product architecture.

Service Component Architecture

Service Component Architecture (SCA) enables a service-oriented architecture and is made available by many corporations including IBM®. SCA is a platform and vendor independent programming model that provides a simple and consistent means for expressing business logic and business data as SOA services, regardless of technical implementation details. In this section, we examine SCA services and data objects.

SCA and service invocation

If you take the three aspects of a programming model, which are data, invocation, and composition, and apply some of the new paradigms of a services-based approach, the new programming model for SOA starts to emerge. Service Component Architecture (SCA) provides a way of invoking business services within SOA solutions.

The architectural constructs that make up a service-oriented architecture include a way to represent the data that is exchanged between services, a mechanism for invoking services, and a way to compose services into larger integrated business applications. Today there are many different programming models for supporting each of these. This situation presents developers with the challenge of not only solving a particular business problem, but also choosing and understanding the appropriate implementation technology. One of the important goals of the WebSphere® Process Server SOA solution is to mitigate these complexities. This is done by converging the various programming models used for implementing service-oriented business applications into a simplified programming model.

This section focuses specifically on the Service Component Architecture (SCA) in WebSphere Process Server as the service-oriented component model for defining and invoking business services. SCA plays an important role in providing an invocation model for the SOA solution in WebSphere Process Server. SCA also plays a role in composing business services into composite business applications.

First, we see that data is primarily represented by Extensible Markup Language (XML) and is programmed with business objects based on the Service Data Object (SDO) specification or through native XML facilities such as XPath or XSLT (Extensible Stylesheet Language Transformation). Second, service invocation maps to Service Component Architecture (SCA). Finally, composition is embodied in process orchestration using Business Process Execution Language (BPEL). The figure shows the three aspects of this new programming model.

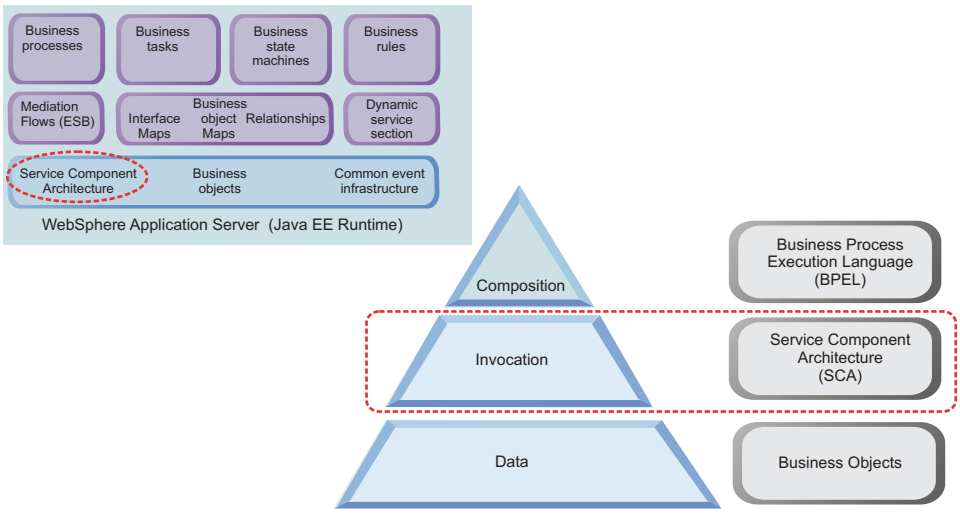


Figure 1. Representing data, invocation, and composition within a programming model for SOA

SCA is aimed at providing a simplified programming model for writing applications that run in a Java EE runtime environment, and is based upon concepts and techniques that are refinements of existing Java EE technology. One of the important aspects of SCA is to enable the separation between application business logic and the implementation details. In order to accomplish this, SCA provides a single abstraction for service types that might already be expressed as session beans, Web services, Java classes, or BPEL. The ability to separate business logic from infrastructure logic is important to help reduce the IT resources needed to build an enterprise application, and give developers more time to work on solving a particular business problem rather than focusing on the details of which implementation technology to use.

Modules

A *module* is a unit of deployment that determines which artifacts are packaged together in an Enterprise Archive (EAR) file. Components within a module are collocated for performance, and can pass their data by reference. A module can be seen as a scoping mechanism; that is, it sets an organizational boundary for artifacts.

A module is a composite of service components, imports, and exports. The service components, imports, and exports reside in the same project and root folder, which also contain the wiring that links the components and the bindings needed for the imports and exports. A module may also contain the implementations and interfaces referenced by its components, imports and exports, or these may be placed in other projects, such as a library project.

There are two types of modules. First, a module called *module* (sometimes referred to as a business integration module) that contains a choice of many component types, often used to support a business process. Second, a module called a *mediation module*, which contains up to one component, one or more mediation flow components, plus zero or more Java components that augment the mediation flow component.

A module may contain one or more mediation flow components.

Why are there two module types? The first type of module is primarily designed for business processes. A mediation module is like a gateway to existing external services, which is common in enterprise service bus architectures. These external services or exports are accessed in a mediation module by imports or service providers. By decoupling client service requesters from service providers by a mediation flow, your applications gain flexibility and resilience, a goal of service-oriented architecture. For example, your mediation flow can log incoming messages, route messages to a specific service determined at run time or transform data to make it suitable to pass to another service. These functions can be added and changed over time without modifying the requester or provider services.

A module results in a service application tested and deployed to the WebSphere Process Server. A mediation module results in a service application tested and deployed to either the WebSphere Process Server or the WebSphere Enterprise Service Bus server. Both types of modules support imports and exports.

Implementations, interfaces, business objects, business object maps, roles, relationships, and other artifacts often need to be shared among modules. A *library* is a project used to store these shared resources.

In Figure 2 on page 4, the module contains two service components, each containing an implementation. The module also contains the appropriate interfaces and references required by the service components. The second service component does not contain a reference because it does not invoke any external service.

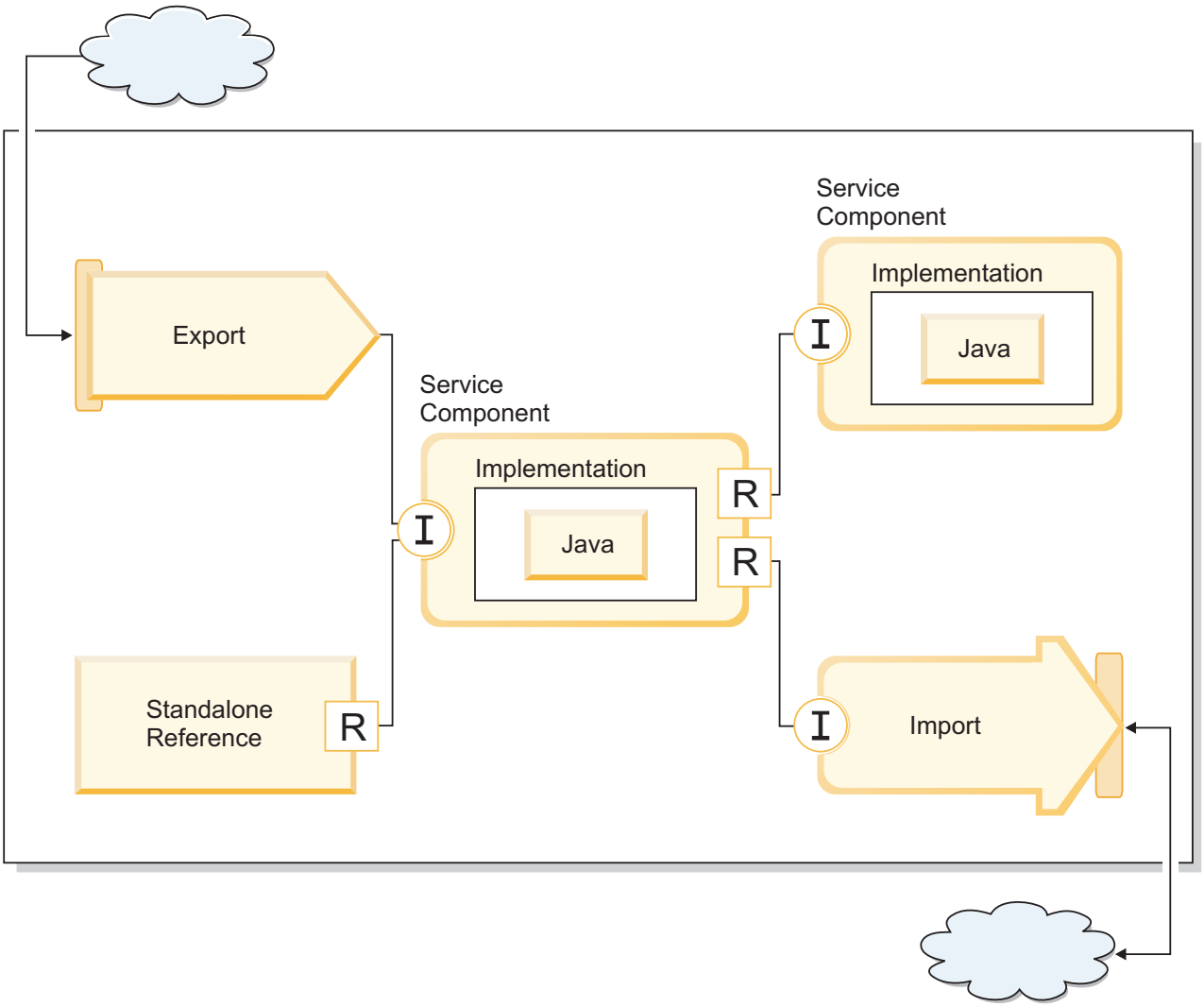
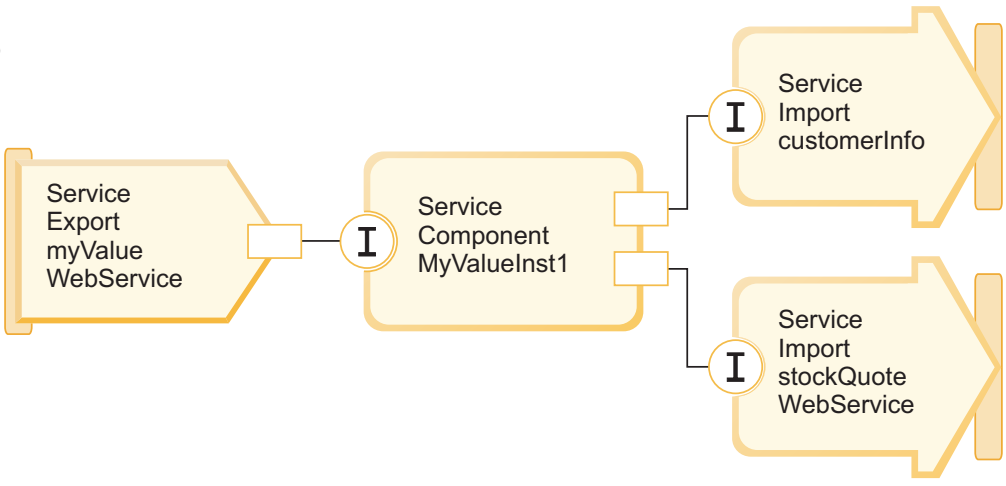


Figure 2. Structure of a module

In Figure 3, the module contains an export, two imports, and a service component that uses them. Wiring is shown linking the interfaces and references.

Figure 3. Service module



Module and mediation module artifacts include:

- Module definition - defines the module.
- Service components - definitions of the services in the module. A service component name inside a module is unique. However, a service component can have an arbitrary display name, which is typically a name more useful to a user.
- Imports - definitions of imports, which are calls to services external to this module. Imports have bindings, which are discussed in the Bindings section.
- Exports - definitions of exports, which are used to expose components to callers that are external to this module. Exports have bindings, which are discussed in the Bindings section.
- References - references from one component to another in the module.
- Stand-alone references - references applications that are not defined as Service Component Architecture components (for example, JavaServer Pages), which enable these applications to interact with Service Component Architecture components. There can be only one stand-alone references artifact per module.
- Other artifacts - these artifacts include WSDL files, Java classes, XSD files, BPEL processes, and so on.

Service components

A service component configures a service implementation. A service component is presented in a standard block diagram.

In addition to providing a consistent syntax and mechanism for service invocation, Service Component Architecture (SCA) is the invocation framework that provides a way for developers to encapsulate service implementations in reusable components. SCA enables developers to define interfaces, implementations, and references in a way that is independent of the technology that is used. This approach gives you the opportunity to bind the elements to whichever technology you choose. SCA separates business logic from infrastructure so that application programmers can focus on solving business problems.

A component consists of an implementation, which is hidden when using WebSphere Integration Developer's tools, one or more interfaces, which defines its inputs, outputs and faults, and zero or more references. A reference identifies the interface of another service or component that this component requires or consumes. An interface may be defined in one of two languages: a WSDL port type or Java. An interface supports synchronous and asynchronous interaction styles. A component's implementation can be in various languages.

The recommended interface type is WSDL and our tutorials and samples consistently use the WSDL interface type. A Java interface, however, is supported and used mostly in the case when a stateless session EJB is imported. Should you develop a top-down Java component, that is, define a component and add the Java implementation later, you should still use a WSDL interface. You cannot mix WSDL-interface-based components with Java-interface-based components.

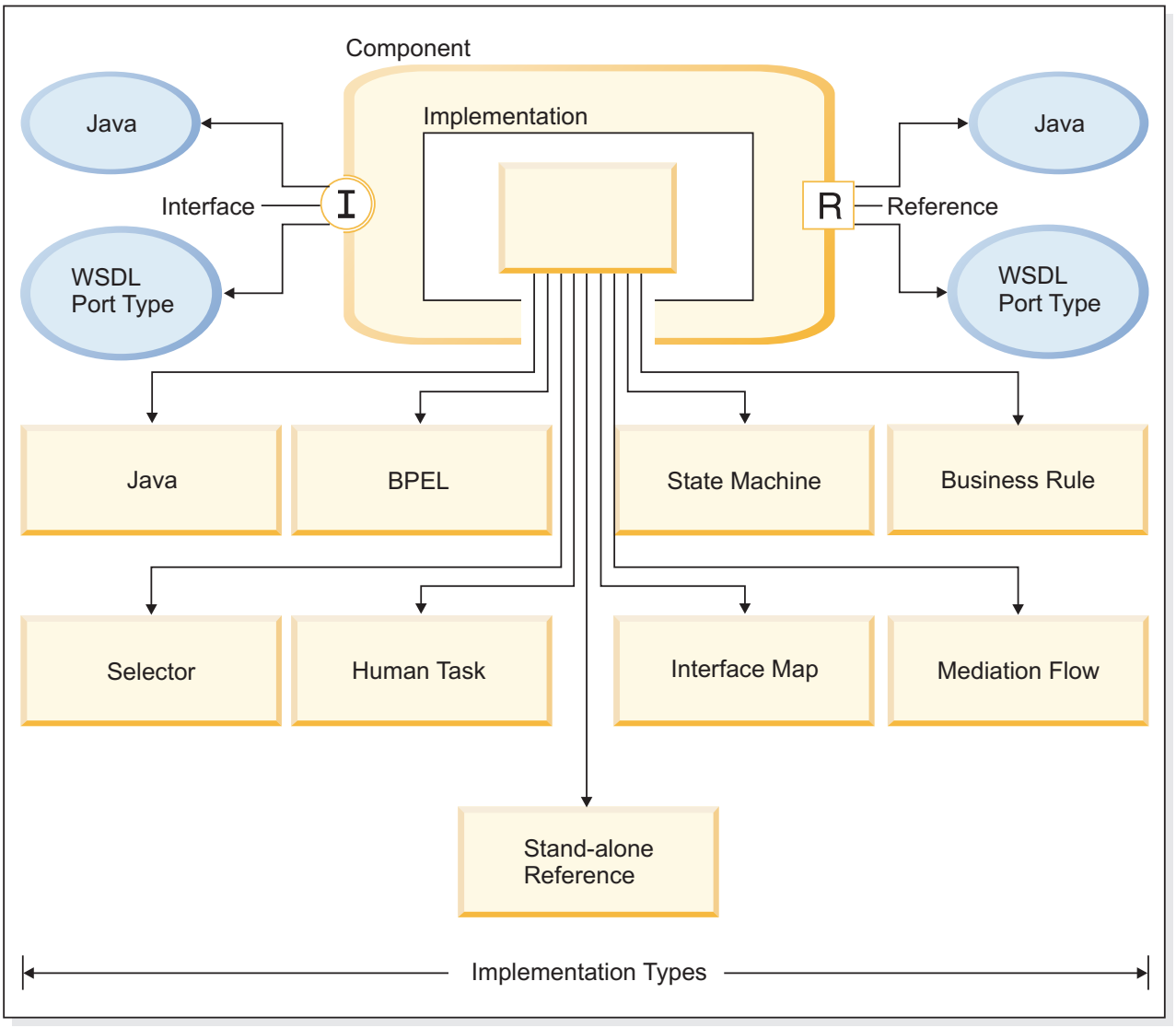


Figure 4. Structure of a component

In Figure 5 on page 7, we have a component in the center. Its implementation, MyValueImpl, is in Java as is its interface. It has two references: another Java interface and a WSDL interface.

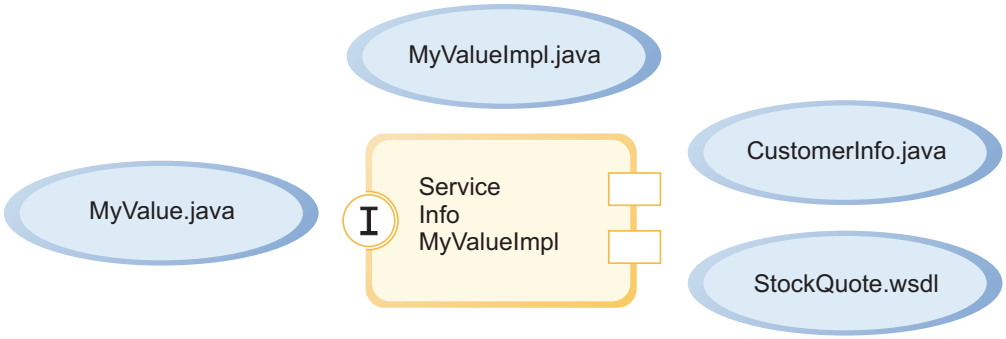


Figure 5. Service component structure

When working with this component, as shown below, you effectively only see the component itself. A reference to this component from another component would be revealed visually by a line to its interface. A reference from this component would be revealed by a line from its reference point to the interface of other component. A reference represents a service that this component consumes. By naming a reference and only specifying its interface, it allows the component implementation author to defer binding that reference to an actual service until later. At that later time, the integration specialist will do so by wiring from the reference to the interface of another component or import. This loose coupling, which allows for deferred binding and the re-use of implementations, is one of the key reasons for using WebSphere Integration Developer's Service Component Architecture.

A component may also have properties and qualifiers. A qualifier is a quality of service (QoS) directive on interfaces and references for the run time.

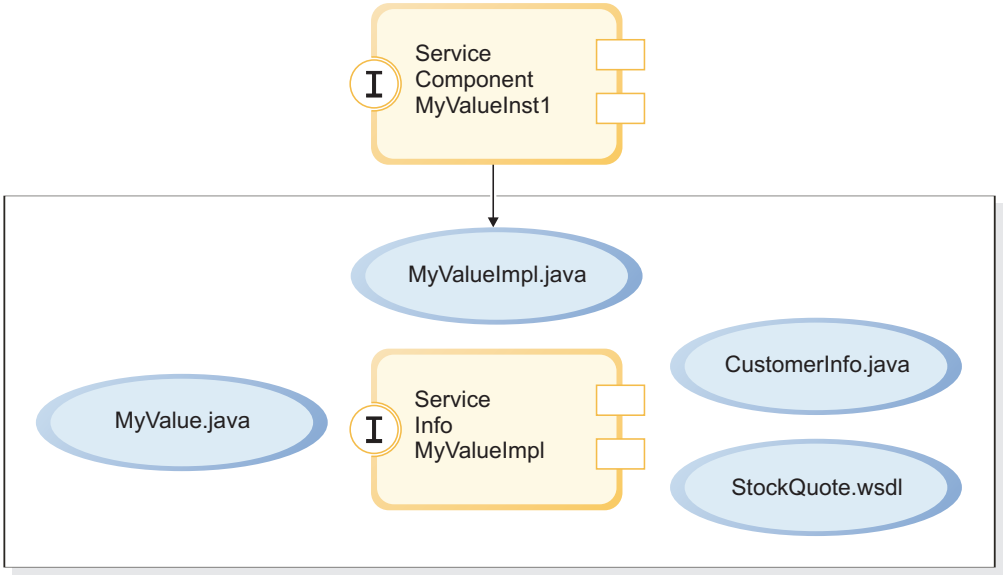


Figure 6. Instance of a service component

Service component implementation types are the implementations of the service components.

WebSphere Integration Developer supports the following implementation artifacts for WebSphere Process Server and WebSphere Enterprise Service Bus:

Table 1. Implementation artifacts

WebSphere Process Server	WebSphere Enterprise Service Bus
Java objects	Java objects
Business processes	Mediation flows
Business state machines	
Business rules	
Selectors	
Human tasks	
Interface maps	
Mediation flows	

Note: Note: Interface maps are deprecated as of WebSphere Process Server version 7.0. You can migrate your existing interface map components in WebSphere Integration Developer to use the functions in the mediation flow component.

The standard component implementations of the services are described in the topics in this section. These implementations appear in services in the assembly editor and or within BPEL processes.

Java objects

An implementation of a component in Java is referred to as a Java object.

One common implementation is a component written in Java. This implementation is sometimes nicknamed a "plain old Java object" or POJO. Generally, this implementation has a WSDL interface type, though this implementation could also have a Java interface. If there are multiple interfaces specified, then you cannot mix WSDL interfaces with Java interfaces. You can, however, "join" an application created with a set of WSDL interfaces to an application with a set of Java interfaces. A sample listed in the samples gallery of the Welcome view shows you how.

When working with a Java object, the code remains hidden from you within the context of the editors.

A Java object can be used in a mediation module. It can be deployed to either a WebSphere Process Server or a WebSphere Enterprise Service Bus server.

BPEL process

A *BPEL process* component implements a business process.

Its implementation language is the industry standard Business Process Execution Language for Web Services (BPEL4WS) and its IBM extensions. A BPEL process implements a potentially long-running service through the use of more elementary services. A BPEL process created in the process editor can do the following things:

- Describe the orchestration of other services using control flow graphs
- Use variables to keep the process state
- Use sophisticated error handling through fault handling
- Support asynchronous events
- Correlate inbound requests with the right instance of a particular process by using correlation sets to mark that business data within the request that identifies the instance (for example, a customer ID)

- Provide extended transactions through sophisticated compensation support

In addition to these standard BPEL items, WebSphere Integration Developer also extends BPEL to include people into a process with *human task* support. For example, this extension could add to a process the requirement that a person approves a loan.

The process editor uses visual representations of BPEL constructs to build your business process quickly and simply.

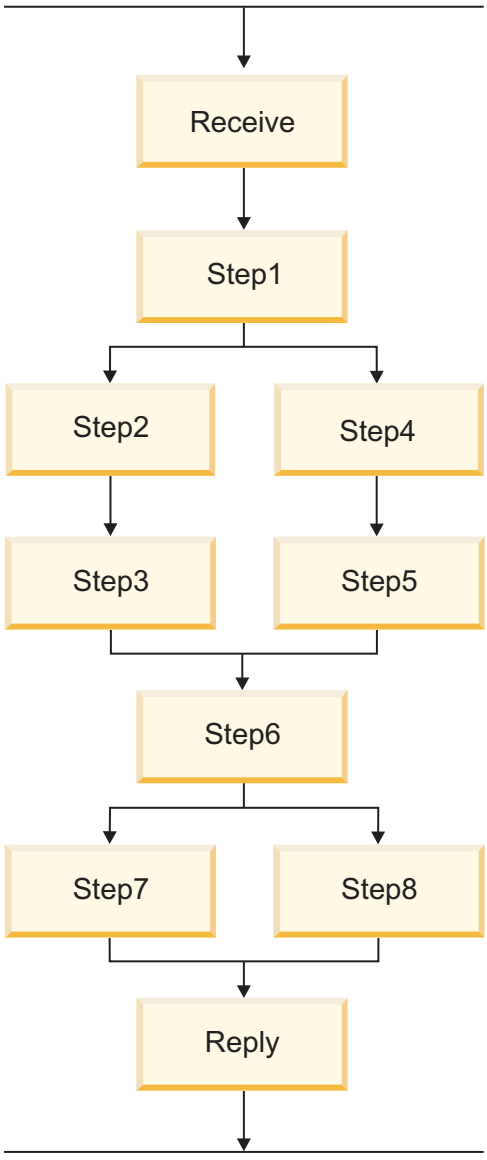


Figure 7. Simple business process

A BPEL process cannot be used in a mediation module. It can only be deployed to a WebSphere Process Server.

State machines

A state machine is an alternative way of creating a business process. A state machine is suited for processes related to changing states rather than a flow of control. A state defines what an artifact can do at a point in time. A *state machine* is an implementation of this set of states.

State machines are a common way of showing a set of interrelated states in a process. A familiar state machine is a drink dispenser. You put some coins into the machine and along with your drink, which hopefully is dispensed, you get your exact change as the state machine mechanically breaks down the coins that need to be returned to you based on the coins you inserted. In the diagram that follows, a typical state machine is shown as created by the state machine editor. In the state machine, an item is purchased and shipped to a customer.

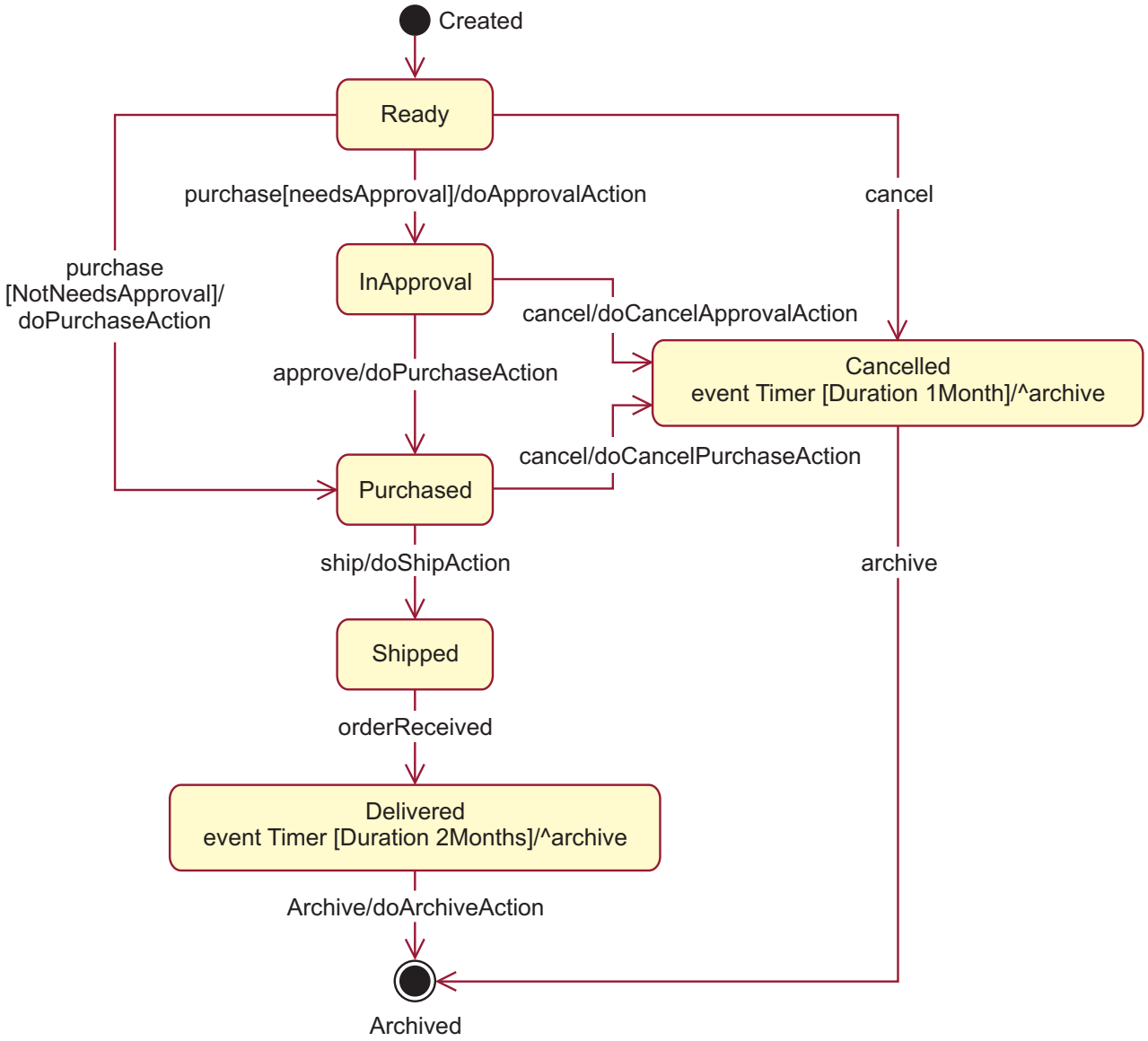


Figure 8. State machine

A state machine cannot be used in a mediation module. It can only be deployed to a WebSphere Process Server.

Business rules

Business rules complement business processes and state machines. If there is condition with a variable, for example, a *business rule* can change the value in that variable at run time. Created by a visual programming language, a business rule makes a decision based on context. The decision can be simple or complex. Business rules are nonprocedural and the rules can be changed independently of an application.

Business rules determine the outcome of a process based on a context. Business rules are used in everyday business situations to make a decision given a specific set of circumstances. This decision may require many rules to cover all the circumstances. Business rules within a business process allow applications to respond quickly to changing business conditions. In an insurance corporation, for example, a business rule for approving car insurance to an applicant could be: *If the applicant is male and over 25 years old, and the car category is sports, and he has been insured with us for the past 5 years, then approve the application for insurance at a fee of $100 per month.*

WebSphere Integration Developer offers a number of approaches to creating business rules. You can create if-then rules or decision tables, all which shape the outcome of your process. These rules are independent of the process itself, meaning that you can change the rules at any time without having to redo your process. For example, based on where your business is located, you might have a rule that says: *If the date is between December 26th and January 1st, then offer a post-holiday sale discount of 20%*. However, if sales continue to be too slow, you could at any time modify the discount to 40%.

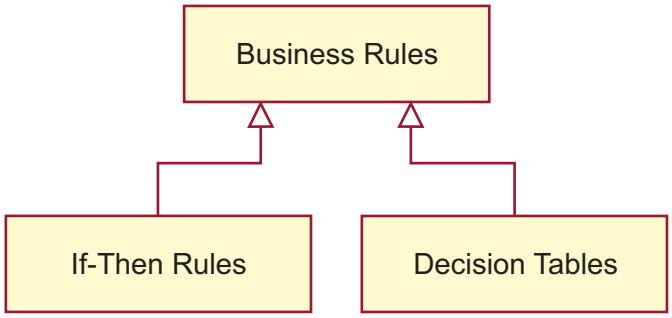


Figure 9. Types of business rules

Business rules cannot be used in a mediation module. They can only be deployed to a WebSphere Process Server.

Selectors

Integrated applications contain many ways to interact. A *selector* is used to route an operation from a client application to one of several possible components for implementation.

Routing to a component is based on dates. For example, here is one route based on a date: *Two weeks before school starts, offer a back-to-school special price on our school-related merchandise.* Businesses have many such routes based on dates. A selector decides to choose one route over another at run time based on a date. For example, if the time is just before school starts, then the previous back-to-school offer would be called. However, if it is the season when school is ending, there could be an offer to prepare children for summer.

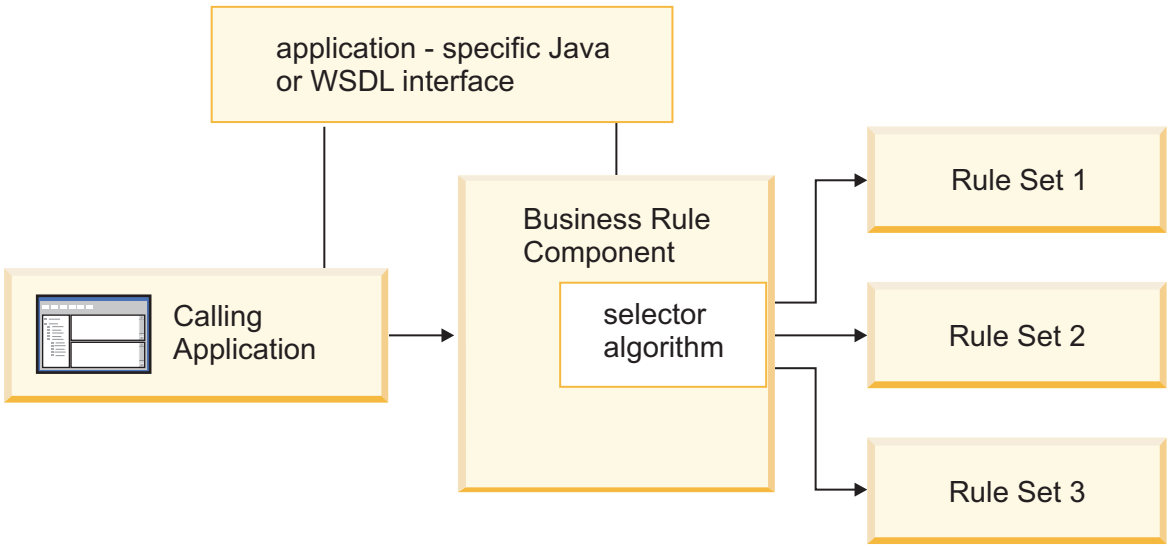


Figure 10. Selecting from a set of business rules

A selector cannot be used in a mediation module. It can only be deployed to a WebSphere Process Server.

Human task

A *human task* component implements a task done by a person. It represents the involvement of a person in a business process.

Occasionally, people need to intervene in a business process. For example, a customer wants to purchase an item that is above their credit limit. A human task lets you intervene and override a business rule that prevents the customer from making the purchase. A human task can have attributes, such as setting the owner of the task, and providing an escalation process in the case that the human specified is not available. The human task component recognizes the reality that many processes require human intervention for tasks like reviewing, researching, and approving.

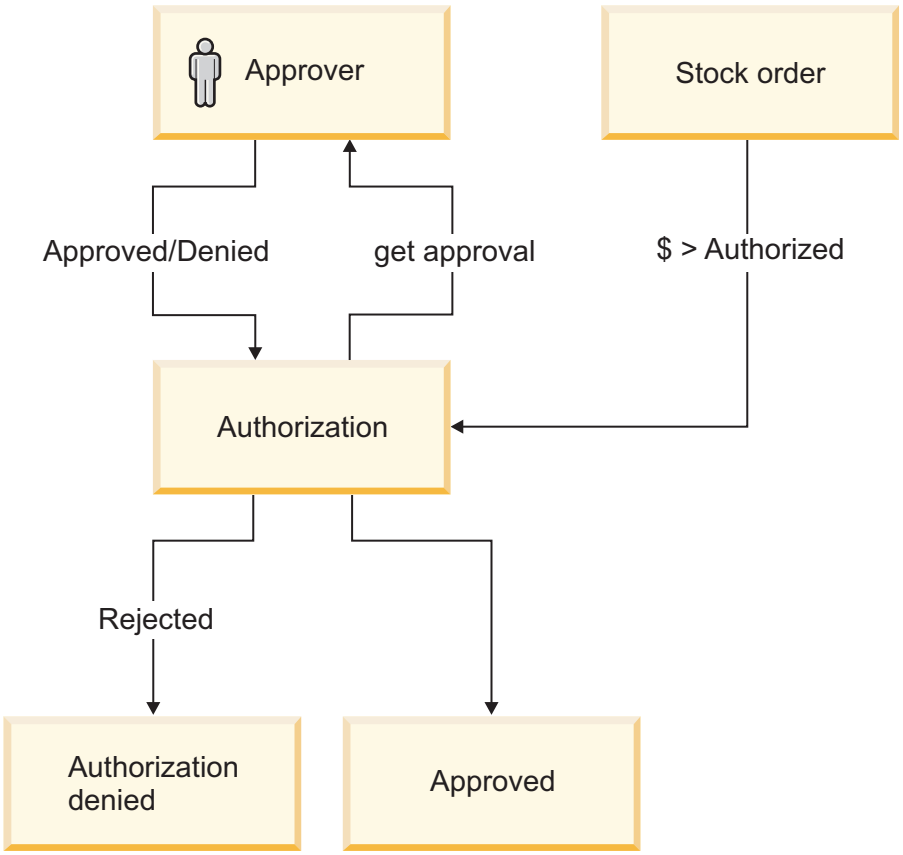


Figure 11. Human task component

A human task cannot be used in a mediation module. It can only be deployed to a WebSphere Process Server.

Interface map

An *interface map* resolves differences between the interfaces of interacting components.

Note: Interface maps are deprecated as of WebSphere Process Server version 7.0. You can migrate your existing interface map components in WebSphere Integration Developer to use the functions in the mediation flow component.

Differences between interfaces in components that need to interact with one another are common. These differences arise because in WebSphere Integration Developer you are often assembling components that were created for different applications. Reusing them to create an application is one of WebSphere Integration Developer's strengths, since otherwise you would be recoding similar components. But you typically must make some adjustments.

For example, two components can have methods that perform basically the same action but have different names such as getCredit and getCreditRating. They also may have different operation names and the operations may have different parameter types. An interface map maps the operations and parameters of these methods so that the differences are resolved and the two components can interact. An interface map is like a bridge between the interfaces of two components allowing them to be wired together despite differences.

An interface map exists independent of the components using it, which means the components themselves do not need to be changed.

An interface map cannot be used in a mediation module. It can only be deployed to a WebSphere Process Server.

Mediation flow

Mediation is a way of mediating or intervening dynamically between services. A *mediation flow* implements a mediation.

Mediation has several useful functions. For example, you can use mediation when you need to transform data from one service into an acceptable format for a subsequent service. Logging lets you log messages from a service before they are sent to the next service. Routing lets you route data from one service into an appropriate service determined by the mediation flow. A mediation operates independently of the services it connects to. A mediation in the assembly editor appears as a mediation flow component between exports and imports.

In the diagram that follows, three service requesters or exports send their output data to the interface of the mediation flow component. The mediation flow component then routes the appropriate data to two service providers or imports.

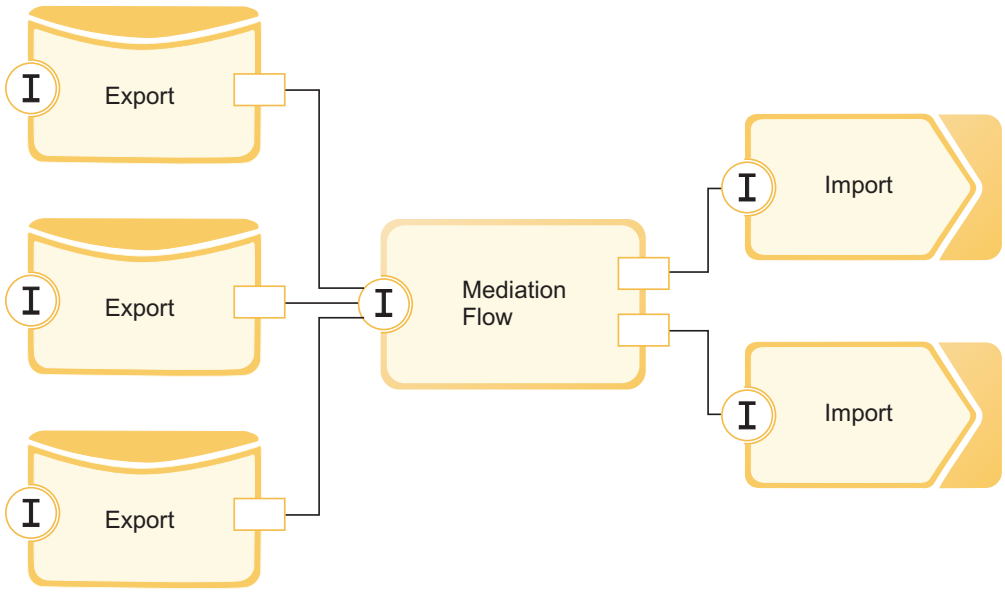


Figure 12. Mediation flow component between three service requesters or exports and two service providers or imports

A mediation flow is a flow-like construct created with the mediation flow editor. Selecting a mediation flow component in the assembly editor, launches the mediation flow editor. In the mediation flow editor, an operation from one service, the service requester or export, is mapped to the operation of another service, the service provider or import, along with functions provided by the mediation flow editor. These functions are called *mediation primitives* and are wired in a mediation flow as shown in the following diagram. Mediation primitives are IBM-supplied or you can create your own custom primitives. Mediation primitives can act on both message content and message context, where context is binding-specific information such as SOAP or JMS headers, or user-defined properties.

In the diagram that follows an operation, applyforLoan, sends a message first to a logging primitive, Log, that records the message. Log sends the message to the Filter primitive, which, depending on the message, routes the message to either a processBusinessLoan operation or a processPersonalLoan operation.

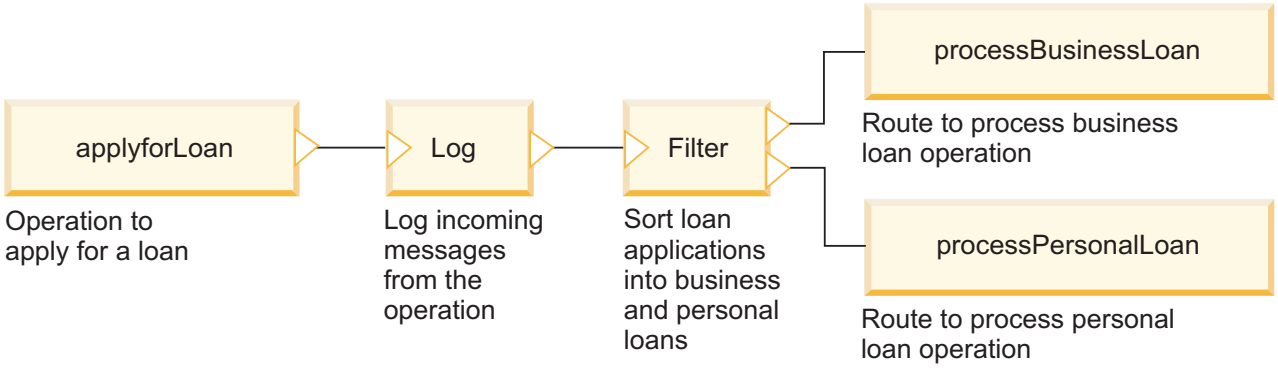


Figure 13. Mediation flow between operations

As discussed in the Modules section, mediation flows can be in either a module or a mediation module. Both types of modules can contain one or more mediation flow components plus zero or more Java components that augment the mediation flow component. A module can be deployed to a WebSphere Process Server. A mediation module can be deployed to either a WebSphere Process Server or a WebSphere Enterprise Service Bus server.

Service qualifiers

An application communicates its quality of service (QoS) needs to the runtime environment by specifying *service qualifiers*. They govern the interaction between a service client and a target service.

Qualifiers can be specified on service component references, interfaces, and implementations. Since declaration of the QoS values is external to an implementation, you can change these values without changing the implementation, or set them differently when several instances of the same implementation are used in different contexts.

These are the categories of qualifiers:

- Transaction - rules for the type of transaction
- Activity session - rules for joining the active session
- Security - rules for permission
- Asynchronous reliability - rules for asynchronous message delivery

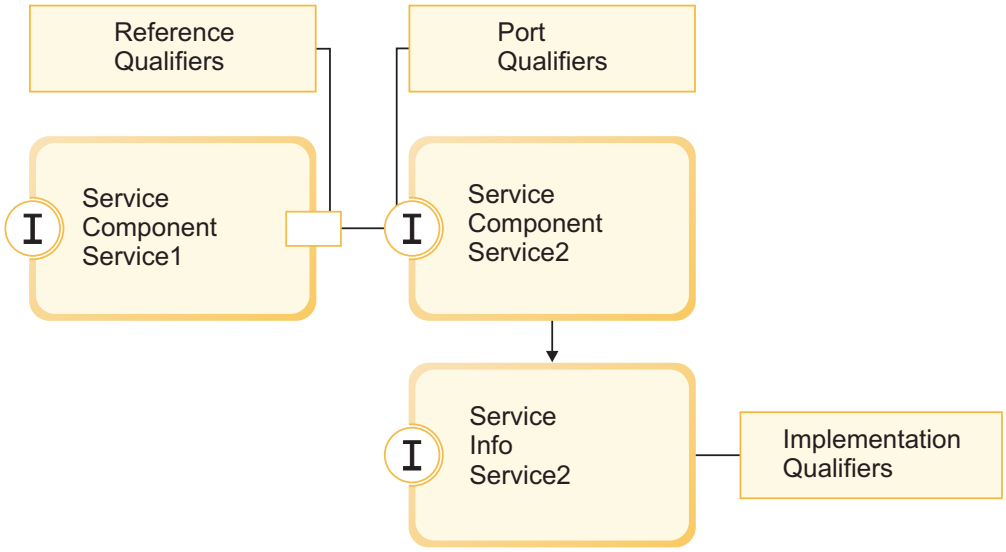


Figure 14. Qualifiers

Stand-alone references

Stand-alone references are references to applications that are not defined as Service Component Architecture components (for example, JavaServer Pages or servlets). Stand-alone references permit these applications to interact with Service Component Architecture components.

Stand-alone references have neither an interface or an implementation (as the implementation is outside the scope of the module). A module can contain no stand-alone references or one stand-alone references artifact. Stand-alone references have the practical value of allowing you to use your existing applications together with Service Component Architecture components created in WebSphere Integration Developer.

Stand-alone references can be used in a mediation module. They can be deployed to either a WebSphere Process Server or a WebSphere Enterprise Service Bus server.

Business objects

Business objects complement Service Component Architecture. Service Component Architecture defines the services as components and the connectivity between them. *Business objects* define the data flowing between components.

Each component passes information as input and output. When a service is invoked, data objects are passed as an XML document with document literal encoding when using a WSDL port type or as a Java object when using a Java interface. Data objects are the preferred form for data and metadata in Service Component Architecture services. Like components, business objects separate the data object from its implementation. For example, a component interacts with purchase orders while the purchase order itself might use JDBC, EJB, and so on, to perform the updates to the data. Business objects let the integration developer focus on working with business artifacts. In fact, service data objects are transparent to the integration developer. They are defined by a service data objects Java Specification Request (JSR).

In Figure 15, business objects are passed from an external service to an export, from an export to a component, from a component to a component, from a component to an import, and from an import to a service. Imports and exports are discussed in the Bindings section.

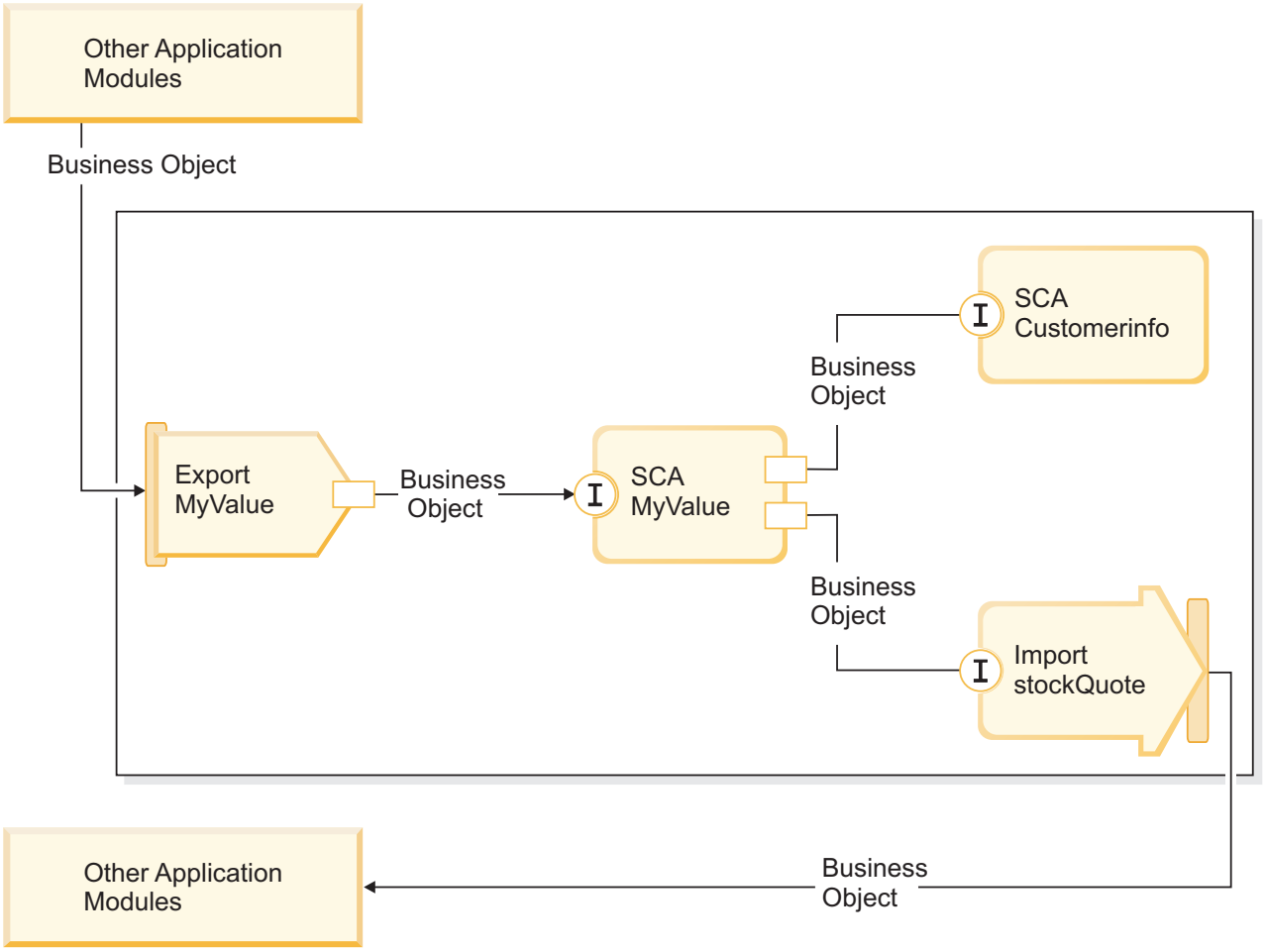


Figure 15. Business objects

Business objects

The computer software industry has developed several programming models and frameworks in which *business objects* provide a natural representation of the business data for application processing.

In general, these business objects:

- Are defined using industry standards
- Transparently map data to database tables or enterprise information systems
- Support remote invocation protocols
- Provide the data programming model foundation for application programming

From a tooling perspective, WebSphere Integration Developer provides developers with one such common business object model for representing different kinds of

business entities from different domains. At development time, this model enables developers to define business objects as XML schema definitions.

At run time, the business data defined by the XML schema definitions is represented as Java Business Objects. In this model, business objects are loosely based on early drafts of the Service Data Object (SDO) specification and provide the complete set of programming model application interfaces required to manipulate business data.

Defining business objects

You define business objects using the business object editor in WebSphere Integration Developer. The business object editor stores the business objects as XML schema definitions.

Using XML schema to define business objects provides several advantages:

- XML schema provide a standards-based data definition model and a foundation for interoperability between disparate heterogeneous systems and applications. XML schema are used in conjunction with the Web Services Description Language (WSDL) to provide standards-based interface contracts among components, applications, and systems.
- XML schema define a rich data definition model for representing business data. This model includes complex types, simple types, user-defined types, type inheritance, and cardinality, among other features.
- Business objects can be defined by business interfaces and data defined in the Web Services Description Language, as well as by XML schema from industry standards organizations or from other systems and applications. WebSphere Integration Developer can import these business objects directly.

WebSphere Integration Developer also provides support for discovering business data in databases and enterprise information systems and then generating the standards-based XML schema business object definition of that business data. Business objects generated in this fashion are often referred to as *application specific business objects* because they mimic the structure of the business data defined in the enterprise information system.

When a process is manipulating data from many different information systems, it can be valuable to transform the disparate representation of business data (for example, CustomerEIS1 and CustomerEIS2 or OrderEIS1 and OrderEIS2) into a single canonical representation (for example, Customer or Order). The canonical representation is often referred to as the *generic business object*.

Business object definitions, particularly for generic business objects, are frequently used by more than one application. To support this reuse, WebSphere Integration Developer allows business objects to be created in libraries that can then be associated with multiple application modules.

The contracts for the services provided and consumed by a Service Component Architecture (SCA) application module as well as the contracts used to create the components within an application module are defined using the Web Services Description Language. A WSDL can represent both the operations and business objects, which are defined by XML schema to represent the business data, of a contract.

Working with business objects

Service Component Architecture (SCA) provides the framework for defining an application module, the services it provides, the services it consumes, and the composition of components that provide the business logic of the application module. Business objects play an important role in the application, defining the business data that is used to describe the service and component contracts and the business data that the components manipulate.

The following diagram depicts an SCA application module and illustrates many of the places in which the developer works with business objects.

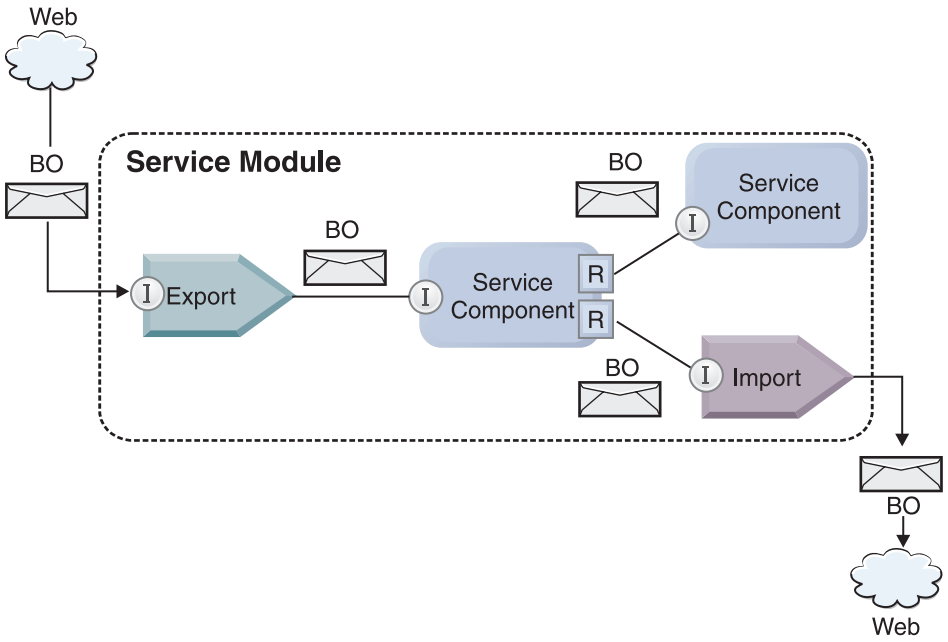


Figure 16. Business objects represent the data that flows between services in an application

Note: This topic describes how business objects are used by SCA application modules. If you are using Java interfaces, the SCA application modules can also process Java objects.

Business object programming model

The business object programming model consists of a set of Java interfaces that represent:

- The business object definition and instance data
- A set of services that support the operations on the business objects

Business object type definitions are represented by the commonj.sdo.Type and commonj.sdo.Property interfaces. The business object programming model provides a set of rules for mapping the XML schema complex type information to the Type interface and each of the elements in the complex type definition to the Property interface.

Business object instances are represented by the commonj.sdo.DataObject interface. The business object programming model is untyped, which means that the same commonj.sdo.DataObject interface can be used to represent different business object definitions, such as Customer and Order. The definition of which properties can be

set and retrieved from each business object is determined by type information defined in the XML schema associated with each business object.

The business object programming model behavior is based on the Service Data Object 2.1 specification. For additional information, see the SDO 2.1 for Java specification, tutorials and javadocs on the Web: http://osoa.org/display/Main/Service+Data+Objects+Specifications.

Business object services support various lifecycle operations (such as creation, equality, parsing, and serialization) on business objects.

For specifics on the business object programming model, see Programming using business object services and Package com.ibm.websphere.bo.

Bindings, data bindings, and data handlers

As shown in Figure 16 on page 19, business data that is used to invoke services provided by SCA application modules is transformed into business objects so that the SCA components can manipulate the business data. Similarly, the business objects manipulated by SCA components are converted into the data format required by the external services.

In some cases, such as the web service binding, the binding used to export and import services automatically transforms the data into the appropriate format. In other cases, such as the JMS binding, developers can provide a data binding or data handler that converts non-native formats into business objects represented by the DataObject interface.

For more information on developing data bindings and data handlers, refer to Data handlers and Data bindings.

Components

SCA components define their provision and consumption service contracts using a combination of the Web Services Description Language and XML schema. The business data that SCA passes between components is represented as business objects using the DataObject interface. SCA verifies that these business object types are compatible with the interface contract defined by the component to be invoked.

The programming model abstractions for manipulating business objects vary from component to component. The POJO component and the mediation flow component Custom primitive provide direct manipulating of the business objects by enabling Java programming directly using the business object programming interfaces and services. Most components provide higher level abstractions for manipulating business objects, but also provide snippets of Java code for defining custom behavior in the business object interfaces and services.

Business objects can be transformed using either the combination of the Interface Flow Mediation and Business Object Map component or the mediation flow component and its XML Map primitive. These business object transformation capabilities are useful for converting application specific business objects to and from generic business objects.

Special business objects

Service message objects and business graphs are two specialized types of business objects that are used for specific application purposes.

Service message object

A service message object (SMO) is a specialized business object that is used by mediation flow components to represent the collection of data associated with a service invocation.

A SMO has a fixed top-level structure consisting of headers, context, body, and attachments (if present).

- Headers carry information related to the service invocation over a particular protocol or binding. Examples are SOAP headers and JMS headers.
- Context data carries additional logical information associated with the invocation while it is being processed by the mediation flow component. This information is typically not part of the application data sent or received by clients.
- The body of the SMO carries the payload business data, which represents the core application message or invocation data in the form of a standard business object.

The SMO can also carry attachment data for Web service invocations using SOAP with attachments.

Mediation flows perform such tasks as request routing and data transformation, and the SMO provides the combined view of header and payload contents in a single unified structure.

Business graph

A business graph is a special business object used to provide support for data synchronization in integration scenarios.

Consider an example in which two enterprise information systems have a representation of a specific order. When the order changes in one system, a message can be sent to the other system to synchronize the order data. Business graphs support the notion of sending just the portion of the order that changed to the other system and annotating it with change-summary information to define the type of change.

In this example, an Order business graph would convey to the other system that one of the line items in the order was deleted and that the projected ship date property of the order was updated.

Business graphs can easily be added to existing business objects in WebSphere Integration Developer. They are most frequently found in scenarios in which WebSphere adapters are being used and to support the migration of WebSphere InterChange Server applications.

Business object parsing mode

WebSphere Integration Developer provides a property on modules and libraries you can use to configure XML parsing mode for business objects to either eager or lazy.

- If the option is set to *eager*, XML byte streams are eagerly parsed to create the business object.
- If the option is set to *lazy*, the business object is created normally, but the actual parsing of the XML byte stream is deferred and partially parsed only when the business object properties are accessed.

In either XML parsing mode, non-XML data is always eagerly parsed to create the business object.

Benefits of using lazy versus eager parsing mode

Some applications benefit from lazy XML parsing mode while others see improved performance with eager parsing mode. It is recommended that you benchmark your application in both parsing modes to determine which mode best suites the specific characteristics of your application.

The following section provides general guidance about the types of applications that benefit from each type of parsing mode:

- Applications benefiting from lazy XML parsing mode

  Applications that parse large XML data streams are likely to see performance improvements when the lazy XML parsing mode is used. The performance benefits increase as the size of the XML byte stream increases and the amount of data from the byte stream that is accessed by the application decreases.

  Note: The business object lazy parsing mode is supported in WebSphere Process Server 7.0.0.3 and later versions. Modules and mediation modules that include mediation flow components are not supported.
- Applications benefiting from eager parsing mode

  The following applications are likely to perform better in eager parsing mode:
  - Applications that parse non-XML data streams
  - Applications that are created using the BOFactory service
  - Applications that parse very small XML messages

Application migration and development considerations

If you are configuring an application that was originally developed using eager parsing mode to now use lazy parsing mode, or if you are planning to switch an application between lazy and eager parsing mode, be aware of the differences between modes and the considerations when switching modes.

Error handling

If the XML byte stream being parsed is ill-formed, parsing exceptions occur.

- In eager XML parsing mode, those exceptions occur as soon as the business object is parsed from the inbound XML stream.
- If lazy XML parsing mode is configured, the parsing exceptions occur latently when the business object properties are accessed and the portion of the XML that is ill-formed is parsed.

To deal with ill-formed XML, select one of the following options:

- Deploy an enterprise service bus on the edges to validate inbound XML
- Author lazy error-detection logic at the point where business object properties are accessed

Exception stacks and messages

Because the eager and lazy XML parsing modes have different underlying implementations, stack traces thrown by the business object programming interfaces and services have the same exception class name, but they might not contain the same exception message or wrapped set of implementation-specific exception classes.

XML serialization format

The lazy XML parsing mode provides a performance optimization that attempts to copy unmodified XML from the inbound byte stream to the outbound byte stream upon serialization. The result is increased performance, but the serialization format of the outbound XML byte stream might be different if the entire business object was updated in lazy XML parsing mode or if it was running in eager XML parsing mode.

Although the XML serialization format might not be precisely syntactically equivalent, the semantic value provided by the business object is equivalent independent of the parsing modes, and XML can be safely passed between applications running in different parsing modes with semantic equivalence.

Business object instance validator

The lazy XML parsing business object mode instance validator provides a higher fidelity validation of business objects, particularly facet validation of property values. Because of these improvements, the lazy parsing mode instance validator catches additional issues that are not caught in eager parsing mode and provides more detailed error messages.

Version 602 XML Maps

Mediation flows originally developed before WebSphere Integration Developer Version 6.1 might contain XSLT primitives that use a map or stylesheet that cannot process directly in lazy XML parsing mode. When an application is migrated for use in lazy XML parsing mode, map files associated with XSLT primitives can be automatically updated by the migration wizard to run in the new mode. However, if an XSLT primitive refers directly to a stylesheet that has been edited manually, the stylesheet is not migrated and cannot process in lazy XML parsing mode.

Private unpublished APIs

If an application is taking advantage of unpublished, private, implementation-specific business object programming interfaces, the application is likely to fail compilation when the parsing mode is switched. In eager parsing mode, these private interfaces are typically business object implementation classes defined by the Eclipse Modeling Framework (EMF).

In all cases, it is recommended that private APIs be removed from the application.

Service Message Object EMF APIs

A mediation component in WebSphere Process Server provides the ability to manipulate message content using the Java classes and interfaces provided in the com.ibm.websphere.sibx.smobo package. In lazy XML parsing mode, the Java interfaces in the com.ibm.websphere.sibx.smobo package can still be used, but

methods that refer directly to Eclipse Modeling Framework (EMF) classes and interfaces or that are inherited from EMF interfaces are likely to fail.

The ServiceMessageObject and its contents cannot be cast to EMF objects in lazy XML parsing mode.

BOMode service

The BOMode service is used to determine whether the currently processing XML parsing mode is eager or lazy.

Migration

All applications before version 7.0.0.0 are running in eager XML parsing mode. When they are runtime migrated using the BPM runtime migration tools, they continue to run in eager XML parsing mode.

To enable an application earlier than version 7.0.0.0 to be configured to use the lazy XML parsing mode, you first use WebSphere Integration Developer to migrate the artifacts of the application. After migration, you then configure the application to use lazy XML parsing.

See Migrating source artifacts for information about migrating artifacts in WebSphere Integration Developer, and see Configuring the business object parsing mode of modules and libraries for information about setting the parsing mode.

Business object property type `QName`

You must modify the application code to work with business objects that contain property of type `QName` if you want your application that used eager parsing to use lazy parsing. In eager parsing mode, WebSphere Process Server uses the Java class `org.eclipse.emf.ecore.xml.type.internal.QName` to define the `QName` type property value. Lazy parsing mode uses the Java class `javax.xml.namespace.QName` to set value for the `QName` type property. Modify the application code when changing the mode from Eager to Lazy for a module by replacing the reference to Java class `org.eclipse.emf.ecore.xml.type.internal.QName` with `javax.xml.namespace.QName`.

Relationships

A relationship is an association between two or more data entities, typically business objects. Relationships can be used to transform data that is equivalent across business objects and other data but that is represented differently, or they can be used to draw associations across different objects found in different applications. They can be shared across applications, across solutions, and even across products.

The relationship service in WebSphere Process Server provides the infrastructure and operations for managing relationships. Because it enables you to deal with business objects regardless of where they reside, it can provide a unified holistic view across all applications in an enterprise, and serve as a building block for BPM solutions. Because relationships are extensible and manageable, they can be used in complex integration solutions.

What are relationships?

A relationship is an association between business objects. Each business object in a relationship is called a *participant* in the relationship. Each participant in the relationship is distinguished from other participants based on the function, or *role*, it serves in that relationship. A relationship contains a list of roles.

The relationship *definition* describes each role and specifies how the roles are related. It also describes the overall "shape" of the relationship. For example, this role can have only one participant, but this other role can have as many participants as necessary. You might define a *car-owner* relationship, for instance, where one owner might own multiple cars. For example, one instance could have the following participants for each of these roles:

- Car (Ferrari)
- Owner (John)

The relationship definition is a template for the relationship *instance*. The instance is the run-time instantiation of the relationship. In the *car-owner* example above, an instance might describe any of the following associations:

- John owns Ferrari
- Sara owns Mazda
- Bob owns Ferrari

Using relationships frees you from the need to custom build relationship tracking persistence within your business logic. For certain scenarios, the relationship service does all the work for you. See the example described in the section on Identity relationships.

Scenarios

Here is a typical example of a situation in which an integration solution might use relationships. A large corporation buys multiple companies, or business units. Each business unit uses different software to monitor personnel and laptops. The company needs a way to monitor its employees and their laptops. It wants a solution that enables them to:

- View all the employees in the various business units as if they were in one database
- Have a single view of all their laptops
- Allow employees to log on to the system and buy a laptop
- Accommodate the different enterprise application systems in the various business units

To accomplish this, the company needs a way to ensure, for example, that John Smith and John A. Smith in different applications are seen as the same employee. For Example, they need a way to consolidate a single entity across multiple application spaces.

More complex relationship scenarios involve building business processes that draw relationships across different objects found in multiple applications. With complex relationship scenarios, the business objects reside in the integration solution, and not in the applications. The relationship service provides a platform for managing relationships persistently. Before the relationship service, you would have to build your own object persistence service. Two examples of complex relationship scenarios are:

- You have a `car` business object with a VIN number in an SAP application, and you want to track the fact that this car is owned by someone else. However, the ownership relationship is with someone in a PeopleSoft application. In this pattern of relationships, you have two solutions and you need to build a cross-bridge between them.
- A large retail company wants to be able to monitor merchandise returned for cash back or credit. There are two different applications involved: an order management system (OMS) for purchases, and a returns management system (RMS) for returns. The business objects reside in more than one application, and you need a way to show the relationships that exist between them.

Common usage patterns

The most common relationship patterns are *equivalence* patterns. These are based on cross-referencing, or correlation. There are two types of relationships that fit this pattern: *non-identity* and *identity*.

- **Non-identity relationships** establish associations between business objects or other data on a one-to-many or many-to-many basis. For each relationship instance, there can be one or more instances of each participant. One type of non-identity relationship is a static lookup relationship. An example of this is a relationship in which `CA` in an SAP application is related to `California` in a Siebel application.
- **Identity relationships** establish associations between business objects or other data on a one-to-one basis. For each relationship instance, there can be only one instance of each participant. Identity relationships capture cross-references between business objects that are semantically equivalent, but that are identified differently within different applications. Each participant in the relationship is associated with a business object that has a value (or combination of values) that uniquely identifies the object. Identity relationships typically transform the key attributes of business objects, such as ID numbers and product codes.

  For example, if you have `car` business objects in SAP, PeopleSoft, and Siebel applications, and you want to build a solution that synchronizes them, you would normally need to introduce hand-built relationship synchronization logic in six maps:

  SAP -> generic

  generic -> SAP

  PeopleSoft-> generic

  generic-> PeopleSoft

  Siebel-> generic

  generic-> Siebel

  However, if you use relationships in your solution, the relationship service provides prebuilt pattern implementations that maintains all these relationship instances for you.

Tools for working with relationships

The *relationship editor* in WebSphere Integration Developer is the tool you use to model and design business integration relationships and roles. For detailed background and task information about creating relationships and using the relationship editor, refer to the WebSphere Integration Developer Information Center.

The *relationship service* is an infrastructure service in WebSphere Process Server that maintains relationships and roles in the system and provides operations for relationship and role management.

The *relationship manager* is the administrative interface for managing relationships. It is accessed through the Relationship Manager pages of the administrative console.

Relationships can be invoked programmatically through the relationship service APIs.

Relationship service

The relationship service stores relationship data in relationship tables, where it keeps track of application-specific values across applications and across solutions. The relationship service provides operations for relationship and role management.

How relationships work

Relationships and roles are defined using the graphical interface of the relationship editor tool in WebSphere Integration Developer. The relationship service stores the correlation data in tables in the relationship database in the default data source that you specify when you configure the relationship service. A separate table (sometimes called a participant table) stores information for each participant in the relationship. The relationship service uses these relationship tables to keep track of the related application-specific values and propagate updated information across all the solutions.

Relationships, which are business artifacts, are deployed within a project or in a shared library. At the first deployment, the relationship service populates the data.

At run time, when maps or other WebSphere Process Server components need a relationship instance, the instances of the relationship are either updated or retrieved, depending on the scenario.

Relationship and role instance data can be manipulated through three means:

- WebSphere Process Server component Java snippet invocations of the relationship service APIs
- Relationship transformations in the WebSphere Process Server business object mapping service
- The relationship manager tool

For detailed background and task information on creating relationships, identifying relationship types, and using the relationship editor, refer to the WebSphere Integration Developer Information Center.

Relationship manager

The relationship manager is the administrative interface for managing relationships. It is accessed through the Relationship Manager pages of the administrative console.

The relationship manager provides a graphical user interface for creating and manipulating relationship and role data at run time. You can manage relationship entities at all levels: relationship instance, role instance, and attribute data and property data levels. With the relationship manager, you can:

- View a list of the relationships in the system and detailed information for individual relationships
- Manage relationship instances:
  - Query relationship data to view subsets of instance data
  - Query relationship data to view subsets of instance data using database views
  - View a list of relationship instances that match a relationship query and detailed information about an instance
  - Edit the property values for a relationship instance
  - Create and delete relationship instances
- Manage roles and role instances:
  - View details about a role or a role instance
  - Edit role instance properties
  - Create and delete role instances for a relationship
  - Roll back relationship instance data to a point in time when you know the data is reliable
- Import data from an existing static relationship into your system, or export data from an existing static relationship to an RI or CSV file
- Remove relationship schema and data from the repository when the application that uses it is uninstalled

Relationships in Network Deployment environments

Relationships can be used in Network Deployment (ND) environments without any extra configuration.

In Network Deployment (ND) environments, relationships are installed in an application cluster. Relationships are then visible within the cluster, and all servers in the cluster have access to the instance data stored in the relationship database. The ability to run the relationship service in an ND environment makes it scalable and highly available.

The relationship manager allows relationships to be managed across different clusters through a centralized administrative interface. You connect the relationship manager to a server in a cluster by selecting its relationship MBean.

Relationship service APIs

Relationships can be invoked programmatically through the relationship service APIs, within or outside of business object maps.

Three API types are available:

- Relationship instance manipulation APIs (including create, update, delete instance data directly)
- Relationship pattern support APIs (including correlate(), correlateforeignKeyLookup)
- Relationship lookup patterns (lookup APIs)

The enterprise service bus in WebSphere Process Server

WebSphere Process Server supports the integration of application services, including the same capabilities as WebSphere Enterprise Service Bus.

Connecting services through an enterprise service bus

With an enterprise service bus (ESB), you can maximize the flexibility of an SOA. Participants in a service interaction are connected to the ESB, rather than directly to one another.

When the service requester connects to the ESB, the ESB takes responsibility for delivering its requests, using messages, to a service provider offering the required function and quality of service. The ESB facilitates requester-provider interactions and addresses mismatched protocols, interaction patterns, or service capabilities. An ESB can also enable or enhance monitoring and management. The ESB provides virtualization and management features that implement and extend the core capabilities of SOA.

The ESB abstracts the following features:

Location and identity
Participants need not know the location or identity of other participants. For example, requesters need not be aware that a request could be serviced by any of several providers; service providers can be added or removed without disruption.

Interaction protocol
Participants need not share the same communication protocol or interaction style. For example, a request expressed as SOAP over HTTP can be serviced by a provider that only understands SOAP over Java Message Service (JMS).

Interface
Requesters and providers need not agree on a common interface. An ESB reconciles differences by transforming request and response messages into a form expected by the provider.

Requesters and providers need not agree on a common interface
An ESB reconciles differences by transforming request messages into a form expected by the provider.

Qualities of (interaction) service
Participants, or systems administrators, declare their quality-of-service requirements, including authorization of requests, encryption and decryption of message contents, automatic auditing of service interactions, and how their requests should be routed (for example, optimizing for speed or cost).

Interposing the ESB between participants enables you to modulate their interaction through a logical construct called a *mediation*. Mediations operate on messages in-flight between requesters and providers. For example, mediations can be used to find services with specific characteristics that a requester is asking for, or to resolve interface differences between requesters and providers. For complex interactions, mediations can be chained sequentially.

An enterprise service bus, with mediations, performs the following actions between requester and service:

- *Routing* messages between services. An enterprise service bus offers a common communication infrastructure that can be used to connect services, and thereby the business functions they represent, without the need for programmers to write and maintain complex connectivity logic.

- *Converting* transport protocols between requester and service. An enterprise service bus provides a consistent, standards-based way to integrate business functions that use different IT standards. This enables integration of business functions that could not normally communicate, such as to connect applications in departmental silos or to enable applications in different companies to participate in service interactions.
- *Transforming* message formats between requester and service. An enterprise service bus enables business functions to exchange information in different formats, with the bus ensuring that the information delivered to a business function is in the format required by that application.
- *Handling* business events from disparate sources. An enterprise service bus supports event-based interactions in addition to the message exchanges to handle service requests.

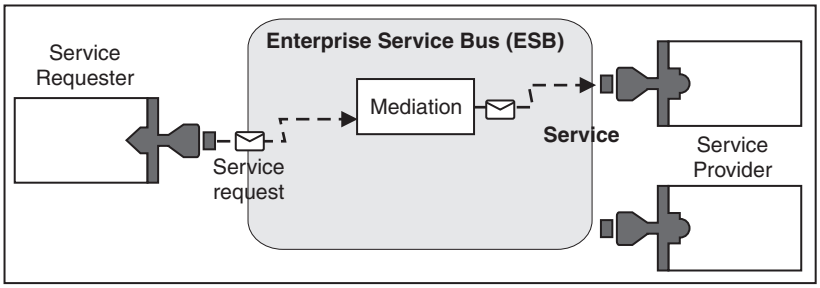


Figure 17. An enterprise service bus. The enterprise service bus is routing messages between applications, which are requesters or providers of services. The bus is converting transport protocols and transforming message formats between requesters and providers. In this figure, each application uses a different protocol (represented by the different geometric shapes of their connectors) and uses different message formats.

By using the enterprise service bus you can concentrate focus on your core business rather than your computer systems. You can change or add to the services when you need to; for example, to respond to changes in the business requirement, to add extra service capacity, or to add new capabilities. You can make the required changes by reconfiguring the bus, with little or no impact to existing services and applications that use the bus.

Enterprise service bus messaging infrastructure

WebSphere Process Server includes enterprise service bus capabilities. WebSphere Process Server supports the integration of service-oriented, message-oriented, and event-driven technologies to provide a standards-based, messaging infrastructure in an integrated enterprise service bus.

The enterprise service capabilities that you can use for your enterprise applications provide not only a transport layer but mediation support to facilitate service interactions. The enterprise service bus is built around open standards and service-oriented architecture (SOA). It is based on the robust Java EE infrastructure and associated platform services provided by IBM WebSphere Application Server Network Deployment.

WebSphere Process Server is powered by the same technology available with IBM WebSphere Enterprise Service Bus. This capability is part of the underlying functionality of WebSphere Process Server, and no additional license for WebSphere Enterprise Service Bus is required to take advantage of these capabilities.

However, you can deploy additional stand-alone licenses of WebSphere Enterprise Service Bus around your enterprise to extend the connectivity reach of the process integration solutions powered by WebSphere Process Server. For example, WebSphere Enterprise Service Bus can be installed closer to an SAP application to host an IBM WebSphere Adapter for SAP and to transform SAP messages before sending that information across the network to a business process choreographed by WebSphere Process Server.

You can deploy WebSphere Enterprise Service Bus around your enterprise to extend the connectivity reach of the process integration solutions powered by separate installations of WebSphere Process Server or other integration solutions as part of a federated ESB. For example, WebSphere Enterprise Service Bus can be installed closer to an SAP application to host an IBM WebSphere Adapter for SAP and to transform SAP messages before sending that information across the network to a business process choreographed by WebSphere Process Server.

Messaging or queue destination hosts

A messaging or queue destination host provides the messaging function within a server. A server becomes the messaging destination host when you configure it as the messaging target.

A messaging engine runs within a server. The messaging engine provides messaging functions and a connection point for applications to connect to the bus. Service Component Architecture (SCA) asynchronous communication, JMS imports and exports, asynchronous internal processing use message queues on the messaging engine.

The deployment environment connects the message source to the message target through the bus when the application modules are deployed. Knowing the message source and message target helps you determine what type of deployment environment you need.

Applications can store persistent data in a data store, which is a set of tables in a database or schema, or in a file store. The messaging engine uses an instance of a JDBC data source to interact with that database.

Configure the messaging destination host when you define your deployment environment by using **Server** from the administrative console or designate the server as the destination host during software installation.

Data stores:

Every messaging engine can use a data store, which is a set of tables in a database or schema that store persistent data.

All of the tables in the data store are held in the same database schema. You can create each data store in a separate database. Alternatively, you can create multiple data stores in the same database, with each data store using a different schema.

A messaging engine uses an instance of a JDBC data source to interact with the database that contains the data store for that messaging engine.

Data sources

Data sources provide the link between applications and relational databases.

Applications use a data source to obtain connections to a relational database. A data source is analogous to the Java EE Connector Architecture (JCA) connection factory, which provides connectivity to other types of enterprise information systems (EIS).

A data source is associated with a JDBC provider, which supplies the driver implementation classes that are required for JDBC connectivity with a specific type of database. Application components transact directly with the data source to obtain connection instances to your database. The connection pool that corresponds to each data source provides connection management.

You can create multiple data sources with different settings, and associate them with the same JDBC provider. For example, you might use multiple data sources to access different databases within the same database application. WebSphere Process Server requires JDBC providers to implement one or both of the following data source interfaces, which are defined by Sun Microsystems. These interfaces enable the application to run in a single-phase or two-phase transaction protocol.

Note: Business Process Choreographer data sources are created using the Business Process Choreographer configuration tools. Refer to Configuring Business Process Choreographer.

- ConnectionPoolDataSource - a data source that supports application participation in local and global transactions, except two-phase commit transactions. When a connection pool data source is involved in a global transaction, transaction recovery is not provided by the transaction manager. The application is responsible for providing the backup recovery process if multiple resource managers are involved.
- XADataSource - a data source that supports application participation in any single-phase or two-phase transaction environment. When this data source is involved in a global transaction, the WebSphere Application Server transaction manager provides transaction recovery.

The following tables provide examples of typical stand-alone and typical deployment environment setups:

Table 2. Typical stand-alone environment setup

Datasource	Component	Scope	JNDI Name
WBI DataSource	CommonDB	Node	jdbc/WPSDB
SCA Application Bus ME data source	SCA ME	Server	jdbc/com.ibm.ws.sib/nlNode01.server1-SCA.APPLICATION.localhostNode01Cell.Bus
Business Process Choreographer data source	BPC	Server	jdbc/BPEDB
Business Process Choreographer ME data source	BPC ME	Server	jdbc/com.ibm.ws.sib/nlNode01.server1-BPC.localhostNode01Cell.Bus
event	CEI	Server	jdbc/cei
CEI ME data source	CEI ME	Server	jdbc/com.ibm.ws.sib/nlNode01.server1-CEI.*cellName*.BUS

Table 3. Typical deployment environment setup

Datasource	Component	Scope	JNDI Name
WBI DataSource	CommonDB	Cell	jdbc/WPSDB
SCA Application Bus ME data source	SCA ME	Cluster	jdbc/com.ibm.ws.sib/clusterone-SCA.APPLICATION.enduranceTestCell01.Bus
Business Process Choreographer data source	BPC	Cluster	jdbc/BPEDB
Business Process Choreographer ME data source	BPC ME	Cluster	jdbc/com.ibm.ws.sib/clusterone-BPC.enduranceTestCell01.Bus
event	CEI	Cluster	jdbc/cei
CEI ME data source	CEI ME	Cluster	jdbc/com.ibm.ws.sib/clusterone-CEI.*cellName*.BUS

For more information on data sources, see “Data sources” in theWebSphere Application Server information center.

JDBC providers:

You can use JDBC providers to interact applications with relational databases.

Applications use JDBC providers to interact with relational databases. The JDBC provider supplies the specific JDBC driver implementation class for access to a specific type of database. To create a pool of connections to that database, you associate a data source with the JDBC provider. Together, the JDBC provider and the data source objects are functionally equivalent to the Java EE Connector Architecture (JCA) connection factory, which provides connectivity with a non-relational database.

Refer to the examples of both Typical stand-alone environment setup and Typical deployment environment setup in the previous topic.

For more information on JDBC providers, see “JDBC providers” in the WebSphere Application Server information center.

Service integration buses for WebSphere Process Server

A service integration bus is a managed communication mechanism that supports service integration through synchronous and asynchronous messaging. A bus consists of interconnecting messaging engines that manage bus resources. It is one of the WebSphere Application Server technologies on which WebSphere Process Server is based.

Some buses are automatically created for use by the system, the Service Component Architecture (SCA) applications that you deploy, and by other components. You can also create buses to support service integration logic or other applications, for example, to support applications that act as service requesters and providers within WebSphere Process Server, or to link to WebSphere MQ.

A bus destination is a logical address to which applications can attach as a producer, consumer, or both. A queue destination is a bus destination that is used for point-to-point messaging.

Each bus can have one or more bus members, each of which is either a server or a cluster.

The *bus topology* is the physical arrangement of application servers, messaging engines, and WebSphere MQ queue managers and the pattern of bus connections and links between them that makes up your enterprise service bus.

Some service integration buses are created automatically to support WebSphere Process Server. Up to four buses are created when you create your deployment environment or configure a server or cluster to support SCA applications. These buses each have three authentication aliases that you must configure.

SCA system bus:

The *SCA system bus* is a service integration bus that is used to host queue destinations for Service Component Architecture (SCA) modules. The SCA run time, which supports mediation modules, uses queue destinations on the system bus as an infrastructure to support asynchronous interactions between components and modules.

The system bus is automatically created when you create a deployment environment or when you configure a server or cluster to support SCA applications. The system bus provides a scope within which resources, such as queue destinations, are configured for mediation modules and interaction endpoints. The bus enables message routing between endpoints. You can specify the quality of service for the bus, including priority and reliability.

The bus name is SCA.SYSTEM.busID.Bus. The authentication alias used for securing this bus is SCA_Auth_Alias.

SCA application bus:

The application bus destinations support the asynchronous communication of WebSphere Business Integration Adapters and other System Component Architecture components.

The application bus is automatically created when you create a deployment environment or when you configure a server or cluster to support SCA applications. The application bus is similar to service integration buses you might create to support service integration logic or other applications.

The bus name is SCA.APPLICATION.busID.Bus. The authentication alias used for securing this bus is SCA_Auth_Alias.

The Common Event Infrastructure bus:

The Common Event Infrastructure bus is used for transmitting common base events, asynchronously, to the configured Common Event Infrastructure server.

The bus name is CommonEventInfrastructure_Bus. The authentication alias used for securing this bus is CommonEventInfrastructureJMSAuthAlias

The Business Process Choreographer bus:

Use the Business Process Choreographer bus name and authentication for internal message transmission.

The Business Process Choreographer bus is used for transmitting messages internally and for business flow manager's Java Messaging Service (JMS) API.

The bus name is BPC.cellName.Bus. The authentication alias is BPC_Auth_Alias

Service applications and service modules

A service module is a Service Component Architecture (SCA) module that provides services in the run time. When you deploy a service module toWebSphere Process Server, you build an associated service application that is packaged as an Enterprise ARchive (EAR) file.

Service modules are the basic units of deployment and can contain components, libraries, and staging modules used by the associated service application. Service modules have exports and, optionally, imports to define the relationships between modules and service requesters and providers. WebSphere Process Server supports modules for business services and mediation modules. Both modules and mediation modules are types of SCA modules. 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.

Deploying a service application

The process of deploying an EAR file containing a service application is the same as the process of deploying any EAR file. You can modify values for mediation parameters at deployment time. After you have deployed an EAR file containing an SCA module, you can view details about the service application and its associated module. You can see how a service module is connected to service requesters (through exports) and service providers (through imports).

Viewing SCA module details

The service module details that you can view depend upon the SCA module. They can include the following attributes.

- SCA module name
- SCA module description
- Associated application name
- SCA module version information, if the module is versioned
- SCA module imports:
  - Import interfaces are abstract definitions that describe how an SCA module accesses a service.
  - Import bindings are concrete definitions that specify the physical mechanism by which an SCA module accesses a service. For example, using SOAP/HTTP.
- SCA module exports:
  - Export interfaces are abstract definitions that describe how service requesters access an SCA module.
  - Export bindings are concrete definitions that specify the physical mechanism by which a service requester accesses an SCA module, and indirectly, a service.
- SCA module properties

Imports and import bindings

Imports define interactions between Service Component Architecture (SCA) modules and service providers. SCA modules use imports to permit components to access external services (services that are outside the SCA module) using a local representation. Import bindings define the specific way that an external service is accessed.

If SCA modules do not need to access external services, they are not required to have imports. Mediation modules usually have one or more imports that are used to pass messages or requests on to their intended targets.

Interfaces and bindings

An SCA module import needs at least one interface, and an SCA module import has a single binding.

- Import interfaces are abstract definitions that define a set of operations using Web Services Description Language (WSDL), an XML language for describing Web services. An SCA module can have many import interfaces.
- Import bindings are concrete definitions that specify the physical mechanism that SCA modules use to access an external service.

Supported import bindings

WebSphere Process Server supports the following import bindings:

- SCA bindings connect SCA modules to other SCA modules. SCA bindings are also referred to as default bindings.
- Web Service bindings permit components to invoke Web services. The supported protocols are SOAP1.1/HTTP, SOAP1.2/HTTP, and SOAP1.1/JMS.

  You can use a SOAP1.1/HTTP or SOAP1.2/HTTP binding based on the Java API for XML Web Services (JAX-WS), which allows interaction with services using document or RPC literal bindings and which uses JAX-WS handlers to customize invocations. A separate SOAP1.1/HTTP binding is provided to allow interaction with services that use an RPC-encoded binding or where there is a requirement to use JAX-RPC handlers to customize invocations.
- HTTP bindings permit you to access applications using the HTTP protocol.
- Enterprise JavaBeans (EJB) import bindings enable SCA components to invoke services provided by Java EE business logic running on a Java EE server.
- Enterprise information system (EIS) bindings provide connectivity between SCA components and an external EIS. This communication is achieved through the use of resource adapters.
- Java Message Service (JMS) 1.1 bindings permit interoperability with the WebSphere Application Server default messaging provider. JMS can exploit various transport types, including TCP/IP and HTTP or HTTPS. The JMS Message class and its five subtypes (Text, Bytes, Object, Stream, and Map) are automatically supported.
- Generic JMS bindings permit interoperability with third-party JMS providers that integrate with the WebSphere Application Server using the JMS Application Server Facility (ASF).
- WebSphere MQ JMS bindings permit interoperability with WebSphere MQ-based JMS providers. The JMS Message class and its five subtypes (Text, Bytes, Object, Stream, and Map) are automatically supported. If you want to use WebSphere MQ as a JMS provider, use WebSphere MQ JMS bindings.

- WebSphere MQ bindings permit interoperability with WebSphere MQ. You can use WebSphere MQ bindings only with remote queue managers by way of a WebSphere MQ client connection; you cannot use them with local queue managers. Use WebSphere MQ bindings if you want to communicate with native WebSphere MQ applications.

Dynamic invocation of services

Services can be invoked through any supported import binding. A service is normally found at an endpoint specified in the import. This endpoint is called a static endpoint. It is possible to invoke a different service by overriding the static endpoint. Dynamic override of static endpoints lets you invoke a service at another endpoint, through any supported import binding. Dynamic invocation of services also permits you to invoke a service where the supported import binding does not have a static endpoint.

An import with an associated binding is used to specify the protocol and its configuration for dynamic invocation. The import used for the dynamic invocation can be wired to the calling component, or it can be dynamically selected at runtime.

For Web service and SCA invocations, it is also possible to make a dynamic invocation without an import, with the protocol and configuration deduced from the endpoint URL. The invocation target type is identified from the endpoint URL. If an import is used, the URL must be compatible with the protocol of the import binding.

- An SCA URL indicates invocation of another SCA module.
- An HTTP or a JMS URL by default indicates invocation of a Web service; for these URLs, it is possible to provide an additional binding type value that indicates that the URL represents an invocation by way of an HTTP or JMS binding.
- For a Web service HTTP URL, the default is to use SOAP 1.1, and a binding type value can be specified that indicates the use of SOAP 1.2.

Exports and export bindings

Exports define interactions between Service Component Architecture (SCA) modules and service requesters. SCA modules use exports to offer services to others. Export bindings define the specific way that an SCA module is accessed by service requesters.

Interfaces and bindings

An SCA module export needs at least one interface.

- Export interfaces are abstract definitions that define a set of operations using Web Services Description Language (WSDL), an XML language for describing Web services. An SCA module can have many export interfaces.
- Export bindings are concrete definitions that specify the physical mechanism that service requesters use to access a service. Usually, an SCA module export has one binding specified. An export with no binding specified is interpreted by the run time as an export with an SCA binding.

Supported export bindings

WebSphere Process Server supports the following export bindings:

- SCA bindings connect SCA modules to other SCA modules. SCA bindings are also referred to as default bindings.
- Web Service bindings permit exports to be invoked as Web services. The supported protocols are SOAP1.1/HTTP, SOAP1.2/HTTP, and SOAP1.1/JMS.

  You can use a SOAP1.1/HTTP or SOAP1.2/HTTP binding based on the Java API for XML Web Services (JAX-WS), which allows interaction with services using document or RPC literal bindings and which uses JAX-WS handlers to customize invocations. A separate SOAP1.1/HTTP binding is provided to allow interaction with services that use an RPC-encoded binding or where there is a requirement to use JAX-RPC handlers to customize invocations.
- HTTP bindings permit exports to be accessed using the HTTP protocol.
- Enterprise JavaBeans (EJB) export bindings allow SCA components to be exposed as EJBs so that Java EE business logic can invoke SCA components otherwise unavailable to them.
- Enterprise information system (EIS) bindings provide connectivity between SCA components and an external EIS. This communication is achieved through the use of resource adapters.
- Java Message Service (JMS) 1.1 bindings permit interoperability with the WebSphere Application Server default messaging provider. JMS can exploit various transport types, including TCP/IP and HTTP or HTTPS. The JMS Message class and its five subtypes (Text, Bytes, Object, Stream, and Map) are automatically supported.
- Generic JMS bindings permit interoperability with third-party JMS providers that integrate with the WebSphere Application Server using the JMS Application Server Facility (ASF).
- WebSphere MQ JMS bindings permit interoperability with WebSphere MQ-based JMS providers. The JMS Message class and its five subtypes (Text, Bytes, Object, Stream, and Map) are automatically supported. If you want to use WebSphere MQ as a JMS provider, use WebSphere MQ JMS bindings.
- WebSphere MQ bindings permit interoperability with WebSphere MQ. You use a remote (or client) connection to connect to an MQ queue manager on a remote machine. A local (or bindings) connection is a direct connection to WebSphere MQ. This can be used only for a connection to an MQ queue manager on the same machine. WebSphere MQ will permit both types of connection, but MQ bindings only support the "remote" (or "client") connection.

Mediation modules

Mediation modules are Service Component Architecture (SCA) modules that can change the format, content, or target of service requests.

Mediation modules operate on messages that are in-flight between service requesters and service providers. You are able to route messages to different service providers and to amend message content or form. Mediation modules can provide functions such as message logging, and error processing that is tailored to your requirements.

You can change certain aspects of mediation modules, from the WebSphere Process Server administrative console, without having to redeploy the module.

Components of mediation modules

Mediation modules contain the following items:

- Imports, which define interactions between SCA modules and service providers. They allow SCA modules to call external services as if they were local. You can view mediation module imports from WebSphere Process Server and modify the binding.
- Exports, which define interactions between SCA modules and service requesters. They allow an SCA module to offer a service and define the external interfaces (access points) of an SCA module. You can view mediation module exports from WebSphere Process Server.
- SCA components, which are building blocks for SCA modules such as mediation modules. You can create and customize SCA modules and components graphically, using WebSphere Integration Developer. After you deploy a mediation module you can customize certain aspects of it from the WebSphere Process Server administrative console, without having to redeploy the module.

  Usually, mediation modules contain a specific type of SCA component called a *mediation flow component*. Mediation flow components define mediation flows.

  A mediation flow component can contain none, one, or a number of mediation primitives. WebSphere Process Server supports a supplied set of mediation primitives that provide functionality for message routing and transformation. If you need additional mediation primitive flexibility, you can use the Custom Mediation primitive to call custom logic.

  The purpose of a mediation module that does not contain a mediation flow component is to transform service requests from one protocol to another. For example, a service request might be made using SOAP/JMS but might need transforming to SOAP/HTTP before sending on.

  Note: You can view and make certain changes to mediation modules from WebSphere Process Server. However, you cannot view or change the SCA components inside a WebSphere Process Server module. Use WebSphere Integration Developer to customize SCA components.

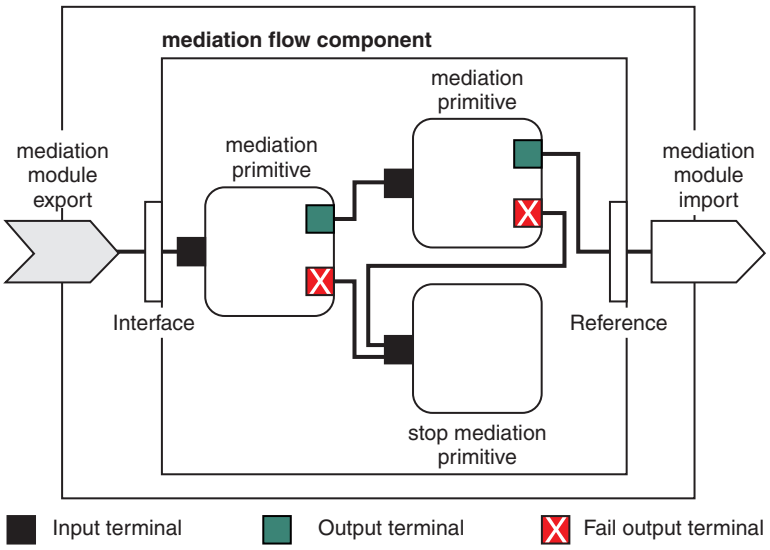


Figure 18. Simplified example of a mediation module. The mediation module contains one mediation flow component, which contains mediation primitives.

- Properties

  Mediation primitives have properties, some of which can be displayed in the administrative console as additional properties of an SCA module.

For mediation primitive properties to be visible from the WebSphere Process Server administrative console, the integration developer must promote the properties. Certain properties lend themselves to being administratively configured and WebSphere Integration Developer describes these as promotable properties, because they can be promoted from the integration cycle to the administrative cycle. Other properties are not suitable for administrative configuration, because modifying them can affect the mediation flow in such a way that the mediation module needs to be redeployed. WebSphere Integration Developer lists the properties that you can choose to promote under the promoted properties of a mediation primitive.

You can use the WebSphere Process Server administrative console to change the value of promoted properties without having to redeploy a mediation module, or restart the server or module.

Generally, mediation flows use property changes immediately. However, if property changes occur in a deployment manager cell, they take effect on each node as that node is synchronized. Also, mediation flows that are in-flight continue to use previous values.

Note: From the administrative console, you can only change property values, not property groups, names or types. If you want to change property groups, names or types, you must use WebSphere Integration Developer.

- A mediation module or dependent library may also define subflows. A subflow encapsulates a set of mediation primitives wire together as a reusable piece of integration logic. A primitive can be added to a mediation flow to invoke a subflow.

Deploying mediation modules

Mediation modules are created using WebSphere Integration Developer, and are generally deployed to WebSphere Process Server inside an enterprise archive (EAR) file.

You can change the value of promoted properties at deployment time.

You can export a mediation module from WebSphere Integration Developer, and cause WebSphere Integration Developer to package the mediation module inside a Java archive (JAR) file, and the JAR file inside an EAR file. You can then deploy the EAR file, by installing a new application from the administrative console.

Mediation modules can be thought of as one entity. However, SCA modules are defined by a number of XML files stored in a JAR file.

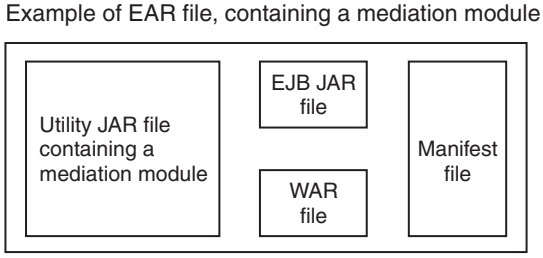


Figure 19. Simplified example of an EAR file containing a mediation module. The EAR file contains JARs. The utility JAR file contains a mediation module.

Mediation primitives

Mediation flow components operate on message flows between service components. The capabilities of a mediation component are implemented by *mediation primitives*, which implement standard service implementation types.

A mediation flow component has one or more flows. For example, one for request and one for reply.

WebSphere Process Server supports a supplied set of mediation primitives, which implement standard mediation capabilities for mediation modules or modules deployed into WebSphere Process Server. If you need special mediation capabilities, you can develop your own custom mediation primitives.

A mediation primitive defines an "in" operation that processes or handles messages that are represented by service message objects (SMOs). A mediation primitive can also define "out" operations that send messages to another component or module.

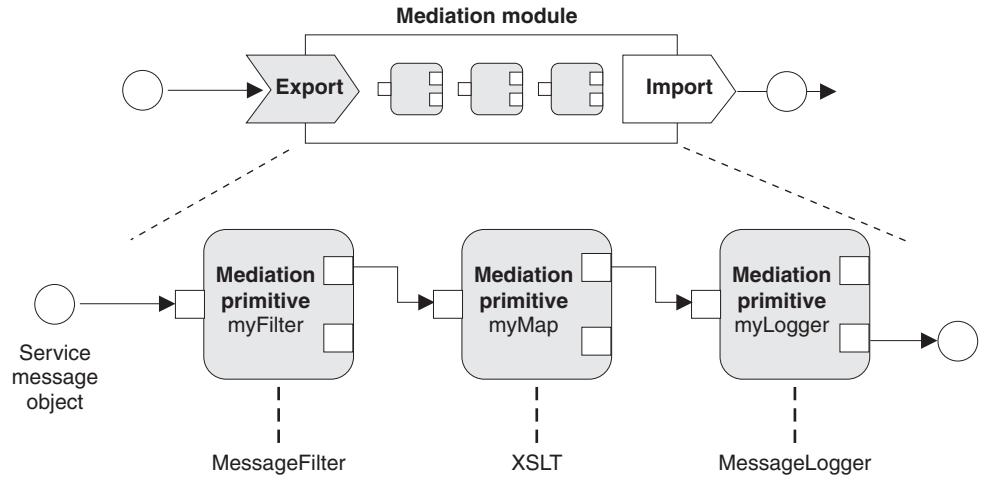


Figure 20. Mediation module containing three mediation primitives

You can use WebSphere Integration Developer to configure mediation primitives and set their properties. Some of these properties can be made visible to the runtime administrator by promoting them. Any mediation primitive property that can be promoted can also be a dynamic property. A dynamic property can be overridden, at run time, using a policy file.

WebSphere Integration Developer also allows you to graphically model and assemble mediation flow components from mediation primitives, and assemble mediation modules or modules from mediation flow components. The administrative console refers to mediation modules and modules as SCA modules.

WebSphere Integration Developer also allows the definition of subflows in modules or their dependent libraries. A subflow can contain any mediation primitive except for the Policy Resolution mediation primitive. A subflow is invoked from a request or response flow, or from another subflow using the Subflow mediation primitive. Properties promoted from mediation primitives in a subflow are exposed as

properties on the Subflow mediation primitives. These may then be promoted again until they reach the module level at which point they can then be modified by the runtime administrator.

Supported mediation primitives

The following set of mediation primitives are supported by WebSphere Process Server:

Business Object Map

Transforms messages.

- Defines message transformations using a business object map, which can be reused.
- Allows you to define message transformations graphically, using the business object map editor.
- Can alter the content of a message.
- Can transform an input message type to a different output message type.

Custom Mediation

Allows you to implement your own mediation logic in Java code. The Custom Mediation primitive combines the flexibility of a user-defined mediation primitive, with the simplicity of a pre-defined mediation primitive. You can create complex transformations and routing patterns by:

- Creating Java code.
- Creating your own properties.
- Adding new terminals.

You can call a service from a Custom Mediation primitive, but the Service Invoke mediation primitive is designed to call services and provides additional functionality, such as retry.

Data Handler

Allows you to transform a part of a message. It is used to convert an element of a message from a physical format to a logical structure or a logical structure to a physical format. The primary usage of the primitive is to convert a physical format, such as a Text string within a JMS Text Message object, into a logical Business Object structure and back again. This mediation is commonly used to:

- Transform a section of the input message from a defined structure to another - an example of this would be were the SMO includes a string value that is comma delimited and you want to parse this into a specific Business Object.
- Alter the message type – an example would be when a JMS export has been configured to use a JMS basic typed data binding and within the mediation module the integration developer decides that the content should be inflated to a specific BO structure.

Database Lookup

Modifies messages, using information from a user-supplied database.

- You must set up a database, data source, and any server authentication settings for the Database Lookup mediation primitive to use. Use the administrative console to help you do this.
- The Database Lookup mediation primitive can read from only one table.
- The specified key column must contain a unique value.

- The data in the value columns must be either a simple XML schema type, or an XML schema type that extends a simple XML schema type.

Endpoint Lookup
Allows for the dynamic routing of requests, by searching for service endpoints in a repository.

- Service endpoint information is retrieved from a WebSphere Service Registry and Repository (WSRR). The WSRR registry can be local or remote.
- You make registry changes from the WSRR administrative console.
- WebSphere Process Server needs to know which registry to use, therefore, you must create WSRR access definitions using the WebSphere Process Server administrative console.

Event Emitter
Enhances monitoring by letting you send events from inside a mediation flow component.

- You can suspend the mediate action by deselecting the check box.
- You can view Event Emitter events using the Common Base Events (CBE) browser on WebSphere Process Server.
- You should only send events at a significant point in a mediation flow, for performance reasons.
- You can define the parts of the message that the event contains.
- The events are sent in the form of Common Base Events and are sent to a Common Event Infrastructure server.
- To fully use the Event Emitter information, event consumers need to understand the structure of the Common Base Events. The Common Base Events has an overall schema, but this does not model the application specific data, which is contained in the extended data elements. To model the extended data elements, the WebSphere Integration Developer tools generate a Common Event Infrastructure event catalog definition file for each of the configured Event Emitter mediation primitives. Event catalog definition files are export artifacts that are provided to help you; they are not used by WebSphere Integration Developer or by the WebSphere Process Server runtime. You should refer to the event catalog definition files when you create applications to consume Event Emitter events.
- You can specify other monitoring from WebSphere Process Server. For example, you can monitor events to be emitted from imports and exports.

Fail Stops a particular path in the flow, and generates an exception.

Fan In Helps aggregate (combine) messages.

- Can only be used in combination with the Fan Out mediation primitive.
- Together, the Fan Out and Fan In mediation primitives allow aggregation of data into one output message.
- The Fan In mediation primitive receives messages until a decision point is reached, then one message is output.
- The shared context should be used to hold aggregation data.

Fan Out
Helps split and aggregate (combine) messages.

- Together, the Fan Out and Fan In mediation primitives allow aggregation of data into one output message.

- In iterate mode, the Fan Out mediation primitive lets you iterate through a single input message that contains a repeating element. For each occurrence of the repeating element, a message is sent.
- The shared context should be used to hold aggregation data.

HTTP Header Setter

Provides a mechanism for managing headers in HTTP messages.

- Can create, set, copy, or delete HTTP message headers.
- Can set multiple actions to change multiple HTTP headers.

MQ Header Setter

Provides a mechanism for managing headers in MQ messages.

- Can create, set, copy, or delete MQ message headers.
- Can set multiple actions to change multiple MQ headers.

SOAP Header Setter

Provides a mechanism for managing headers in SOAP messages.

- Can create, set, copy, or delete SOAP message headers.
- Can set multiple actions to change multiple SOAP headers.

Message Element Setter

Provides a simple mechanism for setting the content of messages.

- Can change, add or delete message elements.
- Does not change the type of the message.
- The data in the value columns must be either a simple XML schema type, or an XML schema type that extends a simple XML schema type.

Message Filter

Routes messages down different paths, based on the message content.

- You can suspend the mediate action by deselecting the check box.

Message Logger

Logs messages in a relational database or through your own custom logger. The messages are stored as XML, therefore, data can be post-processed by XML-aware applications.

- You can suspend the mediate action by deselecting the check box.
- The rational database schema (table structure) is defined by IBM.
- By default, the Message Logger mediation primitive uses the Common database. The runtime maps the data source at `jdbc/mediation/messageLog` to the Common database.
- You can set Handler implementation classes to customize the behavior of the custom logger. Optionally, you can provide Formatter implementation classes, Filter implementation classes, or both to customize the behavior of the custom logger.

Policy Resolution

Allows for the dynamic configuration of requests, by searching for service endpoints, and associated policy files, in a repository.

- You can use a policy file to dynamically override the promoted properties of other mediation primitives.
- Service endpoint information and policy information is retrieved from a WebSphere Service Registry and Repository (WSRR). The WSRR registry can be local or remote.
- You make registry changes from the WSRR administrative console.

- WebSphere Process Server needs to know which registry to use, therefore, you must create WSRR access definitions using the WebSphere Process Server administrative console.

Service Invoke
Calls a service from inside a mediation flow, rather than waiting until the end of the mediation flow and using the callout mechanism.
- If the service returns a fault, you can retry the same service or call another service.
- The Service Invoke mediation primitive is a powerful mediation primitive that can be used on its own for simple service calls, or in combination with other mediation primitives for complex mediations.

Set Message Type
During integration development, lets you treat weakly-typed message fields as though they are strongly-typed. A field is weakly-typed if it can contain more than one type of data. A field is strongly-typed if its type and internal structure are known.
- At runtime, the Set Message Type mediation primitive lets you check that the content of a message matches the data types you expect.

Stop Stops a particular path in the flow, without generating an exception.

Type Filter
Allows you to direct messages down a different path of a flow, based on their type.

XSL Transformation
Transforms messages.
- Allows you to perform Extensible Stylesheet Language (XSL) transformations.
- You transform messages using an XSLT 1.0 transformation. The transformation operates on an XML serialization of the message.

Dynamic routing

You can route messages in various ways using endpoints defined at integration time or endpoints determined, dynamically, at run time.

Dynamic routing covers message routing where the flow is dynamic but all possible endpoints are predefined in a Service Component Architecture (SCA) module, and message routing where the flow is dynamic and the endpoint selection is also dynamic. In the latter case, the service endpoints are selected from an external source, at run time.

Dynamic endpoint selection

The run time has the capability to route request and response messages to an endpoint address identified by a message header element. This message header element can be updated by mediation primitives, in a mediation flow. The endpoint address could be updated with information from a registry, a database, or with information from the message itself. Routing of response messages applies only when the response is being sent by a Web service JAX-WS export.

In order for the run time to implement dynamic routing on a request or response, the SCA module must have the `Use dynamic endpoint if set in the message header` property set. Integration developers can set the `Use dynamic endpoint if set in the message header` property or they can promote it (make it visible at run

time), so that the runtime administrator can set it. You can view module properties in the Module Properties window. To see the window, click **Applications** > **SCA Modules** > **Module Properties**. The integration developer gives promoted properties alias names, and these are the names displayed on the administrative console.

Registry

You can use IBM WebSphere Service Registry and Repository (WSRR) to store service endpoint information, and then create SCA modules to retrieve endpoints from the WSRR registry.

When you develop SCA modules, you use the Endpoint Lookup mediation primitive to allow a mediation flow to query a WSRR registry for a service endpoint, or a set of service endpoints. If an SCA module retrieves a set of endpoints then it must use another mediation primitive to select the preferred one.

Mediation policy control of service requests

You can use mediation policies to control mediation flows between service requesters and service providers.

You can control mediation flows using mediation policies stored in IBM WebSphere Service Registry and Repository (WSRR). The implementation of service policy management in WSRR is based on the Web Services Policy Framework (WS-Policy).

In order to control service requests using mediation policies, you must have suitable Service Component Architecture (SCA) modules and mediation policy documents in your WSRR registry.

How to attach a mediation policy to a service request

When you develop an SCA module that needs to make use of a mediation policy, you must include a Policy Resolution mediation primitive in the mediation flow. At run time, the Policy Resolution mediation primitive obtains mediation policy information from the registry. Therefore, an SCA module must contain a mediation flow component in order to support mediation policy control of service requests.

In the registry, you can attach one or more mediation policies to an SCA module, or to a target service used by the SCA module. Attached mediation policies could be used (are in scope) for all service messages processed by that SCA module. The mediation policies can have policy attachments that define conditions. Mediation policy conditions allow different mediation policies to apply in different contexts. In addition, mediation policies can have classifications, which can be used to specify a governance state.

WebSphere Service Registry and Repository

The WebSphere Service Registry and Repository (WSRR) product allows you to store, access, and manage information about service endpoints and mediation policies. You can use WSRR to make your service applications more dynamic, and more adaptable to changing business conditions.

Introduction

Mediation flows can use WSRR as a dynamic lookup mechanism, providing information about service endpoints or mediation policies.

To configure access to WSRR, you create WSRR definition documents using the administrative console. Alternatively, you can use the WSRR administration commands from the wsadmin scripting client. WSRR definitions and their connection properties are the mechanism used to connect to a registry instance, and retrieve a service endpoint or mediation policy.

Service endpoints

You can use WSRR to store information about services that you already use, that you plan to use, or that you want to be aware of. These services might be in your systems, or in other systems. For example, an application could use WSRR to locate the most appropriate service to satisfy its functional and performance needs.

When you develop an SCA module that needs to access service endpoints from WSRR, you must include an Endpoint Lookup mediation primitive in the mediation flow. At run time, the Endpoint Lookup mediation primitive obtains service endpoints from the registry.

Mediation policies

You can also use WSRR to store mediation policy information. Mediation policies can help you to control service requests, by dynamically overriding module properties. If WSRR contains mediation policies that are attached to an object representing either your SCA module or your target service, then the mediation policies could override the module properties. If you want different mediation policies to apply in different contexts, you can create mediation policy conditions.

Note: Mediation policies are concerned with the control of mediation flows, and not with security.

When you develop an SCA module that needs to make use of a mediation policy, you must include a Policy Resolution mediation primitive in the mediation flow. At run time, the Policy Resolution mediation primitive obtains mediation policy information from the registry.

Message Service clients

Message Service clients is available for C/C++ and .NET to enable non-Java applications to connect to the enterprise service bus.

Message Service Clients for C/C++ and .NET provide an API called XMS that has the same set of interfaces as the Java Message Service (JMS) API. Message Service Client for C/C++ contains two implementations of XMS, one for use by C applications and another for use by C++ applications. Message Service Client for .NET contains a fully managed implementation of XMS, which can be used by any .NET compliant language.

You can obtain Message Service Clients for .NET from http://www-01.ibm.com/support/docview.wss?rs=0&q1=IA9H&uid=swg24011756&loc=en_US&cs=utf-8&cc=us&lang=en

You can obtain Message Service Clients for C/C++ from http://www-01.ibm.com/support/docview.wss?rs=0&q1=ia94&uid=swg24007092&loc=en_US&cs=utf-8&cc=us&lang=en.

You can also install and use the Java EE client support from WebSphere Application Server Network Deployment, including Web services Client, EJB Client, and JMS Client.

Printed in USA