WebSphere Process Server for Multiplatforms

バージョン 6.1.0

モジュールの開発とデプロイ

WebSphere Process Server for Multiplatforms

バージョン 6.1.0

モジュールの開発とデプロイ

お願い

本書に記載されている情報をご使用になる前に、本書末尾の『特記事項』セクションに記載されている情報をお読みください。

本書は、WebSphere Process Server for Multiplatforms バージョン 6、リリース 1、モディフィケーション 0 (製品番号 5724-L01)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

IBM 発行のマニュアルに関する情報のページ

`http://www.ibm.com/jp/manuals/`

こちらから、日本語版および英語版のオンライン・ライブラリーをご利用いただけます。また、マニュアルに関するご意見やご感想を、上記ページよりお送りください。今後の参考にさせていただきます。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原　典： WebSphere® Process Server for Multiplatforms
Version 6.1.0
Developing and Deploying Modules

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2008.2

目次

図

表

第 1 部 アプリケーション開発

第 1 章 モジュールの開発の概要

モジュールは、WebSphere® Process Server アプリケーションのデプロイメントの基本単位です。モジュールは、アプリケーションが使用する 1 つ以上のコンポーネント・ライブラリーとステージング・モジュールで構成されています。コンポーネントは、ほかのサービス・コンポーネントを参照することができます。モジュールを開発するには、アプリケーションが必要とするコンポーネント、ステージング・モジュール、およびライブラリー (モジュールによって参照される成果物の集合) が実動サーバー上で使用可能であることを確認する必要があります。

WebSphere Integration Developer は、WebSphere Process Server にデプロイするモジュールを開発するための主要なツールです。ほかの環境でモジュールを開発することもできますが、WebSphere Integration Developer を使用するのが最適な方法です。

WebSphere Process Server は、2 つのタイプのサービス・モジュールをサポートします。ビジネス・サービス用モジュールおよびメディエーション・モジュールです。ビジネス・サービス用モジュールは、プロセスのロジックをインプリメントします。メディエーション・モジュールは、サービス起動をターゲットが理解する形式に変換し、要求をターゲットに渡して結果をオリジネーターに戻すことによって、アプリケーション間の通信を可能にします。

以降のセクションでは、WebSphere Process Server 上でモジュールをインプリメントおよび更新する方法について説明します。

コンポーネントの概要

コンポーネントは、再使用可能なビジネス・ロジックをカプセル化するための基本要素です。サービス・コンポーネントは、インターフェース、参照、インプリメンテーションに関連付けられます。インターフェースは、サービス・コンポーネントと呼び出し側コンポーネントの間の取り決めを定義します。サービス・モジュールは、WebSphere Process Server を使用して、他のモジュールが使用できるようにサービス・コンポーネントをエクスポートしたり、サービス・コンポーネントをインポートして使用したりすることができます。サービス・コンポーネントを呼び出すために、呼び出し側のモジュールはサービス・コンポーネントとのインターフェースを参照します。呼び出し側モジュールからそれぞれのインターフェースへの参照を構成することによって、インターフェースに対する参照が解決されます。

モジュールを開発するには、以下の作業を行う必要があります。

1. モジュール内のコンポーネント用のインターフェースを定義します。
2. サービス・コンポーネントで使用されるビジネス・オブジェクトを定義、変更、または操作します。
3. インターフェースを使用して、サービス・コンポーネントを定義または変更します。

   注: サービス・コンポーネントは、インターフェースを使用して定義されます。

4. 必要に応じて、サービス・コンポーネントをエクスポートまたはインポートします。
5. コンポーネントを使用するモジュールをインストールするために使用する EAR ファイルを作成します。WebSphere Integration Developer のエクスポート EAR 機能を使用してファイルを作成するか、serviceDeploy コマンドを使用して EAR ファイルを作成し、サービス・コンポーネントを使用するサービス・モジュールをインストールします。

開発タイプ

WebSphere Process Server では、サービス指向のプログラミング・パラダイムを促進するコンポーネント・プログラミング・モデルを提供します。このモデルを使用するために、提供者はサービス・コンポーネントのインターフェースをエクスポートします。これにより、利用者はそのインターフェースをインポートして、そのサービス・コンポーネントがローカルであるかのように使用できるようになります。開発者は厳密に型指定されたインターフェースまたは動的型付きインターフェースのいずれかを使用して、サービス・コンポーネントをインプリメントしたり、呼び出したりします。インターフェースとそのメソッドについては、このインフォメーション・センターの『References』のセクションに説明があります。

サービス・モジュールをサーバーにインストールした後、管理コンソールを使用して、アプリケーションからの参照のターゲット・コンポーネントを変更することができます。新しいターゲットは、アプリケーションからの参照が要求しているものと同じビジネス・オブジェクト・タイプを受け入れ、同じ操作を実行する必要があります。

サービス・コンポーネントの開発に関する考慮事項

サービス・コンポーネントを開発する場合は、以下の点を検討してください。

- このサービス・コンポーネントがエクスポートされ、ほかのモジュールによって使用されるかどうか。

  使用される場合、そのコンポーネントに定義したインターフェースを別のモジュールが使用できることを確認してください。
- サービス・コンポーネントを実行するのに比較的長い時間がかかるかどうか。

  長時間かかる場合は、サービス・コンポーネントに非同期のインターフェースをインプリメントすることを検討してください。
- サービス・コンポーネントを分散化することが有益かどうか。

  有益である場合は、サーバーのクラスター上にデプロイされているサービス・モジュール内にサービス・コンポーネントのコピーを配置して、並列処理の利点を活かすことを検討してください。
- アプリケーションが、一相および二相コミット・リソースの混用を必要とするか。

  必要とする場合、アプリケーションの Last Participant サポートを使用可能にしてください。

注: WebSphere Integration Developer を使用してアプリケーションを作成したか、または serviceDeploy コマンドを使用してインストール可能な EAR ファイルを作成した場合、これらのツールは自動的にアプリケーションのサポートを使用可能にします。WebSphere Application Server Network Deployment インフォメーション・センターで、トピック「同一トランザクション内での 1 フェーズ・コミットおよび 2 フェーズ・コミットのリソースの使用」を参照してください。

サービス・モジュールの開発

サービス・コンポーネントは、サービス・モジュール内に含まれていなければなりません。サービス・コンポーネントを含むためのサービス・モジュールを開発することが、ほかのモジュールにサービスを提供するための鍵となります。

始める前に

以下のタスクでは、要件を分析した結果、ほかのモジュールで使用できるように、サービス・コンポーネントをインプリメントすると有益であると判断されていることが前提となっています。

このタスクについて

要件を分析した結果、サービス・コンポーネントの提供と利用が効率的な情報処理手段であると判断できる場合があります。ご使用の環境にとって再使用可能なサービス・コンポーネントが有効であると判断したうえで、サービス・コンポーネントを含むためのサービス・モジュールを作成してください。

プロシージャー

1. ほかのサービス・モジュールで使用できるコンポーネントを特定します。

   サービス・コンポーネントを特定したら、『サービス・コンポーネントの開発』に進みます。
2. ほかのサービス・モジュール内のサービス・コンポーネントを使用できる、アプリケーション内のサービス・コンポーネントを特定します。

   サービス・コンポーネントとそれぞれのターゲット・コンポーネントを特定したら、『コンポーネントの呼び出し』に進みます。
3. クライアント・コンポーネントをワイヤー経由でターゲット・コンポーネントに接続します。

サービス・コンポーネントの開発

ご使用のサーバー内の複数のアプリケーションに再使用可能なロジックを提供するための、サービス・コンポーネントを作成します。

始める前に

この作業では、複数のモジュールで使用できる処理がすでに作成され、特定されていることが前提になっています。

このタスクについて

複数のモジュールで 1 つのサービス・コンポーネントを使用することができます。サービス・コンポーネントをエクスポートすると、インターフェースを介してそのコンポーネントを参照するほかのモジュールが、そのサービス・コンポーネントを利用できるようになります。この作業では、ほかのモジュールがコンポーネントを使用できるように、そのサービス・コンポーネントを作成する方法を説明します。

注: 1 つのサービス・コンポーネントに、複数のインターフェースを設定することができます。

プロシージャー

1. 呼び出し元とサービス・コンポーネントの間のデータの移動のためのデータ・オブジェクトを定義します。

   データ・オブジェクトおよびそのタイプは、呼び出し元とサービス・コンポーネント間のインターフェースの一部となります。
2. 呼び出し元がサービス・コンポーネントを参照するときに使用するインターフェースを定義します。

   このインターフェース定義で、サービス・コンポーネントを指定し、サービス・コンポーネント内のすべての使用可能なメソッドをリストします。
3. インプリメンテーションを定義するクラスを開発します。
   - コンポーネントが長期にわたって実行される (非同期) 場合は、ステップ 4 に進みます。
   - コンポーネントが長期にわたって実行されるものでない (同期) 場合は、ステップ 5 に進みます。
4. 非同期インプリメンテーションを開発します。

   重要: 非同期型コンポーネント・インターフェースでは、joinsTransaction プロパティーを `true` に設定できません。

   a. 同期型サービス・コンポーネントを示すインターフェースを定義します。
   b. サービス・コンポーネントのインプリメンテーションを定義します。
   c. ステップ 6 に進みます。
5. 同期インプリメンテーションを開発します。
   a. 同期型サービス・コンポーネントを示すインターフェースを定義します。
   b. サービス・コンポーネントのインプリメンテーションを定義します。
6. コンポーネントのインターフェース、およびインプリメンテーションを拡張子が .java のファイルに保管します。
7. サービス・モジュールと必要なリソースを JAR ファイルにパッケージ化します。

   ステップ 7 から 9 までの詳しい説明については、このインフォメーション・センターの『実動サーバーへのモジュールのデプロイ』のセクションを参照してください。
8. serviceDeploy コマンドを実行して、アプリケーションを格納するインストール可能な EAR ファイルを作成します。
9. サーバー・ノード上にアプリケーションをインストールします。

10. オプション: ほかのサービス・モジュール内のサービス・コンポーネントを呼び出す場合は、呼び出し元とそれに対応するサービス・コンポーネント間のワイヤーを構成します。

    このインフォメーション・センターの『管理』セクションに、ワイヤーの構成についての説明があります。

コンポーネントの開発例

この例では、1 つのメソッド CustomerInfo をインプリメントする同期型サービス・コンポーネントを示しています。最初のセクションでは、getCustomerInfo というメソッドをインプリメントするサービス・コンポーネントに対するインターフェースを定義しています。

```
public interface CustomerInfo {
 public Customer getCustomerInfo(String customerID);
}
```

以下のコード・ブロックで、サービス・コンポーネントをインプリメントします。

```
public class CustomerInfoImpl implements CustomerInfo {
 public Customer getCustomerInfo(String customerID) {
  Customer cust = new Customer();

  cust.setCustNo(customerID);
  cust.setFirstName("Victor");
  cust.setLastName("Hugo");
  cust.setSymbol("IBM");
  cust.setNumShares(100);
  cust.setPostalCode(10589);
  cust.setErrorMsg("");

  return cust;
 }
}
```

この例では、非同期型サービス・コンポーネントを作成します。コードの最初のセクションでは、getQuote というメソッドをインプリメントするサービス・コンポーネントに対するインターフェースを定義しています。

```
public interface StockQuote {

 public float getQuote(String symbol);
}
```

以下のセクションは、StockQuote に関連したクラスのインプリメンテーションです。

```
public class StockQuoteImpl implements StockQuote {

 public float getQuote(String symbol) {


     return 100.0f;
 }
}
```

以下のコード・セクションは、非同期インターフェース StockQuoteAsync をインプリメントします。

```
public interface StockQuoteAsync {

 // deferred response
 public Ticket getQuoteAsync(String symbol);
 public float getQuoteResponse(Ticket ticket, long timeout);

 // callback
 public Ticket getQuoteAsync(String symbol, StockQuoteCallback callback);
}
```

以下のセクションは、onGetQuoteResponse メソッドを定義するインターフェース StockQuoteCallback です。

```
public interface StockQuoteCallback {

 public void onGetQuoteResponse(Ticket ticket, float quote);
}
```

次のタスク

サービスを起動します。

コンポーネントの呼び出し

モジュールを含むコンポーネントは、WebSphere Process Server クラスターの任意のノード上でコンポーネントを使用することができます。

始める前に

コンポーネントを呼び出す前に、WebSphere Process Server に、コンポーネントを含むモジュールがインストールされていることを確認してください。

このタスクについて

コンポーネントは、コンポーネントの名前を使用し、コンポーネントに適したデータ型を渡すことによって、WebSphere Process Server クラスター内で使用可能なすべてのサービス・コンポーネントを使用することができます。この環境内でコンポーネントを呼び出すには、必要なコンポーネントを見つけてから、そのコンポーネントへの参照を作成する操作が必要です。

注: モジュール内のコンポーネントは、同一のモジュール内のコンポーネントを呼び出すことができ、これはモジュール内呼び出しと呼ばれます。提供側コンポーネント内のインターフェースをエクスポートし、呼び出し側コンポーネント内でインターフェースをインポートすることによって、外部呼び出し (モジュール内呼び出し) をインプリメントしてください。

重要: 呼び出し側モジュールが稼動するサーバーと異なるサーバー上に存在するコンポーネントを呼び出す場合は、サーバーへの追加構成を実行する必要があります。必要な構成は、コンポーネントが非同期に呼び出されるか、同期して呼び出されるかによって異なります。この場合のアプリケーション・サーバーの構成方法は、関連タスクで説明されています。

プロシージャー

1. 呼び出し側モジュールに必要なコンポーネントを判別します。

コンポーネント内のインターフェースの名前と、そのインターフェースに必要なデータ型を書き留めます。

2. データ・オブジェクトを定義します。

   入力または戻りは Java™ クラスでかまいませんが、サービス・データ・オブジェクトが最適です。

3. コンポーネントを探します。

   a. ServiceManager クラスを使用して、呼び出し側モジュールが使用できる参照を取得します。

   b. locateService() メソッドを使用して、コンポーネントを探します。

      インターフェースは、コンポーネントに応じて、Web サービス記述言語 (WSDL) ポート・タイプまたは Java インターフェースのいずれかを使用することができます。

4. コンポーネントを同期式、または非同期に呼び出します。

   Java インターフェースを使用してコンポーネントを呼び出すことも、 invoke() メソッドを使用してコンポーネントを動的に呼び出すこともできます。

5. 戻り値を処理します。

   コンポーネントが例外を生成することがあるので、クライアントでは例外の処理が可能である必要があります。

コンポーネントの呼び出し例

次の例では、 ServiceManager クラスを作成します。

```
ServiceManager serviceManager = new ServiceManager();
```

以下の例は、 ServiceManager クラスを使用して、コンポーネントの参照を含んでいるファイルからコンポーネントのリストを取得します。

```
InputStream myReferences = new FileInputStream("MyReferences.references");
ServiceManager serviceManager = new ServiceManager(myReferences);
```

以下のコードは、StockQuote Java インターフェースをインプリメントするコンポーネントを探します。

```
StockQuote stockQuote = (StockQuote)serviceManager.locateService("stockQuote");
```

以下のコードは、Java または WSDL ポート・タイプ・インターフェースをインプリメントするコンポーネントを探します。呼び出し側モジュールは、Service インターフェースを使用して、コンポーネントと対話します。

ヒント: コンポーネントが Java インターフェースをインプリメントする場合は、コンポーネントをインターフェースまたは invoke() メソッドのいずれかを使用して呼び出すことができます。

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
```

次の例は、別のコンポーネントを呼び出すコード MyValue を示しています。

```
public class MyValueImpl implements MyValue {

 public float myValue throws MyValueException {
```

```
  ServiceManager serviceManager = new ServiceManager();

     // variables
         Customer customer = null;
         float quote = 0;
         float value = 0;

     // invoke
         CustomerInfo cInfo =
      (CustomerInfo)serviceManager.locateService("customerInfo");
         customer = cInfo.getCustomerInfo(customerID);

     if (customer.getErrorMsg().equals("")) {

         // invoke
       StockQuoteAsync sQuote =
      (StockQuoteAsync)serviceManager.locateService("stockQuote");
        Ticket ticket =  sQuote.getQuoteAsync(customer.getSymbol());
    // ... do something else ...
       quote =  sQuote.getQuoteResponse(ticket, Service.WAIT);

         // assign
          value = quote * customer.getNumShares();
     } else {

         // throw
         throw new MyValueException(customer.getErrorMsg());
     }
     // reply
         return value;
 }
}
```

次のタスク

呼び出し側モジュールの参照とコンポーネントのインターフェースの間のワイヤーを構成します。

コンポーネントの動的呼び出し

Web サービス記述言語 (WSDL) ポート・タイプ・インターフェースを指定したコンポーネントをモジュールから呼び出す場合、モジュールは invoke() メソッドを使用して、そのコンポーネントを動的に呼び出す必要があります。

始める前に

この操作では、呼び出し側コンポーネントがコンポーネントを動的に呼び出すことが前提となっています。

このタスクについて

WSDL ポート・タイプ・インターフェースの場合は、呼び出し側コンポーネントは invoke() メソッドを使用して、コンポーネントを呼び出す必要があります。呼び出し側モジュールから、この方法で Java インターフェースを指定したコンポーネントも呼び出すことができます。

プロシージャー

1. 必要なコンポーネントを含んでいるモジュールを判別します。
2. コンポーネントが必要とする配列を判別します。

入力配列は、次の 3 つのタイプのいずれかです。
- 大文字の Java プリミティブ型、またはこの型の配列
- 通常の Java クラス、またはクラスの配列
- サービス・データ・オブジェクト (SDO)

3. コンポーネントからの応答を収容する配列を定義します。

   応答配列は、入力配列と同じタイプでかまいません。

4. invoke() メソッドを使用して、必要なコンポーネントを呼び出し、配列オブジェクトをそのコンポーネントに渡します。
5. 結果を処理します。

コンポーネントの動的呼び出しの例

以下の例では、モジュールは invoke() メソッドを使用して、大文字の Java プリミティブ・データ型を使用するコンポーネントを呼び出します。

```
Service service = (Service)serviceManager.locateService("multiParamInf");

  Reference reference = service.getReference();

  OperationType methodMultiType =
    reference.getOperationType("methodWithMultiParameter");

  Type t = methodMultiType.getInputType();

  BOFactory boFactory = (BOFactory)serviceManager.locateService
    ("com/ibm/websphere/bo/BOFactory");

  DataObject paramObject = boFactory.createbyType(t);

  paramObject.set(0,"input1")
  paramObject.set(1,"input2")
  paramObject.set(2,"input3")

  service.invoke("methodMultiParamater",paramObject);
```

次の例では、WSDL ポート・タイプ・インターフェースをターゲットとして持つ呼び出しメソッドを使用します。

```
Service serviceOne = (Service)serviceManager.locateService("multiParamInfWSDL");

 DataObject dob = factory.create("http://MultiCallWSServerOne/bos", "SameBO");
   dob.setString("attribute1", stringArg);

 DataObject wrapBo = factory.createByElement
  ("http://MultiCallWSServerOne/wsdl/ServerOneInf", "methodOne");
   wrapBo.set("input1", dob); //wrapBo encapsulates all the parameters of methodOne
   wrapBo.set("input2", "XXXX");
   wrapBo.set("input3", "yyyy");

 DataObject resBo= (DataObject)serviceOne.invoke("methodOne", wrapBo);
```

モジュールとターゲットの分離の概要

モジュールを開発する際、複数のモジュールが使用できるサービスを識別します。このようにしてサービスにてこ入れすることにより、開発サイクルとコストを最小化します。多数のモジュールによって使用されるサービスがある場合は、ターゲットがアップグレードされた場合に新規サービスへの切り替えが呼び出しモジュールに対して透過的になるように、呼び出しモジュールをターゲットから分離する必要

があります。このトピックでは、単純な呼び出しモデルと分離された呼び出しモデルを対比して、分離がどのように役立つかを示す例を提供します。特定の例について説明しますが、これが、ターゲットからモジュールを分離する唯一の方法というわけではありません。

単純な呼び出しモデル

モジュールを開発する際、その他のモジュールにあるサービスを使用することができます。これは、モジュールにサービスをインポートしてからそのサービスを呼び出すことによって実行します。インポートされたサービスは、WebSphere Integration Developer で、または管理コンソール内のサービスをバインディングすることによって、その他のモジュールによってエクスポートされたサービスに「関連付け」られます。『単純な呼び出しモデル』は、このモデルを示しています。

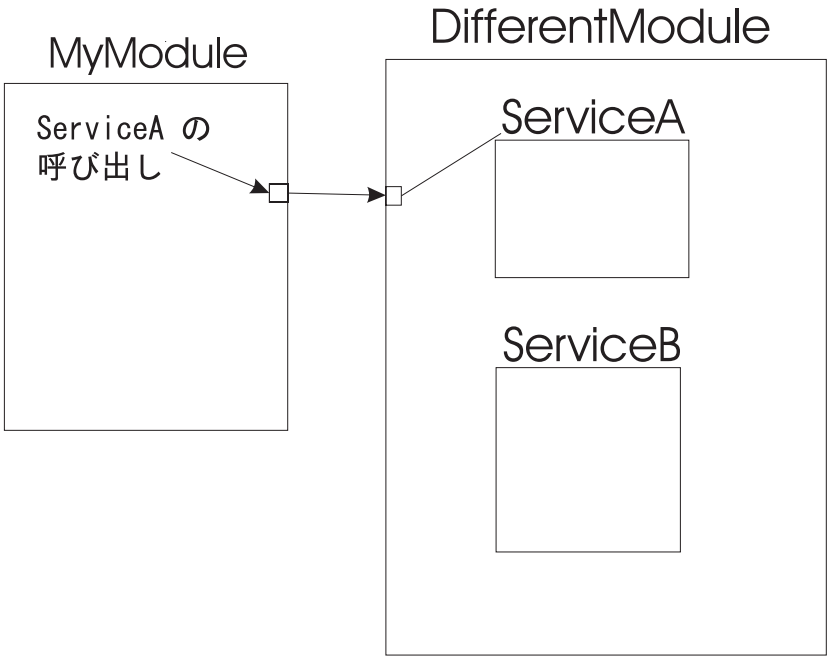


図 1. 単純な呼び出しモデル

分離された呼び出しモデル

呼び出しモジュールを停止せずに呼び出しのターゲットを変更するには、呼び出しのターゲットから呼び出しモジュールを分離します。この場合、モジュールそのものではなくダウンストリーム・ターゲットを変更しているので、ターゲットの変更中もモジュールが処理を続行できます。『アプリケーションの分離の例』は、分離によって、呼び出しモジュールの状況に影響を与えずにターゲットを変更する方法を示します。

アプリケーションの分離の例

単純な呼び出しモデルを使用した場合、同一のサービスを呼び出す複数のモジュールは、『単一のサービスを呼び出す複数のアプリケーション』のようになります。MODA、MODB、および MODC はすべて CalculateFinalCost を呼び出します。

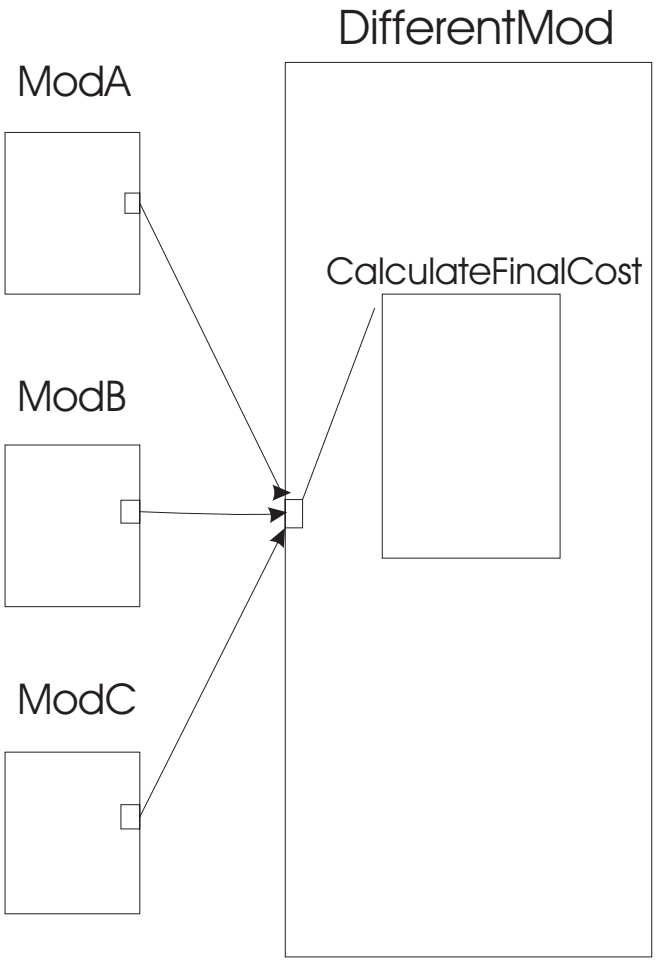


図 2. 単一のサービスを呼び出す複数のアプリケーション

CalculateFinalCost によって提供されるサービスは、そのサービスを使用するすべてのモジュールに新規コストが反映されるように、更新する必要があります。開発チームは、新規サービス UpdatedCalculateFinal を構築およびテストして、変更を取り込みます。新規サービスは実動に移す準備ができています。分離を使用しない場合は、UpdateCalculateFinal を呼び出すために、 CalculateFinalCost を呼び出すモジュールをすべて更新する必要があります。分離を使用した場合、実行する必要があるのは、バッファー・モジュールをターゲットに接続するバインディングを変更することだけです。

注: このようにサービスを変更することにより、オリジナルのサービスを、それを必要とするその他のモジュールに提供し続けることができます。

分離を使用して、アプリケーションとターゲットの間でバッファー・モジュールを作成します (『UpdateCalculateFinal を呼び出す分離された呼び出しモデル』を参照してください)。

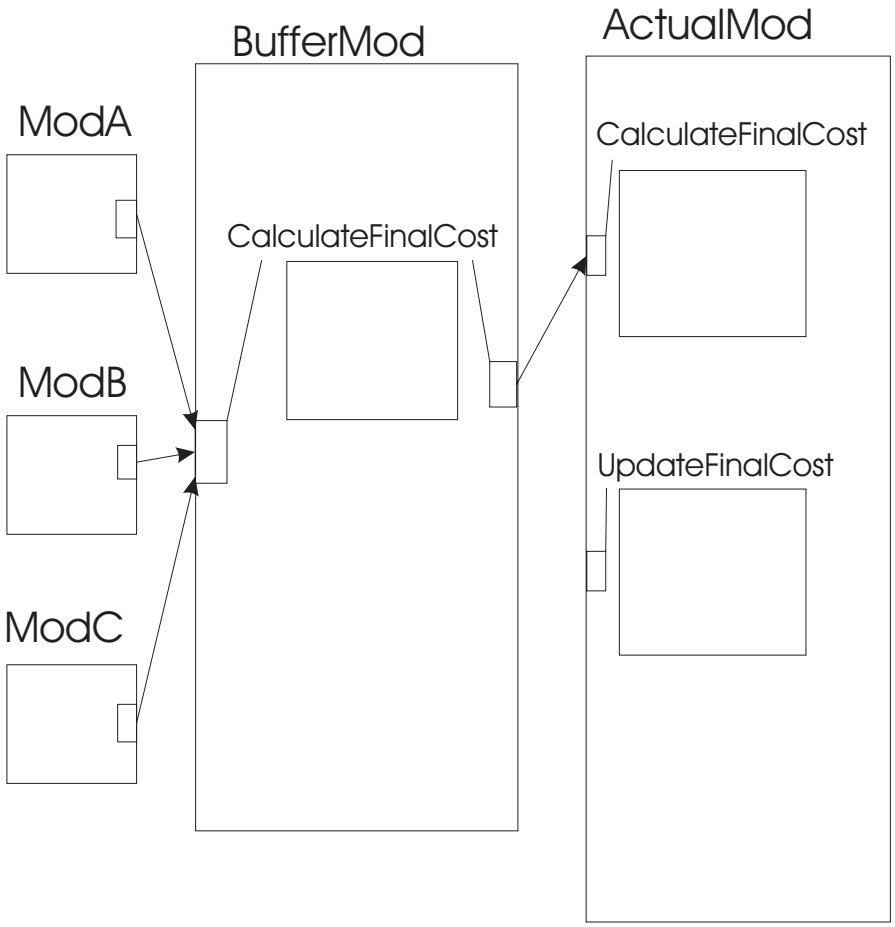


図 3. UpdateCalculateFinal を呼び出す分離された呼び出しモデル

このモデルで、呼び出しモジュールは変わりません。実行する必要があるのは、バインディングをバッファー・モジュール・インポートからターゲットへ変更することだけです (『UpdatedCalculateFinal を呼び出す分離された呼び出しモデル』を参照してください)。

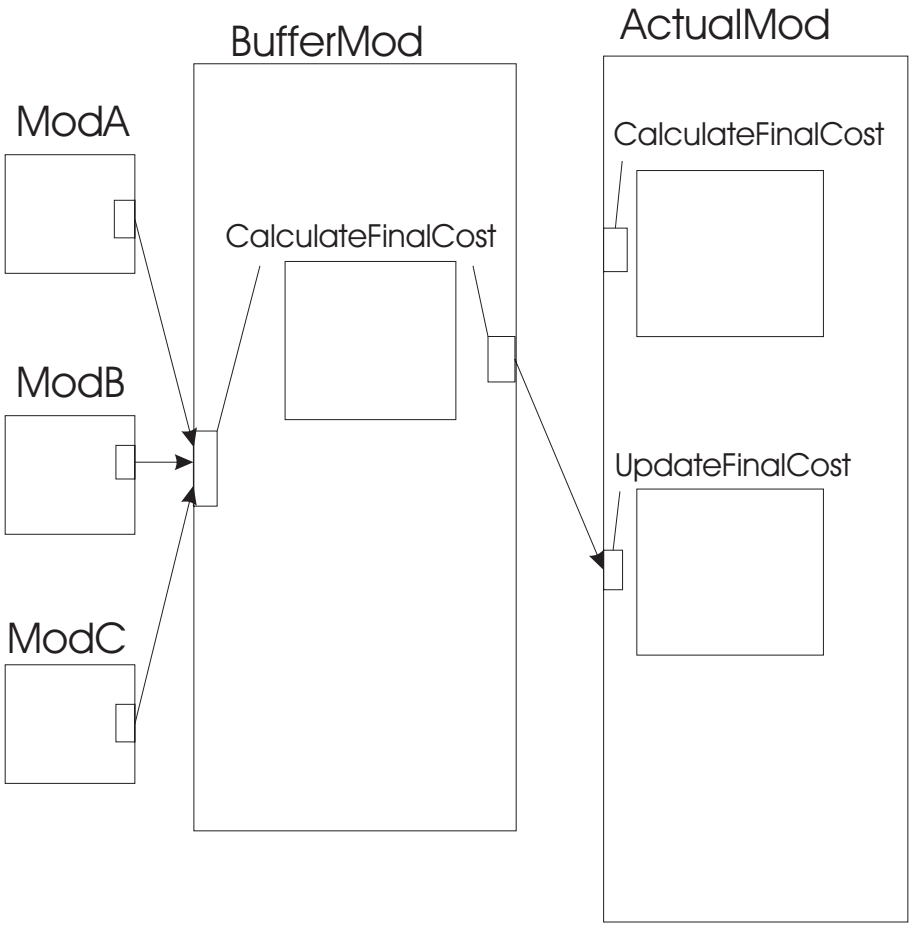


図 4. UpdatedCalculateFinal を呼び出す分離された呼び出しモデル

バッファー・モジュールがターゲットを同期で呼び出す場合、元のアプリケーションに戻される結果は、バッファー・モジュール (メディエーション・モジュール、またはビジネス・モジュール用のサービス) を再始動したときに、新規ターゲットから送信されます。バッファー・モジュールがターゲットを非同期で呼び出す場合、元のアプリケーションに戻される結果は、次の呼び出し時に新規ターゲットから送信されます。

関連タスク

ターゲットの変更

参照のターゲットを変更すると、アプリケーションを再コンパイルおよび再インストールすることなく、進化したコンポーネントをそのままアプリケーションで利用できるという柔軟性がもたらされます。

HTTP バインディング

HTTP バインディングは、Service Component Architecture (SCA) の HTTP 接続を提供するためのものです。これにより、既存または新規作成された HTTP アプリケーションを、サービス指向アーキテクチャー (SOA) 環境内に組み込むことができます。

さらに、SCA ランタイム環境のネットワークは、既存の HTTP インフラストラクチャーを通して通信できます。

HTTP バインディングは、以下のような HTTP 機能を公開します。

- メッセージがメディエーション・コンポーネントに渡される際、 HTTP 形式とメッセージ・ヘッダー情報が保持されます。これにより、HTTP アプリケーションのプログラマー、ユーザー、および管理者に、より分かりやすいビューが提供されます。
- 既存のデータ・バインディング・フレームワークが HTTP の規則のために拡張され、SCA メッセージと HTTP メッセージ・ヘッダーおよび本文との間でマッピングが提供されます。
- 共通 HTTP 機能の範囲をサポートするよう、インポートおよびエクスポートを構成できます。
- HTTP インポートまたはエクスポートを含む SCA モジュールをインストールすると、自動的に、HTTP 接続が可能になるようランタイム環境が適切に構成されます。

HTTP インポートおよびエクスポートを作成する方法について詳しくは、インフォメーション・センターの「**WebSphere Integration Developer**」> 「**統合アプリケーションの開発 (Developing integration applications)**」>「**HTTP データのバインディング (HTTP data binding)**」 を参照してください。

関連タスク

HTTP バインディングの表示

アプリケーションをデプロイした後、HTTP バインディングを検査して、それが正しいかどうかを確認することができます。

HTTP エクスポート・バインディングの変更

管理コンソールを使用すると、HTTP エクスポート・バインディングの構成を変更するときに、元のソースを変更してからアプリケーションを再デプロイする必要がありません。

HTTP インポート・バインディングの変更

管理コンソールを使用すると、HTTP インポート・バインディングの構成を変更するときに、元のソースを変更してからアプリケーションを再デプロイする必要がありません。

生成される Service Component Architecture インプリメンテーションのオーバーライド

場合によっては、システムが作成した Java コードとサービス・データ・オブジェクト (SDO) 間の変換では、要件に合わないことがあります。デフォルトの Service Component Architecture (SCA) クラスのインプリメンテーションを独自のインプリメンテーションに置き換えるには、以下の手順を実行します。

始める前に

WebSphere Integration Developer または genMapper コマンドを使用して Java から Web サービス記述言語 (WSDL) 型への変換を生成していることを確認します。

このタスクについて

生成されたコードを、要件に合うコードで置き換えることにより、Java 型から WSDL 型にマップする生成されたコンポーネントをオーバーライドします。独自の Java クラスを定義している場合は、独自のマップを使用することを考慮してください。以下の手順を使用して変更を行います。

プロシージャー

1. 生成されたコンポーネントを見つけます。 コンポーネントの名前は *java_class*Mapper.component です。
2. テキスト・エディターを使用してコンポーネントを編集します。
3. 生成されたコードをコメント化し、独自のメソッドを指定します。

   コンポーネントのインプリメンテーションを含むファイル名は変更しないでください。

以下は、置き換え対象の生成されたコンポーネントの例です。

```
private DataObject javatodata_setAccount_output(Object myAccount) {

  // このコードをカスタム・マッピングのためオーバーライドできます。
  // このコードをコメント化してカスタム・コードを書き込みます。

  // コンバーターに渡される Java 型は、コンバーターが作成を試行します。
  // この Java 型を変更することもできます。


 return SDOJavaObjectMediator.java2Data(myAccount);


}
```

コンポーネントおよびその他のファイルを、含んでいるモジュールが存在するディレクトリーにコピーし、WebSphere Integration Developer のコンポーネントをワイヤリングするか、serviceDeploy コマンドを使用してエンタープライズ・アーカイブ (EAR) ファイルを生成します。

関連概念

19 ページの『Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール』
生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

関連資料

Java から XML への変換

システムは、事前定義のルールを使用して、Java 型に基づいて XML を生成します。


genMapper コマンド

genMapper コマンドは、Service Component Architecture (SCA) 参照を Java インターフェースにブリッジするコンポーネントを生成する場合に使用します。

サービス・データ・オブジェクトから Java への変換のオーバーライド

場合によっては、システムが作成するサービス・データ・オブジェクト (SDO) と Java 型オブジェクト間の変換では、要件に合わないことがあります。デフォルトのインプリメンテーションを独自のインプリメンテーションに置き換えるには、以下の手順を実行します。

始める前に

WebSphere Integration Developer または genMapper コマンドを使用して WSDL から Java 型への変換を生成していることを確認します。

このタスクについて

生成されたコードを、要件に合うコードで置き換えることにより、WSDL 型から Java 型にマップする生成済みコンポーネントをオーバーライドします。独自の Java クラスを定義している場合は、独自のマップを使用することを考慮してください。以下の手順を使用して変更を行います。

プロシージャー

1. 生成されたコンポーネントを見つけます。 コンポーネントの名前は *java_class*Mapper.component です。
2. テキスト・エディターを使用してコンポーネントを編集します。
3. 生成されたコードをコメント化し、独自のメソッドを指定します。

   コンポーネントのインプリメンテーションを含むファイル名は変更しないでください。

以下は、置き換え対象の生成されたコンポーネントの例です。

```
 private Object datatojava_get_customerAcct(DataObject myCustomerID,
    String integer)
{

  // このコードをカスタム・マッピングのためオーバーライドできます。
  // このコードをコメント化してカスタム・コードを書き込みます。

  // コンバーターに渡される Java 型は、コンバーターが作成を試行します。
  // この Java 型を変更することもできます。


  return SDOJavaObjectMediator.data2Java(customerID, integer) ;

}
```

コンポーネントおよびその他のファイルを、含んでいるモジュールが存在するディレクトリーにコピーし、WebSphere Integration Developer のコンポーネントをワイヤリングするか、serviceDeploy コマンドを使用してエンタープライズ・アーカイブ (EAR) ファイルを生成します。

関連概念

『Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール』
生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

関連資料


Java から XML への変換

システムは、事前定義のルールを使用して、Java 型に基づいて XML を生成します。


genMapper コマンド

genMapper コマンドは、Service Component Architecture (SCA) 参照を Java インターフェースにブリッジするコンポーネントを生成する場合に使用します。

Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール

生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

基本の型およびクラス

ランタイムは、サービス・データ・オブジェクトと基本の Java 型およびクラス間で単純な変換を実行します。基本の型およびクラスは以下のとおりです。

- Char または java.lang.Character
- ブール
- Java.lang.Boolean
- Byte または java.lang.Byte
- Short または java.lang.Short
- Int または java.lang.Integer
- Long または java.lang.Long
- Float または java.lang.Float
- Double または java.lang.Double
- Java.lang.String
- Java.math.BigInteger
- Java.math.BigDecimal
- Java.util.Calendar
- Java.util.Date
- Java.xml.namespace.QName
- Java.net.URI

- Byte[]

ユーザー定義の Java クラスおよび配列

Java クラスまたは配列から SDO に変換する場合、ランタイムが作成するデータ・オブジェクトは、Java 型のパッケージ名を反転して生成される URI を持ち、Java クラス名と同じ型を持ちます。例えば、Java クラス com.ibm.xsd.Customer が SDO に変換されると、URI は http://xsd.ibm.com であり、Customer 型を持ちます。次にランタイムは Java クラス・メンバーのコンテンツを検査し、値を SDO 内のプロパティーに割り当てます。

SDO から Java 型に変換する場合、ランタイムは URI を反転させてパッケージ名を生成し、SDO の型と同じ型の名前を生成します。例えば、型 Customer と、URI http://xsd.ibm.com を持つデータ・オブジェクトは、Java パッケージ com.ibm.xsd.Customer のインスタンスを生成します。次にランタイムは、SDO のプロパティーから値を抽出し、それらのプロパティーを Java クラスのインスタンス内のフィールドに割り当てます。

Java クラスがユーザー定義のインターフェースである場合、生成されるコードをオーバーライドして、ランタイムがインスタンス化できる具象クラスを作成する必要があります。ランタイムが具象クラスを作成できない場合、例外が発生します。

Java.lang.Object

Java 型が java.lang.Object の場合、生成される型は xsd:anyType です。モジュールは、すべての SDO で、このインターフェースを呼び出すことができます。ランタイムは、具象クラスを見つけることができる場合は、ユーザー定義の Java クラスと配列の場合と同様にその具象クラスをインスタンス化しようとします。見つからない場合、ランタイムは SDO を Java インターフェースに渡します。

メソッドが java.lang.Object 型を戻しても、ランタイムは、メソッドが具象型を戻す場合のみ、SDO に変換します。ランタイムは、ユーザー定義の Java クラスおよび配列から SDO への変換でも同様の変換を使用します (次の段落を参照)。

Java クラスまたは配列から SDO に変換する場合、ランタイムが作成するデータ・オブジェクトは、Java 型のパッケージ名を反転して生成される URI を持ち、Java クラス名と同じ型を持ちます。例えば、Java クラス com.ibm.xsd.Customer が SDO に変換されると、URI は http://xsd.ibm.com であり、Customer 型を持ちます。次にランタイムは Java クラス・メンバーのコンテンツを検査し、値を SDO 内のプロパティーに割り当てます。

いずれの場合でも、ランタイムが変換を完了できない場合は例外が発生します。

Java.util コンテナー・クラス

Vector、HashMap、HashSet などのような具象 Java コンテナー・クラスに変換する場合、ランタイムは該当するコンテナー・クラスをインスタンス化します。ランタイムは、ユーザー定義の Java クラスおよび配列で使用したのと同様の方法で、コンテナー・クラスにデータを取り込みます。ランタイムが具象 Java クラスを見つけられない場合、ランタイムはコンテナー・クラスに SDO を取り込みます。

具象 Java コンテナー・クラスから SDO に変換する場合、ランタイムは、『Java から XML への変換』に示される、生成されたスキーマを使用します。

Java.util インターフェース

java.util パッケージ内の特定のコンテナー・インターフェースの場合、ランタイムは以下の具象クラスをインスタンス化します。

表 1. WSDL 型から Java クラスへの変換

インターフェース	デフォルトの具象クラス
コレクション	HashSet
マップ	HashMap
リスト	ArrayList
セット	HashSet

関連タスク

16 ページの『生成される Service Component Architecture インプリメンテーションのオーバーライド』
場合によっては、システムが作成した Java コードとサービス・データ・オブジェクト (SDO) 間の変換では、要件に合わないことがあります。デフォルトの Service Component Architecture (SCA) クラスのインプリメンテーションを独自のインプリメンテーションに置き換えるには、以下の手順を実行します。

18 ページの『サービス・データ・オブジェクトから Java への変換のオーバーライド』
場合によっては、システムが作成するサービス・データ・オブジェクト (SDO) と Java 型オブジェクト間の変換では、要件に合わないことがあります。デフォルトのインプリメンテーションを独自のインプリメンテーションに置き換えるには、以下の手順を実行します。

関連資料

Java から XML への変換

システムは、事前定義のルールを使用して、Java 型に基づいて XML を生成します。

genMapper コマンド

genMapper コマンドは、Service Component Architecture (SCA) 参照を Java インターフェースにブリッジするコンポーネントを生成する場合に使用します。

第 2 章 ビジネス・プロセスおよびタスク用クライアント・アプリケーションの開発

モデル化ツールを使用して、ビジネス・プロセスやタスクを作成しデプロイすることができます。そのようなプロセスとタスクは実行時に相互作用し、例えば、プロセスが開始すると、タスクが要求されて完了します。プロセスおよびタスクとは、Business Process Choreographer Explorer を使用して対話できますが、Business Process Choreographer API を使用して、このような対話用にカスタマイズしたクライアントを開発することもできます。

このタスクについて

これらのクライアントは、Business Process Choreographer Explorer JavaServer Faces (JSF) コンポーネントを活用する Enterprise JavaBeans™ (EJB) クライアント、Web サービス・クライアント、または Web クライアントです。これらのクライアントを開発するために、Business Process Choreographer は、Enterprise JavaBeans (EJB) API と Web サービス用インターフェースを提供しています。EJB API は、別の EJB アプリケーションを含むすべての Java アプリケーションによってアクセスできます。 Web サービス用インターフェースには、Java 環境または Microsoft® .Net 環境のいずれかからアクセスできます。

ビジネス・プロセスおよびヒューマン・タスク用 EJB クライアント・アプリケーションの開発

EJB API は、WebSphere Process Server 上にインストールされているビジネス・プロセスやヒューマン・タスクを処理する EJB クライアント・アプリケーションを開発するための汎用的な方法をいくつか提供します。

このタスクについて

この Enterprise JavaBeans (EJB) API を使用すれば、以下を実行するためのクライアント・アプリケーションを作成できます。

- プロセスやタスクのライフ・サイクルの、開始から完了後の削除までの管理
- アクティビティーやプロセスの修復
- ワークグループのメンバーに対するワークロードの管理および配布

EJB API は、次の 2 種類のステートレス・セッション・エンタープライズ Bean として提供されます。

- BusinessFlowManagerService インターフェースは、ビジネス・プロセス・アプリケーション用のメソッドを備えています。
- HumanTaskManagerService インターフェースは、タスク・ベースのアプリケーション用のメソッドを備えています。

EJB API の詳細は、com.ibm.bpe.api パッケージおよび com.ibm.task.api パッケージの中の Javadoc を参照してください。

以下のステップは、 EJB アプリケーションの開発に必要なアクションの概要です。

プロシージャー

1. アプリケーションが提供する機能を決定します。
2. 使用するセッション Bean を決定します。

   アプリケーションでインプリメントするシナリオに応じて、2 つのセッション Bean のうちの 1 つ、または両方を使用することができます。
3. アプリケーションのユーザーが必要とする許可権限を判別します。

   アプリケーションのユーザーには、アプリケーションに組み込まれたメソッドを呼び出し、これらのメソッドが戻すオブジェクトとそのオブジェクトの属性を表示するための、適切な許可のロールが割り当てられている必要があります。該当するセッション Bean のインスタンスを作成するときに、WebSphere Application Server がコンテキストとそのインスタンスを関連付けます。コンテキストには、呼び出し元のプリンシパル ID、グループ・メンバーシップ・リスト、およびロールについての情報が含まれています。この情報は、それぞれの呼び出しごとに、呼び出し元の権限を確認するために使用されます。

   Javadoc には、各メソッドの許可情報が含まれています。
4. アプリケーションをレンダリングする方法を決めます。

   EJB API は、ローカル側でもリモート側でも呼び出すことができます。
5. アプリケーションを開発します。
   a. EJB API にアクセスします。
   b. EJB API を使用して、プロセスまたはタスクと対話します。
      - データを照会します。
      - データで作業を行います。

EJB API へのアクセス

Enterprise JavaBeans (EJB) API は、次の 2 種類のステートレス・セッション・エンタープライズ Bean として提供されます。ビジネス・プロセス・アプリケーションおよびタスク・アプリケーションは、 Bean のホーム・インターフェースを介して、適切なセッション・エンタープライズ Bean にアクセスします。

このタスクについて

BusinessFlowManagerService インターフェースは、ビジネス・プロセス・アプリケーション用のメソッドを備え、HumanTaskManagerService インターフェースは、タスク・ベースのアプリケーション用のメソッドを備えています。このアプリケーションは、別の Enterprise JavaBeans (EJB) アプリケーションを含む任意の Java アプリケーションを指します。

セッション Bean のリモート・インターフェースにアクセスする

EJB クライアント・アプリケーションは、Bean のリモート・ホーム・インターフェースを介して、セッション Bean のリモート・インターフェースにアクセスします。

このタスクについて

セッション Bean は、プロセス・アプリケーションに対しては BusinessFlowManager セッション Bean、タスク・アプリケーションに対しては HumanTaskManager セッション Bean のいずれかである可能性があります。

プロシージャー

1. セッション Bean のリモート・インターフェースへの参照をアプリケーション・デプロイメント記述子に追加します。 参照を以下のファイルの 1 つに追加します。
   - Java 2 Platform Enterprise Edition (J2EE) クライアント・アプリケーションの場合は、application-client.xml ファイル
   - Web アプリケーションの場合は、web.xml ファイル
   - Enterprise JavaBeans (EJB) アプリケーションの場合は、ejb-jar.xml ファイル

   プロセス・アプリケーションの場合のリモート・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-ref>
    <ejb-ref-name>ejb/BusinessFlowManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
    <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
   </ejb-ref>
   ```

   タスク・アプリケーションの場合のリモート・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-ref>
    <ejb-ref-name>ejb/HumanTaskManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.ibm.task.api.HumanTaskManagerHome</home>
    <remote>com.ibm.task.api.HumanTaskManager</remote>
   </ejb-ref>
   ```

   WebSphere Integration Developer を使用して EJB 参照をデプロイメント記述子に追加する場合、EJB 参照のバインディングが、アプリケーションのデプロイ時に自動的に作成されます。EJB 参照の追加について詳しくは、WebSphere Integration Developer の文書を参照してください。
2. 生成されたスタブをアプリケーションにパッケージします。

   ご使用のアプリケーションが、BPEContainer アプリケーションまたは TaskContainer アプリケーションを実行しているのと異なる Java 仮想マシン (JVM) で実行されている場合、以下のアクションを完了します。
   a. プロセス・アプリケーションの場合、<*install_root*>/ProcessChoreographer/client/bpe137650.jar ファイルを、ご使用のアプリケーションのエンタープライズ・アーカイブ (EAR) ファイルにパッケージします。
   b. タスク・アプリケーションの場合、<*install_root*>/ProcessChoreographer/client/task137650.jar ファイルを、ご使用のアプリケーションの EAR ファイルにパッケージします。
   c. アプリケーション・モジュールのマニフェスト・ファイル内の **Classpath** パラメーターを、JAR ファイルを含めるように設定します。

アプリケーション・モジュールは、J2EE アプリケーション、Web アプリケーション、または EJB アプリケーションの可能性があります。

d. ビジネス・プロセスまたはヒューマン・タスクで複合データ型を使用しており、クライアントが EJB アプリケーションや Web アプリケーションで動作しない場合は、対応する XSD または WSDL ファイルをアプリケーションの EAR ファイルにパッケージします。

3. Java Naming and Directory Interface (JNDI) からセッション Bean のリモート・ホーム・インターフェースを見つけます。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

// Lookup the remote home interface of the BusinessFlowManager bean
Object result =
       initialContext.lookup("java:comp/env/ejb/BusinessFlowManagerHome");

// Convert the lookup result to the proper type
   BusinessFlowManagerHome processHome =
           (BusinessFlowManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,BusinessFlowManagerHome.class);
```

セッション Bean のリモート・ホーム・インターフェースには、EJB オブジェクトの create メソッドが含まれます。 このメソッドは、セッション Bean のリモート・インターフェースを戻します。

4. セッション Bean のリモート・インターフェースにアクセスします。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
BusinessFlowManager process = processHome.create();
```

セッション Bean へのアクセス権は、呼び出し元が Bean が提供するすべてのアクションを実行できることを保証するものではありません。呼び出し元には、そのアクションに対する許可も必要になります。セッション Bean のインスタンスが作成されると、コンテキストはセッション Bean のそのインスタンスと関連付けられます。コンテキストは、呼び出し元のプリンシパル ID とグループ・メンバーシップ・リストを含み、呼び出し元が Business Process Choreographer J2EE のロールの 1 つを持っているかどうかを示します。このコンテキストを使用して、グローバル・セキュリティーが設定されていない場合でも、呼び出しごとに呼び出し元の権限を確認します。グローバル・セキュリティーが設定されていない場合、呼び出し元のプリンシパル ID の値は UNAUTHENTICATED になります。

5. サービス・インターフェースによって公開されたビジネス関数を呼び出します。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
process.initiate("MyProcessModel",input);
```

アプリケーションからの呼び出しは、トランザクションとして実行されます。トランザクションは、以下のいずれかの方法で確立されて終了します。

- WebSphere Application Server から自動的に (デプロイメント記述子が TX_REQUIRED を指定)。
- アプリケーションから明示的に。アプリケーションの呼び出しを 1 つのトランザクションにバンドルすることができます。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

// On successful return, commit the transaction
transaction.commit();
```

ヒント: データベース・ロック競合を防ぐには、並列で以下のようなステートメントの実行を避けるようにします。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

getActivityInstance メソッドおよびその他の読み取り操作は、読み取りロックを設定します。この例では、アクティビティー・インスタンス上の読み取りロックは、アクティビティー・インスタンス上の更新ロックにアップグレードされます。これにより、トランザクションが並列で実行されるときに、データベース・デッドロックが発生することがあります。

例

以下に、ステップ 3 から 5 でタスク・アプリケーションを探す方法の例を示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

// Lookup the remote home interface of the HumanTaskManager bean
Object result =
         initialContext.lookup("java:comp/env/ejb/HumanTaskManagerHome");

// Convert the lookup result to the proper type
HumanTaskManagerHome taskHome =
           (HumanTaskManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,HumanTaskManagerHome.class);

...
//Access the remote interface of the session bean.
HumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

セッション Bean のローカル・インターフェースにアクセスする

EJB クライアント・アプリケーションは、Bean のローカル・ホーム・インターフェースを介してセッション Bean のローカル・インターフェースにアクセスします。

このタスクについて

セッション Bean は、プロセス・アプリケーションに対しては BusinessFlowManager セッション Bean、ヒューマン・タスク・アプリケーションに対しては HumanTaskManager セッション Bean のいずれかである可能性があります。

プロシージャー

1. セッション Bean のローカル・インターフェースへの参照をアプリケーション・デプロイメント記述子に追加します。 参照を以下のファイルの 1 つに追加します。
   - Java 2 Platform Enterprise Edition (J2EE) クライアント・アプリケーションの場合は、application-client.xml ファイル
   - Web アプリケーションの場合は、web.xml ファイル
   - Enterprise JavaBeans (EJB) アプリケーションの場合は、ejb-jar.xml ファイル

   プロセス・アプリケーションの場合のローカル・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-local-ref>
    <ejb-ref-name>ejb/LocalBusinessFlowManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
    <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
   </ejb-local-ref>
   ```

   タスク・アプリケーションの場合のローカル・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-local-ref>
    <ejb-ref-name>ejb/LocalHumanTaskManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
    <local>com.ibm.task.api.LocalHumanTaskManager</local>
   </ejb-local-ref>
   ```

   WebSphere Integration Developer を使用して EJB 参照をデプロイメント記述子に追加する場合、EJB 参照のバインディングが、アプリケーションのデプロイ時に自動的に作成されます。EJB 参照の追加について詳しくは、WebSphere Integration Developer の文書を参照してください。
2. Java Naming and Directory Interface (JNDI) からセッション Bean のローカル・ホーム・インターフェースを見つけます。

   以下の例では、プロセス・アプリケーションでのこのステップを示します。

   ```
   // Obtain the default initial JNDI context
   InitialContext initialContext = new InitialContext();

   // Lookup the local home interface of the BusinessFlowManager bean

    LocalBusinessFlowManagerHome processHome =
         (LocalBusinessFlowManagerHome)initialContext.lookup
         ("java:comp/env/ejb/LocalBusinessFlowManagerHome");
   ```

   セッション Bean のローカル・ホーム・インターフェースには、EJB オブジェクトの create メソッドが含まれます。このメソッドは、セッション Bean のローカル・インターフェースを戻します。
3. セッション Bean のローカル・インターフェースにアクセスします。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
LocalBusinessFlowManager process = processHome.create();
```

セッション Bean へのアクセス権は、呼び出し元が Bean が提供するすべてのアクションを実行できることを保証するものではありません。呼び出し元には、そのアクションに対する許可も必要になります。セッション Bean のインスタンスが作成されると、コンテキストはセッション Bean のそのインスタンスと関連付けられます。コンテキストは、呼び出し元のプリンシパル ID とグループ・メンバーシップ・リストを含み、呼び出し元が Business Process Choreographer J2EE のロールの 1 つを持っているかどうかを示します。このコンテキストを使用して、グローバル・セキュリティーが設定されていない場合でも、呼び出しごとに呼び出し元の権限を確認します。グローバル・セキュリティーが設定されていない場合、呼び出し元のプリンシパル ID の値は UNAUTHENTICATED になります。

4. サービス・インターフェースによって公開されたビジネス関数を呼び出します。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
process.initiate("MyProcessModel",input);
```

アプリケーションからの呼び出しは、トランザクションとして実行されます。トランザクションは、以下のいずれかの方法で確立されて終了します。

- WebSphere Application Server から自動的に (デプロイメント記述子が TX_REQUIRED を指定)。
- アプリケーションから明示的に。アプリケーションの呼び出しを 1 つのトランザクションにバンドルすることができます。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

// On successful return, commit the transaction
transaction.commit();
```

ヒント: データベース・デッドロックを防ぐには、並列で以下のようなステートメントの実行を避けるようにします。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

getActivityInstance メソッドおよびその他の読み取り操作は、読み取りロックを設定します。この例では、アクティビティー・インスタンス上の読み取りロックは、アクティビティー・インスタンス上の更新ロックにアップグレードされま

す。これにより、トランザクションが並列で実行されるときに、データベース・デッドロックが発生することがあります。

例

以下に、ステップ 2 から 4 でタスク・アプリケーションを探す方法の例を示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

//Lookup the local home interface of the HumanTaskManager bean
LocalHumanTaskManagerHome taskHome =
        (LocalHumanTaskManagerHome)initialContext.lookup
        ("java:comp/env/ejb/LocalHumanTaskManagerHome");

...
//Access the local interface of the session bean
LocalHumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

ビジネス・プロセスおよびタスク関連のオブジェクトの照会

クライアント・アプリケーションは、ビジネス・プロセスとタスク関連オブジェクトを操作します。データベース内のビジネス・プロセス・オブジェクトおよびタスク関連のオブジェクトを照会して、これらのオブジェクトの特定のプロパティーを取得することができます。

このタスクについて

Business Process Choreographer を構成すると、リレーショナル・データベースは、ビジネス・プロセス・コンテナーおよびタスク・コンテナーの両方に関連付けられます。このデータベースは、ビジネス・プロセスとタスクの管理用のすべてのテンプレート (モデル) とインスタンス (ランタイム) のデータを保管します。そのデータを照会するには、SQL 形式の構文を使用します。

単発の照会を実行して、オブジェクトの特定のプロパティーを取得することができます。また、頻繁に使用する照会を保管しておいて、この保管照会文をアプリケーションに組み込むこともできます。

ビジネス・プロセスおよびタスク関連オブジェクトに対する照会

サービス API の query メソッドまたは queryAll メソッドを使用して、ビジネス・プロセスとタスクについて保管されている情報を取得します。

すべてのユーザーが query メソッドを呼び出すことができます。このメソッドは、作業項目が存在しているオブジェクトのプロパティーを戻します。 queryAll メソッドは、J2EE ロール BPESystemAdministrator、TaskSystemAdministrator、BPESystemMonitor、TaskSystemMonitor のいずれかが割り当てられているユーザーのみが呼び出すことができます。このメソッドは、データベースに格納されているすべてのオブジェクトのプロパティーを戻します。

すべての API 照会は SQL 照会にマップされます。生成される SQL 照会の形式は、次の要因によって異なります。

- 照会が、J2EE ロールが割り当てられているユーザーによって呼び出されたかどうか。
- 照会対象オブジェクト。オブジェクトのプロパティーを照会するために、事前定義データベース・ビューが提供されています。
- from 文節、結合条件、およびユーザー固有のアクセス制御条件の挿入。

照会には、カスタム・プロパティーと変数プロパティーの両方を含めることができます。複数のカスタム・プロパティーまたは変数プロパティーを照会に含める場合、対応するデータベース表で自己結合が行われます。データベース・システムによっては、これらの query() 呼び出しによりパフォーマンスへの影響が出ることがあります。

また、createStoredQuery メソッドを使用して、Business Process Choreographer データベースに照会を保管することもできます。保管照会文を定義する場合は、照会基準を指定します。この基準は、保管照会文が実行されるとき、すなわち実行時にデータがアセンブルされるときに動的に適用されます。保管照会文にパラメーターが含まれている場合は、照会を実行するときにこれらのパラメーターも解決されます。

Business Process Choreographer API について詳しくは、プロセス関連メソッドの com.ibm.bpe.api パッケージおよびタスク関連メソッドの com.ibm.task.api パッケージ内にある Javadoc を参照してください。

API query メソッドの構文:

Business Process Choreographer API の照会の構文は、SQL 照会の構文に似ています。照会には、select 文節、where 文節、order-by 文節、スキップ・タプル・パラメーター、しきい値パラメーター、および時間帯パラメーターを組み込むことができます。

照会の構文はオブジェクト・タイプによって異なります。以下の表は、異なるオブジェクト・タイプごとの構文を示しています。

表 2.

オブジェクト	構文
プロセス・テンプレート	`ProcessTemplateData[] queryProcessTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);`
タスク・テンプレート	`TaskTemplate[] queryTaskTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);`
ビジネス・プロセスとタスク関連データ	`QueryResultSet query (java.lang.String selectClause, java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer skipTuples java.lang.Integer threshold, java.util.TimeZone timezone);`

select 文節:

照会関数の select 文節は、照会によって戻されるオブジェクト・プロパティーを示します。

select 文節は、照会結果を記述します。これは、戻すオブジェクト・プロパティー (結果の列) を識別する名前のリストを指定します。構文は SQL SELECT 文節の構文と似ており、コンマを使用して文節のパーツを区切ります。文節の各パーツは、事前定義されているビューのいずれか 1 つの列を指定する必要があります。列は、ビュー名と列名を使用して完全に指定される必要があります。QueryResultSet オブジェクトで戻される列は、select 文節で指定されている列と同じ順序で表示されます。

select 文節は、AVG()、SUM()、MIN()、または MAX() などの SQL 集約関数はサポートしていません。

複数の名前と値の対のプロパティー (カスタム・プロパティーや、照会可能な変数のプロパティーなど) を選択する場合は、ビュー名に 1 桁のカウンターを追加します。このカウンターは 1 から 9 の値を取ることができます。

select 文節の例

- ″WORK_ITEM.OBJECT_TYPE, WORK_ITEM.REASON″

  関連オブジェクトのオブジェクト・タイプ、および作業項目の割り当て理由を取得します。

- ″DISTINCT WORK_ITEM.OBJECT_ID″

  呼び出し元が作業項目を所有しているオブジェクトの ID すべてを重複なしで取得します。

- ″ACTIVITY.TEMPLATE_NAME, WORK_ITEM.REASON″

  呼び出し元が作業項目を所有しているアクティビティーの名前、およびその割り当て理由を取得します。

- ″ACTIVITY.STATE, PROCESS_INSTANCE.STARTER″

  アクティビティーの状態、およびその関連プロセス・インスタンスのスターターを取得します。

- ″DISTINCT TASK.TKIID, TASK.NAME″

  呼び出し元が作業項目を所有しているタスクの ID と名前すべてを重複なしで取得します。

- ″TASK_CPROP1.STRING_VALUE, TASK_CPROP2.STRING_VALUE″

  さらに where 文節でも指定されているカスタム・プロパティーの値を取得します。

- ″QUERY_PROPERTY1.STRING_VALUE, QUERY_PROPERTY2.INT_VALUE

  照会できる変数のプロパティーの値を取得します。これらの部分は、さらに where 文節でも指定されています。

- ″COUNT(DISTINCT TASK.TKIID)″

  where 文節の条件を満たす固有のタスクの作業項目の数を数えます。

where 文節:

照会関数の中の where 文節は、照会ドメインに適用するフィルター基準を記述します。

where 文節の構文は、SQL WHERE 文節の構文に似ています。文節から明示的に SQL を追加したり、API where 文節に述部を結合したりする必要はありません。これらの構成要素は照会の実行時に自動的に追加されます。フィルター基準を適用しない場合は、where 文節に `null` を指定する必要があります。

where 文節構文は以下のものをサポートします。

- キーワード: AND、OR、NOT
- 比較演算子: =、<=、<、<>、>、>=、LIKE

  LIKE 操作では、照会されるデータベースに定義されているワイルドカード文字がサポートされます。
- 設定操作: IN

以下の規則も適用されます。

- オブジェクト ID 定数を `ID('string-rep-of-oid')` に指定します。
- `BIN('UTF-8 string')` としてバイナリー定数を指定します。
- 整数列挙型の代わりにシンボリック定数を使用します。例えば、アクティビティー状態式 `ACTIVITY.STATE=2` を指定する代わりに、`ACTIVITY.STATE=ACTIVITY.STATE.STATE_READY` を指定します。
- 比較ステートメント内のプロパティーの値に単一引用符 (') が含まれる場合、例えば `"TASK_CPROP.STRING_VALUE='d''automatisation'"` のように、引用符を二重にしてください。
- ビュー名に 1 桁のサフィックスを追加して、複数の名前と値の対のプロパティー (カスタム・プロパティーなど) を参照します。例: `"TASK_CPROP1.NAME='prop1' AND "TASK_CPROP2.NAME='prop2'"`
- タイム・スタンプ定数を `TS('yyyy-mm-ddThh:mm:ss')` に指定します。現在日付を参照するには、タイム・スタンプを `CURRENT_DATE` に指定します。

  タイム・スタンプには、最低でも日付または時間の値を指定する必要があります。
  - 日付のみを指定すると、時間値はゼロに設定されます。
  - 時間のみを指定すると、日付は現在の日付に設定されます。
  - 日付を指定する場合、年は 4 桁の定数で構成する必要があります。月および日の値はオプションです。欠落している月および日の値は、01 に設定されます。例えば、`TS('2003')` は `TS('2003-01-01T00:00:00')` と同じです。
  - 日付を指定する場合、この値は 24 時間制で記述されます。例えば、現在日付が 2003 年 1 月 1 日の場合、`TS('T16:04')` または `TS('16:04')` は、`TS('2003-01-01T16:04:00')` と同じです。

where 文節の例

- オブジェクト ID と既存の ID の比較

  ```
  "WORK_ITEM.WIID = ID('_WI:800c00ed.df8d7e7c.feffff80.38')"
  ```

この型の where 文節は、通常、直前の呼び出しの既存オブジェクト ID を使用して、動的に作成されます。このオブジェクト ID が *wiid1* 変数に保管されている場合、文節の構文は次のようになります。

```
"WORK_ITEM.WIID = ID('" + wiid1.toString() + "')"
```

- タイム・スタンプの使用

```
"ACTIVITY.STARTED >= TS('2002-06-1T16.00.00')"
```

- シンボリック定数の使用

```
"WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER"
```

- ブール値 true および false の使用

```
"ACTIVITY.BUSINESS_RELEVANCE = TRUE"
```

- カスタム・プロパティーの使用

```
"TASK_CPROP1.NAME = 'prop1' AND " TASK_CPROP1.STRING_VALUE = 'v1' AND
 TASK_CPROP2.NAME = 'prop2' AND " TASK_CPROP2.STRING_VALUE = 'v2'"
```

order-by 文節:

照会関数内の order-by 文節は、照会結果セットのソート基準を指定します。

結果をソートするビューの列のリストを指定できます。これらの列は、ビューと列の名前で完全に修飾されている必要があります。select 文節に含まれている列を指定することをお勧めします。

order-by 文節の構文は、SQL の order-by 文節の構文と似ており、文節の各パーツをコンマで区切ります。列を昇順にソートする場合は `ASC` を指定し、列を降順にソートする場合は `DESC` を指定します。照会結果セットをソートしない場合は、order-by 文節で `null` を指定する必要があります。

ソート基準はサーバーに適用されます。つまり、ソートにサーバーのロケールが使用されます。複数の列を指定すると、照会結果セットはまず最初の列の値で順序付けされ、次に 2 番目の列の値で順序付けされる、という具合に続きます。SQL 照会のように、order-by 文節の列を、位置によって指定することはできません。

order-by 文節の例

- ″PROCESS_TEMPLATE.NAME″

  照会結果を、プロセス・テンプレート名でアルファベット順にソートします。
- ″PROCESS_INSTANCE.CREATED, PROCESS_INSTANCE.NAME DESC″

  照会結果を作成日でソートし、特定の日付の場合はその結果を、プロセス・インスタンス名でアルファベット順の逆順にソートします。
- ″ACTIVITY.OWNER, ACTIVITY.TEMPLATE_NAME, ACTIVITY.STATE″

  照会結果を、アクティビティー所有者、アクティビティー・テンプレート名、アクティビティーの状態の順でソートします。

スキップ・タプル・パラメーター:

スキップ・タプル・パラメーターは、無視して照会結果セットで呼び出し元に戻さない照会結果セット・タプルの数を指定します。この数は、照会結果セットの先頭から数えます。

このパラメーターをしきい値パラメーターと一緒に使用してクライアント・アプリケーションでページングをインプリメントします (例えば、最初の 20 項目を取得し、それから次の 20 項目を取得し、以下同様に項目を取得します)。

このパラメーターを null に設定したときに、しきい値パラメーターを設定していないと、すべての適格なタプルが戻されます。

スキップ・タプル・パラメーターの例

- new Integer(5)

  最初の 5 つの適格なタプルを戻さないように指定します。

しきい値パラメーター:

照会関数のしきい値パラメーターは、照会の結果セットとしてサーバーからクライアントに戻されるオブジェクトの数を制限します。

実動シナリオでの照会結果セットには数千から数百万の項目が含まれる可能性があるので、常にしきい値を指定するのがベスト・プラクティスです。しきい値パラメーターは、例えば少数の項目のみが 1 度に表示されるグラフィカル・ユーザー・インターフェースなどで有用な場合があります。しきい値パラメーターを適宜設定すると、データベース照会が高速化し、サーバーからクライアントへ転送する必要のあるデータが少なくなります。

このパラメーターを null に設定したときに、スキップ・タプル・パラメーターを設定していないと、適格なオブジェクトがすべて戻されます。

しきい値パラメーターの例

- new Integer(50)

  50 個の適格なタプルを戻すように指定します。

時間帯パラメーター:

照会関数の時間帯パラメーターは、照会内のタイム・スタンプ定数の時間帯を定義します。

照会を開始するクライアントと照会を処理するサーバーの間で、時間帯が異なることがあります。時間帯パラメーターを使用して、where 文節で使用されるタイム・スタンプ定数の時間帯を、例えば地方時を指定するように指定します。照会の結果セットで戻される日付には、照会で指定したものと同じ時間帯が設定されます。

このパラメーターを null に設定すると、タイム・スタンプ定数は協定世界時 (UTC) と想定されます。

時間帯パラメーターの例

- ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 (String)null,
                 (Integer)null,
                 java.util.TimeZone.getDefault() );
  ```

  2005 年 1 月 1 日の 17:40 地方時より後に開始されたアクティビティーのオブジェクト ID を戻します。
  ```

- ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 (String)null, (Integer)null, (TimeZone)null);
  ```

  2005 年 1 月 1 日の 17:40 UTC より後に開始されたアクティビティーのオブジェクト ID を戻します。この指定は、例えば東部標準時より 6 時間早い時間です。

保管照会文のパラメーター:

保管照会文は、データベースに保管され、名前で識別される照会のことです。適格なタプルは、照会が実行されるときに動的にアセンブルされます。保管照会文を再使用可能にするには、実行時に解決される照会定義のパラメーターを使用できます。

例えば、顧客名を保管するカスタム・プロパティーを定義したとします。特定の顧客 ACME Co. に関連したタスクを戻すように、照会を定義することができます。この情報を照会する場合、照会内の where 文節は以下の例のようになります。

```
String whereClause =
  "TASK.STATE = TASK.STATE.STATE_READY
  AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
  AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = 'ACME Co.'";
```

顧客 BCME Ltd. も検索できるようにこの照会を再使用可能にするには、カスタム・プロパティーの値に対してパラメーターを使用できます。パラメーターをタスク照会に追加する場合、以下の例のようになります。

```
String whereClause =
  "TASK.STATE = TASK.STATE.STATE_READY
  AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
  AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = '@param1'";
```

`@param1` パラメーターは、query メソッドに受け渡されるパラメーターのリストから、実行時に解決されます。以下の規則は、照会内でのパラメーターの使用に適用されます。

- パラメーターは where 文節でのみ使用できる。
- パラメーターはストリングである。
- パラメーターは実行時にストリングの置換を使用して置き換えられる。特殊文字が必要な場合、where 文節に指定するか、実行時にパラメーターの一部として受け渡す必要があります。
- パラメーター名は、ストリング `@param` を整数と連結したもので構成される。最小番号は 1 で、実行時に照会 API に受け渡されるパラメーターのリスト内の最初の項目を指します。
- パラメーターは where 文節内で複数回使用することができ、出現するすべてのパラメーターは同じ値で置き換えられます。

関連タスク

65 ページの『保管照会文の管理』
保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

照会結果:
```

照会結果セットには、照会の結果が入ります。

結果セットのエレメントは、呼び出し元により指定された where 文節の条件を満たし、かつ呼び出し元に対し表示が許可されているオブジェクトのプロパティーです。エレメントは、API の次のメソッドを使用して相対的に読み取るか、あるいは最初と最後のメソッドを使用して絶対的に読み取ります。照会結果セットの暗黙カーソルは初めは最初のエレメントの前に配置されるため、エレメントを読み取る前に、最初または次のメソッドのいずれかを呼び出す必要があります。 size メソッドを使用して、セット内のエレメント数を判別することができます。

照会結果セットのエレメントは、作業項目とそれに関連する参照オブジェクト (アクティビティー・インスタンスやプロセス・インスタンスなど) の選択済み属性を構成します。 QueryResultSet エレメントの最初の属性 (列) は、照会要求の select 文節で指定されている最初の属性の値を指定します。 QueryResultSet エレメントの 2 番目の属性 (列) は、照会要求の select 文節で指定されている 2 番目の属性の値を指定する、という具合に続きます。

属性の値は、その属性タイプと互換性のあるメソッドを呼び出すことによって、また、適切な列インデックスを指定することによって、取得することができます。列インデックスの番号付けは 1 から始まります。

属性タイプ	**メソッド**
ストリング	getString
OID	getOID
タイム・スタンプ	getTimestamp getString getTimestampAsLong
整数	getInteger getShort getLong getString getBoolean
ブール	getBoolean getShort getInteger getLong getString
byte[]	getBinary

例:

以下の照会が実行されます。

```
QueryResultSet resultSet = process.query("ACTIVITY.STARTED,
                                         ACTIVITY.TEMPLATE_NAME AS NAME,
                                         WORK_ITEM.WIID, WORK_ITEM.REASON",
                                         (String)null, (String)null,
                                         (Integer)null, (TimeZone)null);
```

戻される照会結果セットには、以下の 4 つの列があります。
- 列 1 はタイム・スタンプ

- 列 2 はストリング
- 列 3 はオブジェクト ID
- 列 4 は整数

以下のメソッドを使用して、属性値を取得することができます。

```
while (resultSet.next())
{
 java.util.Calendar activityStarted = resultSet.getTimestamp(1);
 String templateName = resultSet.getString(2);
 WIID wiid = (WIID) resultSet.getOID(3);
 Integer reason = resultSet.getInteger(4);
}
```

結果セットの表示名を、例えば、印刷されるテーブルの見出しなどに使用することができます。これらの名前は、ビューの列名、または照会の AS 文節で定義された名前です。以下のメソッドを使用して、例中の表示名を取得することができます。

```
resultSet.getColumnDisplayName(1) returns "STARTED"
resultSet.getColumnDisplayName(2) returns "NAME"
resultSet.getColumnDisplayName(3) returns "WIID"
resultSet.getColumnDisplayName(4) returns "REASON"
```

ユーザー固有のアクセス条件:

SQL SELECT ステートメントが API 照会から生成されるときに、ユーザー固有のアクセス条件が追加されます。この条件により、呼び出し元により指定されている条件に一致し、呼び出し元に対し許可されているオブジェクトのみが呼び出し元に戻されます。

追加されるアクセス条件は、ユーザーがシステム管理者であるかどうかによって異なります。

システム管理者以外のユーザーが呼び出した照会

生成される SQL WHERE 文節では、API where 文節と、ユーザー固有のアクセス制御条件が結合されます。この照会は、ユーザーに対しアクセスが許可されているオブジェクト、つまりユーザーが作業項目を所有しているオブジェクトのみを取得します。作業項目とは、ビジネス・オブジェクト (タスクやプロセスなど) の許可ロールへのユーザーまたはユーザー・グループの割り当てを表します。例えば、ユーザー John Smith が特定のタスクの潜在的所有者ロールのメンバーである場合、この関係を表す作業項目オブジェクトがあります。

例えば、グループ作業項目が無効な場合に、システム管理者以外のユーザーがタスクを照会すると、以下のアクセス条件が WHERE 文節に追加されます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'user'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

したがって John Smith が、自身が潜在的所有者であるタスクのリストを取得する場合、API where 文節は次のようになります。

```
"WORK_ITEM.REASON == WORK_ITEM.REASON.REASON_POTENTIAL_OWNER"
```

この API where 文節により、SQL ステートメントに次のアクセス条件が追加されます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'JohnSmith'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
AND WI.REASON = 1
```

つまり、John Smith が、自身がプロセス・リーダーまたはプロセス管理者であるアクティビティーやタスクと、作業項目を所有していないアクティビティーやタスクを表示するには、 PROCESS_INSTANCE ビューのプロパティー (PROCESS_INSTANCE.PIID など) を照会の select、where、または order-by 文節に追加する必要があります。

グループ作業項目が有効な場合は、ユーザーに対し、グループがアクセスできるオブジェクトへのアクセスを許可するアクセス条件が WHERE 文節に追加されます。

システム管理者が呼び出した照会

システム管理者は、query メソッドを呼び出し、作業項目が関連付けられているオブジェクトを取得できます。この場合、生成される SQL 照会には WORK_ITEM ビューとの結合が追加されますが、WORK_ITEM.OWNER_ID のアクセス制御条件は追加されません。

この場合、タスクの SQL 照会には以下が含まれます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
```

queryAll 照会

このタイプの照会を呼び出すことができるのは、システム管理者またはシステム・モニターのみです。アクセス制御条件も WORK_ITEM ビューとの結合も追加されません。このタイプの照会では、すべてのオブジェクトの全データが戻されます。

query メソッドと queryAll メソッドの例:

以下の例は、標準的な各種 API 照会の構文と、照会の実行時に生成される関連 SQL ステートメントを示します。

例: 作動可能状態のタスクの照会:

この例では、query メソッドを使用して、ログオン・ユーザーが作業可能なタスクを取得する方法を示します。

John Smith は、自分に割り当てられているタスクを一覧表示します。ユーザーがタスクの作業を行うには、そのタスクが作動可能状態になっている必要があります。ログオン・ユーザーには、そのタスクの潜在的所有者作業項目も必要です。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT TASK.TKIID",
           "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
             AND " +
           "TASK.STATE = TASK.STATE.STATE_READY AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
            (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.KIND IN ( 101, 105 )
   AND    TA.STATE = 2
   AND    WI.REASON = 1
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND  WI.EVERYBODY = true )
```

API 照会を特定のプロセスのタスク (sampleProcess など) に限定する場合、この照会は次のようになります。

```
query( "DISTINCT TASK.TKIID",
           "PROCESS_TEMPLATE.NAME = 'sampleProcess' AND "+
           "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
             AND " +
           "TASK.STATE = TASK.STATE.STATE_READY AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
            (String)null, (String)null, (Integer)null, (TimeZone)null )
```

例: 要求済み状態のタスクの照会:

この例では、query メソッドを使用して、ログオン・ユーザーが要求したタスクを取得する方法を示します。

ユーザー John Smith は、自身が要求したタスクのうち、まだ要求済み状態であるタスクを検索します。「John Smith により要求された」ことを指定する条件は TASK.OWNER = 'JohnSmith' です。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT TASK.TKIID",
           "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
           "TASK.OWNER = 'JohnSmith'",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.STATE = 8
   TA.OWNER = 'JohnSmith'
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

タスクが要求されると、タスクの所有者に対して作業項目が作成されます。したがって、John Smith が要求したタスクを取得する照会のもう 1 つの作成方法として、TASK.OWNER = 'JohnSmith' を使用する代わりに、次の条件を照会に追加する方法があります。

```
WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER
```

照会は、次のコード・スニペットのようになります。

```
query( "DISTINCT TASK.TKIID",
           "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.STATE = 8
   AND    WI.REASON = 4
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

John は休暇に入るため、所属するチームのリーダーである Anne Grant が、John の現在の作業割り当てを確認するとします。Anne にはシステム管理者権限が付与されています。呼び出す照会は、John が呼び出した照会と同じです。ただし Anne は管理者であるため、生成される SQL ステートメントが異なります。生成される SQL ステートメントを次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  TA.TKIID = WI.OBJECT_ID =
   AND    TA.STATE = 8
   AND   TA.OWNER = 'JohnSmith')
```

Anne は管理者であるため、アクセス制御条件が WHERE 文節に追加されません。

例: エスカレーションの照会:

この例では、query メソッドを使用して、ログオン・ユーザーのエスカレーションを取得する方法を示します。

タスクがエスカレートされると、エスカレーション受信者作業項目が作成されます。ユーザー Mary Jones が、Mary 自身にエスカレートされたタスクのリストを表示するとします。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT ESCALATION.ESIID, ESCALATION.TKIID",
         "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_ESCALATION_RECEIVER",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT ESCALATION.ESIID, ESCALATION.TKIID
  FROM   ESCALATION ESC, WORK_ITEM WI
  WHERE  ESC.ESIID = WI.OBJECT_ID
  AND    WI.REASON  = 10
  AND
 ( WI.OWNER_ID = 'MaryJones' OR WI.OWNER_ID = null AND  WI.EVERYBODY = true )
```

例: queryAll メソッドの使用:

この例では、queryAll メソッドを使用して、1 つのプロセス・テンプレートに属するアクティビティーをすべて取得する方法を示します。

queryAll メソッドは、システム管理者権限またはシステム・モニター権限が付与されているユーザーだけが使用できます。プロセス・テンプレート sampleProcess に属するすべてのアクティビティーを取得する照会の queryAll メソッド呼び出しを、次のコード・スニペットに示します。

```
queryAll( "DISTINCT ACTIVITY.AIID",
       "PROCESS_TEMPLATE.NAME = 'sampleProcess'",
       (String)null, (String)null, (Integer)null, (TimeZone)null )
```

この API 照会から生成される SQL 照会を、次のコード・スニペットに示します。

```
SELECT DISTINCT ACTIVITY.AIID
  FROM   ACTIVITY AI, PROCESS_TEMPLATE PT
  WHERE  AI.PTID = PT.PTID
  AND    PT.NAME = 'sampleProcess'
```

この呼び出しは管理者により実行されるため、生成される SQL ステートメントにアクセス制御条件は追加されません。WORK_ITEM ビューとの結合も追加されません。つまり、この照会では、プロセス・テンプレートのすべてのアクティビティー (作業項目のないアクティビティーを含む) が取得されます。

例: 照会への照会プロパティーの組み込み:

この例では、query メソッドを使用して、ビジネス・プロセスに属するタスクを取得する方法を示します。このプロセスに対して定義されている照会プロパティーを、検索に組み込むとします。

例えば、1 つのビジネス・プロセスに属し、作動可能状態にあるヒューマン・タスクをすべて検索するとします。プロセスには照会プロパティー **customerID** とその値 CID_12345、および名前空間があります。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
                   PROCESS_INSTANCE.NAME",
          " QUERY_PROPERTY.NAME = 'customerID' AND " +
          " QUERY_PROPERTY.STRING_VALUE = 'CID_12345' AND " +
          " QUERY_PROPERTY.NAMESPACE =
             'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

2 番目の照会プロパティー (**Priority** など) と特定の名前空間を照会に追加する場合、照会の query メソッド呼び出しは次のようになります。

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
                   PROCESS_INSTANCE.NAME",
          " QUERY_PROPERTY1.NAME = 'customerID' AND " +
          " QUERY_PROPERTY1.STRING_VALUE = 'CID_12345' AND " +
          " QUERY_PROPERTY1.NAMESPACE =
                 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " QUERY_PROPERTY2.NAME = 'Priority' AND " +
          " QUERY_PROPERTY2.NAMESPACE =
                 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

複数の照会プロパティーを照会に追加する場合は、コード・スニペットに示されているように、追加する各プロパティーに番号を付ける必要があります。ただし、カスタム・プロパティーの照会を実行するとパフォーマンスに影響します。照会に含まれているカスタム・プロパティーの数に応じてパフォーマンスが低下します。

例: 照会へのカスタム・プロパティーの組み込み:

この例では、query メソッドを使用して、カスタム・プロパティーが指定されたタスクを取得する方法を示します。

例えば、カスタム・プロパティー **customerID** とその値 CID_12345 が指定されており、作動可能状態にあるヒューマン・タスクをすべて検索するとします。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query ( " DISTINCT TASK.TKIID ",
          " TASK_CPROP.NAME = 'customerID' AND " +
          " TASK_CPROP.STRING_VALUE = 'CID_12345' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

タスクとそのカスタム・プロパティーを取得する場合、照会の query メソッド呼び出しは次のようになります。

```
query ( " DISTINCT TASK.TKIID, TASK_CPROP.NAME, TASK_CPROP.STRING_VALUE",
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TA.TKIID , TACP.NAME , TACP.STRING_VALUE
   FROM  TASK TA LEFT JOIN TASK_CPROP TACP ON (TA.TKIID = TACP.TKIID),
         WORK_ITEM WI
   WHERE WI.OBJECT_ID = TA.TKIID
   AND    TA.KIND IN ( 101, 105 )
   AND    TA.STATE = 2
   AND  (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID IS NULL AND WI.EVERYBODY = 1 )
```

この SQL ステートメントには、TASK ビューと TASK_CPROP ビューの外部結合が含まれています。つまり、WHERE 文節の条件を満たすタスクは、カスタム・プロパティーが含まれていない場合でも取得されます。

ビジネス・プロセス・オブジェクトおよびヒューマン・タスク・オブジェクトの照会のための事前定義ビュー:

ビジネス・プロセス・オブジェクトおよびヒューマン・タスク・オブジェクト用に、事前定義データベース・ビューが提供されています。これらのオブジェクトの参照データを照会する場合は、これらのビューを使用します。

事前定義ビューを使用する場合は、ビューの列に明示的に述部を結合する必要はありません。これらの構成要素は自動的に追加されます。このデータを照会するには、サービス API (BusinessFlowManagerService または HumanTaskManagerService) の汎用照会関数を使用することができます。HumanTaskManagerDelegate API の対応するメソッド、または ExecutableQuery インターフェースのインプリメンテーションによって提供される定義済み照会を使用することもできます。

注: 説明されていない列がビューに含まれていることがあります。このような列は内部でのみ使用される列です。

ACTIVITY ビュー:

この定義済みデータベース・ビューは、アクティビティーの照会に使用します。

表 3. ACTIVITY ビュー内の列

列名	タイプ	コメント
PIID	ID	プロセス・インスタンス ID。
AIID	ID	アクティビティー・インスタンス ID。
PTID	ID	プロセス・テンプレート ID。
ATID	ID	アクティビティー・テンプレート ID。
KIND	整数	アクティビティーの種類。指定可能な値は、以下のとおりです。 KIND_INVOKE (21) KIND_RECEIVE (23) KIND_REPLY (24) KIND_THROW (25) KIND_RETHROW (46) KIND_TERMINATE (26) KIND_WAIT (27) KIND_COMPENSATE (29) KIND_SEQUENCE (30) KIND_EMPTY (3) KIND_SWITCH (32) KIND_WHILE (34) KIND_PICK (36) KIND_FLOW (38) KIND_SCOPE (40) KIND_SCRIPT (42) KIND_STAFF (43) KIND_ASSIGN (44) KIND_CUSTOM (45) KIND_FOR_EACH_PARALLEL (49) KIND_FOR_EACH_SERIAL (47)

表 3. ACTIVITY ビュー内の列 (続き)

列名	タイプ	コメント
COMPLETED	タイム・スタンプ	アクティビティーの完了時刻。
ACTIVATED	タイム・スタンプ	アクティビティーが活動化された時刻。
FIRST_ACTIVATED	タイム・スタンプ	そのアクティビティーが初めて活動化された時刻。
STARTED	タイム・スタンプ	アクティビティーの開始時刻。
STATE	整数	アクティビティーの状態。指定可能な値は、以下のとおりです。 STATE_INACTIVE (1) STATE_READY (2) STATE_RUNNING (3) STATE_PROCESSING_UNDO (14) STATE_SKIPPED (4) STATE_FINISHED (5) STATE_FAILED (6) STATE_TERMINATED (7) STATE_CLAIMED (8) STATE_TERMINATING (9) STATE_FAILING (10) STATE_WAITING (11) STATE_EXPIRED (12) STATE_STOPPED (13)
OWNER	ストリング	所有者のプリンシパル ID。
DESCRIPTION	ストリング	アクティビティー・テンプレートの説明にプレースホルダーが含まれている場合、この列には、解決済みのプレースホルダーを所有するアクティビティー・インスタンスの説明が入ります。
TEMPLATE_NAME	ストリング	関連するアクティビティー・テンプレートの名前。
TEMPLATE_DESCR	ストリング	関連するアクティビティー・テンプレートの説明。
BUSINESS_RELEVANCE	ブール	アクティビティーがビジネスと関係があるかどうかを指定します。指定可能な値は、以下のとおりです。 **TRUE** アクティビティーはビジネスと関係があります。アクティビティー状況は、Business Process Choreographer Explorer で確認できます。 **FALSE** アクティビティーはビジネスと関係がありません。

表 3. ACTIVITY ビュー内の列 (続き)

列名	タイプ	コメント
EXPIRES	タイム・スタンプ	アクティビティーの有効期限が切れる日時。アクティビティーの有効期限が切れている場合は、このイベントが発生した日時。

ACTIVITY_ATTRIBUTE ビュー:

この定義済みデータベース・ビューは、アクティビティーのカスタム・プロパティーの照会に使用します。

表 4. ACTIVITY_ATTRIBUTE ビュー内の列

列名	タイプ	コメント
AIID	ID	カスタム・プロパティーを所有するアクティビティー・インスタンスの ID。
NAME	ストリング	カスタム・プロパティーの名前。
VALUE	ストリング	カスタム・プロパティーの値。

ACTIVITY_SERVICE ビュー:

この定義済みデータベース・ビューは、アクティビティー・サービスの照会に使用します。

表 5. ACTIVITY_SERVICE ビュー内の列

列名	タイプ	コメント
EIID	ID	イベント・インスタンスの ID。
AIID	ID	イベントを待機しているアクティビティー・インスタンスの ID。
PIID	ID	イベントが含まれているプロセス・インスタンスの ID。
VTID	ID	イベントを説明するサービス・テンプレートの ID。
PORT_TYPE	ストリング	ポート・タイプの名前。
NAME_SPACE_URI	ストリング	ネーム・スペースの URI。
OPERATION	ストリング	サービスのオペレーション名。

APPLICATION_COMP ビュー:

この定義済みデータベース・ビューは、アプリケーション・コンポーネント ID、およびタスクのデフォルト設定の照会に使用します。

表 6. APPLICATION_COMP ビュー内の列

列名	タイプ	コメント
ACOID	ストリング	アプリケーション・コンポーネントの ID。

表 6. APPLICATION_COMP ビュー内の列 (続き)

列名	タイプ	コメント
BUSINESS_ RELEVANCE	ブール	コンポーネントのデフォルトのタスク・ビジネス関連ポリシー。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** タスクはビジネスと関係があり、監査されます。 **FALSE** タスクはビジネスとの関係がなく、監査されません。
NAME	ストリング	アプリケーション・コンポーネントの名前。
SUPPORT_ AUTOCLAIM	ブール	コンポーネントのデフォルトの自動要求ポリシー。この属性が TRUE に設定されている場合、潜在的な所有者が単一のユーザーであれば、タスクを自動的に要求することができます。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。
SUPPORT_CLAIM_ SUSP	ブール	中断されているタスクを要求できるかどうかを判断するコンポーネントのデフォルト設定。この属性が TRUE に設定されている場合は、中断されているタスクを要求することができます。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。
SUPPORT_ DELEGATION	ブール	コンポーネントのデフォルトのタスク代行ポリシー。この属性が TRUE に設定されている場合は、タスクの作業項目割り当てを変更することができます。すなわち、作業項目の作成、削除、または転送が可能です。
SUPPORT_ FOLLOW_ON	ブール	コンポーネントのデフォルトの後続タスク・ポリシー。この属性が TRUE に設定されている場合は、タスクの後続のタスクを作成できます。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。
SUPPORT_ SUB_TASK	ブール	コンポーネントのデフォルトのサブタスク・ポリシー。この属性が TRUE に設定されている場合は、タスクのサブタスクを作成できます。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。

ESCALATION ビュー:

この定義済みデータベース・ビューは、エスカレーションのデータの照会に使用します。

表 7. ESCALATION ビュー内の列

列名	タイプ	コメント
ESIID	ストリング	エスカレーション・インスタンスの ID。

表 7. ESCALATION ビュー内の列 (続き)

列名	タイプ	コメント
ACTION	整数	エスカレーションによって起動されるアクション。指定可能な値は、以下のとおりです。 **ACTION_CREATE_WORK_ITEM (1)** それぞれのエスカレーションに対する受信者の作業項目を作成します。 **ACTION_SEND_EMAIL (2)** それぞれのエスカレーション受信者に E メールを送信します。 **ACTION_CREATE_EVENT (3)** イベントを作成および公表します。
ACTIVATION_STATE	整数	対応するタスクが以下のいずれかの状態になると、エスカレーション・インスタンスが作成されます。 **ACTIVATION_STATE_READY (2)** ヒューマン・タスクまたは参加タスクは、要求を受ける準備ができていることを示します。 **ACTIVATION_STATE_RUNNING (3)** 親タスクが開始され、実行中であることを示します。 **ACTIVATION_STATE_CLAIMED (8)** タスクが要求されることを明示します。 **ACTIVATION_STATE_WAITING_ FOR_SUBTASK (20)** タスクがサブタスクの完了を待っていることを指定します。
ACTIVATION_TIME	タイム・スタンプ	エスカレーションが活動化される時刻。
AT_LEAST_ EXP_STATE	整数	エスカレーションによって予期されるタスクの状態。タイムアウトが発生した場合、タスク状態がこの属性の値と比較されます。指定可能な値は、以下のとおりです。 **AT_LEAST_EXPECTED_STATE_CLAIMED (8)** タスクが要求されることを明示します。 **AT_LEAST_EXPECTED_STATE_ENDED (20)** タスクが最終状態 (FINISHED、FAILED、TERMINATED、または EXPIRED) にあることを明示します。 **AT_LEAST_EXPECTED_STATE_ SUBTASKS_COMPLETED (21)** タスクのサブタスクすべてが完了していることを指定します。
ESTID	ストリング	対応するエスカレーション・テンプレートの ID。
FIRST_ESIID	ストリング	チェーン内の最初のエスカレーションの ID。

表 7. ESCALATION ビュー内の列 (続き)

列名	タイプ	コメント
INCREASE_PRIORITY	整数	タスクの優先度を増やす方法を示します。指定可能な値は、以下のとおりです。 **INCREASE_PRIORITY_NO (1)** タスクの優先度は増えません。 **INCREASE_PRIORITY_ONCE (2)** タスクの優先度は 1 度に 1 ずつ増えます。 **INCREASE_PRIORITY_REPEATED (3)** タスクの優先度は、エスカレーションが繰り返されるごとに 1 ずつ増えます。
NAME	ストリング	エスカレーションの名前。
STATE	整数	エスカレーションの状態。指定可能な値は、以下のとおりです。 STATE_INACTIVE (1) STATE_WAITING (2) STATE_ESCALATED (3) STATE_SUPERFLUOUS (4)
TKIID	ストリング	エスカレーションが所属するタスク・インスタンス ID。

ESCALATION_CPROP ビュー:

この定義済みデータベース・ビューは、エスカレーションのカスタム・プロパティーを照会するために使用します。

表 8. ESCALATION_CPROP ビュー内の列

列名	タイプ	コメント
ESIID	ストリング	エスカレーション ID。
NAME	ストリング	プロパティーの名前。
DATA_TYPE	ストリング	非ストリング・カスタム・プロパティーのクラスのタイプ。
STRING_VALUE	ストリング	String 型のカスタム・プロパティーの値。

ESCALATION_DESC ビュー:

この定義済みデータベース・ビューは、エスカレーションのマルチリンガル記述データを照会するために使用します。

表 9. ESCALATION_DESC ビュー内の列

列名	タイプ	コメント
ESIID	ストリング	エスカレーション ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
DESCRIPTION	ストリング	タスク・テンプレートの説明。

表 9. ESCALATION_DESC ビュー内の列 (続き)

列名	タイプ	コメント
DISPLAY_NAME	ストリング	エスカレーションの記述名。

ESC_TEMPL ビュー:

この定義済みデータベース・ビューは、エスカレーション・テンプレートのデータを照会するために使用します。

表 10. ESC_TEMPL ビュー内の列

列名	タイプ	コメント
ESTID	ストリング	エスカレーション・テンプレートの ID。
ACTION	整数	エスカレーションによって起動されるアクション。指定可能な値は、以下のとおりです。 **ACTION_CREATE_WORK_ITEM (1)** それぞれのエスカレーションに対する受信者の作業項目を作成します。 **ACTION_SEND_EMAIL (2)** それぞれのエスカレーション受信者に E メールを送信します。 **ACTION_CREATE_EVENT (3)** イベントを作成および公表します。
ACTIVATION_STATE	整数	対応するタスクが以下のいずれかの状態になると、エスカレーション・インスタンスが作成されます。 **ACTIVATION_STATE_READY (2)** ヒューマン・タスクまたは参加タスクは、要求を受ける準備ができていることを示します。 **ACTIVATION_STATE_RUNNING (3)** 親タスクが開始され、実行中であることを示します。 **ACTIVATION_STATE_CLAIMED (8)** タスクが要求されることを明示します。 **ACTIVATION_STATE_WAITING_ FOR_SUBTASK (20)** タスクがサブタスクの完了を待っていることを指定します。

表 10. ESC_TEMPL ビュー内の列 (続き)

列名	タイプ	コメント
AT_LEAST_ EXP_STATE	整数	エスカレーションによって予期されるタスクの状態。タイムアウトが発生した場合、タスク状態がこの属性の値と比較されます。指定可能な値は、以下のとおりです。 **AT_LEAST_EXPECTED_STATE_CLAIMED (8)** タスクが要求されることを明示します。 **AT_LEAST_EXPECTED_STATE_ENDED (20)** タスクが最終状態 (FINISHED、FAILED、TERMINATED、または EXPIRED) にあることを明示します。 **AT_LEAST_EXPECTED_STATE_ SUBTASKS_COMPLETED (21)** タスクのサブタスクすべてが完了していることを指定します。
CONTAINMENT_ CTX_ID	ストリング	エスカレーション・テンプレートがインライン・タスク・テンプレートに属する場合、包含コンテキストはプロセス・テンプレートです。エスカレーション・テンプレート・コンテキストがスタンドアロン・タスク・テンプレートに属する場合、包含コンテキストはタスク・テンプレートです。
FIRST_ESTID	ストリング	エスカレーション・テンプレート・チェーンの 1 番目のエスカレーション・テンプレートの ID です。
INCREASE_PRIORITY	整数	タスクの優先度を増やす方法を示します。指定可能な値は、以下のとおりです。 **INCREASE_PRIORITY_NO (1)** タスクの優先度は増えません。 **INCREASE_PRIORITY_ONCE (2)** タスクの優先度は 1 度に 1 ずつ増えます。 **INCREASE_PRIORITY_REPEATED (3)** タスクの優先度は、エスカレーションが繰り返されるごとに 1 ずつ増えます。
NAME	ストリング	エスカレーション・テンプレートの名前。
PREVIOUS_ESTID	ストリング	エスカレーション・テンプレート・チェーンの直前のエスカレーション・テンプレートの ID です。
TKTID	ストリング	エスカレーション・テンプレートが所属するタスク・テンプレートの ID。

ESC_TEMPL_CPROP ビュー:

この定義済みデータベース・ビューは、エスカレーション・テンプレートのカスタム・プロパティーを照会するために使用します。

表 11. ESC_TEMPL_CPROP ビュー内の列

列名	タイプ	コメント
ESTID	ストリング	エスカレーション・テンプレートの ID。
NAME	ストリング	プロパティーの名前。
TKTID	ストリング	エスカレーション・テンプレートが所属するタスク・テンプレートの ID。
DATA_TYPE	ストリング	非ストリング・カスタム・プロパティーのクラスのタイプ。
VALUE	ストリング	String 型のカスタム・プロパティーの値。

ESC_TEMPL_DESC ビュー:

この定義済みデータベース・ビューは、エスカレーション・テンプレートのマルチリンガル記述データを照会するために使用します。

表 12. ESC_TEMPL_DESC ビュー内の列

列名	タイプ	コメント
ESTID	ストリング	エスカレーション・テンプレートの ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
TKTID	ストリング	エスカレーション・テンプレートが所属するタスク・テンプレートの ID。
DESCRIPTION	ストリング	タスク・テンプレートの説明。
DISPLAY_NAME	ストリング	エスカレーションの記述名。

PROCESS_ATTRIBUTE ビュー:

この定義済みデータベース・ビューは、プロセスのカスタム・プロパティーの照会に使用します。

表 13. PROCESS_ATTRIBUTE ビュー内の列

列名	タイプ	コメント
PIID	ID	カスタム・プロパティーを所有するプロセス・インスタンスの ID。
NAME	ストリング	カスタム・プロパティーの名前。
VALUE	ストリング	カスタム・プロパティーの値。

PROCESS_INSTANCE ビュー:

この定義済みデータベース・ビューは、プロセス・インスタンスの照会に使用します。

表 14. PROCESS_INSTANCE ビュー内の列

列名	タイプ	コメント
PTID	ID	プロセス・テンプレート ID。
PIID	ID	プロセス・インスタンス ID。

表 14. PROCESS_INSTANCE ビュー内の列 (続き)

列名	タイプ	コメント
NAME	ストリング	プロセス・インスタンスの名前。
STATE	整数	プロセス・インスタンスの状態。指定可能な値は、以下のとおりです。 STATE_READY (1) STATE_RUNNING (2) STATE_FINISHED (3) STATE_COMPENSATING (4) STATE_INDOUBT (10) STATE_FAILED (5) STATE_TERMINATED (6) STATE_COMPENSATED (7) STATE_COMPENSATION_FAILED (12) STATE_TERMINATING (8) STATE_FAILING (9) STATE_SUSPENDED (11)
CREATED	タイム・スタンプ	プロセス・インスタンスの作成時刻。
STARTED	タイム・スタンプ	プロセス・インスタンスの開始時刻。
COMPLETED	タイム・スタンプ	プロセス・インスタンスの完了時刻。
PARENT_PIID	ID	親プロセス・インスタンスの ID。
PARENT_NAME	ストリング	親プロセス・インスタンスの名前。
TOP_LEVEL_PIID	ID	トップレベル・プロセス・インスタンスのプロセス・インスタンス ID。トップレベルのプロセス・インスタンスがない場合、これは、現行プロセス・インスタンスのプロセス・インスタンス ID になります。
TOP_LEVEL_NAME	ストリング	トップレベル・プロセス・インスタンスの名前。トップレベルのプロセス・インスタンスがない場合、これは、現行プロセス・インスタンスの名前になります。
STARTER	ストリング	プロセス・インスタンスのスターターのプリンシパル ID。
DESCRIPTION	ストリング	プロセス・テンプレートの説明にプレースホルダーが含まれている場合、この列には、解決済みのプレースホルダーを所有するプロセス・インスタンスの説明が入ります。
TEMPLATE_NAME	ストリング	関連するプロセス・テンプレートの名前。
TEMPLATE_DESCR	ストリング	関連するプロセス・テンプレートの説明。
RESUMES	タイム・スタンプ	プロセス・インスタンスが自動的に再開される時刻。

PROCESS_TEMPLATE ビュー:

この定義済みデータベース・ビューは、プロセス・テンプレートの照会に使用します。

表 15. PROCESS_TEMPLATE ビュー内の列

列名	タイプ	コメント
PTID	ID	プロセス・テンプレート ID。
NAME	ストリング	プロセス・テンプレートの名前。
VALID_FROM	タイム・スタンプ	プロセス・テンプレートのインスタンス化が可能になる時刻。
TARGET_NAMESPACE	ストリング	プロセス・テンプレートのターゲット・ネーム・スペース。
APPLICATION_NAME	ストリング	プロセス・テンプレートが所属するエンタープライズ・アプリケーションの名前。
VERSION	ストリング	ユーザー定義のバージョン。
CREATED	タイム・スタンプ	プロセス・テンプレートがデータベース内に作成される時刻。
STATE	整数	プロセス・インスタンスの作成にプロセス・テンプレートを使用できるかどうかを指定します。指定可能な値は、以下のとおりです。 STATE_STARTED (1) STATE_STOPPED (2)
EXECUTION_MODE	整数	このプロセス・テンプレートから派生したプロセス・インスタンスの実行方法を指定します。指定可能な値は、以下のとおりです。 EXECUTION_MODE_MICROFLOW (1) EXECUTION_MODE_LONG_RUNNING (2)
DESCRIPTION	ストリング	プロセス・テンプレートの説明。
COMP_SPHERE	整数	プロセス・テンプレート内の microflow のインスタンスの補正の振る舞いを指定します。既存の補正範囲を結合するか、補正範囲を作成するかのいずれかです。 指定可能な値は、以下のとおりです。 COMP_SPHERE_REQUIRED (2) COMP_SPHERE_SUPPORTS (4)
DISPLAY_NAME	ストリング	プロセスの記述名。

QUERY_PROPERTY ビュー:

この定義済みデータベース・ビューは、プロセス・レベル変数の照会に使用します。

表 16. QUERY_PROPERTY ビュー内の列

列名	タイプ	コメント
PIID	ID	プロセス・インスタンス ID。
VARIABLE_NAME	ストリング	プロセス・レベル変数の名前。

表 16. QUERY_PROPERTY ビュー内の列 (続き)

列名	タイプ	コメント
NAME	ストリング	照会プロパティーの名前。
NAMESPACE	ストリング	照会プロパティーのネームスペース。
GENERIC_VALUE	ストリング	次のいずれかの定義済みタイプにマップしないプロパティー・タイプのストリング表記。 STRING_VALUE、 NUMBER_VALUE、 DECIMAL_VALUE、または TIMESTAMP_VALUE。
STRING_VALUE	ストリング	プロパティー・タイプがストリング・タイプにマップされる場合、これがストリングの値です。
NUMBER_VALUE	整数	プロパティー・タイプが整数タイプにマップされる場合、これが整数の値です。
DECIMAL_VALUE	10 進数	プロパティー・タイプが浮動小数点タイプにマップされる場合、これが 10 進数の値です。
TIMESTAMP_VALUE	タイム・スタンプ	プロパティー・タイプがタイム・スタンプ・タイプにマップされる場合、これがタイム・スタンプの値です。

TASK ビュー:

この定義済みデータベース・ビューは、タスク・オブジェクトの照会に使用します。

表 17. TASK ビュー内の列

列名	タイプ	コメント
TKIID	ID	タスク・インスタンスの ID。
ACTIVATED	タイム・スタンプ	タスクが活動化された時刻。
APPLIC_ DEFAULTS_ID	ID	タスクのデフォルト値を指定するアプリケーション・コンポーネントの ID。
APPLIC_NAME	ストリング	タスクが所属するエンタープライズ・アプリケーションの名前。
BUSINESS_ RELEVANCE	ブール	タスクがビジネスと関係があるかどうかを指定します。 属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** タスクはビジネスと関係があり、監査されます。 **FALSE** タスクはビジネスとの関係がなく、監査されません。

表 17. TASK ビュー内の列 (続き)

列名	タイプ	コメント
COMPLETED	タイム・スタンプ	タスクが完了した時刻。
CONTAINMENT_CTX_ID	ID	このタスクの包含コンテキスト。この属性は、タスクのライフ・サイクルを決定します。タスクの包含コンテキストを削除すると、タスク・テンプレートも削除されます。
CTX_AUTHORIZATION	整数	タスクの所有者がタスク・コンテキストにアクセスできるようにします。指定可能な値は、以下のとおりです。 **AUTH_NONE** 関連コンテキスト・オブジェクトに対する許可権限はありません。 **AUTH_READER** 関連コンテキスト・オブジェクトに関する操作に、プロセス・インスタンスのプロパティーの読み取りなどのリーダー権限が必要です。
DUE	タイム・スタンプ	タスクの期限時刻。
EXPIRES	タイム・スタンプ	タスクの有効期限が切れる日付。
FIRST_ACTIVATED	タイム・スタンプ	タスクが初めて活動化された時刻。
FOLLOW_ON_TKIID	ID	後続のタスクのインスタンスの ID。
HIERARCHY_POSITION	整数	指定可能な値は、以下のとおりです。 **HIERARCHY_POSITION_TOP_TASK (0)** タスク階層の最上位タスク。 **HIERARCHY_POSITION_SUB_TASK (1)** タスクは、タスク階層のサブタスク。 **HIERARCHY_POSITION_FOLLOW_ON_TASK (2)** タスクは、タスク階層の後続のタスク。
IS_AD_HOC	ブール	このタスクが実行時に動的に作成されたのか、またはタスク・テンプレートから作成されたのかを示します。
IS_ESCALATED	ブール	このタスクのエスカレーションが発生済みかどうかを示します。
IS_INLINE	ブール	タスクがビジネス・プロセス内のインラインのタスクであるかどうかを示します。
IS_WAIT_FOR_SUB_TK	ブール	親タスクが、サブタスクが終了状態になるのを待機しているかどうかを示します。

表 17. TASK ビュー内の列 (続き)

列名	タイプ	コメント
KIND	整数	タスクの種類。指定可能な値は、以下のとおりです。 **KIND_HUMAN (101)** タスクが、人の手で作成され、処理されるコラボレーション・タスクであることを示します。 **KIND_WPC_STAFF_ACTIVITY (102)** タスクが、WebSphere Business Integration Server Foundation バージョン 5 ビジネス・プロセスのスタッフ・アクティビティーであるヒューマン・タスクであることを示します。 **KIND_ORIGINATING (103)** タスクが人からコンピューターへの対話をサポートし、人がサービスを作成、開始、および始動できる*呼び出し*タスクであることを示します。 **KIND_PARTICIPATING (105)** タスクがコンピューターから人への対話をサポートし、人がサービスを実装できる予定タスクであることを示します。 **KIND_ADMINISTRATIVE (106)** タスクが管理タスクであることを示します。
LAST_MODIFIED	タイム・スタンプ	タスクの最終変更時刻。
LAST_STATE_CHANGE	タイム・スタンプ	タスクの状態の最終変更時刻。
NAME	ストリング	タスクの名前。
NAME_SPACE	ストリング	タスクのカテゴリー化に使用されるネーム・スペース。
ORIGINATOR	ストリング	タスク・オリジネーターのプリンシパル ID。
OWNER	ストリング	タスクの所有者のプリンシパル ID。
PARENT_CONTEXT_ID	ストリング	このタスクの親コンテキスト。この属性は、呼び出し側アプリケーション・コンポーネント内の対応するコンテキストに対するキーを提供します。親コンテキストは、そのタスクを作成するアプリケーション・コンポーネントによって設定されます。
PRIORITY	整数	タスクの優先順位。
RESUMES	タイム・スタンプ	タスクが自動的に再開される時刻。
STARTED	タイム・スタンプ	タスクが開始された時刻 (STATE_RUNNING、STATE_CLAIMED)。
STARTER	ストリング	タスク・スターターのプリンシパル ID。

表 17. TASK ビュー内の列 (続き)

列名	タイプ	コメント
STATE	整数	タスクの状態。指定可能な値は、以下のとおりです。 **STATE_READY (2)** そのタスクは要求を受ける準備ができたことを示します。 **STATE_RUNNING (3)** タスクが開始され、実行中であることを示します。 **STATE_FINISHED (5)** タスクが正常に完了したことを示します。 **STATE_FAILED (6)** タスクが正常に完了しなかったことを示します。 **STATE_TERMINATED (7)** 外部要求または内部要求が原因で、タスクが終了したことを示します。 **STATE_CLAIMED (8)** タスクが要求されることを示します。 **STATE_EXPIRED (12)** 指定された期間を超えたため、タスクが終了したことを示します。 **STATE_FORWARDED (101)** タスクが後続のタスクで完了したことを示します。
SUPPORT_ AUTOCLAIM	ブール	このタスクが単一ユーザーに割り当てられている場合に、自動的に要求されるかどうかを示します。
SUPPORT_CLAIM_ SUSP	ブール	このタスクが中断されている場合に、要求可能かどうかを示します。
SUPPORT_ DELEGATION	ブール	このタスクが、作業項目の作成、削除、または転送によって、作業代行をサポートするかどうかを示します。
SUPPORT_ FOLLOW_ON	ブール	このタスクが後続のタスクの作成をサポートするかどうかを示します。
SUPPORT_SUB_TASK	ブール	このタスクがサブタスクの作成をサポートするかどうかを示します。
SUSPENDED	ブール	タスクが中断されているかどうかを示します。
TKTID	ID	タスク・テンプレート ID。
TOP_TKIID	ID	これがサブタスクの場合、親タスク上位インスタンス ID。
TYPE	ストリング	タスクのカテゴリー化に使用されるタイプ。

TASK_CPROP ビュー:

この定義済みデータベース・ビューは、タスク・オブジェクトのカスタム・プロパティーを照会するために使用します。

表 18. TASK_CPROP ビュー内の列

列名	タイプ	コメント
TKIID	ストリング	タスク・インスタンス ID。
NAME	ストリング	プロパティーの名前。
DATA_TYPE	ストリング	非ストリング・カスタム・プロパティーのクラスのタイプ。
STRING_VALUE	ストリング	String 型のカスタム・プロパティーの値。

TASK_DESC ビュー:

この定義済みデータベース・ビューは、タスク・オブジェクトのマルチリンガル記述データを照会するために使用します。

表 19. TASK_DESC ビュー内の列

列名	タイプ	コメント
TKIID	ストリング	タスク・インスタンス ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
DESCRIPTION	ストリング	タスクの説明。
DISPLAY_NAME	ストリング	タスクの記述名。

TASK_TEMPL ビュー:

この定義済みデータベース・ビューは、タスクのインスタンスを生成するために使用できるデータを保持します。

表 20. TASK_TEMPL ビュー内の列

列名	タイプ	コメント
TKTID	ストリング	タスク・テンプレート ID。
VALID_FROM	タイム・スタンプ	インスタンス化にタスク・テンプレートが使用できるようになる時刻。
APPLIC_ DEFAULTS_ID	ストリング	タスク・テンプレートのデフォルト値を指定するアプリケーション・コンポーネントの ID。
APPLIC_NAME	ストリング	タスク・テンプレートが所属するエンタープライズ・アプリケーションの名前。
BUSINESS_ RELEVANCE	ブール	タスク・テンプレートがビジネスと関係があるかどうかを指定します。 属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** タスクはビジネスと関係があり、監査されます。 **FALSE** タスクはビジネスとの関係がなく、監査されません。

表 20. TASK_TEMPL ビュー内の列 (続き)

列名	タイプ	コメント
CONTAINMENT_ CTX_ID	ID	このタスク・テンプレートの包含コンテキスト。この属性は、タスク・テンプレートのライフ・サイクルを決定します。包含コンテキストを削除すると、タスク・テンプレートも削除されます。
CTX_ AUTHORIZATION	整数	タスクの所有者がタスク・コンテキストにアクセスできるようにします。指定可能な値は、以下のとおりです。 **AUTH_NONE** 関連コンテキスト・オブジェクトに対する許可権限はありません。 **AUTH_READER** 関連コンテキスト・オブジェクトに関する操作に、プロセス・インスタンスのプロパティーの読み取りなどのリーダー権限が必要です。
DEFINITION_NAME	ストリング	Task Execution Language (TEL) ファイルのタスク・テンプレート定義の名前。
DEFINITION_NS	ストリング	TEL ファイルのタスク・テンプレート定義のネーム・スペース。
IS_AD_HOC	ブール	このタスク・テンプレートが実行時に動的に作成されたのかどうか、またはタスクが EAR ファイルの一部としてデプロイされたときを示します。
IS_INLINE	ブール	このタスク・テンプレートがビジネス・プロセス内のタスクとしてモデル化されるかどうかを示します。
KIND	整数	このタスク・テンプレートから派生したタスクの種類。指定可能な値は、以下のとおりです。 **KIND_HUMAN (101)** タスクが、人の手で作成され、処理されるコラボレーション・タスクであることを示します。 **KIND_ORIGINATING (103)** タスクが人からコンピューターへの対話をサポートし、人がサービスを作成、開始、および始動できる*呼び出し*タスクであることを示します。 **KIND_PARTICIPATING (105)** タスクがコンピューターから人への対話をサポートし、人がサービスを実装できる予定タスクであることを示します。 **KIND_ADMINISTRATIVE (106)** タスクが管理タスクであることを示します。
NAME	ストリング	タスク・テンプレートの名前。

表 20. TASK_TEMPL ビュー内の列 (続き)

列名	タイプ	コメント
NAMESPACE	ストリング	タスク・テンプレートのカテゴリー化に使用されるネーム・スペース。
PRIORITY	整数	タスク・テンプレートの優先順位。
STATE	整数	タスク・テンプレートの状態。指定可能な値は、以下のとおりです。 **STATE_STARTED (1)** タスク・テンプレートをタスク・インスタンスの作成に使用できることを明示します。 **STATE_STOPPED (2)** タスク・テンプレートが停止されたことを明示します。この状態のタスク・テンプレートからタスク・インスタンスを作成することはできません。
SUPPORT_AUTOCLAIM	ブール	このタスク・テンプレートから派生したタスクが 1 人のユーザーに割り当てられた場合、タスクを自動的に要求できるかどうかを示します。
SUPPORT_CLAIM_SUSP	ブール	このタスク・テンプレートから派生したタスクが中断された場合、タスクを自動的に要求できるかどうかを示します。
SUPPORT_DELEGATION	ブール	このタスク・テンプレートから派生したタスクが、作業項目の作成、削除、または転送を使用して作業代行をサポートするかどうかを示します。
SUPPORT_FOLLOW_ON	ブール	タスク・テンプレートが後続のタスクの作成をサポートするかどうかを示します。
SUPPORT_SUB_TASK	ブール	タスク・テンプレートがサブタスクの作成をサポートするかどうかを示します。
TYPE	ストリング	タスク・テンプレートのカテゴリー化に使用されるタイプ。

TASK_TEMPL_CPROP ビュー:

この定義済みデータベース・ビューは、タスク・テンプレートのカスタム・プロパティーを照会するために使用します。

表 21. TASK_TEMPL_CPROP ビュー内の列

列名	タイプ	コメント
TKTID	ストリング	タスク・テンプレート ID。
NAME	ストリング	プロパティーの名前。
DATA_TYPE	ストリング	非ストリング・カスタム・プロパティーのクラスのタイプ。
STRING_VALUE	ストリング	String 型のカスタム・プロパティーの値。

TASK_TEMPL_DESC ビュー:

この定義済みデータベース・ビューは、タスク・テンプレート・オブジェクトのマルチリンガル記述データを照会するために使用します。

表 22. TASK_TEMPL_DESC ビュー内の列

列名	タイプ	コメント
TKTID	ストリング	タスク・テンプレート ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
DESCRIPTION	ストリング	タスク・テンプレートの説明。
DISPLAY_NAME	ストリング	タスク・テンプレートの記述名。

WORK_ITEM ビュー:

この定義済みデータベース・ビューは、作業項目の照会や、プロセス、タスクおよびエスカレーション用の許可データの照会に使用します。

表 23. WORK_ITEM ビュー内の列

列名	タイプ	コメント
WIID	ID	作業項目 ID。
OWNER_ID	ストリング	所有者のプリンシパル ID。
GROUP_NAME	ストリング	関連するグループ・ワーク・リストの名前。
EVERYBODY	ブール	この作業項目を全員が所有するかどうかを指定します。

表 23. WORK_ITEM ビュー内の列 (続き)

列名	タイプ	コメント
OBJECT_TYPE	整数	関連オブジェクトのタイプ。指定可能な値は、以下のとおりです。 **OBJECT_TYPE_ACTIVITY (1)** その作業項目がアクティビティー用に作成されたものであることを明示します。 **OBJECT_TYPE_PROCESS_INSTANCE (3)** その作業項目がプロセス・インスタンス用に作成されたものであることを明示します。 **OBJECT_TYPE_TASK_INSTANCE (5)** その作業項目がタスク用に作成されたものであることを明示します。 **OBJECT_TYPE_TASK_TEMPLATE (6)** その作業項目がタスク・テンプレート用に作成されたものであることを明示します。 **OBJECT_TYPE_ESCALATION_ INSTANCE (7)** その作業項目がエスカレーション・インスタンス用に作成されたものであることを明示します。 **OBJECT_TYPE_APPLICATION_ COMPONENT (9)** その作業項目がアプリケーション・コンポーネント用に作成されたものであることを示します。
OBJECT_ID	ID	関連オブジェクト (例えば、関連プロセスやタスクなど) の ID。
ASSOC_OBJECT_TYPE	整数	ASSOC_OID 属性によって参照されるオブジェクトのタイプ。例えば、タスク、プロセス、または外部オブジェクトなど。OBJECT_TYPE 属性の値を使用します。
ASSOC_OID	ID	作業項目のあるオブジェクト関連オブジェクトの ID。例えば、この作業項目が作成されたアクティビティー・インスタンスを含んでいるプロセス・インスタンスの、プロセス・インスタンス ID など。

表 23. WORK_ITEM ビュー内の列 (続き)

列名	タイプ	コメント
REASON	整数	作業項目の割り当て理由。指定可能な値は、以下のとおりです。 REASON_POTENTIAL_STARTER (5) REASON_POTENTIAL_INSTANCE_ CREATOR (11) REASON_POTENTIAL_STARTER (1) REASON_EDITOR (2) REASON_READER (3) REASON_ORIGINATOR (9) REASON_OWNER (4) REASON_STARTER (6) REASON_ESCALATION_RECEIVER (10) REASON_ADMINISTRATOR (7)
CREATION_TIME	タイム・スタンプ	作業項目が作成された日時。

照会で変数を使用することによるデータのフィルタリング

照会結果は、照会基準に一致するオブジェクトを戻します。この結果を、変数の値でフィルタリングすることもできます。

このタスクについて

実行時にプロセスが使用する変数を、そのプロセス・モデルで定義することができます。これらの変数で、照会可能なパートを宣言します。

例えば、John Smith が保険会社のサービス番号を呼び出して、損傷を受けた車に対する保険請求の進捗状況を問い合わせるとします。請求の管理者はカスタマー ID でその請求を検索します。

プロシージャー

1. オプション: プロセス内の照会可能な変数のプロパティーをリストします。

   プロセス・テンプレート ID を使用して、プロセスを特定します。照会可能な変数がわかっている場合は、このステップはスキップしてください。

   ```
   List variableProperties = process.getQueryProperties(ptid);
   for (int i = 0; i < variableProperties.size(); i++)
   {
      QueryProperty queryData = (QueryProperty)variableProperties.get(i);
      String variableName = queryData.getVariableName();
      String name         = queryData.getName();
      int mappedType      = queryData.getMappedType();
      ...
   }
   ```

2. フィルター基準に一致する変数を持つプロセス・インスタンスをリストします。

   このプロセスでは、カスタマー ID は照会可能な変数 customerClaim の一部としてモデル化されます。そのため、カスタマー ID を使用すれば問題の請求を見つけることができます。

```
QueryResultSet result = process.query
    ("PROCESS_INSTANCE.NAME, QUERY_PROPERTY.STRING_VALUE",
     "QUERY_PROPERTY.VARIABLE_NAME = 'customerClaim' AND " +
     "QUERY_PROPERTY.NAME = 'customerID' AND " +
     "QUERY_PROPERTY.STRING_VALUE like 'Smith%'",
      (String)null, (Integer)null,
      (Integer)null, (TimeZone)null );
```

このアクションによって戻される照会結果セットには、プロセス・インスタンス名と、ID が Smith で始まる顧客のカスタマー ID の値が含まれています。

保管照会文の管理

保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

このタスクについて

保管照会文は、データベースに保管され、名前で識別される照会のことです。専用の保管照会文と共通の保管照会文の名前を同じにすることができます。異なる複数の所有者の専用保管照会文を同じ名前にすることもできます。

保管照会文は、ビジネス・プロセス・オブジェクト、タスク・オブジェクト、またはこの 2 つのオブジェクト・タイプの組み合わせたものを対象とします。

関連概念

36 ページの『保管照会文のパラメーター』
保管照会文は、データベースに保管され、名前で識別される照会のことです。適格なタプルは、照会が実行されるときに動的にアセンブルされます。保管照会文を再使用可能にするには、実行時に解決される照会定義のパラメーターを使用できます。

共通保管照会文の管理:

共通保管照会文がシステム管理者によって作成されます。この照会は、全ユーザーが使用できます。

このタスクについて

システム管理者は、共通保管照会文を作成、表示、および削除できます。API 呼び出しでユーザー ID を指定しないと、その保管照会文は共通保管照会文と見なされます。

プロシージャー

1. 共通の保管照会文を作成します。

   例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、CustomerOrdersStartingWithA という名前を付けて保管します。

```
process.createStoredQuery("CustomerOrdersStartingWithA",
             "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
             "PROCESS_INSTANCE.NAME LIKE 'A%'",
             "PROCESS_INSTANCE.NAME",
              (Integer)null, (TimeZone)null);
```

この保管照会文の結果として、A で始まるプロセス・インスタンス名すべてのソート済みリストが、関連付けられたプロセス・インスタンス ID (PIID) とともに戻されます。

2. 保管照会文で定義された照会を実行します。

```
QueryResultSet result = process.query("CustomerOrdersStartingWithA",
               new Integer(0));
```

このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. 使用可能な共通保管照会文の名前をリストします。

以下のコードの断片では、戻される照会のリストを共通照会のみに限定する方法を示しています。

```
String[] storedQuery = process.getStoredQueryNames(StoredQueryData.KIND_PUBLIC);
```

4. オプション: 特定の保管照会文で定義された照会を検査します。

専用の保管照会文には、共通の保管照会文と同じ名前を付けることができます。名前が同じである場合は、専用の保管照会文が戻されます。以下のコードの断片では、指定した名前の共通照会のみを戻す方法を示しています。タスク・ベース・オブジェクトでこの照会を実行する場合は、戻されるオブジェクト・タイプとして、StoredQueryData ではなく StoredQuery を指定してください。

```
StoredQueryData storedQuery = process.getStoredQuery
   (StoredQueryData.KIND_PUBLIC, "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();String owner = storedQuery.getOwner();
```

5. 共通の保管照会文を削除します。

以下のコードの断片では、ステップ 1 で作成した保管照会文の削除方法を示しています。

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

他のユーザーの専用保管照会文の管理:

専用照会はどのユーザーでも作成できます。この照会は、照会の所有者とシステム管理者しか使用できません。

このタスクについて

システム管理者は、特定ユーザーに属する専用の保管照会文を管理できます。

プロシージャー

1. ユーザー ID Smith の専用保管照会文を作成します。

例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、Smith というユーザー ID で CustomerOrdersStartingWithA という名前を付けて保管します。

```
process.createStoredQuery("Smith", "CustomerOrdersStartingWithA",
             "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
             "PROCESS_INSTANCE.NAME LIKE 'A%'",
             "PROCESS_INSTANCE.NAME",
              (Integer)null, (TimeZone)null,
              (List)null, (String)null);
```

この保管照会文の結果として、A で始まるプロセス・インスタンス名すべてのソート済みリストが、関連付けられたプロセス・インスタンス ID (PIID) とともに戻されます。

2. 保管照会文で定義された照会を実行します。

```
QueryResultSet result = process.query
                       ("Smith", "CustomerOrdersStartingWithA",
                         (Integer)null, (Integer)null, (List)null);
new Integer(0));
```

このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. 特定のユーザーに属する専用照会の名前のリストを取得します。

例えば、以下のコードの断片では、ユーザー Smith に属する専用照会のリストを取得する方法を示しています。

```
String[] storedQuery = process.getStoredQueryNames("Smith");
```

4. 特定の照会の詳細を表示します。

以下のコードの断片では、ユーザー Smith が所有する照会 CustomerOrdersStartingWithA の詳細を表示する方法を示しています。

```
StoredQuery storedQuery = process.getStoredQuery
   ("Smith", "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

5. 専用の保管照会文を削除します。

以下のコードの断片では、ユーザー Smith が所有する専用照会を削除する方法を示しています。

```
process.deleteStoredQuery("Smith", "CustomerOrdersStartingWithA");
```

専用保管照会文の操作:

システム管理者でなくても、自分専用の保管照会文は作成、実行、および削除できます。また、システム管理者が作成した共通の保管照会文を使用することもできます。

プロシージャー

1. 専用の保管照会文を作成します。

例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、固有の名前を付けて保管します。ユーザー ID が指定されない場合、その保管照会文はログオン・ユーザーの専用保管照会文と見なされます。

```
process.createStoredQuery("CustomerOrdersStartingWithA",
             "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
             "PROCESS_INSTANCE.NAME LIKE 'A%'",
             "PROCESS_INSTANCE.NAME",
              (Integer)null, (TimeZone)null);
```

この照会は、文字 A で始まるプロセス・インスタンス名、および関連したプロセス・インスタンス ID (PIID) をすべてソートしたリストにして戻します。

2. 保管照会文で定義された照会を実行します。

```
QueryResultSet result = process.query("CustomerOrdersStartingWithA",
               new Integer(0));
```

このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. ログオン・ユーザーがアクセスできる保管照会文の名前のリストを取得します。

以下のコードの断片では、ユーザーがアクセスできる共通の保管照会文と専用の保管照会文の両方を取得する方法を示しています。

```
String[] storedQuery = process.getStoredQueryNames();
```

4. 特定の照会の詳細を表示します。

以下のコードの断片では、ユーザー Smith が所有する照会 CustomerOrdersStartingWithA の詳細を表示する方法を示しています。

```
StoredQuery storedQuery = process.getStoredQuery
   ("CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

5. 専用の保管照会文を削除します。

以下のコード断片は、専用保管照会文を削除する方法を示しています。

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

ビジネス・プロセス用のアプリケーションの開発

ビジネス・プロセスは、ビジネス・ゴールを達成するために特定のシーケンスで呼び出される、ビジネス関連の一連のアクティビティーです。プロセスに対する標準のアクションに対応したアプリケーションを開発する方法を示した例が提供されています。

このタスクについて

ビジネス・プロセスは、microflow または長期にわたって実行するプロセスのいずれかです。

- microflow は、同期して実行される短期実行のビジネス・プロセスです。結果は即時に呼び出し元に戻されます。
- 長期実行の割り込み可能プロセスは、まとめてチェーニングされるアクティビティーのシーケンスとして実行されます。プロセスで特定の構成要素を使用するとプロセス・フローが中断し、例えば、ヒューマン・タスクの呼び出し、同期バインディングを使用したサービスの呼び出し、またはタイマー駆動アクティビティーの使用などが割り込みます。

プロセスの並列分岐は通常、非同期でナビゲートされるので、並列分岐のアクティビティーは平行して実行されます。アクティビティーのタイプとトランザクションの設定に応じて、アクティビティーを独自のトランザクションで実行することができます。

プロセス・インスタンスに対するアクションに必要なロール

BusinessFlowManager インターフェースへのアクセス権は、呼び出し元がプロセスに対するすべてのアクションを実行できることは保証しません。呼び出し元は、アクションを実行する許可が与えられているロールを使用して、クライアント・アプリケーションにログオンする必要があります。

次の表に、それぞれのロールで実行できるプロセス・インスタンス上のアクションを示します。

アクション	呼び出し元のプリンシパルのロール		
	リーダー	スターター	管理者
createMessage	x	x	x
createWorkItem			x
delete			x
deleteWorkItem			x
forceTerminate			x
getActiveEventHandlers	x		x
getActivityInstance	x		x
getAllActivities	x		x
getAllWorkItems	x		x
getClientUISettings	x	x	x
getCustomProperties	x	x	x
getCustomProperty	x	x	x
getCustomPropertyNames	x	x	x
getFaultMessage	x	x	x
getInputClientUISettings	x	x	x
getInputMessage	x	x	x
getOutputClientUISettings	x	x	x
getOutputMessage	x	x	x
getProcessInstance	x	x	x
getVariable	x	x	x
getWaitingActivities	x	x	x
getWorkItems	x		x
restart			x
resume			x
setCustomProperty		x	x
setVariable			x
suspend			x
transferWorkItem			x

ビジネス・プロセス・アクティビティーのアクションに必要なロール

BusinessFlowManager インターフェースへのアクセス権は、呼び出し元がアクティビティーに対するすべてのアクションを実行できることを保証するものではありません。呼び出し元は、アクションを実行する許可が与えられているロールを使用して、クライアント・アプリケーションにログオンする必要があります。

次の表に、それぞれのロールで実行できるアクティビティー・インスタンス上のアクションを示します。

アクション	呼び出し元のプリンシパルのロール				
	リーダー	編集者	潜在的な所有者	所有者	管理者
cancelClaim				x	x
claim			x		x
complete				x	x
createMessage	x	x	x	x	x
createWorkItem					x
deleteWorkItem					x
forceComplete					x
forceRetry					x
getActivityInstance	x	x	x	x	x
getAllWorkItems	x	x	x	x	x
getClientUISettings	x	x	x	x	x
getCustomProperties	x	x	x	x	x
getCustomProperty	x	x	x	x	x
getCustomPropertyNames	x	x	x	x	x
getFaultMessage	x	x	x	x	x
getFaultNames	x	x	x	x	x
getInputMessage	x	x	x	x	x
getOutputMessage	x	x	x	x	x
getVariable	x	x	x	x	x
getVariableNames	x	x	x	x	x
getInputVariableNames	x	x	x	x	x
getOutputVariableNames	x	x	x	x	x
getWorkItems	x	x	x	x	x
setCustomProperty		x		x	x
setFaultMessage		x		x	x
setOutputMessage		x		x	x
setVariable					x
transferWorkItem				x 潜在的な所有者または管理者に対してのみ	x

ビジネス・プロセスのライフ・サイクルの管理

プロセスを開始できる Business Process Choreographer API メソッドが呼び出されると、プロセス・インスタンスが生成されます。プロセス・インスタンスのすべてのアクティビティーが終了状態になるまで、プロセス・インスタンスのナビゲーションは続きます。ライフ・サイクルを管理するために、プロセス・インスタンスでさまざまなアクションを実行できます。

このタスクについて

プロセスに対する以下の標準のライフ・サイクル・アクションに対応したアプリケーションを開発する方法を示した例が提供されています。

ビジネス・プロセスの開始:

ビジネス・プロセスを開始する方法は、プロセスが microflow であるか長期実行プロセスであるかによって異なります。プロセスを開始するサービスも、プロセスの開始方法にとって重要です。プロセスに固有の開始サービスを 1 つ設定するか、複数の開始サービスを設定することができます。

このタスクについて

microflow や長期実行プロセスを開始する標準のシナリオに対応したアプリケーションを開発する方法を示した例が提供されています。

固有の開始サービスを含む microflow の実行:

microflow は、receive アクティビティーまたは pick アクティビティーから開始できます。開始サービスが固有であるのは、microflow が receive アクティビティーを使って開始された場合、または pick アクティビティー内に 1 つの onMessage 定義のみがある場合です。

このタスクについて

microflow によって要求/応答操作がインプリメントされている場合、つまり、プロセスに応答が入っている場合、call メソッドを使用してそのプロセスを実行し、その呼び出しでパラメーターとしてプロセス・テンプレート名を渡すことができます。

microflow が片方向操作である場合は、sendMessage メソッドを使用してプロセスを実行します。このメソッドは、次の例には含まれていません。

プロシージャー

1. オプション: プロセス・テンプレートをリストして、実行するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
       PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
 "PROCESS_TEMPLATE.NAME",
  new Integer(50),
  (TimeZone)null);
```

結果は名前でソートされます。call メソッドによって開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

```
ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
                           (template.getID(),
                            template.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}

//run the process
ClientObjectWrapper output = process.call(template.getName(), input);
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

   このアクションによって、プロセス・テンプレート CustomerTemplate のインスタンスが作成され、一部の顧客データが受け渡されます。この操作は、プロセスが完了してからでないと戻りません。プロセスの結果 OrderNo が、呼び出し元に戻されます。

非固有の開始サービスを含む microflow の実行:

microflow は、receive アクティビティーまたは pick アクティビティーから開始できます。 microflow が複数の onMessage 定義を含む pick アクティビティーを使用して開始された場合、開始サービスは固有ではありません。

このタスクについて

microflow によって要求/応答操作がインプリメントされている場合、つまり、プロセスに応答が入っている場合、call メソッドを使用してそのプロセスを実行し、その呼び出しで開始サービスの ID を渡すことができます。

microflow が片方向操作である場合は、sendMessage メソッドを使用してプロセスを実行します。このメソッドは、次の例には含まれていません。

プロシージャー

1. オプション: プロセス・テンプレートをリストして、実行するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
       PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
 "PROCESS_TEMPLATE.NAME",
  new Integer(50),
  (TimeZone)null);
```

結果は名前でソートされます。microflow として開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が、照会から戻されます。

2. 呼び出すべき開始サービスを判別します。

   この例では、最初に検出されたテンプレートを使用します。

```
ProcessTemplateData template = processTemplates[0];
ActivityServiceTemplateData[] startActivities =
        process.getStartActivities(template.getID());
```

3. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

```
ActivityServiceTemplateData activity = startActivities[0];
//create a message for the service to be called
ClientObjectWrapper input =
      process.createMessage(activity.getServiceTemplateID(),
                            activity.getActivityTemplateID(),
                            activity.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//run the process
ClientObjectWrapper output = process.call(activity.getServiceTemplateID(),
                                          activity.getActivityTemplateID(),
                                          input);
//check the output of the process, for example, an order number
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

   このアクションによって、プロセス・テンプレート CustomerTemplate のインスタンスが作成され、一部の顧客データが受け渡されます。この操作は、プロセスが完了してからでないと戻りません。プロセスの結果 OrderNo が、呼び出し元に戻されます。

固有の開始サービスを含む長期実行プロセスの開始:

開始サービスが固有の場合、initiate メソッドを使用して、プロセス・テンプレート名をパラメーターとして渡すことができます。これは、長期実行プロセスが、単一の receive アクティビティーまたは pick アクティビティーのいずれかを使用して開始する、および単一の pick アクティビティーが 1 つのみの onMessage 定義を持つ場合にあてはまります。

プロシージャー

1. オプション: プロセス・テンプレートをリストして、開始するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
     ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
      "PROCESS_TEMPLATE.NAME",
       new Integer(50),
       (TimeZone)null);
   ```

   結果は名前でソートされます。initiate メソッドによって開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。
2. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。プロセス・インスタンス名を指定する場合、アンダースコアーで開始しないようにする必要があります。プロセス・インスタンス名が指定されていない場合、ストリング・フォーマットのプロセス・インスタンス ID (PIID) が名前として使用されます。

   ```
   ProcessTemplateData template = processTemplates[0];
   //create a message for the single starting receive activity
   ClientObjectWrapper input = process.createMessage
                               (template.getID(),
                               template.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   //start the process
   PIID piid = process.initiate(template.getName(), "CustomerOrder", input);
   ```

   このアクションによって、インスタンス CustomerOrder が作成され、一部の顧客データが受け渡されます。プロセスが開始されると、新規プロセス・インスタンスのオブジェクト ID を呼び出し元に戻します。

   プロセス・インスタンスのスターターは、要求の呼び出し元に設定されます。このユーザーは、このプロセス・インスタンスの作業項目を受信します。プロセス・インスタンスのプロセス管理者、リーダー、および編集者が決定され、プロセス・インスタンスの作業項目を受信します。追加のアクティビティー・インスタンスが決定されます。これらは自動的に開始されるか、または human task、receive、pick アクティビティーの場合、作業項目が潜在的な所有者に対して作成されます。

非固有の開始サービスを含む長期実行プロセスの開始:

長期実行プロセスは、複数の開始 receive アクティビティーまたは pick アクティビティーを介して開始することができます。initiate メソッドを使用して、プロセスを開始することができます。例えば、プロセスが複数の receive または pick アクティビティー、または複数の onMessage 定義を持つ pick アクティビティーから開始される場合など、開始サービスが固有のものではない場合、呼び出されるサービスを識別する必要があります。

プロシージャー

1. オプション: プロセス・テンプレートをリストして、開始するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
     ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
      "PROCESS_TEMPLATE.NAME",
       new Integer(50),
       (TimeZone)null);
   ```

   結果は名前でソートされます。長期実行プロセスとして開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 呼び出すべき開始サービスを判別します。

   ```
   ProcessTemplateData template = processTemplates[0];
   ActivityServiceTemplateData[] startActivities =
           process.getStartActivities(template.getID());
   ```

3. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。プロセス・インスタンス名を指定する場合、アンダースコアーで開始しないようにする必要があります。プロセス・インスタンス名が指定されていない場合、ストリング・フォーマットのプロセス・インスタンス ID (PIID) が名前として使用されます。

   ```
   ActivityServiceTemplateData activity = startActivities[0];
   //create a message for the service to be called
   ClientObjectWrapper input = process.createMessage
                               (activity.getServiceTemplateID(),
                                activity.getActivityTemplateID(),
                                activity.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   //start the process
   PIID piid = process.sendMessage(activity.getServiceTemplateID(),
                                   activity.getActivityTemplateID(),
                                   input);
   ```

   このアクションによって、インスタンスが作成され、一部の顧客データが受け渡されます。プロセスが開始されると、新規プロセス・インスタンスのオブジェクト ID を呼び出し元に戻します。

   プロセス・インスタンスの開始は、要求の呼び出し元に設定され、プロセス・インスタンスの作業項目を受信します。プロセス・インスタンスのプロセス管理者、リーダー、および編集者が決定され、プロセス・インスタンスの作業項目を受信します。追加のアクティビティー・インスタンスが決定されます。これらは自動的に開始されるか、または human task、receive、pick アクティビティーの場合、作業項目が潜在的な所有者に対して作成されます。

ビジネス・プロセスの中断と再開:

長期にわたって実行するトップレベルのプロセス・インスタンスを実行中に中断し、再開して完了することができます。

始める前に

呼び出し元は、プロセス・インスタンスの管理者、またはビジネス・プロセス管理者でなければなりません。プロセス・インスタンスを中断するには、プロセス・インスタンスが実行状態または失敗状態でなければなりません。

このタスクについて

例えば、プロセスで後で使用されるバックエンド・システムへのアクセスを構成するために、プロセス・インスタンスを中断することがあります。プロセスの前提条件を満たしていれば、そのプロセス・インスタンスを再開することができます。また、プロセスを中断し、プロセス・インスタンスの失敗の原因となっている問題を修正して、問題が修正されたら再開することもできます。

プロシージャー

1. 中断する実行中のプロセス CustomerOrder を取得します。

   ```
   ProcessInstanceData processInstance =
                       process.getProcessInstance("CustomerOrder");
   ```

2. プロセス・インスタンスを中断します。

   ```
   PIID piid = processInstance.getID();
   process.suspend( piid );
   ```

   このアクションにより、指定したトップレベルのプロセス・インスタンスが中断します。プロセス・インスタンスは、中断状態になります。 autonomy 属性が child に設定されたサブプロセスは、実行中、失敗、終了中、または補正中の状態であれば、中断されます。このプロセス・インスタンスに関連するインライン・タスクも中断されますが、このプロセス・インスタンスに関連するスタンドアロン・タスクは中断されません。

   この状態では、開始されたアクティビティーはまだ完了することはできますが、新規のアクティビティーはアクティブ化されません。例えば、要求済み状態の human task アクティビティーは完了することができます。

3. プロセス・インスタンスを再開します。

   ```
   process.resume( piid );
   ```

   このアクションにより、プロセス・インスタンスとそのサブプロセスが中断前の状態に戻ります。

ビジネス・プロセスの再開:

完了、終了、失敗、補正のいずれかの状態にあるプロセス・インスタンスを再開させることができます。

始める前に

呼び出し元は、プロセス・インスタンスの管理者、またはビジネス・プロセス管理者でなければなりません。

このタスクについて

プロセス・インスタンスの再開は、プロセス・インスタンスを初めて開始する手順と同様です。ただし、プロセス・インスタンスの再開時には、プロセス・インスタンス ID が認識されているため、インスタンスの入力メッセージが使用可能です。

プロセスに、プロセス・インスタンスを作成可能な複数の receive アクティビティーまたは pick アクティビティー (receive choice アクティビティーとも呼ばれる) が含まれる場合、これらのアクティビティーに属するすべてのメッセージを使用して、プロセス・インスタンスを再始動します。これらのアクティビティーのいずれかが、要求/応答操作をインプリメントする場合、関連する reply アクティビティーがナビゲートされると、応答が再度送信されます。

プロシージャー

1. 再開させるプロセスを取得します。

```
ProcessInstanceData processInstance =
                    process.getProcessInstance("CustomerOrder");
```

2. プロセス・インスタンスを再開します。

```
PIID piid = processInstance.getID();
process.restart( piid );
```

   このアクションにより、指定されたプロセス・インスタンスが再開されます。

プロセス・インスタンスの終了:

プロセス管理者権限を持つユーザーが、リカバリー不能状態として認識されているトップレベルのプロセス・インスタンスを終了する必要がある場合があります。プロセス・インスタンスは、未解決のサブプロセスやアクティビティーがあってもこれらを待たずに即時に終了するため、プロセス・インスタンスの終了は例外的な場合にのみ行ってください。

プロシージャー

1. 終了するプロセス・インスタンスを検索します。

```
ProcessInstanceData processInstance =
       process.getProcessInstance("CustomerOrder");
```

2. プロセス・インスタンスを終了します。

   プロセス・インスタンスを終了する場合、補正を使用してプロセス・インスタンスを終了することも、補正を使用せずに終了することもできます。

   補正を使用してプロセス・インスタンスを終了するには、以下のようにします。

```
PIID piid = processInstance.getID();
process.forceTerminate(piid, CompensationBehaviour.INVOKE_COMPENSATION);
```

   補正を使用しないでプロセス・インスタンスを終了するには、以下のようにします。

```
PIID piid = processInstance.getID();
process.forceTerminate(piid);
```

   補正を使用してプロセス・インスタンスを終了する場合、プロセスの補正は、障害が最上位スコープで発生したかのように実行されます。補正を使用せずにプロ

セス・インスタンスを終了する場合、プロセス・インスタンスはアクティビティー、予定タスク、またはインライン呼び出しタスクが正常に終了するのを待たずに、即時に終了されます。

プロセスおよびプロセスに関連するスタンドアロン・タスクによって開始されるアプリケーションは、強制終了要求によって終了されません。そのようなアプリケーションを終了させる場合は、プロセスによって開始されるアプリケーションを明示的に終了するステートメントをプロセス・アプリケーションに追加する必要があります。

プロセス・インスタンスの削除:

完了済みのプロセス・インスタンスは、プロセス・モデル内のプロセス・テンプレートに対応するプロパティーが設定されていれば、Business Process Choreographer データベースから自動的に削除されます。例えば、監査ログに書き込まれていないプロセス・インスタンスのデータを照会する場合などは、プロセス・インスタンスをデータベースに保存しておくことができます。ただし、格納されたプロセス・インスタンスのデータは、ディスク・スペースとパフォーマンスに影響を与えるだけでなく、同じ相関セット値を使用するプロセス・インスタンスも作成されなくなります。したがって、プロセス・インスタンス・データは、データベースから定期的に削除してください。

このタスクについて

プロセス・インスタンスを削除するには、プロセス管理者権限が必要であり、そのプロセス・インスタンスは、トップレベルのプロセス・インスタンスでなければなりません。

以下の例では、完了したプロセス・インスタンスをすべて削除する方法が示されています。

プロシージャー

1. 完了したプロセス・インスタンスをリストします。

```
QueryResultSet result =
     process.query("DISTINCT PROCESS_INSTANCE.PIID",
                   "PROCESS_INSTANCE.STATE =
                           PROCESS_INSTANCE.STATE.STATE_FINISHED",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、完了したプロセス・インスタンスをリストした照会結果セットを戻します。

2. 完了したプロセス・インスタンスを削除します。

```
while (result.next() )
{
   PIID piid = (PIID) result.getOID(1);
   process.delete(piid);
}
```

   このアクションにより、選択したプロセス・インスタンスとそのインライン・タスクがデータベースから削除されます。

human task アクティビティーの処理

ビジネス・プロセス内の human task アクティビティーは、作業項目を通じて、組織内のさまざまな人に割り当てられます。プロセスが開始されると、潜在的な所有者に対して作業項目が作成されます。

このタスクについて

human task アクティビティーが活動状態にされると、アクティビティー・インスタンスと、関連した予定タスクの両方が作成されます。 human task アクティビティーおよび作業項目管理の処理は、Human Task Manager に委任されます。アクティビティー・インスタンスの状態変更はすべてタスク・インスタンスに反映され、その反対にタスク・インスタンスの状態変更はアクティビティー・インスタンスに反映されます。

潜在的な所有者がアクティビティーを要求します。このユーザーは、関係のある情報の提供とアクティビティーの完了に対して責任があります。

プロシージャー

1. 作業の準備ができている、ログオン・ユーザーに属するアクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
                    ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
                    WORK_ITEM.REASON =
                         WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、ログオン・ユーザーが作業することができるアクティビティーが含まれる照会結果セットを戻します。

2. 作業対象のアクティビティーを要求します。

```
if (result.size() > 0)
{
 result.first();
 AIID aiid = (AIID) result.getOID(1);
 ClientObjectWrapper input = process.claim(aiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

   アクティビティーが要求されると、アクティビティーの入力メッセージが戻されます。

3. アクティビティーの作業が終了したら、アクティビティーを完了します。 アクティビティーは、正常に完了することも、障害メッセージが表示されて完了することもあります。アクティビティーが正常に完了した場合、出力メッセージが渡されます。アクティビティーが失敗した場合、アクティビティーは失敗状態または停止状態に置かれ、障害メッセージが渡されます。これらのアクションに対し

て、適切なメッセージを作成する必要があります。メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

a. アクティビティーを正常に完了するには、出力メッセージを作成します。

```
ActivityInstanceData activity = process.getActivityInstance(aiid);
ClientObjectWrapper output =
      process.createMessage(aiid, activity.getOutputMessageTypeName());
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the activity
process.complete(aiid, output);
```

このアクションは、オーダー番号が含まれる出力メッセージを設定します。

b. 障害が発生した場合にアクティビティーを完了するには、障害メッセージを作成します。

```
//retrieve the faults modeled for the human task activity
List faultNames = process.getFaultNames(aiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      process.createMessage(aiid, faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
}

process.complete(aiid, (String)faultNames.get(0), myFault);
```

このアクションは、アクティビティーを失敗状態または停止状態のいずれかに設定します。プロセス・モデル内のアクティビティーの **continueOnError** パラメーターが真に設定されている場合、アクティビティーは失敗状態に置かれ、ナビゲーションが続行されます。 **continueOnError** パラメーターが FALSE に設定されているときに、周囲の有効範囲で障害がキャッチされない場合、そのアクティビティーは停止状態になります。この状態では、強制完了または強制再試行を使用してアクティビティーを修復できます。

単独ユーザー・ワークフローの処理

ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。このタイプのワークフローには、並列パスは存在しません。 completeAndClaimSuccessor API は、このタイプのワークフローの処理をサポートします。

このタスクについて

オンライン・ブックストアでは、購入者は一連の操作を完了することで本を注文します。この一連の操作は、human task アクティビティー (予定タスク) として実装できます。購入者が複数の書籍を注文する場合は、これが次の human task アクティビティーの要求に相当します。このタイプのワークフローは、ページ・フローとも呼ばれます。ユーザー・インターフェース定義が、ユーザー・インターフェースのダイアログのフローを制御するアクティビティーと関連付けられているためです。

completeAndClaimSuccessor API は human task アクティビティーを完了し、ログオン・ユーザーの同じプロセス・インスタンスで次のアクティビティーを要求します。そして、次に要求したアクティビティーの情報 (処理される入力メッセージなど) を戻します。次のアクティビティーは、完了したアクティビティーと同じトランザクション内で使用可能になるため、プロセス・モデル内のすべての human task アクティビティーのトランザクション動作が participates に設定される必要があります。

この例を、Business Flow Manager API と Human Task Manager API の両方を使用する例と比較してください。

プロシージャー

1. アクティビティー・シーケンスで最初のアクティビティーを要求します。

```
//
//Query the list of activities that can be claimed by the logged-on user
//
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
                    ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
                    ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
                    WORK_ITEM.REASON =
                         WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                    (String)null, (Integer)null, (TimeZone)null);
...
//
//Claim the first activity
//
if (result.size() > 0)
{
 result.first();
 AIID aiid = (AIID) result.getOID(1);
 ClientObjectWrapper input = process.claim(aiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

   アクティビティーが要求されると、アクティビティーの入力メッセージが戻されます。

2. アクティビティーの作業が終了したら、そのアクティビティーを完了して次のアクティビティーを要求します。

アクティビティーを完了するには、出力メッセージを渡します。出力メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

```
ActivityInstanceData activity = process.getActivityInstance(aiid);
ClientObjectWrapper output =
      process.createMessage(aiid, activity.getOutputMessageTypeName());
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the activity and claim the next one
CompleteAndClaimSuccessorResult successor =
   process.completeAndClaimSuccessor(aiid, output);
```

このアクションは、オーダー番号が含まれる出力メッセージを設定し、シーケンス内の次のアクティビティーを要求します。後続アクティビティーに AutoClaim が設定されており、有効なパスが複数存在する場合は、後続アクティビティーのすべてが要求され、ランダムなアクティビティーが次のアクティビティーとして戻されます。このユーザーに割り当て可能な後続アクティビティーが他にない場合は、NULL が戻されます。

後に続くことができる並列パスがプロセスに含まれ、これらのパスに、ログオン・ユーザーが潜在的な所有者である human task アクティビティーが複数含まれる場合、ランダムなアクティビティーが自動的に要求され、次のアクティビティーとして戻されます。

3. 次のアクティビティーを処理します。

```
String name = successor.getActivityName();

ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject()!=
               null && nextInput.getObject() instanceof DataObject )
{
  activityInput = (DataObject)input.getObject();
  // read the values
  ...
}

aiid = successor.getAIID();
```

4. アクティビティーを完了する場合は、ステップ 2 に進みます。

関連タスク

114 ページの『ヒューマン・タスクを含む単一の個人ワークフローの処理』
ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。この例では、一連の human task アクティビティー (予定タスク) として、書籍を注文するための一連のアクションを実装する方法を示しています。ワークフローの処理には、Business Flow Manager と Human Task Manager API の両方が使用されます。

待機中のアクティビティーへのメッセージの送信

インバウンド・メッセージ・アクティビティー (receive アクティビティー、pick アクティビティーの onMessage、イベント・ハンドラーの onEvent) を使用して、実行

中のプロセスを「外の世界」からのイベントと同期することができます。例えば、情報に対する要求に応えたお客様からの E メールの受信は、このようなイベントとみなされます。

このタスクについて

親タスクを使用して、アクティビティーにメッセージを送信できます。

プロシージャー

1. 特定のプロセス・インスタンス ID を持つプロセス・インスタンスのログオンしたユーザーからのメッセージを待っているアクティビティー・サービス・テンプレートをリストします。

   ```
   ActivityServiceTemplateData[] services = process.getWaitingActivities(piid);
   ```

2. 最初の待機サービスにメッセージを送信します。

   最初のサービスを、ユーザーがサービスを提供しようとするサービスと想定します。呼び出し元は、メッセージを受信するアクティビティーの潜在的なスターター、またはプロセス・インスタンスの管理者である必要があります。

   ```
   VTID vtid = services[0].getServiceTemplateID();
   ATID atid = services[0].getActivityTemplateID();
   String inputType = services[0].getInputMessageTypeName();

   // create a message for the service to be called
     ClientObjectWrapper message =
           process.createMessage(vtid,atid,inputMessageTypeName);
     DataObject myMessage = null;
     if ( message.getObject()!= null && message.getObject() instanceof DataObject )
     {
       myMessage = (DataObject)message.getObject();
       //set the strings in the message, for example, chocolate is to be ordered
       myMessage.setString("Order", "chocolate");
     }

     // send the message to the waiting activity
     process.sendMessage(vtid, atid, message);
   }
   ```

   このアクションによって、指定されたメッセージを待機アクティビティー・サービスに送信し、一部のオーダー・データを渡します。

   また、プロセス・インスタンス ID を指定して、メッセージが指定されたプロセス・インスタンスに送信されたことを確認することもできます。プロセス・インスタンス ID が指定されていない場合、メッセージは、アクティビティー・サービス、およびメッセージの相関値によって識別されたプロセス・インスタンスに送信されます。プロセス・インスタンス ID が指定された場合、相関値を使用して検出されたプロセス・インスタンスがチェックされ、指定されたプロセス・インスタンス ID であることが確認されます。

イベントの処理

ビジネス・プロセス全体とビジネス・プロセスの各スコープを、関連するイベントの発生時に呼び出されるイベント・ハンドラーと関連付けることができます。 プロセスによりイベント・ハンドラーを使用して、Web サービス操作を提供できるという点で、イベント・ハンドラーは、receive アクティビティーや pick アクティビティーと似ています。

このタスクについて

イベント・ハンドラーは、対応するスコープが実行中である限り、何度でも呼び出すことができます。また、イベント・ハンドラーの複数インスタンスを並行して活動化することができます。

以下のコードの断片では、あるプロセス・インスタンス用のアクティブなイベント・ハンドラーを取得する方法、および入力メッセージを送信する方法を示しています。

プロシージャー

1. プロセス・インスタンス ID のデータを判別し、そのプロセスのアクティブなイベント・ハンドラーをリストします。

```
ProcessInstanceData processInstance =
       process.getProcessInstance( "CustomerOrder2711");
EventHandlerTemplateData[] events = process.getActiveEventHandlers(
                                      processInstance.getID() );
```

2. 入力メッセージを送信します。

   この例では、最初に検出されたイベント・ハンドラーを使用します。

```
EventHandlerTemplateData event = null;
if ( events.length > 0 )
{
    event = events[0];

    // create a message for the service to be called
    ClientObjectWrapper input = process.createMessage(
    event.getID(), event.getInputMessageTypeName());

    if (input.getObject() != null && input.getObject() instanceof DataObject )
    {
       DataObject inputMessage = (DataObject)input.getObject();
       // set content of the message, for example, a customer name, order number
       inputMessage.setString("CustomerName", "Smith");
       inputMessage.setString("OrderNo", "2711");

        // send the message
       process.sendMessage( event.getProcessTemplateName(),
                            event.getPortTypeNamespace(),
                            event.getPortTypeName(),
                            event.getOperationName(),
     input );
     }
 }
```

   このアクションにより、指定されたメッセージがプロセスのアクティブなイベント・ハンドラーに送信されます。

プロセスの結果の分析

プロセスは、Web Services Description Language (WSDL) の片方向操作または要求/応答操作としてモデル化される Web サービス操作を公開できます。片方向インターフェースを使用する長期実行プロセスの結果は、そのプロセスに出力がないため、getOutputMessage メソッドを使用して取り出すことはできません。ただし、代わりに変数の内容を照会できます。

このタスクについて

プロセスの結果は、プロセス・インスタンスが派生したプロセス・テンプレートが、派生したプロセス・インスタンスの自動削除を指定しない場合にのみ、データベースに保管されます。

プロシージャー

プロセスの結果を分析し、例えば、オーダー番号などを確認します。

```
QueryResultSet result = process.query
                       ("PROCESS_INSTANCE.PIID",
                        "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
                         PROCESS_INSTANCE.STATE =
                                 PROCESS_INSTANCE.STATE.STATE_FINISHED",
                         (String)null, (Integer)null, (TimeZone)null);
if (result.size() > 0)
{
  result.first();
  PIID piid = (PIID) result.getOID(1);
  ClientObjectWrapper output = process.getOutputMessage(piid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject )
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

アクティビティーの修復

長期実行プロセスには、やはり長期間実行されるアクティビティーが含まれる場合があります。これらのアクティビティーでは、catch されていないエラーが発生して、停止状態になる可能性があります。実行状態のアクティビティーが、反応していないように見える可能性もあります。どちらの場合でも、プロセス管理者は、プロセスのナビゲーションを継続できるように、いくつかの方法でアクティビティーを処理することができます。

このタスクについて

Business Process Choreographer API は、アクティビティーの修復のために、forceRetry メソッドおよび forceComplete メソッドを提供しています。アクティビティーの修復アクションをアプリケーションに追加する方法を示した例が提供されています。

アクティビティーの強制完了: このタスクについて

長期実行プロセスのアクティビティーで、障害が発生することがあります。これらの障害が、囲んでいるスコープ内で障害ハンドラーによって catch されておらず、関連したアクティビティー・テンプレートが、エラー発生時にアクティビティーが停止するように指定している場合、アクティビティーは修復することができるように停止状態になります。この状態で、アクティビティーの完了を強制することができます。

例えば、アクティビティーが応答しない場合、実行状態のアクティビティーを強制的に完了することもできます。

特定のタイプのアクティビティーでは、追加要件が存在します。

human task アクティビティー
送信されるはずだったメッセージ、または引き起こされるはすだった障害など、強制完了呼び出しでパラメーターを渡すことができます。

script アクティビティー
強制完了呼び出しで、パラメーターを渡すことはできません。ただし、修復する必要がある変数を設定する必要があります。

invoke アクティビティー
invoke アクティビティーが実行状態の場合、サブプロセスでない非同期サービスを呼び出す invoke アクティビティーを強制的に完了することもできます。例えば、非同期サービスが呼び出されて応答がない場合、こうすることがあります。

プロシージャー

1. 停止状態の停止アクティビティーをリストします。

```
QueryResultSet result =
     process.query("DISTINCT ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    PROCESS_INSTANCE.NAME='CustomerOrder'",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、 CustomerOrder プロセス・インスタンスに対して停止アクティビティーを戻します。

2. 例えば、停止した human task アクティビティーなどのアクティビティーを完了します。

   この例では、出力メッセージが渡されます。

```
if (result.size() > 0)
{
   result.first();
   AIID aiid = (AIID) result.getOID(1);
   ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper output =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( output.getObject()!= null && output.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)output.getObject();
     //set the parts in your message, for example, an order number
     myMessage.setInt("OrderNo", 4711);
   }

   boolean continueOnError = true;
   process.forceComplete(aiid, output, continueOnError);
}
```

   このアクションによって、アクティビティーが完了します。エラーが発生すると、**continueOnError** パラメーターに基づいて、障害が forceComplete 要求によって発生する場合に実行されるアクションが決まります。

   例では、**continueOnError** が true です。この値は、障害が発生した場合にアクティビティーが失敗状態になることを意味します。障害は、処理されるかプロセス・スコープに到達するまで、アクティビティーの囲んでいるスコープに伝搬されます。次にプロセスは障害状態になり、最終的に失敗状態になります。

停止されたアクティビティーの再試行: このタスクについて

長期実行プロセスのアクティビティーに、囲んでいるスコープ内で catch されていない障害が発生した場合、関連したアクティビティー・テンプレートが、エラー発生時にアクティビティーの停止を指定しているときは、アクティビティーは停止状態になり、修復することができます。アクティビティーの実行を再試行することができます。

アクティビティーが使用する変数を設定することができます。また、script アクティビティーの例外と共に、アクティビティーが予想したメッセージなど、強制再試行呼び出しのパラメーターを渡すこともできます。

プロシージャー

1. 停止アクティビティーをリストします。

```
QueryResultSet result =
      process.query("DISTINCT ACTIVITY.AIID",
                    "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                     PROCESS_INSTANCE.NAME='CustomerOrder'",
                     (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、 CustomerOrder プロセス・インスタンスに対して停止アクティビティーを戻します。

2. 例えば、停止した human task アクティビティーなどのアクティビティーの実行を再試行します。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper input =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
    {
      myMessage = (DataObject)input.getObject();
      //set the strings in your message, for example, chocolate is to be ordered
      myMessage.setString("OrderNo", "chocolate");
    }

   boolean continueOnError = true;
   process.forceRetry(aiid, input, continueOnError);
}
```

   このアクションによって、アクティビティーが再試行されます。エラーが発生した場合、**continueOnError** パラメーターによって、forceRetry 要求の処理中にエラーが発生した場合に実行するアクションが決まります。

   例では、**continueOnError** が true です。この場合、forceRetry 要求の処理中にエラーが発生すると、アクティビティーは失敗状態になります。障害は、処理されるかプロセス・スコープに到達するまで、アクティビティーの囲んでいるスコープに伝搬されます。次にプロセスは障害状態になり、プロセス状態が失敗状態で終了する前にプロセス・レベルの障害ハンドラーが実行されます。

BusinessFlowManagerService インターフェース

BusinessFlowManagerService インターフェースは、クライアント・アプリケーションから呼び出すことができるビジネス・プロセス機能を公開します。

BusinessFlowManagerService インターフェースから呼び出すことができるメソッドは、プロセスまたはアクティビティーの状態、およびそのメソッドが含まれているアプリケーションを使用するユーザーの権限によって異なります。ビジネス・プロセス・オブジェクトを操作するための main メソッドを、以下にリストします。これらのメソッドおよび BusinessFlowManagerService インターフェースで使用可能なその他のメソッドについての詳細は、com.ibm.bpe.api パッケージ内の Javadoc を参照してください。

プロセス・テンプレート

プロセス・テンプレートは、バージョン付けされ、デプロイされ、インストールされるプロセス・モデルで、ビジネス・プロセスの仕様を含んでいます。これは、例えば sendMessage() などの適切な要求を発行することによって、インスタンス化および開始することができます。プロセス・インスタンスの実行は、サーバーによって自動的に駆動されます。

表 24. プロセス・テンプレート用の API メソッド

メソッド	説明
getProcessTemplate	指定されたプロセス・テンプレートを取得します。
queryProcessTemplates	データベースに保管されているプロセス・テンプレートを取得します。

プロセス・インスタンス

以下の API メソッドは、プロセス・インスタンスの開始に関連しています。

表 25. プロセス・インスタンスの開始に関連する API メソッド

メソッド	説明
call	マイクロフローを作成および実行します。
callWithReplyContext	指定されたプロセス・テンプレートから、固有の開始サービスでのマイクロフローまたは固有の開始サービスでの長期実行プロセスを作成および実行します。呼び出しは、結果を非同期で待ちます。
callWithUISettings	マイクロフローを作成および実行し、出力メッセージとクライアント・ユーザー・インターフェース (UI) の設定を戻します。
initiate	プロセス・インスタンスを作成し、そのプロセス・インスタンスの処理を開始します。このメソッドは、長時間実行プロセスに使用します。このメソッドは、応答不要送信を適用するマイクロフローに対しても使用できます。

表 25. プロセス・インスタンスの開始に関連する API メソッド (続き)

メソッド	説明
sendMessage	指定されたメッセージを、指定されたアクティビティー・サービスおよびプロセス・インスタンスに送信します。同じ相関セット値を持つプロセス・インスタンスが存在しない場合には作成されます。このプロセスは、固有または非固有のどちらかの開始サービスを持つことができます。
getStartActivities	指定されたプロセス・テンプレートからプロセス・インスタンスを開始できるアクティビティーに関する情報を戻します。
getActivityServiceTemplate	指定されたアクティビティー・サービス・テンプレートを取得します。

表 26. プロセス・インスタンスのライフ・サイクルを制御するための API メソッド

メソッド	説明
suspend	実行状態または失敗状態にある、長期実行中のトップレベルのプロセス・インスタンスの実行を中断します。
resume	中断状態にある、長期実行中のトップレベルのプロセス・インスタンスの実行を再開します。
restart	完了、失敗、または終了状態にある、長期実行中のトップレベルのプロセス・インスタンスを再始動します。
forceTerminate	指定されたトップレベルのプロセス・インスタンスと、子 autonomy を含むそのサブプロセス、およびその実行中のアクティビティー、要求済みのアクティビティー、または待機中のアクティビティーを終了します。
delete	指定されたトップレベルのプロセス・インスタンスと、子 autonomy を含むそのサブプロセスを削除します。
query	データベースから、検索基準に一致するプロパティーを取得します。

アクティビティー

invoke アクティビティーの場合、プロセス・モデルで、それらのアクティビティーがエラー状態でも続行されるように指定できます。continueOnError フラグが FALSE に設定されているときに未処理エラーが発生すると、そのアクティビティーは停止状態になります。その場合は、プロセス管理者が、そのアクティビティーを修復することができます。continueOnError フラグおよびそれに関連する修復機能は、例えば、invoke アクティビティーが失敗することがある長期実行プロセスなどで使用することができますが、補正および障害処理のモデル化には、かなりの労力が必要です。

アクティビティーの操作および修復には、以下のメソッドが使用可能です。

表 27. アクティビティー・インスタンスのライフ・サイクルを制御するための API メソッド

メソッド	説明
claim	準備ができたアクティビティー・インスタンスを要求し、ユーザーがそのアクティビティーを使用できるようにします。
cancelClaim	アクティビティー・インスタンスの要求を取り消します。
complete	アクティビティー・インスタンスを完了します。
completeAndClaimSuccessor	アクティビティー・インスタンスを完了し、ログオン担当者の同じプロセス・インスタンス内の次のアクティビティーを要求します。
forceComplete	実行状態または停止状態にあるアクティビティー・インスタンスを強制的に完了します。
forceRetry	実行状態または停止状態にあるアクティビティー・インスタンスを強制的に反復します。
query	データベースから、検索基準に一致するプロパティーを取得します。

変数およびカスタム・プロパティー

このインターフェースは、変数の値を取得および設定するための get および set メソッドを提供します。指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスに関連付けたり、指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスから取得することもできます。カスタム・プロパティーの名前および値は、java.lang.String 型である必要があります。

表 28. 変数およびカスタム・プロパティーの API メソッド

メソッド	説明
getVariable	指定された変数を取得します。
setVariable	指定された変数を設定します。
getCustomProperty	指定されたアクティビティーまたはプロセス・インスタンスの指定されたカスタム・プロパティーを取得します。
getCustomProperties	指定されたアクティビティーまたはプロセス・インスタンスのカスタム・プロパティーを取得します。
getCustomPropertyNames	指定されたアクティビティーまたはプロセス・インスタンスのカスタム・プロパティーの名前を取得します。
setCustomProperty	指定されたアクティビティーまたはプロセス・インスタンスのカスタム固有値を保管します。

ヒューマン・タスク用のアプリケーションの開発

タスクは、コンポーネントが人をサービスとして呼び出したり、人がサービスを呼び出すための手段となります。ヒューマン・タスクに関する標準的なアプリケーションの例が提供されています。

このタスクについて

Human Task Manager API について詳しくは、com.ibm.task.api パッケージにある Javadoc を参照してください。

同期インターフェースを起動する呼び出しタスクの開始

呼び出しタスクは、Service Component Architecture (SCA) コンポーネントに関連付けられます。タスクは、開始されると SCA コンポーネントを起動します。呼び出しタスクを同期的に開始するのは、関連した SCA コンポーネントを同期的に呼び出せる場合に限ってください。

このタスクについて

このような SCA コンポーネントは、Microflow や単純な Java クラスなどとして実装できます。

このシナリオでは、タスク・テンプレートのインスタンスが作成され、一部の顧客データが渡されます。両方向操作が戻るまで、タスクは実行状態のままです。タスクの結果である OrderNo が呼び出し元に戻されます。

プロシージャー

1. オプション: タスク・テンプレートをリストして、実行する呼び出しタスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
   ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
    "TASK_TEMPL.NAME",
     new Integer(50),
     (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みの派生元テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```

3. タスクを作成して、タスクを同期実行します。

タスクを同期実行するには、両方向の操作であることが必要です。例では、createAndCallTask メソッドを使用してタスクを作成および実行します。

```
ClientObjectWrapper output = task.createAndCallTask( template.getName(),
                                                     template.getNamespace(),
                                                     input);
```

4. タスクの結果を分析します。

```
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

非同期インターフェースを起動する呼び出しタスクの開始

呼び出しタスクは、Service Component Architecture (SCA) コンポーネントに関連付けられます。タスクは、開始されると SCA コンポーネントを起動します。呼び出しタスクを非同期的に開始するのは、関連した SCA コンポーネントを非同期的に呼び出せる場合に限ってください。

このタスクについて

このような SCA コンポーネントは、長時間実行プロセスや片方向操作などとして実装できます。

このシナリオでは、タスク・テンプレートのインスタンスが作成され、一部の顧客データが渡されます。

プロシージャー

1. オプション: タスク・テンプレートをリストして、実行する呼び出しタスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
     ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
      "TASK_TEMPL.NAME",
       new Integer(50),
       (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みの派生元テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```

3. タスクを作成して、非同期に実行します。

例では、createAndStartTask メソッドを使用してタスクを作成および実行します。

```
task.createAndStartTask( template.getName(),
                         template.getNamespace(),
                         input,
                         (ReplyHandlerWrapper)null);
```

タスク・インスタンスの作成と開始

このシナリオでは、コラボレーション・タスク (API ではヒューマン・タスクともいう) を定義するタスク・テンプレートのインスタンスを作成し、タスク・インスタンスを開始する方法を示します。

プロシージャー

1. オプション: タスク・テンプレートをリストして、実行するコラボレーション・タスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
   ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_HUMAN",
    "TASK_TEMPL.NAME",
     new Integer(50),
     (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みのタスク・テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```

3. コラボレーション・タスクを作成し、開始します。この例では、応答ハンドラーは指定されません。

   この例では、createAndStartTask メソッドを使用してタスクを作成し、開始します。

   ```
   TKIID tkiid = task.createAndStartTask( template.getName(),
                                          template.getNamespace(),
                                          input,
                                          (ReplyHandlerWrapper)null);
   ```

   タスク・インスタンスに関連する人に対して作業項目が作成されます。例えば、潜在的な所有者は、新規タスク・インスタンスを要求できます。

4. タスク・インスタンスを要求します。

   ```
   ClientObjectWrapper input2 = task.claim(tkiid);
   DataObject taskInput = null ;
   if ( input2.getObject()!= null && input2.getObject() instanceof DataObject )
   {
   ```

```
  taskInput = (DataObject)input2.getObject();
  // read the values
  ...
}
```

タスク・インスタンスが要求されると、タスクの入力メッセージが戻されます。

予定タスクまたはコラボレーション・タスクの処理

予定タスク (API では参加タスクともいう) またはコラボレーション・タスク (API ではヒューマン・タスクともいう) は、作業項目を通じて組織内のさまざまな人に割り当てられます。プロセスが human task アクティビティーにナビゲートしたときなどに、予定タスクとそれに関連した作業項目が作成されます。

このタスクについて

潜在的な所有者の中の 1 人が、作業項目に関連したタスクを要求します。このユーザーは、関係のある情報の提供とタスクの完了に対して責任があります。

プロシージャー

1. 作業の準備ができている、ログオン・ユーザーに属するタスクをリストします。

```
QueryResultSet result =
     task.query("TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_READY AND
                (TASK.KIND = TASK.KIND.KIND_PARTICIPATING OR
                 TASK.KIND = TASK.KIND.KIND_HUMAN)AND
                 WORK_ITEM.REASON =
                   WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                 (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、ログオン・ユーザーが作業することができるタスクが含まれる照会結果セットを戻します。

2. 作業対象のタスクを要求します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper input = task.claim(tkiid);
  DataObject taskInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    taskInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

   タスクが要求されると、タスクの入力メッセージが戻されます。

3. タスクの作業が完了した場合、タスクを完了します。

   タスクは、正常に完了することも、障害メッセージが表示されて完了することもあります。タスクが正常に完了した場合、出力メッセージが渡されます。タスクが正常に完了しなかった場合、障害メッセージが渡されます。これらのアクションに対して、適切なメッセージを作成する必要があります。

   a. タスクを正常に完了するには、出力メッセージを作成します。

```
ClientObjectWrapper output =
      task.createOutputMessage(tkiid);
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the task
task.complete(tkiid, output);
```

このアクションは、オーダー番号が含まれる出力メッセージを設定します。タスクは、完了状態になります。

b. 障害が発生した場合にタスクを完了するには、障害メッセージを作成します。

```
//retrieve the faults modeled for the task
List faultNames = task.getFaultNames(tkiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      task.createFaultMessage(tkiid, (String)faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
}

task.complete(tkiid, (String)faultNames.get(0), myFault);
```

このアクションにより、エラー・コードを含む障害メッセージが設定されます。タスクは、失敗状態になります。

タスク・インスタンスの中断と再開

コラボレーション・タスク・インスタンス (API ではヒューマン・タスクともいう) または予定タスク・インスタンス (API では*参加タスク*ともいう) を中断できます。

始める前に

タスク・インスタンスは、作動可能状態または要求済み状態にすることができます。これをエスカレートすることが可能です。呼び出し元は、タスク・インスタンスの所有者、オリジネーター、または管理者である必要があります。

このタスクについて

タスク・インスタンスは、実行中に中断することができます。そうすることによって、例えば、タスクの完了に必要な情報を収集することができます。情報が使用可能になったら、タスク・インスタンスを再開できます。

プロシージャー

1. ログオン・ユーザーによって要求されたタスクのリストを取得します。

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.STATE = TASK.STATE.STATE_CLAIMED",
                                   (String)null,
                                   (Integer)null,
                                   (TimeZone)null);
```

このアクションにより、ログオン・ユーザーによって要求されたタスクのリストを含む照会結果セットが戻されます。

2. タスク・インスタンスを中断します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  task.suspend(tkiid);
}
```

このアクションにより、指定されたタスク・インスタンスが中断されます。タスク・インスタンスは中断状態になります。

3. プロセス・インスタンスを再開します。

```
task.resume( tkiid );
```

このアクションにより、タスク・インスタンスが中断前の状態になります。

タスクの結果の分析

予定タスク (API では*参加*タスクともいう) またはコラボレーション・タスク (API ではヒューマン・タスクともいう) は非同期に実行します。タスク開始時に応答ハンドラーが指定された場合、タスク完了時に自動的に出力メッセージが戻されます。応答ハンドラーが指定されていない場合、メッセージを明示的に検索する必要があります。

このタスクについて

タスクの結果は、そのタスク・インスタンスの派生元となったタスク・テンプレートに、派生したタスク・インスタンスの自動削除が指定されていない場合にのみ、データベースに保管されます。

プロシージャー

タスクの結果を分析します。

例では、正常に完了したタスクのオーダー番号を確認する方法を示します。

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.NAME = 'CustomerOrder' AND
                                    TASK.STATE = TASK.STATE.STATE_FINISHED",
                                    (String)null, (Integer)null, (TimeZone)null);
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper output = task.getOutputMessage(tkiid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject)
  {
```

```
    myOutput  = (DataObject)output.getObject();
    int order = myOutput.getInt("OrderNo");
  }
}
```

タスク・インスタンスの終了

管理者権限を有する担当者が、リカバリー不能状態になったと分かったタスク・インスタンスを終了する必要が生じる場合があります。タスク・インスタンスは即時に終了されるので、例外的な状況の場合にのみタスク・インスタンスを終了してください。

プロシージャー

1. 終了するタスク・インスタンスを検索します。

   ```
   Task taskInstance = task.getTask(tkiid);
   ```

2. タスク・インスタンスを終了します。

   ```
   TKIID tkiid = taskInstance.getID();
   task.terminate(tkiid);
   ```

   タスク・インスタンスは、未解決のタスクを待機しないで即時に終了します。

タスク・インスタンスの削除

タスク・インスタンスは、インスタンスの派生元となった関連タスク・テンプレートに自動削除が指定されている場合にのみ、完了時に自動的に削除されます。この例では、完了して自動的に削除されなかったタスク・インスタンスすべてを削除する方法を示します。

プロシージャー

1. 完了したタスク・インスタンスをリストします。

   ```
   QueryResultSet result =
        task.query("DISTINCT TASK.TKIID",
                   "TASK.STATE = TASK.STATE.STATE_FINISHED",
                    (String)null, (Integer)null, (TimeZone)null);
   ```

   このアクションにより、完了したタスク・インスタンスをリストした照会の結果セットが戻されます。

2. 完了したタスク・インスタンスを削除します。

   ```
   while (result.next() )
   {
    TKIID tkiid = (TKIID) result.getOID(1);
    task.delete(tkiid);
   }
   ```

要求されたタスクの解放

潜在的な所有者がタスクを要求した場合、この所有者にはタスクの完了に対する責任があります。ただし、要求されたタスクを、別の潜在的所有者が要求できるように解放しなければならない場合があります。

このタスクについて

管理者権限を持つユーザーが、要求されたタスクを解放する必要がある場合があります。この状態は、例えば、タスクを完了する必要があるが、タスクの所有者が不在の場合に発生する可能性があります。タスクの所有者も、要求されたタスクを解放できます。

プロシージャー

1. 特定のユーザー (例では、Smith) 所有の、要求されたタスクをリストします。

   ```
   QueryResultSet result =
        task.query("DISTINCT TASK.TKIID",
                   "TASK.STATE = TASK.STATE.STATE_CLAIMED AND
                    TASK.OWNER = 'Smith'",
                    (String)null, (Integer)null, (TimeZone)null);
   ```

   このアクションは、指定されたユーザーの Smith が要求したタスクをリストする照会結果セットを戻します。

2. 要求されたタスクを解放します。

   ```
   if (result.size() > 0)
   {
     result.first();
     TKIID tkiid = (TKIID) result.getOID(1);
     task.cancelClaim(tkiid, true);
   }
   ```

   このアクションは、タスクを作動可能状態に戻すので、他の潜在的な所有者の 1 人が要求することができます。元の所有者が設定した出力データまたは障害データはすべて保持されます。

作業項目の管理

アクティビティー・インスタンスまたはタスク・インスタンスの存続時間中に、オブジェクトに関連したユーザーのセットが変わる場合があります。例えば、社員が休暇に入ったり、新しい社員を雇用したり、またはワークロードを異なった方法で分散させる必要がある場合です。このような変更に対応するために、作業項目を作成、削除、または転送するようにアプリケーションを開発することができます。

このタスクについて

作業項目は、特定の理由でのユーザーまたはユーザーのグループへのオブジェクトの割り当てを表します。通常、このオブジェクトは human task アクティビティー・インスタンス、プロセス・インスタンス、またはタスク・インスタンスのいずれかです。理由は、ユーザーがオブジェクトに対して持っているロールから派生します。ユーザーは、オブジェクトとの関連において異なるロールを持つことができ、作業項目はこのようなロールそれぞれに対して作成されるため、1 つのオブジェクトが複数の作業項目を持つことが可能です。例えば、予定タスク・インスタンスでは、管理者、読者、編集者、所有者の作業項目を同時に持つことができます。

作業項目を管理するために実行可能なアクションは、ユーザーのロールによって異なります。例えば、管理者は作業項目を作成、削除、および転送できますが、タスク所有者は作業項目の転送のみが可能です。

- 作業項目を作成します。

  ```
  // query the task instance for which an additional
  // administrator is to be specified
  QueryResultSet result = task.query("TASK.TKIID",
  ```

```
                                    "TASK.NAME='CustomerOrder'",
                                     (String)null, (Integer)null,
                                     (TimeZone)null);
if ( result.size() > 0 )
{
  result.first();
  // create the work item
  task.createWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_ADMINISTRATOR,"Smith");
}
```

このアクションによって、管理者のロールがあるユーザー Smith の作業項目が作成されます。

- 作業項目を削除します。

```
// query the task instance for which a work item is to be deleted
QueryResultSet result = task.query("TASK.TKIID",
                                    "TASK.NAME='CustomerOrder'",
                                     (String)null, (Integer)null,
                                     (TimeZone)null);
if ( result.size() > 0 )
{
  result.first();
  // delete the work item
  task.deleteWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_READER,"Smith");
}
```

このアクションによって、リーダーのロールがあるユーザー Smith の作業項目が削除されます。

- 作業項目を転送します。

```
// query the task that is to be rescheduled
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.NAME='CustomerOrder' AND
                 TASK.STATE=TASK.STATE.STATE_READY AND
                 WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER AND
                 WORK_ITEM.OWNER_ID='Miller'",
                 (String)null, (Integer)null, (TimeZone)null);
if ( result.size() > 0 )
{
  result.first();
  // transfer the work item from user Miller to user Smith
  // so that Smith can work on the task
  task.transferWorkItem((TKIID)(result.getOID(1)),
                           WorkItem.REASON_POTENTIAL_OWNER,"Miller","Smith");
}
```

このアクションによって、作業項目をユーザー Smith に転送するので、 Smith は作業項目で作業することができます。

実行時のタスク・テンプレートおよびタスク・インスタンスの作成

通常、WebSphere Integration Developer などのモデル化ツールを使用して、タスク・テンプレートを作成します。次に、タスク・テンプレートを WebSphere Process Server にインストールし、例えば Business Process Choreographer Explorer を使用して、これらのテンプレートからインスタンスを作成します。ただし、実行時にヒューマン・タスクまたは参加タスクのインスタンスまたはテンプレートを作成することもできます。

このタスクについて

例えば、アプリケーションのデプロイ時にタスク定義が使用不可である場合、ワークフローに含まれるタスクがまだ認識されていない場合、またはタスクがユーザーのグループ間の随時のコラボレーションを対象とする必要がある場合などに、このようにすることがあります。

臨時の予定タスクまたはコラボレーション・タスクをモデル化できます。それには、com.ibm.task.api.TaskModel クラスのインスタンスを作成し、それを使用して再使用可能なタスク・テンプレートを作成するか、または 1 回実行のタスク・インスタンスを直接作成します。 TaskModel クラスのインスタンスを作成するために、一連のファクトリー・メソッドが com.ibm.task.api.ClientTaskFactory ファクトリー・クラスで使用可能です。実行時のヒューマン・タスクのモデル化は、Eclipse Modeling Framework (EMF) に基づいています。

プロシージャー

1. createResourceSet ファクトリー・メソッドを使用して org.eclipse.emf.ecore.resource.ResourceSet を作成します。
2. オプション: 複雑なメッセージ・タイプを使用する予定の場合、org.eclipse.xsd.XSDFactory (ファクトリー・メソッド getXSDFactory() を使用して取得できる) を使ってそのメッセージ・タイプを定義するか、または loadXSDSchema ファクトリー・メソッドを使用して既存の XML スキーマを直接インポートできます。

   複雑なタイプを WebSphere Process Server が使用できるようにするには、そのタイプをエンタープライズ・アプリケーションの一部としてデプロイします。
3. タイプ javax.wsdl.Definition の Web サービス記述言語 (WSDL) 定義を作成またはインポートします。

   createWSDLDefinition メソッドを使用して新規の WSDL 定義を作成できます。次に、それをポート・タイプおよび操作に追加できます。また、loadWSDLDefinition ファクトリー・メソッドを使用して、既存の WSDL 定義を直接インポートできます。
4. createTTask ファクトリー・メソッドを使用してタスク定義を作成します。

   より複雑なタスク・エレメントを追加または操作する場合は、getTaskFactory ファクトリー・メソッドを使って取得できる com.ibm.wbit.tel.TaskFactory クラスを使用できます。
5. createTaskModel ファクトリー・メソッドを使用してタスク・モデルを作成し、それをステップ 1 で作成されたリソース・バンドルに渡します。リソース・バンドルはその間に作成されたその他のすべての成果物を集約します。
6. オプション: TaskModel validate メソッドを使用してモデルを検証します。

結果

TaskModel パラメーターを持つ Human Task Manager EJB API create メソッドの 1 つを使用して、再使用可能なタスク・テンプレートを作成するか、または 1 回実行のタスク・インスタンスを作成します。

単純 Java 型を使用するランタイム・タスクの作成:

この例では、インターフェースで単純 Java 型 (例えば String オブジェクト) のみを使用するランタイム・タスクを作成します。

このタスクについて

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

プロシージャー

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. 操作の WSDL 定義を作成し、記述を追加します。

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
         ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );

// create a port type
PortType portType = factory.createPortType( definition, "doItPT" );

// create an operation; the input and output messages are of type String:
// a fault message is not specified
Operation operation = factory.createOperation
        ( definition, portType, "doIt",
          new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
          new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
          (Map)null );
```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);
```

```
// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

   ```
   TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
   ```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

   ```
   ValidationProblem[] validationProblems = taskModel.validate();
   ```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。アプリケーションが単純 Java 型のみを使用するため、アプリケーション名を指定する必要はありません。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     atask.createTask( taskModel, (String)null, "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, (String)null );
     ```

結果

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

複合タイプを使用するランタイム・タスクの作成:

この例では、インターフェースで複合タイプを使用するランタイム・タスクを作成します。複合タイプは既に定義されています。すなわち、クライアントのローカル・ファイル・システムには、複合タイプの記述を含む XSD ファイルがあります。

このタスクについて

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

プロシージャー

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

   ```
   ClientTaskFactory factory = ClientTaskFactory.newInstance();
   ResourceSet resourceSet = factory.createResourceSet();
   ```

2. 複合タイプの XSD 定義をリソース・セットに追加して、操作の定義時に使用できるようにします。

   ファイルは、コードが実行されたロケーションの相対位置に配置されます。

   ```
   factory.loadXSDSchema( resourceSet, "InputBO.xsd" );
   factory.loadXSDSchema( resourceSet, "OutputBO.xsd" );
   ```

3. 操作の WSDL 定義を作成し、記述を追加します。

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
         ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );

// create a port type
PortType portType = factory.createPortType( definition, "doItPT" );

// create an operation; the input message is an InputBO and
// the output message an OutputBO;
// a fault message is not specified
Operation operation = factory.createOperation
        ( definition, portType, "doIt",
          new QName( "http://Input", "InputBO" ),
          new QName( "http://Output", "OutputBO" ),
          (Map)null );
```

4. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

5. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

6. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

7. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

8. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。アプリケー

ションが Business Process Choreographer によってロードされるように、アプリケーションにダミー・タスクまたはダミー・プロセスも含まれるようにしてください。

- 以下の断片はタスク・インスタンスを作成します。

```
task.createTask( taskModel, "BOapplication", "HTM" );
```

- 以下の断片はタスク・テンプレートを作成します。

```
task.createTaskTemplate( taskModel, "BOapplication" );
```

結果

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

既存のインターフェースを使用するランタイム・タスクの作成:

この例は、既に定義されているインターフェースを使用するランタイム・タスクを作成します。すなわち、クライアントのローカル・ファイル・システムには、インターフェースの記述を含むファイルがあります。

このタスクについて

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

プロシージャー

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. 操作の WSDL 定義および記述にアクセスします。

   インターフェース記述は、コードが実行されたロケーションの相対位置に配置されます。

```
Definition definition = factory.loadWSDLDefinition(
                          resourceSet, "interface.wsdl"  );
PortType portType = definition.getPortType(
                    new QName( definition.getTargetNamespace(), "doItPT" ) );
Operation operation = portType.getOperation
                              ("doIt", (String)null, (String)null);
```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。アプリケーションが Business Process Choreographer によってロードされるように、アプリケーションにダミー・タスクまたはダミー・プロセスも含まれるようにしてください。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     task.createTask( taskModel, "BOapplication", "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, "BOapplication" );
     ```

結果

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

呼び出し側アプリケーションのインターフェースを使用するランタイム・タスクの作成:

この例では、呼び出し側アプリケーションに属するインターフェースを使用するランタイム・タスクを作成します。例えば、ランタイム・タスクがビジネス・プロセスの Java 断片で作成され、プロセス・アプリケーションからのインターフェースを使用します。

このタスクについて

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

プロシージャー

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();

// specify the context class loader so that following resources are found
ResourceSet resourceSet = factory.createResourceSet
              ( Thread.currentThread().getContextClassLoader() );
```

2. 操作の WSDL 定義および記述にアクセスします。

   収容パッケージ JAR ファイル内でパスを指定します。

```
Definition definition = factory.loadWSDLDefinition( resourceSet,
                           "com/ibm/workflow/metaflow/interface.wsdl" );
                      PortType portType = definition.getPortType(
                      new QName( definition.getTargetNamespace(), "doItPT" ) );
Operation operation = portType.getOperation
                                  ("doIt", (String)null, (String)null);
```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     task.createTask( taskModel, "WorkflowApplication", "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, "WorkflowApplication" );
     ```

結果

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

HumanTaskManagerService インターフェース

HumanTaskManagerService インターフェースは、ローカルまたはリモート・クライアントから呼び出すことができるタスク関連機能を公開します。

呼び出すことができるメソッドは、タスクの状態、およびそのメソッドが含まれているアプリケーションを使用する人物の権限によって異なります。タスク・オブジェクトを操作するための main メソッドを、以下にリストします。これらのメソッドおよび HumanTaskManagerService インターフェースで使用可能なその他のメソッドについての詳細は、com.ibm.task.api パッケージ内の Javadoc を参照してください。

タスク・テンプレート

タスク・テンプレートを扱う作業には、以下のメソッドが使用可能です。

表 29. タスク・テンプレート用の API メソッド

メソッド	説明
getTaskTemplate	指定されたタスク・テンプレートを取得します。
createAndCallTask	指定されたタスク・テンプレートからタスク・インスタンスを作成および実行し、同期式に結果を待機します。
createAndStartTask	指定されたタスク・テンプレートからタスク・インスタンスを作成および開始します。
createTask	指定されたタスク・テンプレートからタスク・インスタンスを作成します。
createInputMessage	指定されたタスク・テンプレート用の入力メッセージを作成します。例えば、タスクの開始に使用できるメッセージを作成します。

表 29. タスク・テンプレート用の API メソッド (続き)

メソッド	説明
queryTaskTemplates	データベースに保管されているタスク・テンプレートを取得します。

タスク・インスタンス

タスク・インスタンスを扱う作業には、以下のメソッドが使用可能です。

表 30. タスク・インスタンス用の API メソッド

メソッド	説明
getTask	タスク・インスタンスを取得します。任意の状態のタスク・インスタンスを取得できます。
callTask	呼び出しタスクを同期で開始します。
startTask	既に作成済みのタスクを開始します。
suspend	コラボレーション・タスクまたは予定タスクを中断します。
resume	コラボレーション・タスクまたは予定タスクを再開します。
terminate	指定されたタスク・インスタンスを終了します。呼び出しタスクを終了する場合、このアクションは、呼び出されたサービスには影響しません。
delete	指定されたタスク・インスタンスを削除します。
claim	処理のためタスクを要求します。
update	タスク・インスタンスを更新します。
complete	タスク・インスタンスを完了します。
cancelClaim	別の潜在的な所有者が処理できるように、要求されたタスク・インスタンスをリリースします。
createWorkItem	タスク・インスタンスの作業項目を作成します。
transferWorkItem	指定された所有者に作業項目を転送します。
deleteWorkItem	作業項目を削除します。

エスカレーション

エスカレーションを扱う作業には、以下のメソッドが使用可能です。

表 31. エスカレーションで使用できる API メソッド

メソッド	説明
getEscalation	指定されたエスカレーション・インスタンスを取得します。

カスタム・プロパティー

タスク、タスク・テンプレート、およびエスカレーションには、すべてカスタム・プロパティーを指定できます。このインターフェースは、カスタム・プロパティーの値を取得および設定するための get および set メソッドを提供します。指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスに関連付けたり、指定されたプロパティーをタスク・インスタンスから取得することもできます。カスタム・プロパティーの名前および値は、java.lang.String 型である必要があります。次のメソッドは、タスク、タスク・テンプレート、およびエスカレーションの場合に有効です。

表 32. 変数およびカスタム・プロパティーの API メソッド

メソッド	説明
getCustomProperty	指定されたタスク・インスタンスの指定された 1 つのカスタム・プロパティーを取得します。
getCustomProperties	指定されたタスク・インスタンスのカスタム・プロパティー (複数) を取得します。
getCustomPropertyNames	タスク・インスタンスのすべてのカスタム・プロパティーの名前を取得します。
setCustomProperty	指定されたタスク・インスタンスのカスタム固有値を保管します。

各タスクで許可されるアクション:

タスクに対して実行可能なアクションは、予定タスク、コラボレーション・タスク、呼び出しタスク、または管理タスクのいずれであるかによって異なります。

HumanTaskManager インターフェースで提供されるすべてのアクションが、すべての種類のタスクに対して使用できるわけではありません。次の表に、タスクの種類ごとに実行可能なアクションを示します。

アクション	タスクの種類			
	予定タスク	コラボレーション・タスク	呼び出しタスク	管理タスク
callTask			X	
cancelClaim	X	X[1]		
claim	X	X[1]		
complete	X	X[1]		X
completeWithFollowOnTask[4]	X	X[1]		
completeWithFollowOnTask[5]		X[3]	X[3]	
createFaultMessage	X	X	X	X
createInputMessage	X	X	X	X
createOutputMessage	X	X	X	X
createWorkItem	X	X[1]	X	X
delete	X[1]	X[1]	X	X[1]
deleteWorkItem	X	X[1]	X	X

アクション	タスクの種類			
	予定タスク	コラボレーション・タスク	呼び出しタスク	管理タスク
getCustomProperty	X	X[1]	X	X
getDocumentation	X	X[1]	X	X
getFaultNames	X	X[1]		
getFaultMessage	X	X[1]	X	
getInputMessage	X	X[1]	X	
getOutputMessage	X	X[1]	X	
getUsersInRole	X	X[1]	X	X
getTask	X	X[1]	X	X
getUISettings	X	X[1]	X	X
resume	X	X[1]		
setCustomProperty	X	X[1]	X	X
setFaultMessage	X	X[1]		
setOutputMessage	X	X[1]		
startTask	X[1]	X[1]	X	X
startTaskAsSubtask[6]	X	X[1]		
startTaskAsSubtask[7]		X[3]	X[3]	
suspend	X	X[1]		
suspendWithCancelClaim	X	X[1]		
terminate	X[1]	X[1]	X[1]	
transferWorkItem	X	X[1]	X	X
update	X	X[1]	X	X

注:

1. スタンドアロン・タスク、臨時タスク、およびタスク・テンプレートの場合のみ
2. スタンドアロン・タスク、ビジネス・プロセス内のインライン・タスク、および臨時タスクの場合のみ
3. スタンドアロン・タスクおよび臨時タスクの場合のみ
4. 後続のタスクを持つことが可能なタスクの種類
5. 後続のタスクとして使用可能なタスクの種類
6. サブタスクを持つことが可能なタスクの種類
7. サブタスクとして使用可能なタスクの種類

ビジネス・プロセスおよびヒューマン・タスク用アプリケーションの開発

ほとんどのビジネス・プロセス・シナリオには担当者が関係します。例えば、プロセスが開始または管理されるとき、あるいは human task アクティビティーが実行されるときには、ビジネス・プロセスに担当者の対話が必要となります。これらのシナリオをサポートするために、Business Flow Manager API と Human Task Manager API の両方を使用する必要があります。

このタスクについて

ビジネス・プロセス・シナリオに担当者を組み込むために、ビジネス・プロセスに以下の種類のタスクを含めることができます。

- インライン呼び出しタスク (API では*親タスク*ともいう)。

  すべての receive アクティビティー、pick アクティビティーの各 onMessage エレメント、およびイベント・ハンドラーの各 onEvent エレメントに対して呼び出しタスクを提供できます。次いで、このタスクはプロセスの開始、または実行中のプロセス・インスタンスとの通信を許可されているユーザーを制御します。
- 管理タスク。

  プロセスの管理またはプロセスの失敗したアクティビティーに対する管理操作の実行を許可されたユーザーを指定するための管理タスクを提供できます。
- 予定タスク (API では*参加タスク*ともいう)。

  予定タスクは human task アクティビティーを実装します。このタイプのアクティビティーにより、プロセスに担当者を含めることができます。

ビジネス・プロセス内の human task アクティビティーは、担当者がビジネス・プロセス・シナリオで実行する予定タスクを表します。 Business Flow Manager API と Human Task Manager API の両方を使用して、以下のシナリオを実現できます。

- ビジネス・プロセスとは、プロセスに属するすべてのアクティビティー (予定タスクによって表される human task アクティビティーを含む) のコンテナーです。プロセス・インスタンスが作成されると、固有オブジェクト ID (PIID) が割り当てられます。
- human task アクティビティーがプロセス・インスタンスの実行中に活動状態にされると、アクティビティー・インスタンスが作成されます。これはその固有オブジェクト ID (AIID) によって識別されます。同時に、インライン予定タスク・インスタンスも作成されます。これはそのオブジェクト ID (TKIID) によって識別されます。 human task アクティビティーとタスク・インスタンスの関係は、次のようにオブジェクト ID を使用して実現されます。
  - アクティビティー・インスタンスの予定タスク ID が、関連した予定タスクの TKIID に設定されます。
  - タスク・インスタンスの包含コンテキスト ID が、関連したアクティビティー・インスタンスを含むプロセス・インスタンスの PIID に設定されます。
  - タスク・インスタンスの親コンテキスト ID が、関連したアクティビティー・インスタンスの AIID に設定されます。
- すべてのインライン予定タスク・インスタンスのライフ・サイクルは、プロセス・インスタンスによって管理されます。プロセス・インスタンスが削除されると、タスク・インスタンスも削除されます。言い換えると、包含コンテキスト ID がプロセス・インスタンスの PIID に設定されているすべてのタスクが自動的に削除されます。

開始できるプロセス・テンプレートまたはアクティビティーの判別

ビジネス・プロセスは、Business Flow Manager API の call、initiate、または sendMessage メソッドの呼び出しにより開始できます。プロセスに 1 つの開始アクティビティーしかない場合は、パラメーターとしてプロセス・テンプレート名を必

要とするメソッド・シグニチャーを使用できます。プロセスに複数の開始アクティビティーがある場合は、開始アクティビティーを明示的に識別する必要があります。

このタスクについて

ビジネス・プロセスをモデル化する場合、モデラーは、ユーザーのサブセットだけしかプロセス・テンプレートからプロセス・インスタンスを作成できないように決定することができます。これは、インライン呼び出しタスクをプロセスの開始アクティビティーに関連付け、許可制限をそのタスクに指定することで実行できます。タスクの潜在的スターターまたは管理者だけが、タスクのインスタンスの (およびプロセス・テンプレートのインスタンスの) 作成を許可されます。

インライン呼び出しタスクが開始アクティビティーと関連付けられていない場合、または許可制限がタスクに指定されていない場合、誰でも開始アクティビティーを使用してプロセス・インスタンスを作成できます。

プロセスは、それぞれが異なる潜在的スターターまたは管理者に対する担当者照会を持つ、複数の開始アクティビティーを持つことができます。これはつまり、ユーザーがアクティビティー A を使用したプロセスの開始は許可されるが、アクティビティ B を使用しては許可されないということです。

プロシージャー

1. Business Flow Manager API を使用して、開始済み状態のプロセス・テンプレートの現行バージョンのリストを作成します。

   ヒント: queryProcessTemplates メソッドは、開始されていないアプリケーションの一部であるプロセス・テンプレートだけを除外します。そのため、結果をフィルター処理せずにこのメソッドを使用する場合、メソッドはどのような状態にあるかに関係なく、すべてのバージョンのプロセス・テンプレートを戻します。

   ```
   // current timestamp in UTC format, converted to yyyy-mm-ddThh:mm:ss
   String now = (new UTCDate()).toXsdString();
   String whereClause = "PROCESS_TEMPLATE.STATE =
                         PROCESS_TEMPLATE.STATE.STATE_STARTED AND
                         PROCESS_TEMPLATE.VALID_FROM =
                         (SELECT MAX(VALID_FROM) FROM PROCESS_TEMPLATE
                          WHERE NAME=PROCESS_TEMPLATE.NAME AND
                          VALID_FROM <= TS('" + now + "'))";

   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
                                          ( whereClause,
                                            "PROCESS_TEMPLATE.NAME",
                                            (Integer)null, (TimeZone)null);
   ```

   結果はプロセス・テンプレート名でソートされます。
2. プロセス・テンプレートのリストと、ユーザーが許可されている開始アクティビティーのリストを作成します。

   プロセス・テンプレートのリストには、単一の開始アクティビティーがあるプロセス・テンプレートが含まれます。これらのアクティビティーは、保護されていないか、またはログオン・ユーザーによる開始が許可されていないかのいずれかです。あるいは、少なくとも 1 つの開始アクティビティーにより開始できるプロセス・テンプレートを収集することもできます。

ヒント: プロセス管理者は、プロセス・インスタンスを開始することもできます。テンプレートの完全なリストを入手するには、プロセス・テンプレートと関連する管理タスク・テンプレートを読み取り、ログオン・ユーザーが管理者であるかを確認することも必要です。

```
List authorizedProcessTemplates = new ArrayList();
List authorizedActivityServiceTemplates = new ArrayList();
```

3. プロセス・テンプレートごとに、開始アクティビティーを判別します。

```
for( int i=0; i<processTemplates.length; i++ )
{
  ProcessTemplateData template = processTemplates[i];
  ActivityServiceTemplateData[] startActivities =
                process.getStartActivities(template.getID());
```

4. 開始アクティビティーごとに、関連したインライン呼び出しタスク・テンプレートの ID を取得します。

```
for( int j=0; j<startActivities.length; j++ )
{
  ActivityServiceTemplateData activity = startActivities[j];
  TKTID tktid = activity.getTaskTemplateID();
```

   a. 呼び出しタスク・テンプレートが存在しない場合、プロセス・テンプレートはこの開始アクティビティーによっては保護されません。

      この場合、この開始アクティビティーを使用して、誰でもプロセス・インスタンスを作成できます。

```
boolean isAuthorized = false;
       if ( tktid == null )
    {
      isAuthorized = true;
      authorizedActivityServiceTemplates.add(activity);
    }
```

   b. 呼び出しタスク・テンプレートが存在する場合は、Human Task Manager API を使用して、ログオン・ユーザーの許可を検査します。

      例ではログオン・ユーザーは Smith です。ログオン・ユーザーは、呼び出しタスクの潜在的なスターターまたは管理者である必要があります。

```
if ( tktid != null )
    {
      isAuthorized =
        task.isUserInRole
            (tkid, "Smith", WorkItem.REASON_POTENTIAL_STARTER) ||
        task.isUserInRole(tktid, "Smith", WorkItem.REASON_ADMINISTRATOR);

      if ( isAuthorized )
      {
        authorizedActivityServiceTemplates.add(activity);
      }
    }
```

      ユーザーに指定されたロールがある場合、またはそのロールの担当者割り当て基準が指定されていない場合、isUserInRole メソッドは値 true を戻します。

5. プロセス・テンプレート名だけを使用してプロセスが開始できるかを確認します。

```
if ( isAuthorized && startActivities.length == 1 )
    {
      authorizedProcessTemplates.add(template);
    }
```

6. ループを終了します。

```
  } // end of loop for each activity service template
}   // end of loop for each process template
```

ヒューマン・タスクを含む単一の個人ワークフローの処理

ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。この例では、一連の human task アクティビティー (予定タスク) として、書籍を注文するための一連のアクションを実装する方法を示しています。ワークフローの処理には、Business Flow Manager と Human Task Manager API の両方が使用されます。

このタスクについて

オンライン・ブックストアでは、購入者は一連の操作を完了することで本を注文します。この一連の操作は、human task アクティビティー (予定タスク) として実装できます。購入者が複数の書籍を注文する場合は、これが次の human task アクティビティーの要求に相当します。一連のタスクに関する情報は Business Flow Manager によって保守されますが、タスク自体は Human Task Manager によって保守されます。

この例を、Business Flow Manager API のみを使用する例と比較してください。

プロシージャー

1. Business Flow Manager API を使用して、作業対象となるプロセス・インスタンスを取得します。

   この例では、CustomerOrder プロセスのインスタンスです。

```
ProcessInstanceData processInstance =
   process.getProcessInstance("CustomerOrder");
String piid = processInstance.getID().toString();
```

2. Human Task Manager API を使用して、指定されたプロセス・インスタンスの一部である、準備のできた予定タスク (種類は参加) を照会します。

   タスクの包含コンテキスト ID を使用して、含まれているプロセス・インスタンスを指定します。単一の個人ワークフローの場合、照会によって、一連の human task アクティビティーの最初の human task アクティビティーに関連付けられている予定タスクが戻されます。

```
//
 // Query the list of to-do tasks that can be claimed by the logged-on user
 // for the specified process instance
//
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.CONTAINMENT_CTX_ID = ID('" + piid + "') AND
                 TASK.STATE = TASK.STATE.STATE_READY AND
                 TASK.KIND = TASK.KIND.KIND_PARTICIPATING AND
                 WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                (String)null, (Integer)null, (TimeZone)null);
```

3. 戻される予定タスクを要求します。

```
if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 ClientObjectWrapper input = task.claim(tkiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    taskInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

タスクが要求されると、タスクの入力メッセージが戻されます。

4. 予定タスクに関連付けられた human task アクティビティーを判別します。

   以下のメソッドのいずれかを使用して、アクティビティーをそのタスクに相関させます。

   - task.getActivityID メソッド:

     ```
     AIID aiid = task.getActivityID(tkiid);
     ```

   - タスク・オブジェクトの一部である親コンテキスト ID:

     ```
     AIID aiid = null;
     Task taskInstance = task.getTask(tkiid);

     OID oid = taskInstance.getParentContextID();
     if ( oid != null and oid instanceof AIID )
     {
       aiid = (AIID)oid;
     }
     ```

5. タスクでの作業が終了したら、Business Flow Manager API を使用してタスクとそれに関連する human task アクティビティーを完了し、プロセス・インスタンス内の次の human task アクティビティーを要求します。

   human task アクティビティーを完了するには、出力メッセージを渡します。出力メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

   ```
   ActivityInstanceData activity = process.getActivityInstance(aiid);
   ClientObjectWrapper output =
         process.createMessage(aiid, activity.getOutputMessageTypeName());
   DataObject myMessage = null ;
   if ( output.getObject()!= null && output.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)output.getObject();
     //set the parts in your message, for example, an order number
     myMessage.setInt("OrderNo", 4711);
   }

   //complete the human task activity and its associated to-do task,
   // and claim the next human task activity
   CompleteAndClaimSuccessorResult successor =
      process.completeAndClaimSuccessor(aiid, output);
   ```

   このアクションは、オーダー番号が含まれる出力メッセージを設定し、シーケンス内の次の human task アクティビティーを要求します。後続アクティビティーに AutoClaim が設定されており、有効なパスが複数存在する場合は、後続アクティビティーのすべてが要求され、ランダムなアクティビティーが次のアクティ

ビティーとして戻されます。このユーザーに割り当て可能な後続アクティビティーが他にない場合は、NULL が戻されます。

後に続くことができる並列パスがプロセスに含まれ、これらのパスに、ログオン・ユーザーが潜在的な所有者である human task アクティビティーが複数含まれる場合、ランダムなアクティビティーが自動的に要求され、次のアクティビティーとして戻されます。

6. 次の human task アクティビティーを処理します。

```
ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject()!=
               null && nextInput.getObject() instanceof DataObject )
{
  activityInput = (DataObject)input.getObject();
  // read the values
  ...
}

aiid = successor.getAIID();
```

7. ステップ 5 を続行して、human task アクティビティーを完了し、次の human task アクティビティーを取得します。

関連タスク

80 ページの『単独ユーザー・ワークフローの処理』
ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。このタイプのワークフローには、並列パスは存在しません。 completeAndClaimSuccessor API は、このタイプのワークフローの処理をサポートします。

例外および障害の処理

BPEL プロセスでは、プロセス内のさまざまな箇所で障害が発生する可能性があります。

このタスクについて

Business Process Execution Language (BPEL) の障害には次の発生原因があります。

- Web サービスの呼び出し (Web Services Description Language (WSDL) の障害)
- throw アクティビティー
- Business Process Choreographer によって認識される BPEL 標準障害

これらの障害を処理する機構が存在します。プロセス・インスタンスによって生成される障害を処理するには、以下の手段のいずれかを使用します。

- 対応する障害ハンドラーへの制御の引き渡し
- プロセス内の前の処理の補正
- プロセスの停止と状態の修復の依頼 (強制再試行、強制完了)

BPEL プロセスは、プロセスによって指定される操作の呼び出し元に障害を戻す可能性もあります。プロセス内の障害は、障害名と障害データを指定して reply アクティビティーとしてモデル化できます。これらの障害は、チェック例外として API 呼び出し元に戻されます。

BPEL プロセスで BPEL 障害を処理しない場合、または API 例外が発生した場合は、ランタイム例外が API 呼び出し元に戻されます。API 例外の例には、インスタンスの作成元となるプロセス・モデルが存在しない場合などがあります。

障害および例外の処理は、以下のタスクで説明します。

API 例外の処理

このタスクについて

BusinessFlowManagerService インターフェースまたは HumanTaskManagerService インターフェースのメソッドが正常に完了しない場合、エラーの原因を示す例外がスローされます。この例外を特別に処理して、呼び出し元にガイダンスを提供することができます。

ただし、例外のサブセットのみを特別に処理し、その他の潜在的な例外に対しては汎用のガイダンスを提供するのが一般的です。すべての固有の例外は、汎用の ProcessException または TaskException から継承しています。最終の `catch(ProcessException)` または `catch(TaskException)` ステートメントを使用して汎用の例外を catch することが*最良実例* です。このステートメントでは、発生する可能性があるその他すべての例外を考慮に入れるため、このステートメントによって、ご使用のアプリケーション・プログラムの上位互換性を確保することができます。

アクティビティーに設定された障害の検査

プロシージャー

1. 失敗状態または停止状態のタスク・アクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "(ACTIVITY.STATE = ACTIVITY.STATE.STATE_FAILED OR
                     ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED) AND
                     ACTIVITY.KIND=ACTIVITY.KIND.KIND_STAFF",
                     (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、失敗または停止のアクティビティーが含まれる照会結果セットを戻します。

2. 障害の名前を読み取ります。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ClientObjectWrapper faultMessage = process.getFaultMessage(aiid);
  DataObject fault = null ;
  if ( faultMessage.getObject() != null && faultMessage.getObject()
       instanceof DataObject )
  {
    fault = (DataObject)faultMessage.getObject();
    Type type = fault.getType();
    String name = type.getName();
    String uri = type.getURI();
  }
}
```

   これは、障害名を戻します。また、障害名を取得する代わりに、停止アクティビティーの未処理の例外を分析することもできます。

停止した invoke アクティビティーで発生した障害の検査

このタスクについて

アクティビティーで障害が発生した場合、障害タイプによってそのアクティビティーの修復のために実行できるアクションが決まります。

プロシージャー

1. 停止状態の human task アクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    ACTIVITY.KIND=ACTIVITY.KIND.KIND_INVOKE",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、停止された invoke アクティビティーが含まれる照会結果セットを戻します。

2. 障害の名前を読み取ります。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);

  ProcessException excp = activity.getUnhandledException();
  if ( excp instanceof ApplicationFaultException )
  {
   ApplicationFaultException fault = (ApplicationFaultException)excp;
   String faultName = fault.getFaultName();
  }
}
```

Web サービス API クライアント・アプリケーションの開発

Web サービス API を介してビジネス・プロセス・アプリケーションとヒューマン・タスク・アプリケーションにアクセスするクライアント・アプリケーションを開発できます。

このタスクについて

クライアント・アプリケーションは、Java Web サービスや Microsoft .NET などのすべての Web サービス・クライアント環境で開発できます。

概要: Web サービス

Web サービスは、クライアント・アプリケーションとデータを交換するために、オープンな XML ベース標準およびトランスポート・プロトコルを使用する Web ベースのエンタープライズ・アプリケーションです。 Web サービスにより、言語および環境に中立なプログラミング・モデルを使用することができます。

Web サービスでは、次のコア・テクノロジーを使用します。

- XML (Extensible Markup Language)。 XML は、データ独立性の問題を解決します。これを使用してデータを記述し、さらにアプリケーションまたはプログラミング言語との間でデータをマップします。

- WSDL (Web サービス記述言語)。この XML ベースの言語を使用して、基礎となるアプリケーションの記述を作成します。この記述は、基礎となるアプリケーションとその他の Web 対応アプリケーションとの間のインターフェースとして機能することにより、アプリケーションを Web サービスに変換します。
- SOAP (Simple Object Access Protocol)。SOAP は、Web 用のコアとなる通信プロトコルで、ほとんどの Web サービスはこのプロトコルを使用して相互に対話します。

Web サービス・コンポーネントおよび一連の制御

多くのクライアント・サイドおよびサーバー・サイドのコンポーネントは、 Web サービスの要求と応答を表す一連の制御に関与します。

標準的な一連の制御は以下のとおりです。

1. クライアント・サイド:
   a. クライアント・アプリケーション (ユーザーによって提供される) は、Web サービスの要求を発行します。
   b. プロキシー・クライアント (ユーザーによっても提供されるが、クライアント・サイド・ユーティリティーを使用して自動的に生成することが可能) は、サービス要求を SOAP 要求エンベロープでラップします。
   c. クライアント・サイド開発インフラストラクチャーは、Web サービスのエンドポイントとして定義された URL に要求を転送します。
2. ネットワークは、HTTP または HTTPS を使用して Web サービス・エンドポイントに要求を送信します。
3. サーバー・サイド:
   a. 汎用 Web サービス API は、要求を受信し、デコードします。
   b. 要求は、汎用の Business Flow Manager または Human Task Manager コンポーネントによって直接処理されるか、指定されたビジネス・プロセスまたはヒューマン・タスクに転送されます。
   c. 戻されたデータは SOAP 応答エンベロープでラップされます。
4. ネットワークは、HTTP または HTTPS を使用してクライアント・サイド環境に応答を送信します。
5. クライアント・サイドに戻る:
   a. クライアント・サイド開発インフラストラクチャーは、SOAP 応答エンベロープをアンラップします。
   b. プロキシー・クライアントはデータを SOAP 応答から抽出して、それをクライアント・アプリケーションに受け渡します。
   c. クライアント・アプリケーションは、必要に応じて、戻されたデータを処理します。

Web サービス API の概要

Web サービス API により、Business Process Choreographer 環境で実行しているビジネス・プロセスおよびヒューマン・タスクに Web サービスを使用してアクセスするクライアント・アプリケーションを開発できます。

Business Process Choreographer Web サービス API は、次の 2 つの別々の Web サービス・インターフェース (WSDL ポート・タイプ) を提供します。

- Business Flow Manager API。クライアント・アプリケーションが microflow および長期間のプロセスと対話できるようにします。例えば、以下の操作を実行できます。
  - プロセス・テンプレートとプロセス・インスタンスを作成
  - 既存のプロセスの要求
  - ID によるプロセスの照会

  実行可能なアクションの詳細なリストについては、 68 ページの『ビジネス・プロセス用のアプリケーションの開発』を参照してください。
- Human Task Manager API。クライアント・アプリケーションは以下の操作を実行できます。
  - タスクの作成と開始
  - 既存のタスクの要求
  - タスクの完了
  - ID によるタスクの照会
  - タスクの集合の照会

  実行可能なアクションの詳細なリストについては、 91 ページの『ヒューマン・タスク用のアプリケーションの開発』を参照してください。

クライアント・アプリケーションは、これらの Web サービス・インターフェースの一方または両方を使用できます。

例

Human Task Manager Web サービス API にアクセスして参加ヒューマン・タスクを処理するクライアント・アプリケーションの場合に考えられる処理の概要を次に示します。

1. クライアント・アプリケーションは、query Web サービス呼び出しを WebSphere Process Server に対して発行します。これにより、ユーザーによって処理される参加タスクのリストを要求します。
2. 参加タスクのリストが、SOAP/HTTP 応答エンベロープで戻されます。
3. クライアント・アプリケーションは、claim Web サービス呼び出しを発行して、いずれかの参加タスクを要求します。
4. WebSphere Process Server は、タスクの入力メッセージを戻します。
5. クライアント・アプリケーションは、complete Web サービス呼び出しを発行して、出力メッセージまたは障害メッセージのあるタスクを完了します。

ビジネス・プロセスとヒューマン・タスクの要件

WebSphere Integration Developer により Business Process Choreographer で実行するために開発されるビジネス・プロセスとヒューマン・タスクは、Web サービス API を介してアクセスできるようにするために特定の規則に準拠する必要があります。

要件は以下のとおりです。

1. ビジネス・プロセスとヒューマン・タスクのインターフェースは、Java API for XML-based RPC (JAX-RPC 1.1) 仕様で定義される「document/literal wrapped」スタイルを使用して定義する必要があります。これは、WID によって開発されるすべてのビジネス・プロセスとヒューマン・タスクのデフォルト・スタイルです。
2. Web サービス操作のビジネス・プロセスとヒューマン・タスクによって公開される障害メッセージは、XML スキーマ・エレメントにより定義される単一の WSDL メッセージ部を構成する必要があります。以下に例を示します。

   ```
   <wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
   ```

   関連情報

   Java API for XML based RPC (JAX-RPC) のダウンロード・ページ

   Which style of WSDL should I use?

クライアント・アプリケーションの開発

クライアント・アプリケーションの開発プロセスは多数のステップで構成されています。

プロシージャー

1. Business Flow Manager API、Human Task Manager API、またはその両方の中から、クライアント・アプリケーションで使用する必要がある Web サービス API を決定します。
2. WebSphere Process Server 環境から必要なファイルをエクスポートします。 あるいは、WebSphere Process Server クライアント CD からファイルをコピーすることもできます。
3. 選択したクライアント・アプリケーション開発環境で、エクスポートした成果物を使用してプロキシー・クライアントを生成します。
4. オプション: ヘルパー・クラス を生成します。 ヘルパー・クラスは、クライアント・アプリケーションが WebSphere サーバーの具象プロセスまたはタスクと直接に対話する場合に必要とされます。ただし、クライアント・アプリケーションが汎用タスク (照会の発行など) を実行するだけの場合は、ヘルパー・クラスは不要です。
5. クライアント・アプリケーション用のコードを開発します。
6. 必要なセキュリティー・メカニズムをクライアント・アプリケーションに追加します。

成果物のコピー

クライアント・アプリケーションを作成しやすくするには、WebSphere 環境から数多くの成果物をコピーする必要があります。

これらの成果物は、次の 2 つの方法で入手できます。

- 成果物を WebSphere Process Server 環境で公開してエクスポートします。
- WebSphere Process Server クライアント CD からファイルをコピーします。

サーバー環境での成果物の公開とエクスポート

クライアント・アプリケーションを開発して Web サービス API にアクセスするには、WebSphere サーバー環境であらかじめ数多くの成果物を公開し、エクスポートしておく必要があります。

このタスクについて

エクスポートする成果物は以下のとおりです。

- Web サービス API を構成するポート・タイプと操作について記述した Web Service Definition Language (WSDL) ファイル。
- WSDL ファイル内のサービスとメソッドによって参照されるデータ型の定義を記述した XML Schema Definition (XSD) ファイル。
- ビジネス・オブジェクトを記述した追加の WSDL ファイルと XSD ファイル。ビジネス・オブジェクトは、WebSphere サーバーで実行される具象ビジネス・プロセスまたはヒューマン・タスクについて記述します。これらの追加ファイルは、クライアント・アプリケーションが Web サービス API を介して具象ビジネス・プロセスまたはヒューマン・タスクと直接に対話する必要がある場合にのみ必要とされます。クライアント・アプリケーションが、照会の発行などの汎用タスクを実行するだけの場合は不要です。

これらの成果物を公開した後は、それらの成果物をクライアント・プログラミング環境にコピーし、プロキシー・クライアントおよびヘルパー・クラスの生成で使用できるようにする必要があります。

Web サービス・エンドポイント・アドレスの指定:

Web サービス・エンドポイント・アドレスは、クライアント・アプリケーションが Web サービス API にアクセスするために指定する必要のある URL です。エンドポイント・アドレスは、クライアント・アプリケーション用のプロキシー・クライアントを生成するためにエクスポートする WSDL ファイルに書き込まれます。

このタスクについて

使用する Web サービス・エンドポイント・アドレスは、WebSphere サーバーの構成によって異なります。

- シナリオ 1。単一の WebSphere サーバー。指定する WebSphere エンドポイント・アドレスは、サーバーのホスト名とポート番号です (例: **host1:9080**)。
- シナリオ 2。複数のサーバーで構成される WebSphere クラスターの場合。指定する WebSphere エンドポイント・アドレスは、Web サービス API をホスティングするサーバーのホスト名とポートです (例: **host2:9081**)。
- シナリオ 3。Web サーバーをフロントエンドとして使用する場合。指定する WebSphere エンドポイント・アドレスは、Web サーバーのホスト名とポートです (例: **host:80**)。

デフォルトでは、Web サービス・エンドポイント・アドレスの形式は *protocol*:**//***host*:*port*/*context_root*/*fixed_path* です。各部の意味は、次のとおりです。

- *protocol*。クライアント・アプリケーションと WebSphere サーバー間で使用される通信プロトコル。デフォルト・プロトコルは HTTP です。代わりに、より安全な HTTPS (HTTP over SSL) プロトコルを使用する方法を選択できます。HTTPS の使用をお勧めします。
- *host:port*。Web サービス API をホスティングするマシンへのアクセスに使用するホスト名とポート番号。この値は、クライアント・アプリケーションがアプリケーションに直接アクセスするか、Web サーバー・フロントエンド経由でアクセスするかなど、場合によって異なります。
- *context_root*。「コンテキスト・ルート」には、任意の値を選択できます。ただし、選択する値は個々の WebSphere セル内で固有でなければなりません。デフォルト値は、「node_server/cluster」サフィックスを使用して、ネーミング競合のリスクを回避しています。
- *fixed_path* は、`/sca/com/ibm/bpe/api/BFMWS` (Business Flow Manager API の場合) または `/sca/com/ibm/task/api/HTMWS` (Human Task Manager API の場合) で、変更できません。

Web サービス・エンドポイント・アドレスは、最初は、ビジネス・プロセス・コンテナーまたはヒューマン・タスク・コンテナーの構成時に指定されます。

プロシージャー

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」を選択します。

   注: 「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」を選択して、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから、「**BPEContainer**」(ビジネス・プロセス・コンテナーの場合) または「**TaskContainer**」(ヒューマン・タスク・コンテナーの場合) を選択します。
4. 「**追加プロパティー**」リストから、「**HTTP エンドポイント URL 情報を提供 (Provide HTTP endpoint URL information)**」を選択します。
5. リストからデフォルトのプレフィックスのいずれか 1 つを選択するか、カスタム・プレフィックスを入力します。 クライアント・アプリケーションを、Web サービス API をホスティングするアプリケーション・サーバーに直接接続する場合は、デフォルト・プレフィックス・リストのプレフィックスを使用します。そうでない場合は、カスタム・プレフィックスを指定します。
6. 「**適用**」をクリックして、選択したプレフィックスを SCA モジュールにコピーします。
7. 「**OK**」をクリックします。 URL 情報がワークスペースに保管されます。

結果

管理コンソールで現行値を表示できます (例えばビジネス・プロセス・コンテナーの場合は、「**エンタープライズ・アプリケーション**」 → 「**BPEContainer**」 → 「**デプロイメント記述子の表示**」)。

エクスポートされた WSDL ファイルでは、soap:address エレメントの location 属性に、指定した Web サービス・エンドポイント・アドレスが含まれています。以下に例を示します。

```
<wsdl:service name="BFMWSService">
  <wsdl:port name="BFMWSPort" binding="this:BFMWSBinding">
   <soap:address location=
         "https://myserver:9080/WebServicesAPIs/sca/com/ibm/bpe/api/BFMWS"/>
```

関連概念

137 ページの『セキュリティーの追加 (Java Web サービス)』
Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことによって保護する必要があります。

関連タスク

147 ページの『セキュリティーの追加 (.NET)』
Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことにより保護できます。

WSDL ファイルの公開:

Web サービス記述言語 (WSDL) ファイルには、Web サービス API で使用できるすべての操作の詳細な説明が含まれています。Business Flow Manager と Human Task Manager の Web サービス API では、別々の WSDL ファイルを使用できます。これらの WSDL ファイルは、まず公開して、その後 WebSphere 環境からご使用の開発環境にコピーし、プロキシー・クライアントの生成に使用することになります。

始める前に

WSDL ファイルを公開する前に、指定した Web サービス・エンドポイント・アドレスが正しいことを確認してください。このアドレスは、クライアント・アプリケーションが Web サービス API へのアクセスに使用する URL です。

このタスクについて

WSDL ファイルの公開が必要なのは一度だけです。

注: WebSphere Process Server クライアント CD を所有している場合は、代わりに、その CD からクライアント・プログラミング環境に直接ファイルをコピーすることもできます。

ビジネス・プロセス WSDL の公開:

管理コンソールを使用して WSDL ファイルを公開します。

プロシージャー

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」を選択します。

   注: 「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」を選択して、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。

3. SCA モジュールまたはアプリケーションのリストから **BPEContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の ZIP ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

結果

エクスポートされる ZIP ファイルは、BPEContainer_WSDLFiles.zip と命名されます。 ZIP ファイルには、Web サービスを記述した WSDL ファイルと、WSDL ファイル内から参照されるすべての XSD ファイルが含まれています。

ヒューマン・タスク WSDL の公開:

管理コンソールを使用して WSDL ファイルを公開します。

プロシージャー

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」を選択します。

   注: 「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」を選択して、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから **TaskContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の ZIP ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

結果

エクスポートされる ZIP ファイルは、TaskContainer_WSDLFiles.zip と命名されます。 ZIP ファイルには、Web サービスを記述した WSDL ファイルと、WSDL ファイル内から参照されるすべての XSD ファイルが含まれています。

ビジネス・オブジェクトのエクスポート:

ビジネス・プロセスおよびヒューマン・タスクには、Web サービスとして外部にアクセスできるよう明確に定義されたインターフェースが用意されています。これらのインターフェースがビジネス・オブジェクトを参照する場合は、インターフェース定義とビジネス・オブジェクトをクライアント・プログラミング環境にエクスポートする必要があります。

このタスクについて

この手順は、クライアント・アプリケーションが対話する必要があるビジネス・オブジェクトごとに、繰り返してください。

WebSphere Process Server では、ビジネス・オブジェクトが、ビジネス・プロセスまたはヒューマン・タスクと対話する要求、応答、および障害メッセージの形式を定義します。これらのメッセージには、複合データ型の定義が含まれている場合もあります。

例えば、ヒューマン・タスクを作成して開始するには、以下の情報項目をタスク・インターフェースに渡す必要があります。

- タスク・テンプレート名
- タスク・テンプレート・ネームスペース
- 入力メッセージ (フォーマット済みのビジネス・データを含む)
- 応答メッセージを戻すための応答ラッパー
- 障害および例外を戻すための障害メッセージ

以上の項目は、単一のビジネス・オブジェクト内でカプセル化されます。Web サービス・インターフェースのすべての操作は、「document/literal wrapped」操作としてモデル化されます。これらの操作の入出力パラメーターは、ラッパー文書の中にカプセル化されます。他のビジネス・オブジェクトは、それに対応する応答および障害のメッセージ形式を定義します。

Web サービスによってビジネス・プロセスまたはヒューマン・タスクを作成および開始するためには、クライアント・サイドのクライアント・アプリケーションがこれらのラッパー・オブジェクトを使用できるようにする必要があります。

そのためには、WebSphere 環境からビジネス・オブジェクトを Web サービス記述言語 (WSDL) ファイルおよび XML スキーマ定義 (XSD) ファイルとしてエクスポートし、データ型定義をクライアント・プログラミング環境にインポートしてから、それをクライアント・アプリケーションで使用できるヘルパー・クラスに変換します。

プロシージャー

1. WebSphere Integration Developer ワークスペースが稼働していない場合は、起動します。
2. エクスポートするビジネス・オブジェクトを含むライブラリー・モジュールを選択します。 ライブラリー・モジュールは、必要なビジネス・オブジェクトを含む圧縮ファイルです。
3. ライブラリー・モジュールをエクスポートします。
4. エクスポートしたファイルをクライアント・アプリケーション開発環境にコピーします。

例

ビジネス・プロセスが以下の Web サービス操作を公開するとします。

```
<wsdl:operation name="updateCustomer">
    <wsdl:input message="tns:updateCustomerRequestMsg"
                name="updateCustomerRequest"/>
    <wsdl:output message="tns:updateCustomerResponseMsg"
                 name="updateCustomerResponse"/>
    <wsdl:fault message="tns:updateCustomerFaultMsg"
                name="updateCustomerFault"/>
  </wsdl:operation>
```

公開は、以下のように定義された WSDL メッセージを介して行われるとします。

```
<wsdl:message name="updateCustomerRequestMsg">
    <wsdl:part element="types:updateCustomer"
               name="updateCustomerParameters"/>
  </wsdl:message>
  <wsdl:message name="updateCustomerResponseMsg">
    <wsdl:part element="types:updateCustomerResponse"
               name="updateCustomerResult"/>
  </wsdl:message>
  <wsdl:message name="updateCustomerFaultMsg">
    <wsdl:part element="types:updateCustomerFault"
               name="updateCustomerFault"/>
  </wsdl:message>
```

具象 ユーザー定義エレメント「types:updateCustomer」、「types:updateCustomerResponse」、および「types:updateCustomerFault」は、クライアント・アプリケーションが実行するすべての*汎用* 命令 (call、sendMessage など) で、<xsd:any> パラメーターを使用して Web サービス API に渡したり Web サービス API から受け取ったりする必要があります。これらのユーザー定義エレメントは、エクスポートされた XSD ファイルから生成されるヘルパー・クラスを使用して、クライアント・アプリケーション上で作成、直列化、および非直列化されます。

関連タスク

143 ページの『BPEL プロセス用ヘルパー・クラスの作成 (.NET)』
特定の Web サービス API 操作では、クライアント・アプリケーションが「document/literal」スタイルのラップ・エレメントを使用する必要があります。クライアント・アプリケーションは、 ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

クライアント CD でのファイルの使用

WebSphere サーバー環境から成果物をエクスポートする代わりに、クライアント・アプリケーションの生成に必要なファイルを WebSphere Process Server クライアント CD からコピーすることができます。

この場合は、Business Flow Manager API または Human Task Manager API のデフォルトの Web サービス・エンドポイント・アドレスを手動で変更する必要があります。

クライアント・アプリケーションが両方の API にアクセスする場合は、両方の API のデフォルト・エンドポイント・アドレスを編集する必要があります。

クライアント CD からのファイルのコピー:

Web サービス API へのアクセスに必要なファイルは、WebSphere Process Server クライアント CD に収録されています。

プロシージャー

1. クライアント CD にアクセスして、ProcessChoreographer¥client ディレクトリーを参照します。
2. 必要なファイルをクライアント・アプリケーション開発環境にコピーします。

   Business Flow Manager API の場合は、次のファイルをコピーします。

   BFMWS.wsdl
   Business Flow Manager Web サービス API で使用可能な Web サービスが記述されています。このファイルには、エンドポイントのアドレスが含まれています。

   BFMIF.wsdl
   Business Flow Manager Web サービス API の各 Web サービスのパラメーターとデータ型が記述されています。

   BFMIF.xsd
   Business Flow Manager Web サービス API で使用されるデータ型が記述されています。

   BPCGEN.xsd
   Business Flow Manager と Human Task Manager の Web サービス API の間で共通のデータ型が記述されています。

   Human Task Manager API の場合は、次のファイルをコピーします。

   HTMWS.wsdl
   Human Task Manager Web サービス API で使用可能な Web サービスが記述されています。このファイルには、エンドポイントのアドレスが含まれています。

   HTMIF.wsdl
   Human Task Manager Web サービス API の各 Web サービスのパラメーターとデータ型が記述されています。

   HTMIF.xsd
   Human Task Manager Web サービス API で使用されるデータ型が記述されています。

   BPCGEN.xsd
   Business Flow Manager と Human Task Manager の Web サービス API の間で共通のデータ型が記述されています。

   注: BPCGen.xsd ファイルは、両方の API で共通です。

各ファイルをコピーした後は、BFMWS.wsdl または HTMWS.wsdl ファイルの Web サービス API のエンドポイント・アドレスを、Web サービス API をホストする WebSphere アプリケーション・サーバーのエンドポイント・アドレスに手動で変更する必要があります。

Web サービス・エンドポイント・アドレスの手動変更:

クライアント CD からファイルをコピーする場合は、WSDL ファイルで指定されるデフォルトの Web サービスのエンドポイント・アドレスを、Web サービス API をホスティングするサーバーのエンドポイント・アドレスに変更する必要があります。

このタスクについて

管理コンソールを使用すると、WSDL ファイルをエクスポートする前に Web サービス・エンドポイント・アドレスを設定することができます。ただし、WSDL ファイルを WebSphere Process Server クライアント CD からコピーする場合は、デフォルトの Web サービス・エンドポイント・アドレスを手動で変更する必要があります。

使用する Web サービス・エンドポイント・アドレスは、WebSphere サーバーの構成によって異なります。

- シナリオ 1。WebSphere サーバーが 1 台だけの場合。指定する WebSphere エンドポイント・アドレスは、サーバーのホスト名とポート番号です (例: **host1:9080**)。
- シナリオ 2。複数のサーバーで構成される WebSphere クラスターの場合。指定する WebSphere エンドポイント・アドレスは、Web サービス API をホスティングするサーバーのホスト名とポートです (例: **host2:9081**)。
- シナリオ 3。Web サーバーをフロントエンドとして使用する場合。指定する WebSphere エンドポイント・アドレスは、Web サーバーのホスト名とポートです (例: **host:80**)。

Business Flow Manager API エンドポイントの変更:

Business Flow Manager API ファイルを WebSphere Process Server クライアント CD からコピーする場合は、デフォルトのエンドポイント・アドレスを手動で編集する必要があります。

プロシージャー

1. クライアント CD からコピーしたファイルが格納されているディレクトリーにナビゲートします。
2. テキスト・エディターか XML エディターで BFMWS.wsdl ファイルを開きます。
3. `soap:address` エレメントを見つけます (ファイル下部)。
4. `location` 属性の値を、Web サービス API が実行されているサーバーの HTTP URL に変更します。 変更するには、次のステップを実行します。
   a. オプションとして `http` を `https` に置き換え、より安全な HTTPS プロトコルを使用できます。
   b. *localhost* を、Web サービス API サーバーのエンドポイント・アドレスのホスト名または IP アドレスに置き換えます。
   c. *9080* を、アプリケーション・サーバーのポート番号に置き換えます。
   d. *BPEContainer_N1_server1* を、Web サービス API を実行しているアプリケーションのコンテキスト・ルートで置換します。 デフォルトのコンテキスト・ルートは、次のエレメントで構成されています。

- *BPEContainer*。アプリケーション名。
- *N1*。ノード名。
- *server1*。サーバー名。

e. URL の固定部 (/sca/com/ibm/bpe/api/BFMWS) は変更しないでください。

例えば、アプリケーションがサーバー **s1.n1.ibm.com** で実行されており、そのサーバーが **9080** ポートで SOAP/HTTP 要求を受け入れている場合は、soap:address エレメントを次のように変更します。

```
<soap:address location="http://si.n1.ibm.com:9080/
                BPEContainer_N1_server1/sca/com/ibm/bpe/api/BFMWS"/>
```

関連概念

137 ページの『セキュリティーの追加 (Java Web サービス)』
Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことによって保護する必要があります。

関連タスク

147 ページの『セキュリティーの追加 (.NET)』
Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことにより保護できます。

Human Task Manager API エンドポイントの変更:

Human Task Manager API ファイルを WebSphere Process Server クライアントCD からコピーする場合は、デフォルトのエンドポイント・アドレスを手動で編集する必要があります。

プロシージャー

1. クライアント CD からコピーしたファイルが格納されているディレクトリーにナビゲートします。
2. テキスト・エディターか XML エディターで HTMWS.wsdl ファイルを開きます。
3. soap:address エレメントを見つけます (ファイル下部)。
4. location 属性の値を、正しいエンドポイント・アドレスに変更します。 変更するには、次のステップを実行します。
   a. オプションとして http を https に置き換え、より安全な HTTPS プロトコルを使用できます。
   b. *localhost* を、Web サービス API サーバーのエンドポイント・アドレスのホスト名または IP アドレスに置き換えます。
   c. *9080* を、アプリケーション・サーバーのポート番号に置き換えます。
   d. *HTMContainer_N1_server1* を、Web サービス API を実行しているアプリケーションのコンテキスト・ルートで置換します。 デフォルトのコンテキスト・ルートは、次のエレメントで構成されています。
      - *HTMContainer*。アプリケーション名。
      - *N1*。ノード名。
      - *server1*。サーバー名。
   e. URL の固定部 (/sca/com/ibm/task/api/HTMWS) は変更しないでください。

例えば、アプリケーションが サーバー **s1.n1.ibm.com** で実行されており、そのサーバーが **9081** ポートで SOAP/HTTPS 要求を受け入れている場合は、soap:address エレメントを次のように変更します。

```
<soap:address location="https://si.n1.ibm.com:9081/
                  HTMContainer_N1_server1/sca/com/ibm/task/api/HTMWS"/>
```

関連概念

137 ページの『セキュリティーの追加 (Java Web サービス)』
Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことによって保護する必要があります。

関連タスク

147 ページの『セキュリティーの追加 (.NET)』
Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことにより保護できます。

Java Web サービス環境でのクライアント・アプリケーションの開発

Java Web サービスと互換性のある Java ベースの開発環境を使用して、Web サービス API 用のクライアント・アプリケーションを開発できます。

プロキシー・クライアントの生成 (Java Web サービス)

Java Web サービス・クライアント・アプリケーションは、*プロキシー・クライアント* を使用して Web サービス API と対話します。

このタスクについて

Java Web サービスのプロキシー・クライアントには、Web サービス要求を実行するためにクライアント・アプリケーションが呼び出す Java Bean クラスが数多く含まれています。プロキシー・クライアントは、サービス・パラメーターを SOAP メッセージにアセンブルし、SOAP メッセージを HTTP 経由で Web サービスに送信し、Web サービスから応答を受信し、戻されたデータをクライアント・アプリケーションに引き渡します。

したがって、基本的にプロキシー・クライアントによってクライアント・アプリケーションは、ローカル機能のようにして Web サービスを呼び出すことができます。

注: プロキシー・クライアントの生成が必要なのは一度だけです。その後、同じ Web サービス API にアクセスするすべてのクライアント・アプリケーションは、同じプロキシー・クライアントを使用できます。

IBM® Web サービス環境では、次の 2 つの方法でプロキシー・クライアントを生成します。

- Rational® Application Developer または WebSphere Integration Developer が統合された開発環境を使用する。
- WSDL2Java コマンド行ツールを使用する。

その他の Java Web サービス開発環境には通常、WSDL2Java ツールまたは独自のクライアント・アプリケーション生成機能が組み込まれています。

Rational Application Developer によるプロキシー・クライアントの生成:

Rational Application Developer の統合された開発環境により、クライアント・アプリケーションのプロキシー・クライアントを生成できます。

始める前に

プロキシー・クライアントを生成するには、その前に、ビジネス・プロセスまたはヒューマン・タスクの Web サービス・インターフェースを記述した WSDL ファイルを WebSphere 環境 (または WebSphere Process Server クライアント CD) からエクスポートし、それをクライアントのプログラミング環境にコピーしておく必要があります。

プロシージャー

1. 該当する WSDL ファイルをプロジェクトに追加します。
   - ビジネス・プロセスの場合:
     a. エクスポート・ファイル BPEContainer_*nodename_servername*_WSDLFiles.zip を一時ディレクトリーに unzip します。
     b. サブディレクトリー META-INF を、unzip されたディレクトリー BPEContainer_*nodename_servername*.ear/b.jar からインポートします。
   - ヒューマン・タスクの場合:
     a. エクスポート・ファイル TaskContainer_*nodename_servername*_WSDLFiles.zip を一時ディレクトリーに unzip します。
     b. サブディレクトリー META-INF を、unzip されたディレクトリー TaskContainer_*nodename_servername*.ear/h.jar からインポートします。

   新規ディレクトリー wsdl およびサブディレクトリー構造がプロジェクト内に作成されます。
2. Web サービス・ウィザード・プロパティーを変更します。
   a. Rational Application Developer で、「**設定 (Preferences)**」 → 「**Web サービス**」 → 「**コード生成 (Code generation)**」 → 「**IBM WebSphere ランタイム**」を選択します。
   b. 「**ラップ・スタイルを使用せずに WSDL から Java を生成 (Generate Java from WSDL using the no wrapped style)**」オプションを選択します。

   注: 「**設定**」メニューで「**Web サービス**」オプションを選択できない場合は、まず「**ウィンドウ**」 → 「**設定**」 → 「**ワークベンチ**」 → 「**機能**」と進んで、必要な機能を有効にする必要があります。「**Web サービス開発者 (Web Service Developer)**」をクリックして、「**OK**」をクリックします。次いで「設定」ウィンドウを再び開き、「**コード生成**」オプションを変更します。
3. 新たに作成された wsdl ディレクトリー内にある BFMWS.WSDL または HTMWS.WSDL ファイルを選択します。
4. 右クリックして、「**Web サービス**」 → 「**クライアントを生成**」を選択します。

   残りのステップを続行する前に、サーバーが開始していることを確認します。

5. 「Web サービス」ウィンドウで「**次へ**」をクリックして、デフォルトをすべて受け入れます。
6. 「Web サービス選択」ウィンドウで「**次へ**」をクリックして、デフォルトをすべて受け入れます。
7. 「クライアント環境構成」ウィンドウで、以下のようにします。
   a. 「**編集**」をクリックして、「Web サービス・ランタイム」オプションを IBM WebSphere に変更します。
   b. 「J2EE のバージョン」オプションを 1.4 に変更します。
   c. 「**OK**」をクリックします。
   d. 「**次へ**」をクリックします。
8. このステップは、ビジネス・プロセス Web サービス API とヒューマン・タスク Web サービス API の両方を含む Web サービス・クライアントを生成する必要がある場合にのみ必要です。なぜなら、両方の WSDL ファイルに重複するメソッドがあるからです。
   a. 「Web サービス・プロキシー」ウィンドウで、「名前空間からパッケージへのカスタム・マッピングを定義」を選択して、「**OK**」をクリックします。
   b. 「Web サービス・クライアントの名前空間からパッケージへのマッピング」ウィンドウで、以下のネーム・スペースおよびパッケージを追加します。

      BFMWS.wsdl の場合:

ネーム・スペース	パッケージ
http://www.ibm.com/xmlns/prod/websphere/business-process/types/6.0	com.ibm.sca.bpe
http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0	com.ibm.sca.bpe
http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0/Binding	com.ibm.sca.bpe
http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/6.0	com.ibm.sca.bpe

      HTMWS.wsdl の場合:

ネーム・スペース	パッケージ
http://www.ibm.com/xmlns/prod/websphere/human-task/types/6.0	com.ibm.sca.task
http://www.ibm.com/xmlns/prod/websphere/human-task/services/6.0	com.ibm.sca.task
http://www.ibm.com/xmlns/prod/websphere/human-task/services/6.0/Binding	com.ibm.sca.task
http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/6.0	com.ibm.sca.task

      上書きを確認するように求められる場合、「**YesToAll**」をクリックします。
9. 「**終了**」をクリックします。

結果

複数のプロキシー、ロケーター、ヘルパー Java クラスで構成されるプロキシー・クライアントが生成され、プロジェクトに追加されます。デプロイメント記述子も更新されます。

WSDL2Java によるプロキシー・クライアントの生成:

WSDL2Java は、プロキシー・クライアントを生成するコマンド行ツールです。プロキシー・クライアントによって、クライアント・アプリケーションのプログラミングが容易になります。

始める前に

プロキシー・クライアントを生成するには、その前に、ビジネス・プロセスまたはヒューマン・タスクの Web サービス API を記述した WSDL ファイルを WebSphere 環境 (または WebSphere Process Server クライアント CD) からエクスポートし、それをクライアントのプログラミング環境にコピーしておく必要があります。

このタスクについて

プロシージャー

1. WSDL2Java ツールを使用してプロキシー・クライアントを生成します。
   タイプ:

   wsdl2java *options WSDLfilepath*

   各部の意味は、次のとおりです。

   - *オプション* は以下のとおりです。

     -noWrappedOperations (-w)
     ラップ操作の検出を使用不可にします。要求および応答メッセージ用の Java Bean が生成されます。

     注: これはデフォルト値ではありません。

     -role (-r)
     値 **client** を指定すると、クライアント・サイド開発用のファイルおよびバインディング・ファイルが生成されます。

     -container (-c)
     使用するクライアント・サイド・コンテナー。有効な引数は以下のとおりです。

     client クライアント・コンテナー

     ejb Enterprise JavaBeans (EJB) コンテナー。

     none コンテナーなし

     web Web コンテナー

     -output (-o)
     生成されたファイルを保管するフォルダー。

     WSDL2Java パラメーターの完全なリストが必要な場合は、**/help** コマンド行スイッチを使用するか、WID/RAD で WSDL2Java ツールのオンライン・ヘルプを参照してください。

   - *WSDLfilepath* は、WebSphere 環境からエクスポートしたか、クライアント CD からコピーした WSDL ファイルのパスとファイル名です。

次の例では、ヒューマン・タスク・アクティビティー Web サービス API 用のプロキシー・クライアントが生成されます。

```
call wsdl2java.bat -r client -c client -noWrappedOperations
                   -output c:¥ws¥proxyClient  c:¥ws¥bin¥HTMWS.wsdl
```

2. 生成されたクラス・ファイルをプロジェクトに組み込みます。

BPEL プロセス用ヘルパー・クラスの作成 (Java Web サービス)

具象 API 要求 (sendMessage、call など) で参照されるビジネス・オブジェクトでは、クライアント・アプリケーションが「document/literal wrapped」スタイル・エレメントを使用する必要があります。クライアント・アプリケーションは、ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

始める前に

ヘルパー・クラスを作成するには、WebSphere Process Server 環境からあらかじめ Web サービス API の WSDL ファイルをエクスポートしておく必要があります。

このタスクについて

Web サービス API の call() 命令や sendMessage() 命令を実行すると、WebSphere Process Server 上で、BPEL プロセスとの対話ができるようになります。call() 命令の入力メッセージは、プロセスの入力メッセージの document/literal ラッパーが提供されると予想します。

BPEL プロセスまたはヒューマン・タスクのヘルパー・クラスを生成する技法は、次に示すように数多く存在します。

1. SoapElement オブジェクトを使用する。

   WebSphere Integration Developer で使用可能な Rational Application Developer 環境では、Web サービス・エンジンが JAX-RPC 1.1 環境をサポートします。JAX-RPC 1.1 では、SoapElement オブジェクトが Document Object Model (DOM) エレメントを拡張するので、DOM API を使用して SOAP メッセージを作成、読み取り、ロード、および保管することが可能です。

   例えば、WSDL ファイルに、次に示すワークフロー・プロセスまたはヒューマン・タスクの入力メッセージが含まれていると想定します。

```
<xsd:element name="operation1">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="input1" nillable="true" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

   プロセスまたはヒューマン・タスク・モジュールを開発する際に、WSDL ファイルが作成されます。

   DOM API を使用して、クライアント・アプリケーション内の対応する SOAP メッセージを作成するには、次のようにします。

```
SOAPFactory soapfactoryinstance = SOAPFactory.newInstance();
SOAPElement soapmessage = soapfactoryinstance.createElement
                          ("operation1", namespaceprefix, interfaceURI);
```

```
SOAPElement inputelement = soapfactoryinstance.createElement("input1");
inputelement.addTextNode( message value);
soapmessage.addChildElement(outputelement);
```

以下の例では、クライアント・アプリケーションで sendMessage 操作用の入力パラメーターを作成する方法を示します。

```
SendMessage inWsend = new SendMessage();
inWsend.setProcessTemplateName(processtemplatename);
inWsend.setPortType(porttype);
inWsend.setOperation(operationname);
inWsend.set_any(soapmessage);
```

2. WebSphere カスタム・データ・バインディング機能を使用する。

   この技法については、次の developerWorks の記事で説明されています。

   - How to choose a custom mapping technology for Web services
   - Developing Web Services with EMF SDOs for complex XML schema

   Interoperability With Patterns and Strategies for Document-Based Web Services

   Web Services support for Schema/WSDL(s) containing optional JAX-RPC 1.0/1.1 XML Schema Types

クライアント・アプリケーションの作成 (Java Web サービス)

クライアント・アプリケーションは、Web サービス API に要求を送信し、Web サービス API からの応答を受信します。プロキシー・クライアントを使用して、複合データ型のフォーマット設定を行う通信クラスおよびヘルパー・クラスを管理すると、クライアント・アプリケーションから、Web サービス・メソッドをローカル機能のように呼び出すことができます。

始める前に

クライアント・アプリケーションの作成を開始する前に、プロキシー・クライアントと必要なヘルパー・クラスを生成します。

このタスクについて

Web サービス対応の開発ツール (IBM Rational Application Developer (RAD) など) を使用すれば、クライアント・アプリケーションを開発できます。どのタイプの Web サービス・アプリケーションをビルドしても、Web サービス API を呼び出すことができます。

プロシージャー

1. 新規クライアント・アプリケーション・プロジェクトを作成します。
2. プロキシー・クライアントを生成し、Java ヘルパー・クラスをプロジェクトに追加します。
3. クライアント・アプリケーションをコーディングします。
4. プロジェクトをビルドします。
5. クライアント・アプリケーションを実行します。

以下の例では、Business Flow Manager Web サービス API の使用方法を示します。

```
// create the proxy
       BFMIFProxy proxy = new BFMIFProxy();
 // prepare the input data for the operation
      GetProcessTemplate iW = new GetProcessTemplate();
      iW.setIdentifier(your_process_template_name);

 // invoke the operation
      GetProcessTemplateResponse oW = proxy.getProcessTemplate(iW);

 // process output of the operation
      ProcessTemplateType ptd = oW.getProcessTemplate();
      System.out.println("getName= " + ptd.getName());
      System.out.println("getPtid= " + ptd.getPtid());
```

関連タスク

131 ページの『プロキシー・クライアントの生成 (Java Web サービス)』
Java Web サービス・クライアント・アプリケーションは、プロキシー・クライアント を使用して Web サービス API と対話します。

135 ページの『BPEL プロセス用ヘルパー・クラスの作成 (Java Web サービス)』
具象 API 要求 (sendMessage、call など) で参照されるビジネス・オブジェクトでは、クライアント・アプリケーションが「document/literal wrapped」 スタイル・エレメントを使用する必要があります。クライアント・アプリケーションは、 ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

セキュリティーの追加 (Java Web サービス)

Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことによって保護する必要があります。

WebSphere Application Server は現在、次に示す Web サービス API のセキュリティー・メカニズムをサポートしています。

- ユーザー名トークン
- Lightweight Third Party Authentication (LTPA)

関連概念

ビジネス・プロセスの許可のロール
ロールとは、同じ許可レベルを共有する担当者の集合です。 ビジネス・プロセスで実行できるアクションは、許可のロールによって異なります。 このロールは、J2EE ロールまたはインスタンス・ベースのロールです。

関連情報

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r1mx/topic/com.ibm.websphere.bpc.610.doc/doc/bpc/c6task_auth.html

ユーザー名トークンのインプリメント:

ユーザー名トークンのセキュリティー・メカニズムでは、ユーザー名とパスワード資格情報を提供します。

このタスクについて

ユーザー名トークンのセキュリティー・メカニズムにより、さまざまなコールバック・ハンドラー を選択してインプリメントできます。選択に応じて、次の違いがあります。

- クライアント・アプリケーションを実行するときに、ユーザー名とパスワードの入力を毎回求められます。
- ユーザー名とパスワードがデプロイメント記述子に書き込まれます。

どちらの場合でも、入力されるユーザー名とパスワードは、対応するビジネス・プロセス・コンテナーまたはヒューマン・タスク・コンテナーの許可ロールの値と一致する必要があります。

ユーザー名とパスワードは要求メッセージ・エンベロープの中にカプセル化されるので、SOAP メッセージ・ヘッダーに「はっきり」示されます。そのため、クライアント・アプリケーションで HTTPS (HTTP over SSL) 通信プロトコルを使用する構成にするよう強くお勧めします。これにより、すべての通信が暗号化されます。HTTPS 通信プロトコルは、Web サービス API のエンドポイント URL アドレスを指定するときに選択できます。

ユーザー名トークンを定義するには、次のステップを実行します。

プロシージャー

1. セキュリティー・トークンを作成します。
   a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。
   b. 「**WS 拡張**」タブをクリックします。
   c. 「**サービス参照**」の下に、次の「Web サービス参照 (Web Service References)」がリストされます。
      - service/BFMWSService (ビジネス・プロセスの場合)
      - service/HTMWSService (ヒューマン・タスクの場合)

      どちらがリストされるかは、プロキシー・クライアントの生成時に、BFMWS.wsdl (ビジネス・プロセスの場合)、HTMWS.wsdl (ヒューマン・タスクの場合)、またはその両方のうちのどれが追加されたかによって決まります。
   d. どちらのサービス参照の場合も、以下のようにします。
      1) いずれかの「**サービス参照**」を選択します。
      2) 「**要求生成プログラム構成**」セクションを展開します。
      3) 「**セキュリティー・トークン**」 サブセクションを展開します。
      4) 「**追加**」をクリックします。「セキュリティー・トークン」ウィンドウが開きます。
      5) 「**名前**」フィールドに、新規のセキュリティー・トークンの名前として **UserNameTokenBFM** または **UserNameTokenHTM** を入力します。
      6) 「**トークン・タイプ**」ドロップダウン・リストで、「**ユーザー名**」を選択します (「**ローカル名**」フィールドにはデフォルト値が自動的に入力されます)。

7) 「**URI**」 フィールドはブランクのままにします。ユーザー名トークンには URI 値は不要です。

8) 「**OK**」をクリックします。

2. 以下のように、トークン生成プログラムを作成します。

a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。

b. 「**WS バインディング**」タブをクリックします。

c. 「**サービス参照**」の下に、前のステップと同じ Web サービス参照がリストされます。

- service/BFMWSService (ビジネス・プロセスの場合)
- service/HTMWSService (ヒューマン・タスクの場合)

d. どちらのサービス参照の場合も、以下のようにします。

1) いずれかの「**サービス参照**」を選択します。

2) 「**セキュリティー要求生成プログラムのバインディング構成**」セクションを展開します。

3) 「**トークン生成プログラム**」サブセクションを展開します。

4) 「**追加**」をクリックします。「トークン生成プログラム」ウィンドウが開きます。

5) 「**名前**」 フィールドに、新規のトークン生成プログラムの名前 (「UserNameTokenGeneratorBFM」または 「UserNameTokenGeneratorHTM」など) を入力します。

6) 「**トークン生成クラス**」フィールドで、トークン生成プログラム・クラス **com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator** が選択されていることを確認します。

7) 「**セキュリティー・トークン**」ドロップダウン・リストで、前に作成した適切なセキュリティー・トークンを選択します。

8) 「**値タイプの使用**」チェック・ボックスを選択します。

9) 「**値のタイプ**」フィールドで、「**ユーザー名トークン (Username Token)**」を選択します(「**ローカル名**」フィールドは、「**ユーザー名トークン**」の選択を反映して自動的に入力されます)。

10) 「**コールバック・ハンドラー**」フィールドで、 「com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler」(クライアント・アプリケーションを実行すると、ユーザー名およびパスワードを入力するようプロンプトが表示される)、または 「com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler」と入力します。

11) **NonPromptCallbackHandler** を選択した場合は、有効なユーザー名とパスワードを、デプロイメント記述子の対応するフィールドに指定する必要があります。

12) 「**OK**」をクリックします。

関連タスク

122 ページの『Web サービス・エンドポイント・アドレスの指定』

Web サービス・エンドポイント・アドレスは、クライアント・アプリケーショ

ンが Web サービス API にアクセスするために 指定する必要のある URL です。エンドポイント・アドレスは、クライアント・アプリケーション用の プロキシー・クライアントを生成するためにエクスポートする WSDL ファイルに書き込まれます。

関連情報

IBM WebSphere Developer Technical Journal: Web services security with WebSphere Application Server V6

LTPA セキュリティー・メカニズムのインプリメント:

Lightweight Third Party Authentication (LTPA) セキュリティー・メカニズムは、クライアント・アプリケーションが以前に確立されたセキュリティー・コンテキストで実行されている場合に使用できます。

このタスクについて

LTPA セキュリティー・メカニズムは、セキュリティー・コンテキストが既に確立されている保護された環境でクライアント・アプリケーションが実行されている場合にのみ使用できます。例えば、クライアント・アプリケーションが Enterprise JavaBeans(EJB) コンテナーで実行されている場合、EJB クライアントは、クライアント・アプリケーションを呼び出すためにあらかじめログインしておく必要があります。セキュリティー・コンテキストはそれから確立されます。その後 EJB クライアント・アプリケーションが Web サービスを呼び出すと、LTPA コールバック・ハンドラーは、セキュリティー・コンテキストから LTPA トークンを取得して SOAP 要求メッセージに追加します。サーバー・サイドでは、LTPA トークンが LTPA メカニズムによって処理されます。

LTPA セキュリティー・メカニズムをインプリメントするには、次のステップを実行します。

プロシージャー

1. WebSphere Integration Developer で使用可能な Rational Application Developer 環境で、「**WS バインディング**」 → 「**セキュリティー要求生成プログラムのバインディング構成**」 → 「**トークン生成プログラム**」を選択します。
2. セキュリティー・トークンを作成します。
   a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。
   b. 「**WS 拡張**」タブをクリックします。
   c. 「**サービス参照**」の下に、次の「**Web サービス参照 (Web Service References)**」がリストされます。
      - service/BFMWSService (ビジネス・プロセスの場合)
      - service/HTMWSService (ヒューマン・タスクの場合)

      どちらがリストされるかは、プロキシー・クライアントの生成時に、BFMWS.wsdl (ビジネス・プロセスの場合)、HTMWS.wsdl (ヒューマン・タスクの場合)、またはその両方のうちのどれが追加されたかによって決まります。

d. どちらのサービス参照の場合も、以下のようにします。
   1) いずれかの「**サービス参照**」を選択します。
   2) 「**要求生成プログラム構成**」セクションを展開します。
   3) 「**セキュリティー・トークン**」 サブセクションを展開します。
   4) 「**追加**」をクリックします。「セキュリティー・トークン」ウィンドウが開きます。
   5) 「**名前**」フィールドに、新規のセキュリティー・トークンの名前として **LTPATokenBFM** または **LTPATokenHTM** を入力します。
   6) 「**トークン・タイプ**」ドロップダウン・リストで、「**LTPAToken**」を選択します (「**URI**」および「**ローカル名**」フィールドには、自動的にデフォルト値が入力されます)。
   7) 「**OK**」をクリックします。

3. 以下のように、トークン生成プログラムを作成します。
   a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。
   b. 「**WS バインディング**」タブをクリックします。
   c. 「**サービス参照**」の下に、前のステップと同じ Web サービス参照がリストされます。
      - service/BFMWSService (ビジネス・プロセスの場合)
      - service/HTMWSService (ヒューマン・タスクの場合)
   d. どちらのサービス参照の場合も、以下のようにします。
      1) いずれかの「**サービス参照**」を選択します。
      2) 「**セキュリティー要求生成プログラムのバインディング構成**」セクションを展開します。
      3) 「**トークン生成プログラム**」サブセクションを展開します。
      4) 「**追加**」をクリックします。「トークン生成プログラム」ウィンドウが開きます。
      5) 「**名前**」 フィールドに、新規のトークン生成プログラムの名前 (「LTPATokenGeneratorBFM」または「LTPATokenGeneratorHTM」など) を入力します。
      6) 「**トークン生成クラス**」フィールドで、トークン生成プログラム・クラス **com.ibm.wsspi.wssecurity.token.LTPATokenGenerator** が選択されていることを確認します。
      7) 「**セキュリティー・トークン**」ドロップダウン・リストで、前に作成した適切なセキュリティー・トークンを選択します。
      8) 「**値タイプの使用**」チェック・ボックスを選択します。
      9) 「**値のタイプ**」フィールドで、「**LTPAToken**」を選択します (「**URI**」および「**ローカル名**」フィールドは、「**LTPA トークン (LTPA Token)**」の選択を反映して自動的に入力されます)。
      10) 「**コールバック・ハンドラー**」フィールドに、「com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler」と入力します。
      11) 「**OK**」をクリックします。

結果

実行時に **LTPATokenCallbackHandler** は、既存のセキュリティー・コンテキストから LTPA トークンを取得して SOAP 要求メッセージに追加します。

トランザクション・サポートの追加 (Java Web サービス)

Java Web サービス・クライアント・アプリケーションを構成して、クライアント・アプリケーションのコンテキストをサービス要求の一部として引き渡すことによってクライアントのトランザクションにサーバー・サイド要求の処理が参加するのを許可することができます。このアトミック・トランザクション・サポートは、Web Services-Atomic Transaction (WS-AT) 仕様で定義されています。

このタスクについて

WebSphere Application Server は、各 Web サービスの API 要求を別々のアトミック・トランザクションとして実行します。クライアント・アプリケーションは、次のいずれかの方法でトランザクション・サポートを使用するように構成できます。

- トランザクションに参加する。サーバー・サイド要求の処理は、クライアント・アプリケーションのトランザクション・コンテキスト内で実行されます。この場合、Web サービス API 要求の実行中やロールバック中にサーバーに問題が発生すると、クライアント・アプリケーションの要求もロールバックされます。
- トランザクション・サポートを使用しない。WebSphere Application Server は、要求を実行するための新規トランザクションを引き続き作成しますが、サーバー・サイド要求の処理は、クライアント・アプリケーションのトランザクション・コンテキストでは実行されません。

.NET 環境でのクライアント・アプリケーションの開発

Microsoft .NET は、Web サービスを使用してアプリケーションを接続する強力な開発環境を提供します。

プロキシー・クライアントの生成 (.NET)

.NET クライアント・アプリケーションは、*プロキシー・クライアント* を使用して Web サービス API と対話します。プロキシー・クライアントのおかげで、クライアント・アプリケーションで、複雑な Web サービス・メッセージング・プロトコルを意識する必要がなくなります。

始める前に

プロキシー・クライアントを作成するには、まず WebSphere 環境からいくつかの WSDL ファイルをエクスポートして、それをクライアントのプログラミング環境にコピーする必要があります。

注: WebSphere Process Server クライアント CD を所有している場合は、代わりにその CD からファイルをコピーできます。

このタスクについて

プロキシー・クライアントは、C# Bean クラスのセットで構成されています。各クラスには、単一の Web サービスで公開されるメソッドおよびオブジェクトがすべて含まれています。サービス・メソッドは、パラメーターを完全な SOAP メッセー

ジにアセンブルし、その SOAP メッセージを HTTP 経由で Web サービスに送信し、Web サービスから応答を受信して、戻されたデータを処理します。

注: プロキシー・クライアントの生成が必要なのは一度だけです。 Web サービス API にアクセスするクライアント・アプリケーションはすべて、以後は同じプロキシー・クライアントを使用できます。

プロシージャー

1. プロキシー・クライアントの生成には WSDL コマンドを使用します。 タイプ:

   wsdl *options WSDLfilepath*

   各部の意味は、次のとおりです。

   - *オプション* は以下のとおりです。

     /language
     プロキシー・クラスの作成に使用する言語を指定できます。デフォルトは C# です。言語引数として、**VB** (Visual Basic)、**JS** (JScript)、または **VJS** (Visual J#) を指定することもできます。

     /output
     該当するサフィックスの付いた出力ファイルの名前。例えば、proxy.cs です。

     /protocol
     プロキシー・クラスでインプリメントされるプロトコル。**SOAP** がデフォルトの設定です。

     WSDL.exe パラメーターの完全なリストが必要な場合は、**/?** コマンド行スイッチを使用するか、Visual Studio で WSDL ツールのオンライン・ヘルプを参照してください。

   - *WSDLfilepath* は、WebSphere 環境からエクスポートしたか、クライアント CD からコピーした WSDL ファイルのパスとファイル名です。

   次の例では、Human Task Manager Web サービス API 用のプロキシー・クライアントが生成されます。

   ```
   wsdl /language:cs /output:proxyclient.cs c:¥ws¥bin¥HTMWS.wsdl
   ```

2. プロキシー・クライアントをダイナミック・リンク・ライブラリー (DLL) ファイルとしてコンパイルします。

BPEL プロセス用ヘルパー・クラスの作成 (.NET)

特定の Web サービス API 操作では、クライアント・アプリケーションが「document/literal」スタイルのラップ・エレメントを使用する必要があります。クライアント・アプリケーションは、ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

始める前に

ヘルパー・クラスを作成するには、WebSphere Process Server 環境からあらかじめ Web サービス API の WSDL ファイルをエクスポートしておく必要があります。

このタスクについて

Web サービス API の call() 命令と sendMessage() 命令により、BPEL プロセスが WebSphere Process Server 内で起動します。call() 命令の入力メッセージは、BPEL プロセスの入力メッセージの document/literal ラッパーが提供されると予想します。BPEL プロセスの必要な Bean とクラスを生成するには、<wsdl:types> エレメントを新規の XSD ファイルにコピーし、xsd.exe ツールを使用してヘルパー・クラスを生成します。

プロシージャー

1. まだ実行していない場合は、WebSphere Integration Developer から BPEL プロセス・インターフェースの WSDL ファイルをエクスポートします。
2. テキスト・エディターまたは XML エディターで WSDL ファイルを開きます。
3. <wsdl:types> エレメントのすべての子エレメントの内容をコピーし、新しいスケルトン XSD ファイルに貼り付けます。
4. XSD ファイル上で xsd.exe ツールを実行します。

   call xsd.exe file.xsd /classes /o

   各部の意味は、次のとおりです。

   file.xsd 変換する XML スキーマ定義ファイル。

   /classes (/c)
   指定した XSD ファイル (複数も可) の内容に対応するヘルパー・クラスを生成します。

   /output (/o)
   生成したファイルの出力ディレクトリーを指定します。このディレクトリーを省略すると、デフォルトでは現行ディレクトリーになります。

   以下に例を示します。

   call xsd.exe ProcessCustomer.xsd /classes /output:c:¥temp
5. 生成されるクラス・ファイルをクライアント・アプリケーションに追加します。例えば Visual Studio を使用する場合は、「**プロジェクト**」 → 「**既存項目の追加 (Add Existing Item)**」メニュー・オプションを実行します。

ProcessCustomer.wsdl ファイルの内容が以下のような場合:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:bons1="http://com/ibm/bpe/unittest/sca"
                  xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  name="ProcessCustomer"
                  targetNamespace="http://ProcessTypes/bpel/ProcessCustomer">
  <wsdl:types>
    <xsd:schema targetNamespace="http://ProcessTypes/bpel/ProcessCustomer"
                xmlns:bons1="http://com/ibm/bpe/unittest/sca"
                xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:import namespace="http://com/ibm/bpe/unittest/sca"
                  schemaLocation="xsd-includes/http.com.ibm.bpe.unittest.sca.xsd"/>
      <xsd:element name="doit">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="input1" nillable="true" type="bons1:Customer"/>
          </xsd:sequence>
```

```
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="doitResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="output1" nillable="true" type="bons1:Customer"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
    <wsdl:message name="doitRequestMsg">
    <wsdl:part element="tns:doit" name="doitParameters"/>
  </wsdl:message>
    <wsdl:message name="doitResponseMsg">
    <wsdl:part element="tns:doitResponse" name="doitResult"/>
  </wsdl:message>
    <wsdl:portType name="ProcessCustomer">
    <wsdl:operation name="doit">
      <wsdl:input message="tns:doitRequestMsg" name="doitRequest"/>
      <wsdl:output message="tns:doitResponseMsg" name="doitResponse"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

結果として出力される XSD ファイルは以下のようになります。

```
<xsd:schema xmlns:bons1="http://com/ibm/bpe/unittest/sca"
                     xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     targetNamespace="http://ProcessTypes/bpel/ProcessCustomer">
 <xsd:import namespace="http://com/ibm/bpe/unittest/sca"
              schemaLocation="Customer.xsd"/>
 <xsd:element name="doit">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="input1" type="bons1:Customer" nillable="true"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="doitResponse">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="output1" type="bons1:Customer" nillable="true"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

関連情報

XML Schema Definition Tool (XSD.EXE) に関する Microsoft の資料

クライアント・アプリケーションの作成 (.NET)

クライアント・アプリケーションは、Web サービス API に要求を送信し、Web サービス API からの応答を受信します。プロキシー・クライアントを使用して、複合データ型のフォーマット設定を行う通信クラスおよびヘルパー・クラスを管理すると、クライアント・アプリケーションから、Web サービス・メソッドをローカル機能のように呼び出すことができます。

始める前に

クライアント・アプリケーションの作成を開始する前に、プロキシー・クライアントと必要なヘルパー・クラスを生成します。

このタスクについて

.NET 対応の開発ツール (Visual Studio .NET など) を使用すれば、.NET クライアント・アプリケーションを開発できます。どのタイプの .NET アプリケーションをビルドしても、汎用の Web サービス API を呼び出すことができます。

プロシージャー

1. 新規クライアント・アプリケーション・プロジェクトを作成します。 例えば、Visual Studio で **WinFX Windows® Application** を作成します。
2. プロジェクト・オプションで、プロキシー・クライアントのダイナミック・リンク・ライブラリー (DLL) ファイルに参照を追加します。 ビジネス・オブジェクト定義を含むヘルパー・クラスすべてをプロジェクトに追加します。例えば Visual Studio では、**「プロジェクト」 → 「既存項目の追加 (Add existing item)」**オプションを実行します。
3. プロキシー・クライアント・オブジェクトを作成します。 以下に例を示します。

```
HTMClient.HTMReference.HumanTaskManagerComponent1Export_HumanTaskManagerHttpService service =
    new HTMClient.HTMReference.HumanTaskManagerComponent1Export_HumanTaskManagerHttpService();
```

4. メッセージで使用され、Web サービスとの間で送受信されるビジネス・オブジェクトのデータ型を宣言します。 以下に例を示します。

```
HTMClient.HTMReference.TKIID id = new HTMClient.HTMReference.TKIID();

ClipBG bg = new ClipBG();
Clip clip = new Clip();
```

5. 特定の Web サービス関数を呼び出し、必要なパラメーターを指定します。 例えば、ヒューマン・タスクを作成して開始するには、以下のようにします。

```
HTMClient.HTMReference.createAndStartTask task =
   new HTMClient.HTMReference.createAndStartTask();
HTMClient.HTMReference.StartTask sTask = new HTMClient.HTMReference.StartTask();

sTask.taskName = "SimpleTask";
sTask.taskNamespace = "http://myProcess/com/acme/task";
sTask.inputMessage = bg;
task.inputTask = sTask;

id = service.createAndStartTask(task).outputTask;
```

6. リモートのプロセスおよびタスクは、永続 ID (上記の例では `id`) で識別されます。 例えば、上記で作成したヒューマン・タスクを要求するには、以下のようにします。

```
HTMClient.HTMReference.claimTask claim = new HTMClient.HTMReference.claimTask();
claim.inputTask = id;
```

関連タスク

142 ページの『プロキシー・クライアントの生成 (.NET)』
.NET クライアント・アプリケーションは、*プロキシー・クライアント* を使用して Web サービス API と対話します。プロキシー・クライアントのおかげで、クライアント・アプリケーションで、 複雑な Web サービス・メッセージング・プロトコルを意識する必要がなくなります。

143 ページの『BPEL プロセス用ヘルパー・クラスの作成 (.NET)』
特定の Web サービス API 操作では、クライアント・アプリケーションが「document/literal」スタイルのラップ・エレメントを使用する必要があります。クライアント・アプリケーションは、 ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

セキュリティーの追加 (.NET)

Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことにより保護できます。

このタスクについて

これらのセキュリティー・メカニズムには、ユーザー名トークン (ユーザー名とパスワード)、またはカスタム・バイナリーと XML ベースのセキュリティー・トークンなどがあります。

プロシージャー

1. Web Services Enhancements (WSE) 2.0 SP3 for Microsoft .NET をダウンロードしてインストールします。 入手先は以下のとおりです。

   http://www.microsoft.com/downloads/details.aspx?familyid= 1ba1f631-c3e7-420a-bc1e-ef18bab66122&displaylang=en
2. 生成されたプロキシー・クライアント・コードを以下のように変更します。

   変更前:

```
public class Export1_MyMicroflowHttpService : System.Web.Services.Protocols.SoapHttpClientProtocol {
```

   変更後:

```
public class Export1_MyMicroflowHttpService : Microsoft.Web.Services2.WebServicesClientProtocol {
```

   注: これらの変更は、WSDL.exe ツールを実行してプロキシー・クライアントを再生成すると失われます。
3. ファイルの先頭に以下の行を追加して、クライアント・アプリケーション・コードを変更します。

```
using System.Web.Services.Protocols;
using Microsoft.Web.Services2;
using Microsoft.Web.Services2.Security.Tokens;
...
```
4. 必要なセキュリティー・メカニズムをインプリメントするコードを追加します。例えば、ユーザー名とパスワード保護を追加するコードは次のとおりです。

```
string user = "U1";
string pwd = "password";
UsernameToken token =
   new UsernameToken(user, pwd, PasswordOption.SendPlainText);

me._proxy.RequestSoapContext.Security.Tokens.Clear();
me._proxy.RequestSoapContext.Security.Tokens.Add(token);
```

ビジネス・プロセスおよびタスク関連のオブジェクトの照会

Web サービス API を使用すると、Business Process Choreographer データベース内のビジネス・プロセス・オブジェクトおよびタスク関連オブジェクトを照会して、これらのオブジェクトの特定のプロパティーを取得することができます。

このタスクについて

Business Process Choreographer データベースは、ビジネス・プロセスおよびタスクの管理用のテンプレート (モデル) とインスタンス (ランタイム) のデータを保管します。

クライアント・アプリケーションは、Web サービス API 経由で照会を発行し、データベースからビジネス・プロセスおよびビジネス・タスクに関する情報を取得することができます。

クライアント・アプリケーションは、1 回限りの照会を発行して、オブジェクトの特定のプロパティーを取得することができます。使用頻度の高い照会は、保管しておくことができます。クライアント・アプリケーションは、このような保管照会文を後で取り出して使用することができます。

ビジネス・プロセスおよびタスク関連オブジェクトに対する照会

Web サービス API の QUERY インターフェースを使用して、ビジネス・プロセスおよびタスクに関する情報を取得します。

クライアント・アプリケーションは、SQL 形式の構文を使用して、データベースを照会します。

Java Web サービスの例

```
string processTemplateName = "ProcessCustomerLR";
query query1 = new query();
query1.selectClause = "DISTINCT PROCESS_INSTANCE.STARTED, PROCESS_INSTANCE.PIID";
query1.whereClause =
       "PROCESS_INSTANCE.TEMPLATE_NAME = '" + processTemplateName + "'";
query1.orderByClause = "PROCESS_INSTANCE.STARTED";
query1.threshold = null;
query1.timeZone = "UTC"; query1.skipTuples = null;
queryResponse queryResponse1 = proxy.query(query1);
```

データベースから取り出した情報は、Web サービス API を使用して*照会結果セット*として戻されます。

以下に例を示します。

```
QueryResultSetType queryResultSet = queryResponse1.queryResultSet;
if (queryResultSet != null) {
   Console.WriteLine("--> QueryResultSetType");
   Console.WriteLine(" . size= " + queryResultSet.size);
   Console.WriteLine(" . numberColumns= " + queryResultSet.numberColumns);
   string indent = " . ";

   // -- the query column info
   QueryColumnInfoType[] queryColumnInfo = queryResultSet.QueryColumnInfo;
   if (queryColumnInfo.Length > 0) {
      Console.WriteLine();
      Console.WriteLine("= . QueryColumnInfoType size= " + queryColumnInfo.Length);
      Console.Write( " | tableName ");
      for (int i = 0; i < queryColumnInfo.Length ; i++) {
```

```
         Console.Write( " | " + queryColumnInfo[i].tableName.PadLeft(20) );
      }
      Console.WriteLine();
      Console.Write( " | columnName ");
      for (int i = 0; i < queryColumnInfo.Length ; i++) {
         Console.Write( " | " + queryColumnInfo[i].columnName.PadLeft(20) );
      }
      Console.WriteLine();
      Console.Write( " | data type ");
      for (int i = 0; i < queryColumnInfo.Length ; i++) {
         QueryColumnInfoTypeType tt = queryColumnInfo[i].type;
         Console.WriteLine( " | " + tt.ToString());
      }
         Console.WriteLine();
   }
   else {
      Console.WriteLine("--> queryColumnInfo= <null>");
   }

   // - the query result values
   string[][] result = queryResultSet.result;
   if (result !=null) {
      Console.WriteLine();
      Console.WriteLine("= . result size= " + result.Length);
      for (int i = 0; i &lt; result.Length; i++) {
        Console.Write(indent +i );
        string[] row = result[i];
        for (int j = 0; j &lt; row.Length; j++ ) {
           Console.Write(" | " + row[j]);
        }
        Console.WriteLine();
      }
   }
   else {
      Console.WriteLine("--> result= <null>");
   }
}
else {
   Console.WriteLine("--> QueryResultSetType= <null>");
}
```

照会関数は、呼び出し元の権限に応じてオブジェクトを戻します。照会結果セットには、呼び出し元が表示する許可を持つオブジェクトのプロパティーが含まれるのみです。

オブジェクトのプロパティーを照会するために、事前定義データベース・ビューが提供されています。プロセス・テンプレートの場合、照会関数には以下の構文があります。

```
ProcessTemplateData[] queryProcessTemplates
                      (java.lang.String whereClause,
                       java.lang.String orderByClause,
                       java.lang.Integer threshold,
                       java.util.TimeZone timezone);
```

タスク・テンプレートの場合、照会関数には以下の構文があります。

```
TaskTemplate[] queryTaskTemplates
               (java.lang.String whereClause,
                java.lang.String orderByClause,
                java.lang.Integer threshold,
                java.util.TimeZone timezone);
```

他のビジネス・プロセスおよびタスク関連オブジェクトの場合、照会関数には以下の構文があります。

```
QueryResultSet query (java.lang.String selectClause,
                      java.lang.String whereClause,
                      java.lang.String orderByClause,
                      java.lang.Integer skipTuples
                      java.lang.Integer threshold,
                      java.util.TimeZone timezone);
```

QUERY インターフェースには、queryAll メソッドも含まれています。このメソッドを使用して、オブジェクトに関係のあるデータすべてを、モニターなどの目的で取得することができます。queryAll メソッドの呼び出し元には、 Java 2 Platform Enterprise Edition (J2EE) ロールの、BPESystemAdministrator、 BPESystemMonitor、TaskSystemAdministrator、または TaskSystemMonitor のいずれかが必要です。オブジェクトの対応する作業項目を使用した許可検査は適用されません。

.NET の例

```
ProcessTemplateType[] templates = null;

   try {
        queryProcessTemplates iW = new queryProcessTemplates();
        iW.whereClause = "PROCESS_TEMPLATE.STATE=PROCESS_TEMPLATE.STATE.STATE_STARTED";
        iW.orderByClause = null;
        iW.threshold = null;
        iW.timeZone = null;

        Console.WriteLine("--> queryProcessTemplates ... ");
        Console.WriteLine("--> query: WHERE " + iW.whereClause + " ORDER BY " +
          iW.orderByClause + " THRESHOLD " + iW.threshold + " TIMEZONE" + iW.timeZone);

        templates = proxy.queryProcessTemplates(iW);

        if (templates.Length < 1) {
          Console.WriteLine("--> No templates found :-(");
        }
        else {
          for (int i = 0; i < templates.Length ; i++) {
            Console.Write("--> found template with ptid: " + templates[i].ptid);
            Console.WriteLine(" and name: " + templates[i].name);
           /*  ... other properties of ProcessTemplateType ... */
          }
        }
      }
      catch( Exception e ) {
        Console.WriteLine("exception= " + e);
      }
```

照会パラメーター:

各照会では、複数の SQL 形式の文節およびパラメーターを指定する必要があります。

照会は以下のもので構成されます。

- select 文節
- where 文節
- order-by 文節
- スキップ・タプル・パラメーター
- しきい値パラメーター

- 時間帯パラメーター

ビジネス・プロセス・オブジェクトおよびヒューマン・タスク・オブジェクトの照会のための事前定義ビュー

ビジネス・プロセス・オブジェクトおよびヒューマン・タスク・オブジェクト用に、事前定義データベース・ビューが提供されています。

これらのオブジェクトの参照データを照会する場合は、これらのビューを使用します。これらのビューを使用する場合は、ビューの列に明示的に述部を結合する必要はありません。これらの構成要素は自動的に追加されます。Web サービス API の照会機能を使用して、このデータを照会できます。

保管照会文の管理

保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

このタスクについて

保管照会文は、データベースに保管され、名前で識別される照会のことです。専用の保管照会文と共通の保管照会文の名前を同じにすることができます。異なる複数の所有者の専用保管照会文を同じ名前にすることもできます。

保管照会文は、ビジネス・プロセス・オブジェクト、タスク・オブジェクト、またはこの 2 つのオブジェクト・タイプの組み合わせたものを対象とします。

共通保管照会文の管理
共通保管照会文がシステム管理者によって作成されます。 この照会は、全ユーザーが使用できます。

他のユーザーの専用保管照会文の管理
専用照会はどのユーザーでも作成できます。この照会は、 照会の所有者とシステム管理者しか使用できません。

専用保管照会文の操作
システム管理者でなくても、自分専用の保管照会文は作成、実行、および削除できます。また、システム管理者が作成した共通の保管照会文を使用することもできます。

JMS クライアント・アプリケーションの開発

Java Messaging Service (JMS) API を介してビジネス・プロセス・アプリケーションにアクセスするクライアント・アプリケーションを開発できます。

このタスクについて

JMS の紹介

WebSphere Process Server バージョン 6.1 は、通信手段として、Java Messaging Service (JMS) プログラミング・インターフェースに基づく非同期メッセージングをサポートします。

JMS は、要求を JMS メッセージとして作成、送信、受信、および読み取るため共通の方法を Java クライアント (クライアント・アプリケーションまたは J2EE アプリケーション) に提供します。

JMS は、以下を行う非同期メッセージングに基づくインターフェースです。

- **point-to-point メッセージングまたはパブリッシュ/サブスクライブ・メッセージング**のいずれかを使用します。メッセージに基づくフレームワークは、情報を明示的に要求しなくても、他のアプリケーションに情報をプッシュできます。同じ情報を多数のサブスクライバーに並行して送達できます。 Business Process Choreographer の JMS インターフェースは point-to-point メッセージングのみサポートします。
- **リズムの独立性**を提供します。JMS フレームワークは非同期に機能しますが、同期要求/応答モードをシミュレートすることもできます。これにより、ソース・システムおよびターゲット・システムは相互に待機しなくても同時に作業できます。この機能は Business Process Choreographer にとって非常に役立ちます。なぜなら、長期実行のビジネス・プロセスと非同期で対話する機能を提供するからです。
- **トランザクション**をサポートします。トランザクションにより、クライアント・アプリケーションは送受信されたメッセージのグループを単一のアトミック単位として処理できます。 JMS トランザクションはサーバーのトランザクション内で実行されます。 Business Process Choreographer の JMS インターフェースの場合、通常はトランザクションごとに単一のメッセージが送受信されます。
- **情報の送達を保証します**。JMS フレームワークはメッセージをトランザクション・モードで管理し、メッセージ送達を保証できます (とはいえ、送達の適時性を保証するものではありません)。 Business Process Choreographer の場合、この信頼できるメッセージ送達機能はビジネス・プロセスを処理するため、特に重要です。
- **異機種フレームワーク間のインターオペラビリティー**を保証します。ソース・アプリケーションとターゲット・アプリケーションは異機種環境で作動でき、それぞれのフレームワークに関連した通信および実行の問題を処理する必要がありません。
- **交換をさらに流動的にします**。メッセージ・モードを切り替えると、きめの細かい情報を交換できます。

ビジネス・プロセスの要件

Business Process Choreographer 上で実行するように WebSphere Integration Developer で開発されたビジネス・プロセスが、JMS API によってアクセス可能となるためには、特定の規則に準拠する必要があります。

要件は以下のとおりです。

1. ビジネス・プロセスのインターフェースは、 Java API for XML-based RPC (JAX-RPC 1.1) 仕様で定義された「document/literal wrapped」スタイルを使用して定義する必要があります。これは、WebSphere Integration Developer で開発するすべてのビジネス・プロセスとヒューマン・タスクのデフォルト・スタイルです。

2. Web サービス操作のビジネス・プロセスとヒューマン・タスクによって公開される障害メッセージは、XML スキーマ・エレメントにより定義される単一の WSDL メッセージ部を構成する必要があります。以下に例を示します。

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

関連情報

Java API for XML based RPC (JAX-RPC) のダウンロード・ページ

Which style of WSDL should I use?

JMS インターフェースへのアクセス

JMS インターフェースを介してメッセージを送受信するには、まずアプリケーションで BPC.cellname.Bus への接続を作成し、セッションを作成し、メッセージ・プロデューサーおよびコンシューマーを生成する必要があります。

このタスクについて

プロセス・サーバーは、point-to-point パラダイムに従う Java Message Service (JMS) メッセージを受け入れます。 JMS メッセージを送受信するアプリケーションは、以下のアクションに従う必要があります。

以下の例では、JMS クライアントは管理対象環境 (EJB、アプリケーション・クライアント、または Web クライアント・コンテナー) で実行していると想定しています。 JMS クライアントを J2SE 環境で実行する場合は、http://www-1.ibm.com/support/docview.wss?uid=swg24012804の「IBM Client for JMS on J2SE with IBM WebSphere Application Server」を参照してください。

プロシージャー

1. BPC.*cellname*.Bus への接続を作成します。 クライアント・アプリケーションの要求用の事前構成された接続ファクトリーは存在しません。つまり、クライアント・アプリケーションは、JMS API の ReplyConnectionFactory を使用するか、または固有の接続ファクトリーを作成するかのいずれかが可能です。作成する場合は接続ファクトリーを検索するために、Java Naming and Directory Interface (JNDI) 参照を使用できます。 JNDI 参照名は、Business Process Choreographer の外部要求キューの構成時に指定したものと同じ名前である必要があります。以下の例では、クライアント・アプリケーションが「jms/clientCF」という固有の接続ファクトリーを作成すると想定しています。

```
//Obtain the default initial JNDI context.
Context initialContext = new InitialContext();

// Look up the connection factory.
// Create a connection factory that connects to the BPC bus.
// Call it, for example, "jms/clientCF".
// Also configure an appropriate authentication alias.
ConnectionFactory connectionFactory =
            (ConnectionFactory)initialcontext.lookup("jms/clientCF");

// Create the connection.
Connection connection = connectionFactory.createConnection();
```

2. セッションを作成し、メッセージ・プロデューサーおよびコンシューマーが作成されるようにします。

```
// Create a transaction session using auto-acknowledgement.
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

3. メッセージを送信するメッセージ・プロデューサーを作成します。 JNDI 参照名は、Business Process Choreographer の外部要求キューの構成時に指定したものと同じ名前である必要があります。

```
// Look up the destination of the Business Process Choreographer input queue to
// send messages to.
Queue sendQueue = (Queue) initialcontext.lookup("jms/BFMJMSAPIQueue");

// Create a message producer.
MessageProducer producer = session.createProducer(sendQueue);
```

4. 応答を受信するメッセージ・コンシューマーを作成します。 応答宛先の JNDI 参照名は、ユーザー定義の宛先を指定できますが、デフォルトの (Business Process Choreographer 定義の) 応答宛先 jms/BFMJMSReplyQueue も指定できます。どちらの場合も、応答宛先は BPC.<cellname>.Bus にしておく必要があります。

```
// Look up the destination of the reply queue.
Queue  replyQueue = (Queue) initialcontext.lookup("jms/BFMJMSReplyQueue");

// Create a message consumer.
MessageConsumer consumer = session.createConsumer(replyQueue);
```

5. メッセージを送信します。

```
// Start the connection.
connection.start();

// Create a message - see the task descriptions for examples - and send it.
// This method is defined elsewhere ...
String payload = createXMLDocumentForRequest();
TextMessage requestMessage = session.createTextMessage(payload);

// Set mandatory JMS header.
// targetFunctionName is the operation name of JMS API
// (for example, getProcessTemplate, sendMessage)
requestMessage.setStringProperty("TargetFunctionName", targetFunctionName);

// Set the reply queue; this is mandatory if the replyQueue
// is not the default queue (as it is in this example).
requestMessage.setJMSReplyTo(replyQueue);

// Send the message.
producer.send(requestMessage);

// Get the message ID.
String jmsMessageID = requestMessage.getJMSMessageID();

session.commit();
```

6. 返信を受信します。

```
// Receive the reply message and analyse the reply.
TextMessage  replyMessage = (TextMessage) consumer.receive();

// Get the payload.
String payload = replyMessage.getText();

session.commit();
```

7. 接続を閉じ、リソースを解放します。

```
 // Final housekeeping; free the resources.
session.close();
connection.close();
```

注: 各トランザクションの後に接続を閉じることは不要です。接続が開始したら、接続が閉じるまでに、要求および応答メッセージはいくつでも交換できます。例では、単一のビジネス・メソッド内に単一の呼び出しがある単純なケースを示しています。

Business Process Choreographer JMS メッセージの構造

各 JMS メッセージのヘッダーおよび本文には事前定義構造がある必要があります。

Java Message Service (JMS) メッセージは以下のもので構成されます。

- メッセージ識別およびルーティング情報用のメッセージ・ヘッダー。
- 目次を保持するメッセージの本文 (ペイロード)。

Business Process Choreographer はテキスト・メッセージ形式のみサポートします。

メッセージ・ヘッダー

JMS は、クライアントが多くのメッセージ・ヘッダー・フィールドにアクセスできるようにします。

Business Process Choreographer JMS クライアントは以下のヘッダー・フィールドを設定できます。

- **JMSReplyTo**

  要求に対する応答を送信する宛先。このフィールドが要求メッセージで指定されていない場合、応答は Export インターフェースのデフォルトの応答宛先に送信されます (Export は、ビジネス・プロセス・コンポーネントのクライアント・インターフェース・レンダリングです)。この宛先は、`initialContext.lookup ("jms/BFMJMSReplyQueue");` を使用して取得できます。
- **TargetFunctionName**

  WSDL 操作の名前 (例えば、"queryProcessTemplates")。このフィールドは常に設定する必要があります。 TargetFunctionName は、ここで説明されている汎用の JMS メッセージ・インターフェースの操作を指定することに注意してください。これを、例えば **call** または **sendMessage** 操作を使用して、間接的に呼び出すことができる具体的なプロセスまたはタスクによって提供される操作と混同しないでください。

また、Business Process Choreographer クライアントは以下のヘッダー・フィールドにアクセスできます。

- **JMSMessageID**

  メッセージを一意的に識別します。メッセージが送信されると、JMS プロバイダーによって設定されます。メッセージの送信前にクライアントが JMSMessageID を設定する場合、JMS プロバイダーによって上書きされます。メッセージの ID が認証目的で必要とされる場合、クライアントはメッセージの送信後に JMSMessageID を取得できます。
- **JMSCorrelationID**

メッセージをリンクします。このフィールドは設定しないでください。 Business Process Choreographer 応答メッセージには、要求メッセージの JMSMessageID が含まれています。

各応答メッセージには以下の JMS ヘッダー・フィールドが含まれています。

- **IsBusinessException**

  WSDL 出力メッセージの場合は ″false″ で、WSDL 障害メッセージの場合は ″true″ です。

ServiceRuntimeExceptions は非同期クライアント・アプリケーションに戻されません。 JMS 要求メッセージの処理中に重大な例外が発生すると、それはランタイム障害となり、この要求メッセージを処理しているトランザクションはロールバックします。その後、JMS 要求メッセージが再度送達されます。メッセージを SCA Export の一部として処理している間 (例えば、メッセージの非直列化中) に障害が早期に発生する場合、SCA Export の受信宛先によって指定された失敗した送達の最大数まで再試行されます。送達の失敗最大数に達した後、要求メッセージは Business Process Choreographer バスのシステム例外宛先に追加されます。しかし、Business Flow Manager の SCA コンポーネントによる要求を実際に処理している間に障害が発生する場合、失敗した要求メッセージは WebSphere Process Server の失敗したイベント管理インフラストラクチャーによって処理されます。つまり、再試行しても例外状態が解決しない場合は、最終的に、失敗したイベント管理データベースに入れられることがあります。

メッセージ本文

JMS メッセージ本文は、操作の文書/リテラル・ラッパー・エレメントを表す XML 文書が入ったストリングです。

有効な要求メッセージ本文の単純な例を以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<_6:queryProcessTemplates xmlns:_6="http://www.ibm.com/xmlns/prod/
      websphere/business-process/services/6.0">
<whereClause>PROCESS_TEMPLATE.STATE IN (1)</whereClause>
</_6:queryProcessTemplates>
```

JMS レンダリングの許可

JMS インターフェースの使用を許可するには、WebSphere Application Server でセキュリティー設定が使用可能になっている必要があります。

ビジネス・プロセス・コンテナーがインストールされている場合、ロール **JMSAPIUser** をユーザー ID にマップする必要があります。このユーザー ID を使用して、すべての JMS API 要求を発行します。例えば、**JMSAPIUser** が 「User A」にマップされる場合、すべての JMS API 要求はプロセス・エンジンにとって「User A」から発生しているように見えます。

JMSAPIUser ロールには以下の権限を割り当てる必要があります。

要求	必要な許可
forceTerminate	プロセス管理者

要求	必要な許可
sendEvent	可能なアクティビティー所有者またはプロセス管理者

注: その他のすべての要求の場合、特殊権限は必要ありません。

特殊権限は、ビジネス・プロセス管理者のロールを持つ人物に付与されます。ビジネス・プロセス管理者は特殊なロールです。これは、プロセス・インスタンスのプロセス管理者とは異なります。ビジネス・プロセス管理者はすべての特権を持っています。

プロセス・スターターのユーザー ID は、プロセス・インスタンスが存在している場合、ユーザー・レジストリーから削除できません。このユーザー ID を削除する場合、このプロセスのナビゲーションが継続できません。 システム・ログ・ファイルに、次のような例外が書き込まれます。

```
no unique ID for: <user ID>
```

JMS API の概要

JMS メッセージ・インターフェース (今後は「JMS API」と表します) により、Business Process Choreographer 環境で実行しているビジネス・プロセスに非同期でアクセスするクライアント・アプリケーションを開発できます。

JMS API により、クライアント・アプリケーションは Microflow および長期実行プロセスと非同期で対話できます。

JMS API は Web サービス API と同じインターフェースを公開します。ただし、以下の例外があります。

- Web サービス API と共に使用する場合、`call` 操作は Microflow を呼び出すためにのみ使用できます。しかし、JMS API を使用する場合、`call` 操作を使用して Microflow と長期実行プロセスの両方を呼び出すことができます。
- 以下の操作は JMS API を使用して公開されません。
  - `callAsync` 操作 (およびそれに関連したコールバック操作)
  - `completeAndClaimSuccessor` 操作および `getParticipatingTask` 操作

例 - 長期実行プロセスの実行

一般のクライアント・アプリケーションが長期実行プロセスで作業する場合、以下の一連のステップを実行します。

1. 153 ページの『JMS インターフェースへのアクセス』の説明に従って、JMS 環境をセットアップします。
2. 以下のように、インストールされたプロセス定義のリストを取得します。
   - `queryProcessTemplates` を送信します。
   - これにより、**ProcessTemplate** オブジェクトのリストが戻されます。
3. 以下のように、開始アクティビティー (`createInstance="yes"` を指定した receive または pick) のリストを取得します。
   - `getStartActivities` を送信します。

- これにより、**InboundOperationTemplate** オブジェクトのリストが戻されます。

4. 入力メッセージを作成します。これは環境に固有で、事前に配置された、プロセス固有の成果物を使用しなければならないことがあります。
5. 以下のように、プロセス・インスタンスを作成します。
   - `sendMessage` を発行します。

   JMS API と共に `call` 操作を使用して、ビジネス・プロセスによって提供される長期実行の要求/応答操作と対話することもできます。この操作では、長期間たった後でも、操作の結果または障害を指定された応答宛先に戻します。そのため、`call` 操作を使用する場合、`query` および `getOutputMessage` 操作を使用してプロセスの出力または障害メッセージを取得する必要はありません。
6. オプションで、以下のステップを繰り返して、プロセス・インスタンスから出力メッセージを取得します。
   - `query` を発行して、終了状態のプロセス・インスタンスを取得します。
   - `getOutputMessage` を発行します。
7. オプションで、以下のように、プロセスによって公開された追加の操作を実行します。
   - `getWaitingActivities` または `getActiveEventHandlers` を発行して、**InboundOperationTemplate** オブジェクトのリストを取得します。
   - 入力メッセージを作成します。
   - `sendMessage` を発行してメッセージを送信します。
8. オプションで、プロセスに対して定義された、またはアクティビティーを含むカスタム・プロパティーを、`getCustomProperties` および `setCustomProperties` で取得および設定します。
9. オプションで、以下のようにプロセス・インスタンスでの作業を終了します。
   - `delete` および `terminate` を送信して、長期実行プロセスでの作業を終了します。

JMS アプリケーションの開発

JMS クライアント・アプリケーションは、Java で Java 2 Enterprise Edition (J2EE) 環境を使用して開発する必要があります。

このタスクについて

JMS クライアント・アプリケーションは要求および応答メッセージを JMS API と交換します。要求メッセージを作成するために、クライアント・アプリケーションは JMS TextMessage メッセージ本文に、対応する操作の文書/リテラル・ラッパーを表す XML エレメントを入力します。

成果物のコピー

JMS クライアント・アプリケーションの作成を補助するために、いくつかの成果物を WebSphere 環境からコピーすることができます。

BOXMLSerializer を使用して JMS メッセージ本文を作成する場合に限り、これらの成果物の使用が必須です。

これらの成果物は、次の 2 つの方法で入手できます。

- 成果物を WebSphere Process Server 環境で公開してエクスポートします。

  WebSphere Process Server 6.1 の場合、すべてのクライアント成果物は *install_root*¥ProcessChoreographer¥client ディレクトリーにあります。 JMS API の場合、成果物は以下のとおりです。

  BFMIF.wsdl

  BFMIF.xsd

  BPCGen.xsd

- WebSphere Process Server クライアント CD からファイルをコピーします。

サーバー環境からの成果物の公開:

JMS API にアクセスするクライアント・アプリケーションを開発できるように、WebSphere サーバー環境から、いくつかの成果物を公開することができます。

このタスクについて

WebSphere Process Server 6.1 の場合、すべてのクライアント成果物は *was_home*¥ProcessChoreographer¥client ディレクトリーにあります。 JMS API の場合、成果物は以下のとおりです。

BFMIF.wsdl

BFMIF.xsd

BPCGen.xsd

これらの成果物は、公開された後、ご使用のクライアント・プログラミング環境にコピーします。

クライアント CD からのファイルのコピー:

JMS API にアクセスするのに必要なファイルは、WebSphere Process Server クライアント CD で入手できます。

プロシージャー

1. クライアント CD を開き、`ProcessChoreographer¥client` ディレクトリーを参照します。
2. 必要なファイルをクライアント・アプリケーション開発環境にコピーします。

   WebSphere Process Server 6.1 の場合、すべてのクライアント成果物は ¥ProcessChoreographer¥client ディレクトリーにあります。 JMS API の場合、成果物は以下のとおりです。

   BFMIF.wsdl

   BFMIF.xsd

   BPCGen.xsd

応答メッセージでのビジネス例外の検査

JMS クライアント・アプリケーションは、すべての応答メッセージのメッセージ・ヘッダーを検査して、ビジネス例外を調べる必要があります。

このタスクについて

JMS クライアント・アプリケーションはまず、応答メッセージのヘッダーの **IsBusinessException** プロパティーを検査する必要があります。

以下に例を示します。

```
// receive response message
Message receivedMessage = ((JmsProxy) getToBeInvokedUponObject().receiveMessage();
String strResponse = ((TextMessage) receivedMessage).getText();

if (receivedMessage.getStringProperty("IsBusinessException") {
  // strResponse is a bussiness fault
  // any api can end w/a processFaultMsg
  // the call api also w/a businessFaultMsg
 }
else {
  // strResponse is the output message
}
```

JSF コンポーネントを使用した、ビジネス・プロセスおよびヒューマン・タスク用 Web アプリケーションの開発

Business Process Choreographer は、いくつかの JavaServer Faces (JSF) コンポーネントを提供します。これらのコンポーネントを拡張および統合して、ビジネス・プロセスおよびヒューマン・タスク機能を Web アプリケーションに追加することができます。

このタスクについて

WebSphere Integration Developer を使用して Web アプリケーションを作成することができます。

プロシージャー

1. 動的プロジェクトを作成し、Web プロジェクト・フィーチャー・プロパティーを変更して JSF 基本コンポーネントを組み込みます。

   Web プロジェクトの作成の詳細については、WebSphere Integration Developer インフォメーション・センターにアクセスしてください。

2. 前提条件である Business Process Choreographer Explorer Java アーカイブ (JAR ファイル) を追加します。

   以下のファイルをプロジェクトの WEB-INF/lib ディレクトリーに追加してください。

   - bpcclientcore.jar
   - bfmclientmodel.jar
   - htmclientmodel.jar
   - bpcjsfcomponents.jar

   Web アプリケーションをリモート・サーバーにデプロイする場合、以下のファイルも追加してください。 Business Process Choreographer API にリモート側でアクセスするには、以下のファイルが必要です。

   - bpe137650.jar

- task137650.jar

WebSphere Process Server では、これらのすべてのファイルは以下のディレクトリーにあります。

- Windows システムの場合: *install_root*¥ProcessChoreographer¥client
- UNIX®、Linux®、および i5/OS® システムの場合: *install_root*/ProcessChoreographer/client

3. 必要な EJB 参照を、Web アプリケーション・デプロイメント記述子 web.xml ファイルに追加します。

```
<ejb-ref id="EjbRef_1">
  <ejb-ref-name>ejb/BusinessProcessHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
  <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
<ejb-ref id="EjbRef_2">
  <ejb-ref-name>ejb/HumanTaskManagerEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.task.api.HumanTaskManagerHome</home>
  <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
<ejb-local-ref id="EjbLocalRef_1">
  <ejb-ref-name>ejb/LocalBusinessProcessHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
  <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
</ejb-local-ref>
<ejb-local-ref id="EjbLocalRef_2">
  <ejb-ref-name>ejb/LocalHumanTaskManagerEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
  <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>
```

4. Business Process Choreographer Explorer JSF コンポーネントを JSF アプリケーションに追加します。
   a. アプリケーションに必要なタグ・ライブラリー参照を JavaServer Pages (JSP) ファイルに追加します。 通常、JSF および HTML タグ・ライブラリーと、JSF コンポーネントに必要とされるタグ・ライブラリーが必要です。
      - `<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>`
      - `<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>`
      - `<%@ taglib uri="http://com.ibm.bpe.jsf/taglib" prefix="bpe" %>`

   b. JSP ページの本体に `<f:view>` タグを追加し、`<f:view>` タグに `<h:form>` タグを追加します。
   c. JSP ファイルに JSF コンポーネントを追加します。

      アプリケーションに応じて、List コンポーネント、Details コンポーネント、CommandBar コンポーネント、または Message コンポーネントを JSP ファイル内に追加します。各コンポーネントの複数のインスタンスを追加することができます。
   d. JSF 構成ファイル内で管理対象 Bean を構成します。

      デフォルトでは、構成ファイルは faces-config.xml ファイルです。このファイルは、Web アプリケーションの WEB-INF ディレクトリーにあります。

JSP ファイルに追加するコンポーネントに応じて、照会およびその他のラッパー・オブジェクトへの参照を JSF 構成ファイルに追加する必要もあります。的確なエラー処理が行われるようにするには、JSF 構成ファイルで、エラー Bean とナビゲーション・ターゲットの両方をエラー・ページ用に定義する必要もあります。

```
<faces-config>
...
<managed-bean>
  <managed-bean-name>BPCError</managed-bean-name>
  <managed-bean-class>com.ibm.bpc.clientcore.util.ErrorBeanImpl
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

...
<navigation-rule>
...
<navigation-case>
<description>
The general error page.
</description>
<from-outcome>error</from-outcome>
<to-view-id>/Error.jsp</to-view-id>
</navigation-case>
...
</navigation-rule>
</faces-config>
```

エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。

e. JSF コンポーネントをサポートするために必要なカスタム・コードをインプリメントします。

5. アプリケーションをデプロイします。

アプリケーションを Network Deployment 環境でデプロイしている場合、ターゲット・リソースの Java Naming and Directory Interface (JNDI) 名を、Business Flow Manager および Human Task Manager API をセル内で検出するための値に変更してください。

- ビジネス・プロセス・コンテナーが同じ管理対象セル内の別のサーバー上で構成されている場合、名前には以下の構造があります。

  cell/*nodes*/*nodename*/*servers*/*servername*/com/ibm/bpe/api/BusinessManagerHome
  cell/*nodes*/*nodename*/*servers*/*servername*/com/ibm/task/api/HumanTaskManagerHome

- ビジネス・プロセス・コンテナーが同じセル内のクラスターで構成されている場合、名前には以下の構造があります。

  cell/*clusters*/*clustername*/com/ibm/bpe/api/BusinessFlowManagerHome
  cell/*clusters*/*clustername*/com/ibm/task/api/HumanTaskManagerHome

EJB 参照を JNDI 名へマップするか、参照を ibm-web-bnd.xmi ファイルへ手動で追加します。

以下の表に、参照バインディングおよびそのデフォルト・マッピングを示します。

表 33. 参照バインディングから JNDI 名へのマッピング

参照バインディング	JNDI 名	コメント
ejb/BusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	リモート・セッション Bean
ejb/LocalBusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	ローカル・セッション Bean
ejb/HumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	リモート・セッション Bean
ejb/LocalHumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	ローカル・セッション Bean

結果

デプロイした Web アプリケーションには、Business Process Choreographer Explorer コンポーネントが提供する機能が含まれています。

次のタスク

プロセスおよびタスク・メッセージにカスタム JSP を使用している場合、JSP をデプロイするために使用される Web モジュールを、カスタム JSF クライアントがマップされるのと同じサーバーにマップする必要があります。

関連概念

165 ページの『JSF コンポーネントでのエラー処理』
JavaServer Faces (JSF) コンポーネントはエラー処理の際に、事前定義された管理対象 Bean である BPCError を活用します。エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。

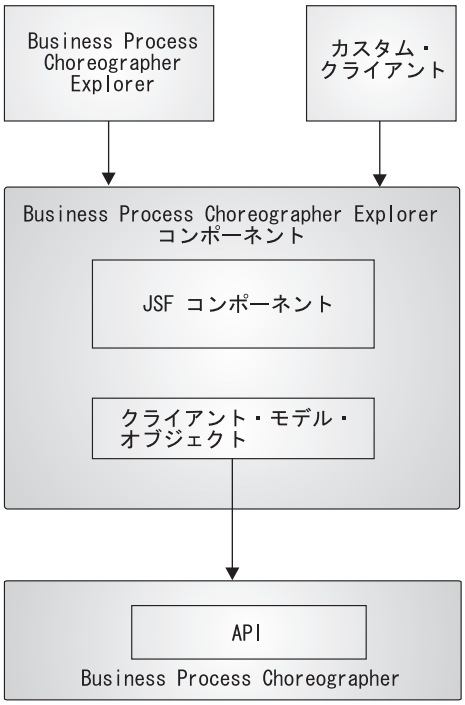
関連タスク

24 ページの『セッション Bean のリモート・インターフェースにアクセスする』
EJB クライアント・アプリケーションは、Bean のリモート・ホーム・インターフェースを介して、セッション Bean のリモート・インターフェースにアクセスします。

Business Process Choreographer Explorer コンポーネント

Business Process Choreographer Explorer コンポーネントは、JavaServer Faces (JSF) テクノロジーに基づく構成可能かつ再使用可能なエレメントの集合です。これらのエレメントを Web アプリケーションに組み込むことができます。これにより、Web アプリケーションは、インストール済みビジネス・プロセスおよびヒューマン・タスク・アプリケーションにアクセスできるようになります。

コンポーネントは、JSF コンポーネントのセットおよびクライアント・モデル・オブジェクトのセットで構成されています。コンポーネントから Business Process Choreographer、Business Process Choreographer Explorer、およびその他のカスタム・クライアントへの関係を、次の図に示します。



JSF コンポーネント

Business Process Choreographer Explorer コンポーネントには、以下の JSF コンポーネントが含まれます。ビジネス・プロセスおよびヒューマン・タスクを操作するための Web アプリケーションをビルドするときに、これらの JSF コンポーネントを JavaServer Pages (JSP) ファイルに組み込みます。

- List コンポーネント

  List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、またはエスカレーションなどの、アプリケーション・オブジェクトのリストをテーブル内に表示します。このコンポーネントには、関連付けられたリスト・ハンドラーがあります。

- Details コンポーネント

  Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。このコンポーネントには、関連付けられた詳細ハンドラーがあります。

- CommandBar コンポーネント

  CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトのいずれかに作動するコマンドを表します。これらのオブジェクトは、リスト・ハンドラーまたは詳細ハンドラーによって提供されます。

- Message コンポーネント

  Message コンポーネントは、サービス・データ・オブジェクト (SDO) または単純型のいずれかを含むことのできるメッセージを表示します。

クライアント・モデル・オブジェクト

クライアント・モデル・オブジェクトは、JSF コンポーネントと共に使用されます。これらのオブジェクトは、基盤となる Business Process Choreographer API のインターフェースの一部をインプリメントし、元のオブジェクトをラップします。クライアント・モデル・オブジェクトは、ラベルの各国語サポートと、一部のプロパティーのコンバーターを提供します。

JSF コンポーネントでのエラー処理

JavaServer Faces (JSF) コンポーネントはエラー処理の際に、事前定義された管理対象 Bean である BPCError を活用します。エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。

この Bean は、com.ibm.bpc.clientcore.util.ErrorBean インターフェースをインプリメントします。エラー・ページが表示されるのは、次のような状態のときです。

- リスト・ハンドラー用に定義された照会の実行中にエラーが発生し、エラーがコマンドの execute メソッドによって ClientException エラーとして生成された場合
- ClientException エラーがコマンドの execute メソッドによって生成され、このエラーが ErrorsInCommandException エラーではなく、 CommandBarMessage インターフェースのインプリメントもしない場合
- コンポーネント内でエラー・メッセージが表示され、メッセージのハイパーリンクに従っている場合

com.ibm.bpc.clientcore.util.ErrorBeanImpl インターフェースのデフォルト・インプリメンテーションを使用できます。

インターフェースは次のように定義されます。

```
public interface ErrorBean {

    public void setException(Exception ex);

    /*
     * This setter method call allows a locale and
     * the exception to be passed. This allows the
     * getExceptionMessage methods to return localized Strings
     *
     */
    public void setException(Exception ex, Locale locale);

    public Exception getException();
    public String getStack();
    public String getNestedExceptionMessage();
    public String getNestedExceptionStack();
    public String getRootExceptionMessage();
    public String getRootExceptionStack();

    /*
     * This method returns the exception message
     * concatenated recursively with the messages of all
     * the nested exceptions.
     */
    public String getAllExceptionMessages();

    /*
     * This method is returns the exception stack
     * concatenated recursively with the stacks of all
```

```
     * the nested exceptions.
     */
    public String getAllExceptionStacks();
}
```

関連概念

171 ページの『List コンポーネントでのエラー処理』
List コンポーネントを使用して、JSF アプリケーション内のリストを表示する場合、com.ibm.bpe.jsf.handler.BPCListHandler クラスが提供するエラー処理機能を利用できます。

クライアント・モデル・オブジェクトのデフォルトのコンバーターおよびラベル

クライアント・モデル・オブジェクトは、Business Process Choreographer API の対応するインターフェースをインプリメントします。

List コンポーネントと Details コンポーネントはあらゆる Bean で機能します。Bean のプロパティーをすべて表示できます。ただし、Bean のプロパティーに使用されるコンバーターとラベルを設定する場合は、 List コンポーネントの `column` タグまたは Details コンポーネントの `property` タグを使用する必要があります。コンバーターとラベルを設定する代わりに、プロパティーのデフォルトのコンバーターとラベルを定義できます。これらを定義するには、次の静的メソッドを定義します。定義できる静的メソッドを次に示します。

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
       getConverter(String property);
```

次の表に、対応する Business Flow Manager API クラスと Human Task Manager API クラスを実装し、そのプロパティーにデフォルトのラベルとコンバーターを提供するクライアント・モデル・オブジェクトを示します。インターフェースをこのようにラップすることにより、ロケールを区別するラベルと、プロパティーのセット用のコンバーターが提供されます。以下の表に、Business Process Choreographer インターフェースから対応するクライアント・モデル・オブジェクトへのマッピングを示します。

表 34. Business Process Choreographer インターフェースからクライアント・モデル・オブジェクトへのマッピング

Business Process Choreographer インターフェース	クライアント・モデル・オブジェクト・クラス
com.ibm.bpe.api.ActivityInstanceData	com.ibm.bpe.clientmodel.bean.ActivityInstanceBean
com.ibm.bpe.api.ActivityServiceTemplateData	com.ibm.bpe.clientmodel.bean.ActivityServiceTemplateBean
com.ibm.bpe.api.ProcessInstanceData	com.ibm.bpe.clientmodel.bean.ProcessInstanceBean
com.ibm.bpe.api.ProcessTemplateData	com.ibm.bpe.clientmodel.bean.ProcessTemplateBean
com.ibm.task.api.Escalation	com.ibm.task.clientmodel.bean.EscalationBean
com.ibm.task.api.Task	com.ibm.task.clientmodel.bean.TaskInstanceBean
com.ibm.task.api.TaskTemplate	com.ibm.task.clientmodel.bean.TaskTemplateBean

JSF アプリケーションへの List コンポーネントの追加

Business Process Choreographer Explorer List コンポーネントを使用して、例えばビジネス・プロセス・インスタンスやタスク・インスタンスなどの、クライアント・モデル・オブジェクトのリストを表示します。

プロシージャー

1. List コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   bpe:list タグを h:form タグに追加します。 bpe:list タグには、モデル属性が含まれていなければなりません。bpe:column タグを bpe:list に追加して、リスト内の各行に表示されるオブジェクトのプロパティーを追加します。

   以下の例では、List コンポーネントを追加してタスク・インスタンスを表示する方法を示します。

   ```
   <h:form>

      <bpe:list model="#{TaskPool}">
         <bpe:column name="name" action="taskInstanceDetails" />
         <bpe:column name="state" />
         <bpe:column name="kind" />
         <bpe:column name="owner" />
         <bpe:column name="originator" />
      </bpe:list>

   </h:form>
   ```

   モデル属性は、TaskPool という管理対象 Bean を参照します。管理対象 Bean は、リストが操作を繰り返す対象となる Java オブジェクトのリストを提供し、次に個々の行を表示します。

2. bpe:list タグで参照されている管理対象 Bean を構成します。

   List コンポーネントの場合、この管理対象 Bean は、com.ibm.bpe.jsf.handler.BPCListHandler クラスのインスタンスでなければなりません。

   以下の例では、TaskPool 管理対象 Bean を構成ファイルに追加する方法を示します。

   ```
   <managed-bean>
   <managed-bean-name>TaskPool</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.jsf.handler.BPCListHandler</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
      <managed-property>
         <property-name>query</property-name>
         <value>#{TaskPoolQuery}</value>
      </managed-property>
      <managed-property>
         <property-name>type</property-name>
         <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
      </managed-property>
   </managed-bean>

   <managed-bean>
   <managed-bean-name>TaskPoolQuery</managed-bean-name>
   <managed-bean-class>sample.TaskPoolQuery</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
      <managed-property>
        <property-name>type</property-name>
   ```

```
     <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>
</managed-bean>

<managed-bean>
<managed-bean-name>htmConnection</managed-bean-name>
<managed-bean-class>com.ibm.task.clientmodel.HTMConnection</managed-bean-class>
<managed-bean-scope>application</managed-bean-scope>
   <managed-property>
      <property-name>jndiName</property-name>
      <value>java:comp/env/ejb/LocalHumanTaskManagerEJB</value>
   </managed-property>
</managed-bean>
```

例では、照会およびタイプの 2 つの構成可能プロパティーが TaskPool に含まれていることを示しています。照会プロパティーの値は、TaskPoolQuery という別の管理対象 Bean を参照しています。タイプ・プロパティーの値は、Bean クラスを指定します。そのクラスのプロパティーは、表示されたリストの列に示されます。関連する照会インスタンスは、プロパティー型を持つことも可能です。プロパティー型が指定された場合、それはリスト・ハンドラーに指定された型と同一でなければなりません。

照会の結果を強い型の Bean のリストとして表すことができる限り、任意のタイプの照会ロジックを JSF アプリケーションに追加できます。例えば、TaskPoolQuery は com.ibm.task.clientmodel.bean.TaskInstanceBean オブジェクトのリストを使用して実装されます。

3. リスト・ハンドラーによって参照される管理対象 Bean 用のカスタム・コードを追加します。

   以下の例では、TaskPool 管理対象 Bean 用のカスタム・コードを追加する方法を示します。

```
public class TaskPoolQuery implements Query {

  public List execute throws ClientException {

    // Examine the faces-config file for a managed bean "htmConnection".
    //
    FacesContext ctx = FacesContext.getCurrentInstance();
    Application  app = ctx.getApplication();
    ValueBinding htmVb = app.createValueBinding("#{htmConnection}");
    htmConnection = (HTMConnection) htmVb.getValue(ctx);
    HumanTaskManagerService taskService =
         htmConnection.getHumanTaskManagerService();

    // Then call the actual query method on the Human Task Manager service.
    //
    QueryResultSet queryResult = taskService.query(
     "DISTINCT TASK.TKIID, TASK.NAME, TASK.KIND, TASK.STATE, TASK.TYPE,"
      + "TASK.STARTED, TASK.ACTIVATED, TASK.DUE, TASK.EXPIRES, TASK.PRIORITY" ,
      "TASK.KIND IN(101,102,105) AND TASK.STATE IN(2)
        AND WORK_ITEM.REASON IN (1)",
      (String)null,
      (Integer)null,
      (TimeZone)null);
    List applicationObjects = transformToTaskList ( queryResult );
    return applicationObjects ;
  }


  private List transformToTaskList(QueryResultSet result) {
```

```
ArrayList array = null;
int entries = result.size();
array = new ArrayList( entries );

// Transforms each row in the QueryResultSet to a task instance beans.
 for (int i = 0; i < entries; i++) {
   result.next();
   array.add( new TaskInstanceBean( result, connection ));
 }
 return array ;
  }
}
```

TaskPoolQuery Bean は、Java オブジェクトのプロパティーを照会します。この Bean は、com.ibm.bpc.clientcore.Query インターフェースをインプリメントする必要があります。リスト・ハンドラーは、内容を最新表示するときに、照会の execute メソッドを呼び出します。呼び出しによって、Java オブジェクトのリストが戻されます。getType メソッドは、戻された Java オブジェクトのクラス名を戻す必要があります。

結果

これで、JSF アプリケーションは、例えば状態、種類、所有者、ユーザーが使用可能なタスク・インスタンスのオリジネーターなどの、要求されたオブジェクトのリストのプロパティーを表示する JavaServer ページを含むようになります。

関連概念

171 ページの『ユーザー固有の時間帯情報』
JavaServer Faces (JSF) コンポーネントは、List コンポーネントのユーザー指定の時間帯情報を処理するためのユーティリティーを提供します。

関連資料

172 ページの『List コンポーネント: タグ定義』
Business Process Choreographer Explorer List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、およびエスカレーションなどの、オブジェクトのリストをテーブル内に表示します。

リストの処理方法

List コンポーネントのインスタンスはすべて com.ibm.bpe.jsf.handler.BPCListHandler クラスのインスタンスに関連しています。

このリスト・ハンドラーは関連するリスト内で選択された項目をトラッキングし、さまざまな種類の項目用の詳細ページにリスト項目を関連付ける通知メカニズムを提供します。リスト・ハンドラーは、`bpe:list` タグの**モデル**属性を介して List コンポーネントにバインドされます。

リスト・ハンドラーの通知メカニズムは、 com.ibm.bpe.jsf.handler.ItemListener インターフェースを使用してインプリメントされます。このインターフェースの実装は、JavaServer Faces (JSF) アプリケーションの構成ファイルに登録できます。

リスト内のリンクがクリックされると、通知がトリガーされます。 **action** 属性が設定されているすべての列についてリンクがレンダリングされます。 **action** 属性の値

は JSF ナビゲーション・ターゲットか、JSF ナビゲーション・ターゲットを戻す JSF アクション・メソッドのいずれかです。

BPCListHandler クラスは、refreshList メソッドを提供します。このメソッドを JSF メソッド・バインディングで使用して、照会を再実行するためのユーザー・インターフェース制御をインプリメントすることができます。

照会のインプリメンテーション

リスト・ハンドラーを使用すると、すべての種類のオブジェクトおよびそれらのプロパティーを表示できます。表示されるリストの内容は、リスト・ハンドラー用に構成された com.ibm.bpc.clientcore.Query インターフェースのインプリメンテーションによって戻されるオブジェクトのリストによって異なります。照会は、BPCListHandler クラスの setQuery メソッドを使用してプログラマチックに設定することもできますし、アプリケーションの JSF 構成ファイルで構成することもできます。

照会は、Business Process Choreographer API に対してだけでなく、コンテンツ管理システムやデータベースなど、アプリケーションからアクセスできるその他の情報ソースに対しても実行できます。要件は、照会の結果が execute メソッドでオブジェクトの `java.util.List` として戻されることだけです。

戻されるオブジェクトのタイプは、照会が定義されたリストの列に表示されるすべてのプロパティーに対して適切な getter メソッドを使用できることを保証する必要があります。戻されるオブジェクトのタイプがリスト定義に適合することを確認するには、 faces 構成ファイルで定義されている BPCListHandler インスタンス上のタイプ・プロパティーの値を、戻されるオブジェクトの完全修飾クラス名に設定します。この名前は、照会インプリメンテーションの getType 呼び出しで戻すことができます。実行時に、リスト・ハンドラーはオブジェクト・タイプが定義に準拠していることを確認します。

リスト内の特定の項目にエラー・メッセージをマップするには、照会によって戻されたオブジェクトが署名 `public Object getID()` でメソッドをインプリメントする必要があります。

デフォルトのコンバーターおよびラベル

照会によって戻される項目は Bean である必要があり、そのクラスは BPCListHandler クラスまたは com.ibm.bpc.clientcore.Query インターフェースの定義でタイプとして指定されたクラスと一致している必要があります。さらに、List コンポーネントは、項目クラスまたはスーパークラスが以下のメソッドを実装しているかどうかを検査します。

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
       getConverter(String property);
```

これらのメソッドが Bean に対して定義されている場合、List コンポーネントはリストのデフォルト・ラベルとして label を使用し、プロパティーのデフォルト・コンバーターとして SimpleConverter を使用します。これらの設定は、`bpe:list` タグの **label** および **converterID** 属性で上書きできます。詳しくは、SimpleConverter インターフェースおよび ColumnTag クラスの Javadoc を参照してください。

ユーザー固有の時間帯情報

JavaServer Faces (JSF) コンポーネントは、List コンポーネントのユーザー指定の時間帯情報を処理するためのユーティリティーを提供します。

BPCListHandler クラスは、 com.ibm.bpc.clientcore.util.User インターフェースを使用して、各ユーザーの時間帯およびロケールに関する情報を取得します。List コンポーネントは、JavaServer Faces (JSF) 構成ファイルでインターフェースのインプリメンテーションが **user** を管理対象 Bean 名として設定されていると予期します。構成ファイル内でこの項目が欠けている場合は、WebSphere Process Server が動作している時間帯が戻されます。

com.ibm.bpc.clientcore.util.User インターフェースは次のように定義されています。

```
public interface User {

    /**
     * The locale used by the client of the user.
     * @return Locale.
     */
    public Locale getLocale();
   /**
    * The time zone used by the client of the user.
    * @return TimeZone.
    */
    public TimeZone getTimeZone();

   /**
    * The name of the user.
    * @return name of the user.
    */
    public String getName();
}
```

List コンポーネントでのエラー処理

List コンポーネントを使用して、JSF アプリケーション内のリストを表示する場合、com.ibm.bpe.jsf.handler.BPCListHandler クラスが提供するエラー処理機能を利用できます。

照会の実行時またはコマンドの実行時に発生するエラー

照会の実行中にエラーが発生した場合、 BPCListHandler クラスは、不十分なアクセス権限によるエラーとその他の例外とを区別します。不十分なアクセス権限によるエラーをキャッチするには、照会の execute メソッドによってスローされる ClientException の **rootCause** パラメーターが com.ibm.bpe.api.EngineNotAuthorizedException または com.ibm.task.api.NotAuthorizedException 例外である必要があります。List コンポーネントは、照会の結果の代わりにエラー・メッセージを表示します。

エラーが不十分なアクセス権限によるものでない場合、 BPCListHandler クラスは例外オブジェクトを、 JSF アプリケーション構成ファイルの BPCError キーで定義した com.ibm.bpc.clientcore.util.ErrorBean インターフェースのインプリメンテーションに渡します。例外が設定されている場合は、エラー・ナビゲーション・ターゲットが呼び出されます。

リストに表示される項目の処理時に発生するエラー

BPCListHandler クラスは、 com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントします。setErrors メソッドでタイプ java.util.Map のマップ・パラメーターを使用して、これらのエラーに関する情報を提供することができます。このマップには、キーとして ID が、値として例外が含まれています。 ID は、エラーの原因となったオブジェクトの getID メソッドによって戻された値である必要があります。マップが設定されていて、リスト内に表示されている項目のいずれかに ID のいずれかが一致する場合は、リスト・ハンドラーによって、エラー・メッセージを含む列が自動的にリストに追加されます。

リスト内のエラー・メッセージが古くなるのを避けるため、エラー・マップをリセットしてください。次の状況では、マップは自動的にリセットされます。

- refreshList メソッドの BPCListHandler クラスが呼び出される。
- BPCListHandler クラスで新規照会が設定されている。
- CommandBar コンポーネントを使用して、リストの項目でアクションがトリガーされている。CommandBar コンポーネントは、エラー処理のメソッドの 1 つとしてこのメカニズムを使用します。

関連概念

165 ページの『JSF コンポーネントでのエラー処理』
JavaServer Faces (JSF) コンポーネントはエラー処理の際に、事前定義された管理対象 Bean である BPCError を活用します。エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。

List コンポーネント: タグ定義

Business Process Choreographer Explorer List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、およびエスカレーションなどの、オブジェクトのリストをテーブル内に表示します。

List コンポーネントは、JSF コンポーネント・タグである `bpe:list` と `bpe:column` から構成されます。`bpe:column` タグは、`bpe:list` タグのサブエレメントです。

コンポーネント・クラス

com.ibm.bpe.jsf.component.ListComponent

構文例

```
<bpe:list model="#{ProcessTemplateList}">
          rows="20"
          styleClass="list"
          headerStyleClass="listHeader"
          rowClasses="normal">

    <bpe:column name="name" action="processTemplateDetails"/>
    <bpe:column name="validFromTime"/>
    <bpe:column name="executionMode" label="Execution mode"/>
    <bpe:column name="state" converterID="my.state.converter"/>
    <bpe:column name="autoDelete"/>
    <bpe:column name="description"/>

</bpe:list>
```

タグ属性

bpe:list タグの本体には、bpe:column タグのみを含めることができます。テーブルがレンダリングされるとき、List コンポーネントは、アプリケーション・オブジェクトのリストで処理を繰り返し、オブジェクトごとにすべての列をレンダリングします。

表 35. bpe:list 属性

属性	必須	説明
buttonStyleClass	いいえ	フッター領域内のボタンのレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル・クラス。
cellStyleClass	いいえ	個々のテーブル・セルのレンダリング用の CSS スタイル・クラス。
checkbox	いいえ	複数の項目を選択するためのチェック・ボックスを提供するかどうかを決定します。この属性には true または false のいずれかの値が使用されます。値が true に設定されている場合は、チェック・ボックス列がレンダリングされます。
headerStyleClass	いいえ	テーブル・ヘッダーのレンダリング用の CSS スタイル・クラス。
model	はい	com.ibm.bpe.jsf.handler.BPCListHandler クラスの管理対象 Bean 用の値バインディング。
rows	いいえ	ページに表示される行数。項目数が行数を超える場合は、テーブルの最後にページ送りボタンが表示されます。この属性では、値の式はサポートされていません。
rowClasses	いいえ	テーブル内の行のレンダリング用の CSS スタイル・クラス。
selectAll	いいえ	この属性が true に設定されている場合は、リスト内のすべての項目がデフォルトで選択されます。
styleClass	いいえ	タイトル、行、およびページ送りボタンを含むテーブル全体のレンダリングの CSS スタイル・クラス。

表 36. bpe:column 属性

属性	必須	説明
action	いいえ	この属性が指定されている場合は、列でリンクがレンダリングされます。リンクがクリックされると、JavaServer Faces アクション・メソッドまたは Faces ナビゲーション・ターゲットが起動されます。シグニチャーが String method() である JavaServer Faces アクション・メソッド。

表 36. bpe:column 属性 (続き)

属性	必須	説明
converterID	いいえ	プロパティー値の変換に使用する Faces コンバーター ID。この属性が設定されていない場合、モデルによりこのプロパティーに設定された Faces コンバーター ID が使用されます。
label	いいえ	列のヘッダーのラベルまたはテーブル・ヘッダー行のセルのラベルとして使用されるリテラルまたは値バインディング式。この属性が設定されていない場合、モデルによりこのプロパティーに設定されたラベルが使用されます。
name	はい	この列に表示されるプロパティーの名前。

JSF アプリケーションへの Details コンポーネントの追加

Business Process Choreographer Explorer Details コンポーネントを使用して、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

プロシージャー

1. Details コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   `bpe:details` タグを `<h:form>` タグに追加します。`bpe:details` タグには、**model** 属性が含まれていなければなりません。`bpe:property` タグを使用して Details コンポーネントにプロパティーを追加することができます。

   以下の例では、Details コンポーネントを追加して、タスク・インスタンスのプロパティーのいくつかを表示する方法を示します。

   ```
   <h:form>

      <bpe:details model="#{TaskInstanceDetails}">
         <bpe:property name="displayName" />
         <bpe:property name="owner" />
         <bpe:property name="kind" />
         <bpe:property name="state" />
         <bpe:property name="escalated" />
         <bpe:property name="suspended" />
         <bpe:property name="originator" />
         <bpe:property name="activationTime" />
         <bpe:property name="expirationTime" />
      </bpe:details>

   </h:form>
   ```

   model 属性は、TaskInstanceDetails という管理対象 Bean を参照します。Bean は、Java オブジェクトのプロパティーを提供します。

2. `bpe:details` タグで参照されている管理対象 Bean を構成します。

   Details コンポーネントの場合、この管理対象 Bean は、com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスのインスタンスでなければなり

ません。このハンドラー・クラスは、Java オブジェクトをラップし、そのパブリック・プロパティーを Details コンポーネントに公開します。

以下の例では、TaskInstanceDetails 管理対象 Bean を構成ファイルに追加する方法を示します。

```
<managed-bean>
   <managed-bean-name>TaskInstanceDetails</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.jsf.handler.BPCDetailsHandler</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>type</property-name>
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>
</managed-bean>
```

例では、TaskInstanceDetails Bean に構成可能な `type` プロパティーが含まれることを示しています。タイプ・プロパティーの値は、Bean クラス (com.ibm.task.clientmodel.bean.TaskInstanceBean) を指定します。そのクラスのプロパティーは、表示された詳細の行に示されます。Bean クラスは任意の JavaBeans クラスにすることができます。Bean がデフォルト・コンバーターおよびプロパティー・ラベルを提供する場合、そのコンバーターおよびラベルが、List コンポーネントの場合と同じようにしてレンダリングに使用されます。

結果

これで、JSF アプリケーションは、例えばタスク・インスタンスの詳細などの、指定されたオブジェクトの詳細を表示する JavaServer ページを含むようになります。

関連資料

『Details コンポーネント: タグ定義』

Business Process Choreographer Explorer Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

Details コンポーネント: タグ定義

Business Process Choreographer Explorer Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

Details コンポーネントは、JSF コンポーネント・タグである `bpe:details` と `bpe:property` から構成されます。`bpe:property` タグは、`bpe:details` タグのサブエレメントです。

コンポーネント・クラス

com.ibm.bpe.jsf.component.DetailsComponent

構文例

```
<bpe:details model="#{MyActivityDetails}">
    <bpe:property name="name"/>
    <bpe:property name="owner"/>
    <bpe:property name="activated"/>
</bpe:details>
```

```
<bpe:details model="#{MyActivityDetails}" style="style" styleClass="cssStyle">
             style="style"
             styleClass="cssStyle"
</bpe:details>
```

タグ属性

bpe:property タグを使用して、表示される属性のサブセットおよびこれらの属性が表示される順序の両方を指定します。詳細タグに属性タブが含まれていない場合、モデル・オブジェクトの使用可能な属性がすべてレンダリングされます。

表 37. bpe:details 属性

属性	必須	説明
columnClasses	いいえ	列のレンダリング用のカスケーディング・スタイル・シート (CSS) のスタイル・クラスをコンマで区切ったリスト。
id	いいえ	コンポーネントの JavaServer Faces ID。
model	はい	com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスの管理対象 Bean 用の値バインディング。
rowClasses	いいえ	行のレンダリング用の、コンマで区切られた CSS スタイル・クラスのリスト。
styleClass	いいえ	HTML エレメントのレンダリングに使用される CSS クラス。

表 38. bpe:property 属性

属性	必須	説明
converterID	いいえ	JavaServer Faces (JSF) 構成ファイルでコンバーターを登録するために使用される ID。
label	いいえ	プロパティーのラベル。この属性が設定されていない場合、クライアント・モデル・クラスによってデフォルト・ラベルが提供されます。
name	はい	表示されるプロパティーの名前。この名前は、対応するクライアント・モデル・クラスで定義されているように、名前付きプロパティーに対応していなければなりません。

JSF アプリケーションへの CommandBar コンポーネントの追加

Business Process Choreographer Explorer CommandBar コンポーネントを使用して、ボタンを含むバーを表示します。これらのボタンは、オブジェクトの詳細ビューまたはリスト内の選択されたオブジェクトで作動するコマンドを表します。

このタスクについて

ユーザーがユーザー・インターフェースのボタンをクリックすると、対応するコマンドが選択されたオブジェクトで実行されます。JSF アプリケーション内の CommandBar コンポーネントを追加および拡張することができます。

プロシージャー

1. CommandBar コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   bpe:commandbar タグを <h:form> タグに追加します。bpe:commandbar タグには、モデル属性が含まれていなければなりません。

   以下の例では、タスク・インスタンス・リストに refresh コマンドと claim コマンドを提供する CommandBar コンポーネントを追加する方法を示します。

   ```
   <h:form>

      <bpe:commandbar model="#{TaskInstanceList}">
         <bpe:command commandID="Refresh" >
                      action="#{TaskInstanceList.refreshList}"
                      label="Refresh"/>

         <bpe:command commandID="MyClaimCommand" >
                      label="Claim" >
                      commandClass="<customcode>"/>
      </bpe:commandbar>

   </h:form>
   ```

   model 属性は、管理対象 Bean を参照します。この Bean は、ItemProvider インターフェースをインプリメントし、選択された Java オブジェクトを提供する必要があります。CommandBar コンポーネントは、通常、同一の JSP ファイル内の List コンポーネントまたは Details コンポーネントのいずれかと共に使用されます。一般に、タグで指定されたモデルは、同一ページの List コンポーネントまたは Details コンポーネントで指定されたモデルと同じです。そのため、例えば List コンポーネントの場合、コマンドはリスト内の選択された項目に対して作動します。

   この例では、 **model** 属性は TaskInstanceList 管理対象 Bean を参照します。この Bean は、タスク・インスタンス・リストの選択されたオブジェクトを提供します。この Bean は、ItemProvider インターフェースをインプリメントする必要があります。このインターフェースは、BPCListHandler クラスおよび BPCDetailsHandler クラスによってインプリメントされます。

2. オプション: bpe:commandbar タグで参照されている管理対象 Bean を構成します。

   CommandBar **model** 属性が、例えばリスト・ハンドラーまたは詳細ハンドラー用に既に構成済みの管理対象 Bean を参照する場合、それ以上の構成は必要ありません。これらのハンドラーのいずれかの構成を変更する場合、または異なる管理対象 Bean を使用する場合は、ItemProvider インターフェースをインプリメントする管理対象 Bean を JSF 構成ファイルに追加してください。

3. カスタム・コマンドをインプリメントするコードを JSF アプリケーションに追加します。

   以下のコード断片は、Command インターフェースを実装するコマンド・クラスの作成方法を示します。このコマンド・クラス (MyClaimCommand) は、JSP ファイル内の bpe:command タグで参照されます。

   ```
   public class MyClaimCommand implements Command {

     public String execute(List selectedObjects) throws ClientException {
   ```

```
    if( selectedObjects != null && selectedObjects.size() > 0 ) {
       try {
        // Determine HumanTaskManagerService from an HTMConnection bean.
        // Configure the bean in the faces-config.xml for easy access
        // in the JSF application.
        FacesContext ctx = FacesContext.getCurrentInstance();
        ValueBinding vb =
          ctx.getApplication().createValueBinding("{htmConnection}");
        HTMConnection htmConnection = (HTMConnection) htmVB.getValue(ctx);
        HumanTaskManagerService htm =
           htmConnection.getHumanTaskManagerService();

        Iterator iter = selectedObjects.iterator() ;
        while( iter.hasNext() ) {
          try {
               TaskInstanceBean task = (TaskInstanceBean) iter.next() ;
               TKIID tiid = task.getID() ;

               htm.claim( tiid ) ;
               task.setState( new Integer(TaskInstanceBean.STATE_CLAIMED ) ) ;

          }
          catch( Exception e ) {
            ;   // Error while iterating or claiming task instance.
                 // Ignore for better understanding of the sample.
          }
        }
      }
      catch( Exception e ) {
        ;  // Configuration or communication error.
          // Ignore for better understanding of the sample
      }
    }
    return null;
  }

 // Default implementations
 public boolean isMultiSelectEnabled() { return false; }
 public boolean[] isApplicable(List itemsOnList) {return null; }
 public void setContext(Object targetModel) {; // Not used here }
}
```

コマンドは以下のように処理されます。

a. ユーザーがコマンド・バーの対応するボタンをクリックすると、コマンドが起動されます。CommandBar コンポーネントは、**model** 属性で指定された項目プロバイダーから選択された項目を検索し、選択されたオブジェクトのリストを commandClass インスタンスの execute メソッドに渡します。

b. **commandClass** 属性は、コマンド・インターフェースをインプリメントするカスタム・コマンド・インプリメンテーションを参照します。つまりこのコマンドは、public String execute(List selectedObjects) throws ClientException メソッドをインプリメントする必要があります。コマンドが戻した結果は、JSF アプリケーションの次のナビゲーション規則を決定するために使用されます。

c. コマンドの完了後、CommandBar コンポーネントは **action** 属性を評価します。**action** 属性は、静的ストリングである場合も、`public String Method()` というシグニチャーの JSF アクション・メソッドへのメソッド・バインディングである場合もあります。**action** 属性を使用して、コマンド・クラスの結果をオーバーライドするか、またはナビゲーション規則の結果を明示的に指

定します。コマンドが ErrorsInCommandException 例外以外の例外を生成した場合、 **action** 属性は処理されません。

d. **commandClass** 属性にコマンド・クラスが指定されていない場合、アクションが即時に呼び出されます。例えば、例の中のリフレッシュ・コマンドの場合、JSF 値式 `#{TaskInstanceList.refreshList}` がコマンドの代わりに呼び出されます。

結果

これで、JSF アプリケーションは、カスタマイズされたコマンド・バーをインプリメントする JavaServer ページを含むようになります。

関連資料

180 ページの『CommandBar コンポーネント: タグ定義』

Business Process Choreographer Explorer CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトに作動します。

コマンドの処理方法

CommandBar コンポーネントを使用して、アプリケーションにアクション・ボタンを追加します。コンポーネントは、ユーザー・インターフェースでのアクション用のボタンを作成し、ボタンがクリックされたときに作成されるイベントを処理します。

これらのボタンは、BPCListHandler クラスや BPCDetailsHandler クラスなど、com.ibm.bpe.jsf.handler.ItemProvider インターフェースによって戻されるオブジェクトで動作する機能を起動します。 CommandBar コンポーネントは、 `bpe:commandbar` タグで**モデル**属性の値によって定義された項目プロバイダーを使用します。

アプリケーションのユーザー・インターフェースのコマンド・バー・セクションにあるボタンをクリックすると、関連するイベントが CommandBar コンポーネントによって次のように処理されます。

1. CommandBar コンポーネントは、イベントを生成したボタンに対して指定された com.ibm.bpc.clientcore.Command インターフェースのインプリメンテーションを示します。
2. CommandBar コンポーネントに関連するモデルが com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントすると、前のイベントからのエラー・メッセージを削除するため、clearErrorMap メソッドが呼び出されます。
3. ItemProvider インターフェースの getSelectedItems メソッドが呼び出されます。戻された項目のリストは、コマンドの execute メソッドに渡され、コマンドが呼び出されます。
4. CommandBar コンポーネントは、 JavaServer Faces (JSF) ナビゲーション・ターゲットを決定します。`bpe:commandbar` タグで**アクション**属性が指定されていない場合は、execute メソッドの戻り値によってナビゲーション・ターゲットが指定されます。**アクション**属性が JSF メソッド・バインディングに設定されている場合は、メソッドによって戻されたストリングがナビゲーション・ターゲットと解釈されます。**アクション**属性は、明示的なナビゲーション・ターゲットも指定します。

CommandBar コンポーネント: タグ定義

Business Process Choreographer Explorer CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトに作動します。

CommandBar コンポーネントは、JSF コンポーネント・タグである bpe:commandbar と bpe:command から構成されます。 bpe:command タグは、bpe:commandbar タグのサブエレメントです。

コンポーネント・クラス

com.ibm.bpe.jsf.component.CommandBarComponent

構文例

```
<bpe:commandbar model="#{TaskInstanceList}">

    <bpe:command
       commandID="Work on"
       label="Work on..."
       commandClass="com.ibm.bpc.explorer.command.WorkOnTaskCommand"
       context="#{TaskInstanceDetailsBean}"/>

    <bpe:command
       commandID="Cancel"
       label="Cancel"
       commandClass="com.ibm.task.clientmodel.command.CancelClaimTaskCommand"
       context="#{TaskInstanceList}"/>

</bpe:commandbar>
```

タグ属性

表 39. bpe:commandbar 属性

属性	必須	説明
buttonStyleClass	いいえ	コマンド・バー内のボタンのレンダリングに使用されるカスケーディング・スタイル・シート (CSS) スタイル・クラス。
id	いいえ	コンポーネントの JavaServer Faces ID。
model	はい	ItemProvider インターフェースをインプリメントする管理対象 Bean に対する値バインディング式。通常、この管理対象 Bean は、同一の JavaServer Pages (JSP) ファイル内の List コンポーネントまたは Details コンポーネントによって CommandBar コンポーネントとして使用される com.ibm.bpe.jsf.handler.BPCListHandler クラスまたは com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスです。
styleClass	いいえ	コマンド・バーのレンダリングに使用される CSS スタイル・クラス。

表 40. bpe:command 属性

属性	必須	説明
action	いいえ	JavaServer Faces アクション・メソッドまたはコマンド・ボタンにより起動される Faces ナビゲーション・ターゲット。アクションにより戻されるナビゲーション・ターゲットは、その他のナビゲーション・ルールをすべて上書きします。このアクションは、例外がスローされない場合、またはコマンドから ErrorsInCommandException 例外がスローされる場合に呼び出されます。
commandClass	いいえ	コマンド・クラスの名前。このクラスのインスタンスは CommandBar コンポーネントにより作成され、コマンド・ボタンが選択されると実行されます。
commandID	はい	コマンドの ID。
context	いいえ	**commandClass** 属性を使用して指定されたコマンドのコンテキストを提供するオブジェクト。コマンド・バーが初めてアクセスされると、コンテキスト・オブジェクトが取得されます。
immediate	いいえ	コマンドが起動される時期を指定します。この属性の値が `true` の場合は、ページの入力が処理される前にコマンドが起動されます。デフォルトは `false` です。
label	はい	コマンド・バーでレンダリングされるボタンのラベル。
rendered	いいえ	ボタンがレンダリングされるかどうかを判別します。属性の値は、ブール値または値の式のいずれかにすることができます。
styleClass	いいえ	ボタンのレンダリングに使用される CSS スタイル・クラス。このスタイルは、コマンド・バーに定義されたボタン・スタイルをオーバーライドします。

JSF アプリケーションへの Message コンポーネントの追加

Business Process Choreographer Explorer Message コンポーネントを使用して、JavaServer Faces (JSF) アプリケーション内で、データ・オブジェクトおよびプリミティブ型をレンダリングします。

このタスクについて

メッセージ型がプリミティブ型である場合、ラベルおよび入力フィールドがレンダリングされます。メッセージ型がデータ・オブジェクトである場合、コンポーネントはオブジェクトを全探索し、オブジェクト内のエレメントをレンダリングします。

プロシージャー

1. Message コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   bpe:form タグを <h:form> タグに追加します。 bpe:form タグには、 model 属性が含まれていなければなりません。

   以下の例では、Message コンポーネントを追加する方法を示します。

```
<h:form>

   <h:outputText value="Input Message" />
   <bpe:form model="#{MyHandler.inputMessage}" readOnly="true" />

   <h:outputText value="Output Message" />
   <bpe:form model="#{MyHandler.outputMessage}" />

</h:form>
```

   Message コンポーネントの **model** 属性は、com.ibm.bpc.clientcore.MessageWrapper オブジェクトを参照します。このラッパー・オブジェクトは、サービス・データ・オブジェクト (SDO) オブジェクトか、または int や boolean などの Java プリミティブ型のいずれかをラップします。例では、メッセージは MyHandler 管理対象 Bean のプロパティーによって提供されます。

2. bpe:form タグで参照されている管理対象 Bean を構成します。

   以下の例では、MyHandler 管理対象 Bean を構成ファイルに追加する方法を示します。

```
<managed-bean>
<managed-bean-name>MyHandler</managed-bean-name>
<managed-bean-class>com.ibm.bpe.sample.jsf.MyHandler</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>

   <managed-property>
      <property-name>type</property-name>
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>

</managed-bean>
```

3. JSF アプリケーションにカスタム・コードを追加します。

   以下の例では、入力メッセージおよび出力メッセージをインプリメントする方法を示します。

```
public class MyHandler implements ItemListener {

  private TaskInstanceBean taskBean;
  private MessageWrapper inputMessage, outputMessage

  /* Listener method, e.g. when a task instance was selected in a list handler.
   * Ensure that the handler is registered in the faces-config.xml or manually.
   */
  public void itemChanged(Object item) {
    if( item instanceof TaskInstanceBean ) {
        taskBean = (TaskInstanceBean) item ;
    }
  }

  /* Get the input message wrapper
   */
  public MessageWrapper getInputMessage() {
```

```
    try{
        inputMessage = taskBean.getInputMessageWrapper() ;
    }
    catch( Exception e ) {
        ;       //...ignore errors for simplicity
    }
    return inputMessage;
  }

  /* Get the output message wrapper
   */
  public MessageWrapper getOutputMessage() {
    // Retrieve the message from the bean. If there is no message, create
    // one if the task has been claimed by the user. Ensure that only
    // potential owners or owners can manipulate the output message.
    try{
        outputMessage = taskBean.getOutputMessageWrapper();
        if( outputMessage == null
         && taskBean.getState() == TaskInstanceBean.STATE_CLAIMED ) {
            HumanTaskManagerService htm = getHumanTaskManagerService();
            outputMessage = new MessageWrapperImpl();
            outputMessage.setMessage(
               htm.createOutputMessage( taskBean.getID() ).getObject()
            );
         }
    }
    catch( Exception e ) {
        ;       //...ignore errors for simplicity
    }
    return outputMessage
  }
}
```

MyHandler 管理対象 Bean は、リスト・ハンドラーへの項目リスナーとして登録できるように、com.ibm.jsf.handler.ItemListener インターフェースをインプリメントします。ユーザーがリスト内の項目をクリックすると、選択された項目について MyHandler Bean が itemChanged(Object item) メソッドで通知されます。ハンドラーは、項目タイプを検査してから、関連した TaskInstanceBean オブジェクトへの参照を保管します。このインターフェースを使用するには、faces-config.xml ファイル内の適切なリスト・ハンドラーの itemListener リストにエントリーを追加します。

MyHandler Bean は、getInputMessage および getOutputMessage メソッドを提供します。これらのメソッドはどちらも、MessageWrapper オブジェクトを戻します。メソッドは、参照されたタスク・インスタンス Bean への呼び出しを委任します。例えばメッセージが設定されていないなどの理由で、タスク・インスタンス Bean がヌルを戻した場合、ハンドラーは新規に空のメッセージを作成して保管します。Message コンポーネントは MyHandler Bean が提供するメッセージを表示します。

結果

これで、JSF アプリケーションは、データ・オブジェクトおよびプリミティブ型をレンダリング可能な JavaServer ページを含むようになります。

関連資料

184 ページの『Message コンポーネント: タグ定義』

Business Process Choreographer Explorer Message コンポーネントは、JavaServer

Faces (JSF) アプリケーション内で、commonj.sdo.DataObject オブジェクトと、整数およびストリングなどのプリミティブ型をレンダリングします。

Message コンポーネント: タグ定義

Business Process Choreographer Explorer Message コンポーネントは、JavaServer Faces (JSF) アプリケーション内で、commonj.sdo.DataObject オブジェクトと、整数およびストリングなどのプリミティブ型をレンダリングします。

Message コンポーネントは、JSF コンポーネント・タグである bpe:form から構成されます。

コンポーネント・クラス

com.ibm.bpe.jsf.component.MessageComponent

構文例

```
<bpe:form model="#{TaskInstanceDetailsBean.inputMessageWrapper}"
          simplification="true" readOnly="true"
          styleClass4table="messageData"
          styleClass4output="messageDataOutput">
</bpe:form>
```

タグ属性

表 41. bpe:form 属性

属性	必須	説明
id	いいえ	コンポーネントの JavaServer Faces ID。
model	はい	commonj.sdo.DataObject オブジェクトまたは com.ibm.bpc.clientcore.MessageWrapper オブジェクトを参照する値バインディング式。
readOnly	いいえ	この属性が true に設定されている場合は、読み取り専用フォームのみがレンダリングされます。デフォルトでは、この属性は false に設定されています。
simplification	いいえ	この属性が true に設定されている場合は、単純型が含まれているプロパティーおよびカーディナリティーがゼロまたは 1 のプロパティーが表示されます。デフォルトでは、この属性は true に設定されています。
style4validinput	いいえ	有効な入力のレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル。
style4invalidinput	いいえ	無効な入力のレンダリング用の CSS スタイル。
styleClass4invalidInput	いいえ	無効な入力のレンダリング用の CSS スタイル・クラス名。
styleClass4output	いいえ	出力エレメントのレンダリング用の CSS スタイル・クラス名。

表 41. bpe:form 属性 (続き)

属性	必須	説明
styleClass4table	いいえ	Message コンポーネントによって提供されるテーブルのレンダリング用の、CSS テーブル・スタイルのクラス名。
styleClass4validInput	いいえ	有効な入力のレンダリング用の CSS スタイル・クラス名。

タスクおよびプロセス・メッセージ用の JSP ページの開発

Business Process Choreographer Explorer インターフェースには、ビジネス・データの表示や入力のためのデフォルトの入力フォームおよび出力フォームが用意されています。 JSP ページを使用して、カスタマイズされた入力フォームおよび出力フォームをカスタマイズできます。

このタスクについて

ユーザー定義の JavaServer Pages (JSP) ページを Web クライアントに組み込むには、WebSphere Integration Developer でヒューマン・タスクをモデル化するときにこれらのページを指定する必要があります。例えば、JSP ページの指定先は、特定のタスクやその入出力メッセージ、特定のユーザーのロールまたはすべてのユーザーのロールのいずれでも構いません。ユーザー定義 JSP ページは、出力データを表示して入力データを収集するために実行時にユーザー・インターフェースに組み込まれます。

カスタマイズ・フォームは自己完結した Web ページではなく、Business Process Choreographer Explorer によって HTML フォームに組み込まれる HTML フラグメントです。例えば、メッセージのすべてのラベルや入力フィールドのフラグメントがこれに該当します。

カスタマイズ・フォームがあるページでボタンをクリックすると、入力データは Business Process Choreographer Explorer に送信されて検証されます。検証は、提供されたプロパティーのタイプとブラウザーで使用されているロケールに基づいて行われます。 入力データを検証できない場合は同じページがもう一度表示され、検証エラーの情報が messageValidationErrors 要求属性に書き込まれます。情報はマップとして提供され、このマップは、無効なプロパティーの XML Path Expression (XPath) を、発生した妥当性検査例外にマップします。

カスタマイズ・フォームを Business Process Choreographer Explorer に追加するには、WebSphere Integration Developer を使用して以下のステップを実行します。

プロシージャー

1. カスタマイズ・フォームを作成します。

   Web インターフェースで使用される、入出力フォーム用のユーザー定義 JSP ページは、メッセージ・データにアクセスする必要があります。 JSP 内の Java 断片または JSP 実行言語を使用して、メッセージ・データにアクセスします。フォーム内のデータは要求コンテキストを介して使用可能です。
2. JSP ページにタスクを割り当てます。

ヒューマン・タスク・エディターでヒューマン・タスクを開きます。 クライアント設定で、ユーザー定義 JSP ページの場所と、カスタマイズ・フォームの適用先のロール (例: 管理者) を指定します。Business Process Choreographer Explorer のクライアント設定は、タスク・テンプレートに格納されます。これらの設定は、実行時にタスク・テンプレートを使用して取得されます。

3. ユーザー定義 JSP ページを Web アーカイブ (WAR ファイル) にパッケージ化します。

   WAR ファイルは、タスクが格納されているモジュールと一緒にエンタープライズ・アーカイブに組み込むことも、個別に配置することもできます。JSP が個別にデプロイされる場合、Business Process Choreographer Explorer またはカスタム・クライアントがデプロイされるサーバー上で JSP を使用可能にしてください。

   プロセスおよびタスク・メッセージにカスタム JSP を使用している場合、JSP をデプロイするために使用される Web モジュールを、カスタム JSF クライアントがマップされるのと同じサーバーにマップする必要があります。

結果

カスタマイズ・フォームは、Business Process Choreographer Explorer で実行時にレンダリングされます。

ユーザー定義 JSP フラグメント

ユーザー定義 JavaServer Pages (JSP) フラグメントは、HTML フォーム・タグに埋め込まれます。Business Process Choreographer Explorer は、実行時にこれらのフラグメントを、レンダリングされるページに埋め込みます。

入力メッセージのユーザー定義 JSP フラグメントは、出力メッセージの JSP フラグメントより先に埋め込まれます。

```
<html....>
   ...
   <form...>
     Input JSP (display task input message)

     Output JSP (display task output message)

   </form>
   ...
</html>
```

ユーザー定義 JSP フラグメントは、HTML フォーム・タグに埋め込まれているため、入力要素を追加できます。 入力要素の名前は、データ要素の XML Path Language (XPath) 表現と一致する必要があります。入力要素の名前には、以下に示す提供されたプレフィックス値を使用してプレフィックスを付けることが重要です。

```
<input id="address"
       type="text"
       name="${prefix}/selectPromotionalGiftResponse/address"
       value="${messageMap['/selectPromotionalGiftResponse/address"]}
       size="60"
       align="left" />
```

プレフィックス値は要求属性として提供されます。この属性により、引用符で囲まれた書式内の入力名は固有であることが保証されます。プレフィックスは Business Process Choreographer Explorer によって次のように生成されるため、変更しないでください。

```
String prefix = (String)request.getAttribute("prefix");
```

プレフィックス要素が設定されるのは、与えられたコンテキストにおいてメッセージが編集できる場合に限ります。 出力データは、ヒューマン・タスクの状態に応じてさまざまな方法で表示できます。例えば、タスクが要求状態にある場合は、出力データを変更できます。ただし、タスクが完了状態にある場合、出力データは表示専用になります。JSP フラグメントでは、プレフィックス要素が存在し、それに応じてメッセージを表示するかどうかをテストできます。 以下の JSTL ステートメントでは、プレフィックス要素が設定されているかどうかをテストする 1 つの方法を示しています。

```
...
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...
<c:choose>
  <c:when test="${not empty prefix}">
    <!--Read/write mode-->
  </c:when>
  <c:otherwise>
    <!--Read-only mode-->
  </c:otherwise>
</c:choose>
```

ヒューマン・タスク機能をカスタマイズするプラグインの作成

Business Process Choreographer では、ヒューマン・タスクの処理中に発生するイベントのイベント処理インフラストラクチャーを提供します。ニーズに合わせて機能を適応させることができるように、プラグイン点も設けられます。サービス・プロバイダー・インターフェース (SPI) を使用してカスタマイズされたプラグインを作成し、イベントの処理とスタッフ照会の処理を行うことができます。

このタスクについて

ヒューマン・タスク API イベントとエスカレーション通知イベント用のプラグインを作成することができます。また、担当者解決から戻される結果を処理するプラグインを作成することもできます。例えば、ピーク時に結果リストにユーザーを追加して、ワークロードのバランスを取ることもできます。

プラグインは、グローバル・レベルのすべてのタスク、アプリケーション・コンポーネントのタスク、タスク・テンプレートに関連付けられたタスクのすべて、または単一のタスク・インスタンスの場合に、異なるレベルで登録できます。

API イベント・ハンドラーの作成

API イベントは、API メソッドがヒューマン・タスクを操作するときに発生します。 API イベント・ハンドラー・プラグインのサービス・プロバイダー・インターフェース (SPI) を使用して、API イベントまたは同等の API イベントを持つ内部イベントが送信するタスク・イベントを処理するプラグインを作成します。

このタスクについて

API イベント・ハンドラーを作成するには、次のステップを実行します。

プロシージャー

1. APIEventHandlerPlugin2 インターフェースをインプリメントするクラス、または APIEventHandler インプリメンテーション・クラスを拡張するクラスを作成します。 このクラスは、他のクラスのメソッドを呼び出すことができます。
   - APIEventHandlerPlugin2 インターフェースを使用する場合は、APIEventHandlerPlugin2 インターフェースおよび APIEventHandlerPlugin インターフェースのすべてのメソッドをインプリメントする必要があります。
   - SPI インプリメンテーション・クラスを拡張する場合は、必要なメソッドを上書きしてください。

   このクラスは、Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

   ヒント: このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。これを呼び出すと、データベース・デッドロックが発生します。
2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   ヘルパー・クラスが数個の J2EE アプリケーションによって使用される場合は、共用ライブラリーとして登録する別の JAR ファイルにこれらのクラスをパッケージできます。
3. JAR ファイルの META-INF/services/ ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java 2 サービス・プロバイダー・インターフェース仕様に準拠します。

   a. `com.ibm.task.spi.`*`plug-in_name`*`APIEventHandlerPlugin` という名前のファイルを作成します。 *plug-in_name* はプラグインの名前です。

      例えば、`Customer` という名前のプラグインが com.ibm.task.spi.APIEventHandlerPlugin インターフェースをインプリメントする場合、構成ファイルの名前は `com.ibm.task.spi.`**`Customer`**`APIEventHandlerPlugin` になります。

   b. このファイルのコメント行、ブランク行以外の最初の行で、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

      例えば、`MyAPIEventHandler` というプラグイン・クラスが `com.customer.plugins` パッケージにある場合は、構成ファイルの最初の行に `com.customer.plugins.MyAPIEventHandler` という項目が含まれていなければなりません。

結果

ファイルには、API イベントを処理するプラグインを含むインストール可能な JAR ファイルと、プラグインのロードで使用できるサービス・プロバイダー構成ファイルがあります。

ヒント: API イベント・ハンドラーおよび通知イベント・ハンドラーの両方登録するために使用可能な eventHandlerName プロパティーは 1 つだけです。API イベント・ハンドラーおよび通知イベント・ハンドラーを両方使用する場合、プラグイン・インプリメンテーションは同じ名前でなければなりません (例えば、SPI インプリメンテーションのイベント・ハンドラー名として `Customer` など)。

いずれのプラグインも、単一のクラスまたは 2 つのクラスを使用して実装できます。いずれの場合も、JAR ファイルの META-INF/services/ ディレクトリーに 2 つのファイルを作成する必要があります (com.ibm.task.spi.CustomerNotificationEventHandlerPlugin と com.ibm.task.spi.CustomerAPIEventHandlerPlugin など)。

プラグインの実装とヘルパー・クラスを単一の JAR ファイルにパッケージします。

次のタスク

次に、プラグインをインストールして登録し、実行時にヒューマン・タスク・コンテナーが使用できるようにする必要があります。API イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

API イベント・ハンドラー

API イベントは、ヒューマン・タスクが変更されるか、その状態が変化するときに発生します。これらの API イベントを処理するために、タスクが変更される前 (pre-event メソッド)、および API 呼び出しが戻る直前 (post-event メソッド) に、イベント・ハンドラーが直接呼び出されます。

pre-event メソッドが ApplicationVetoException 例外をスローすると、API アクションが実行されずに API の呼び出し元にその例外が戻され、イベントに関連付けられたトランザクションがロールバックされます。pre-event メソッドが内部イベントによってトリガーされ、ApplicationVetoException 例外がスローされた場合は、自動要求などの内部イベントが実行されなくても、例外はクライアント・アプリケーションに戻されません。この場合、情報メッセージが SystemOut.log ファイルに書き込まれます。API メソッドが処理中に例外をスローすると、その例外はキャッチされ、post-event メソッドに渡されます。その例外は、post-event メソッドが戻った後に再び呼び出し元に渡されます。

以下の規則が、pre-event メソッドに適用されます。

- pre-event メソッドは、関連した API メソッドまたは内部イベントのパラメーターを受け取ります。
- pre-event メソッドは、ApplicationVetoException 例外をスローして処理の続行を阻止することができます。

以下の規則が、post-event メソッドに適用されます。

- post-event メソッドは、API 呼び出しに提供されたパラメーター、および戻り値を受け取ります。API メソッド・インプリメンテーションによって例外がスローされると、post-event メソッドも例外を受け取ります。
- post-event メソッドは、戻り値を変更できません。
- post-event メソッドは例外をスローできません。ランタイム例外はログに記録されますが、無視されます。

API イベント・ハンドラーをインプリメントするには、APIEventHandlerPlugin インターフェースを拡張する APIEventHandlerPlugin2 インターフェースを使用するか、またはデフォルトの com.ibm.task.spi.APIEventHandler SPI インプリメンテーション・クラスを拡張します。イベント・ハンドラーは、デフォルトのインプリメンテーション・クラスから継承する場合、常に最新バージョンの SPI をインプリメントします。新しいバージョンの Business Process Choreographer にアップグレードする場合は、新しい SPI メソッドを利用すると必要な変更も少なくなります。

通知イベント・ハンドラーと API イベント・ハンドラーの両方がある場合、イベント・ハンドラー名は 1 つしか登録できないので、両方のハンドラーの名前は同じでなければなりません。

通知イベント・ハンドラーの作成

通知イベントは、ヒューマン・タスクがエスカレートされるときに生成されます。Business Process Choreographer には、エスカレーション作業項目の作成や電子メールの送信などのエスカレーションを処理する機能があります。通知イベント・ハンドラーを作成して、エスカレーションの処理方法をカスタマイズすることができます。

このタスクについて

通知イベント・ハンドラーをインプリメントするには、NotificationEventHandlerPlugin インターフェースを使用する方法と、デフォルトの com.ibm.task.spi.NotificationEventHandler サービス・プロバイダー・インターフェース (SPI) インプリメンテーション・クラスを拡張する方法があります。

通知イベント・ハンドラーを作成するには、次のステップを実行します。

プロシージャー

1. NotificationEventHandlerPlugin インターフェースをインプリメントするクラス、または NotificationEventHandler インプリメンテーション・クラスを拡張するクラスを作成します。 このクラスは、他のクラスのメソッドを呼び出すことができます。

   NotificationEventHandlerPlugin インターフェースを使用する場合は、すべてのインターフェース・メソッドをインプリメントする必要があります。SPI インプリメンテーション・クラスを拡張する場合は、必要なメソッドを上書きしてください。

   このクラスは、Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

プラグインは、EscalationUser ロールの権限で呼び出されます。このロールは、ヒューマン・タスク・コンテナーの構成時に定義されます。

ヒント: このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクまたはエスカレーションを更新するメソッドは呼び出さないでください。これを呼び出すと、データベース・デッドロックが発生します。

2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   ヘルパー・クラスが数個の J2EE アプリケーションによって使用される場合は、共用ライブラリーとして登録する別の JAR ファイルにこれらのクラスをパッケージできます。

3. JAR ファイルの META-INF/services/ ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java 2 サービス・プロバイダー・インターフェース仕様に準拠します。

   a. `com.ibm.task.spi.`*`plug-in_name`*`NotificationEventHandlerPlugin` という名前のファイルを作成します。 *plug-in_name* はプラグインの名前です。

      例えば、`HelpDeskRequest` (イベント・ハンドラー名) という名前のプラグインが com.ibm.task.spi.NotificationEventHandlerPlugin インターフェースをインプリメントする場合、構成ファイルの名前は `com.ibm.task.spi.`**`HelpDeskRequest`**`NotificationEventHandlerPlugin` になります。

   b. このファイルのコメント行、ブランク行以外の最初の行で、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

      例えば、`MyEventHandler` というプラグイン・クラスが `com.customer.plugins` パッケージにある場合は、構成ファイルの最初の行に `com.customer.plugins.MyEventHandler` という項目が含まれていなければなりません。

結果

ファイルには、通知イベントを処理するプラグインを含むインストール可能な JAR ファイルと、プラグインのロードで使用できるサービス・プロバイダー構成ファイルがあります。API イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

ヒント: API イベント・ハンドラーおよび通知イベント・ハンドラーの両方登録するために使用可能な eventHandlerName プロパティーは 1 つだけです。API イベント・ハンドラーおよび通知イベント・ハンドラーを両方使用する場合、プラグイン・インプリメンテーションは同じ名前でなければなりません (例えば、SPI インプリメンテーションのイベント・ハンドラー名として `Customer` など)。

いずれのプラグインも、単一のクラスまたは 2 つのクラスを使用して実装できます。いずれの場合も、JAR ファイルの META-INF/services/ ディレクトリーに 2 つ

のファイルを作成する必要があります
(com.ibm.task.spi.CustomerNotificationEventHandlerPlugin と
com.ibm.task.spi.CustomerAPIEventHandlerPlugin など)。

プラグインの実装とヘルパー・クラスを単一の JAR ファイルにパッケージします。

次のタスク

次に、プラグインをインストールして登録し、実行時にヒューマン・タスク・コンテナーが使用できるようにする必要があります。通知イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

担当者照会結果の後処理を行うプラグインの作成

スタッフ解決は、特定のロール (例えばタスクの潜在的な所有者) に割り当てられているユーザーのリストを戻します。担当者解決で戻される担当者照会の結果を変更するプラグインを作成することができます。例えば、ワークロード・バランシングを改善するために、既にワークロードが高いユーザーを照会結果から除去するプラグインを使用することがあります。

このタスクについて

後処理プラグインは 1 つしか所有できません。つまり、そのプラグインですべてのタスクの担当者照会結果を処理する必要があります。 このプラグインでは、ユーザーの追加または除去、あるいはユーザーまたはグループ情報の変更ができます。また、結果タイプを、例えばユーザー・リストからグループに、あるいは全員に、変更することもできます。

プラグインは担当者解決の完了後に実行されるので、機密性またはセキュリティーを保持するための規則が既に適用されています。プラグインは、担当者の解決中に除去されたユーザーについての情報を受け取ります (HTM_REMOVED_USERS マップ・キーで)。 プラグインでこのコンテキスト情報を確実に使用し、規定されている機密性規則またはセキュリティー規則が保持されるようにする必要があります。

担当者照会結果の後処理を実装するには、StaffQueryResultPostProcessorPlugin インターフェースを使用します。このインターフェースには、タスク、エスカレーション、タスク・テンプレート、およびアプリケーション・コンポーネントの照会結果を変更するメソッドが含まれています。

担当者照会結果の後処理を行うプラグインを作成するには、次のステップを実行します。

プロシージャー

1. StaffQueryResultPostProcessorPlugin インターフェースをインプリメントするクラスを作成します。

   すべてのインターフェース・メソッドをインプリメントする必要があります。このクラスは、他のクラスのメソッドを呼び出すことができます。

このクラスは、Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

ヒント: このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。これを呼び出すと、データベース・デッドロックが発生します。

次の例では、SpecialTask というタスクのエディター・ロールを変更する方法を示しています。

```
public StaffQueryResult processStaffQueryResult
                        (StaffQueryResult originalStaffQueryResult,
                                          Task task,
                                          int  role,
                                          Map  context)
{
  StaffQueryResult newStaffQueryResult = originalStaffQueryResult;
  StaffQueryResultFactory staffResultFactory =
                          StaffQueryResultFactory.newInstance();
  if (role == com.ibm.task.api.WorkItem.REASON_EDITOR &&
      task.getName() != null &&
      task.getName().equals("SpecialTask"))
  {
      UserData user = staffResultFactory.newUserData
                           ("SuperEditor",
                             new Locale("en-US"),
                            "SuperEditor@company.com");
      ArrayList userList = new ArrayList();
      userList.add(user);

      newStaffQueryResult = staffResultFactory.newStaffQueryResult(userList);
   }
   return(newStaffQueryResult);
}
```

2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   ヘルパー・クラスが数個の J2EE アプリケーションによって使用される場合は、共用ライブラリーとして登録する別の JAR ファイルにこれらのクラスをパッケージできます。

3. JAR ファイルの META-INF/services/ ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java 2 サービス・プロバイダー・インターフェース仕様に準拠します。

   a. com.ibm.task.spi.*plug-in_name*StaffQueryResultPostProcessorPlugin という名前のファイルを作成します。*plug-in_name* はプラグインの名前です。

      例えば、MyHandler という名前のプラグインが com.ibm.task.spi.StaffQueryResultPostProcessorPlugin インターフェースをインプリメントする場合、構成ファイルの名前は com.ibm.task.spi.**MyHandler**StaffQueryResultPostProcessorPlugin になります。

b. このファイルのコメント行、ブランク行以外の最初の行で、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

例えば、StaffPostProcessor というプラグイン・クラスが com.customer.plugins パッケージにある場合は、構成ファイルの最初の行に com.customer.plugins.StaffPostProcessor という項目が含まれていなければなりません。 担当者照会結果を後処理するプラグインを含むインストール可能 JAR ファイルと、プラグインのロードに使用できるサービス・プロバイダー構成ファイルができます。

4. プラグインをインストールします。

担当者照会結果の後処理プラグインは 1 つしか所有できません。プラグインは、共用ライブラリーとしてインストールする必要があります。

5. プラグインを登録します。
   a. 管理コンソールで、Human Task Manager の「カスタム・プロパティー」ページに移動します (「**アプリケーション・サーバー**」 → 「***server_name***」 → 「**ヒューマン・タスク・コンテナー**」 → 「**カスタム・プロパティー**」)。
   b. **Staff.PostProcessorPlugin** という名前と、プラグインにつけた名前の値 (この例では、MyHandler) を持つカスタム・プロパティーを追加します。

プラグインのインストール

プラグインを使用するには、プラグインをインストールして、タスク・コンテナーがアクセスできるようにする必要があります。

このタスクについて

プラグインのインストール方法は、そのプラグインが 1 つの Java 2 Enterprise Edition (J2EE) アプリケーションでのみ使用されるか、複数のアプリケーションで使用されるかによって異なります。

次のステップのいずれかを実行してプラグインをインストールします。

- 1 つの J2EE アプリケーションによって使用される場合のプラグインのインストール

  プラグインの JAR ファイルをアプリケーションの EAR ファイルに追加します。WebSphere Integration Developer のデプロイメント記述子エディターで、プラグインの JAR ファイルを、主要な Enterprise JavaBeans (EJB) モジュールの J2EE アプリケーション用のプロジェクト・ユーティリティー JAR ファイルとしてインストールします。

- 複数の J2EE アプリケーションによって使用される場合のプラグインのインストール

  JAR ファイルを WebSphere Application Server 共用ライブラリーに入れ、そのライブラリーを、プラグインへのアクセスが必要なアプリケーションと関連付けます。JAR ファイルをネットワーク・デプロイメント環境で使用可能にするには、各サーバーに JAR ファイルを手動で配布してから、セルごとに共用ライブラリーを 1 度インストールします。

次のタスク

これで、プラグインを登録できます。

プラグインの登録

タスク・コンテナー成果物階層のさまざまなレベルにあるプラグインを登録できます。例えば、グローバル・レベルのすべてのタスク、アプリケーション・コンポーネントのすべてのタスク、タスク・テンプレートに関連付けられたタスクのすべて、または単一のタスク・インスタンスの場合です。

このタスクについて

複数のプラグインを登録する場合は、スコープがサポートされます。つまり、タスク・インスタンスなどのタスク・コンテナー成果物階層の下位レベルで登録されるプラグインを、タスク・テンプレートやアプリケーション・コンポーネントなどの上位レベルで登録されるプラグインの代わりに使用します。スコープは、すべての階層レベルでサポートされます。タスク・コンテナーは、階層の最下位レベルで登録されるプラグインを使用します。

プラグインは、次のいずれかの方法で登録できます。

- タスク・モデルにプラグインを登録する。

  WebSphere Integration Developer のタスク・エディターで、タスクのプロパティー領域の「詳細」ページにある「**イベント・ハンドラー名 (Event handler name)**」フィールドにイベント・ハンドラーの名前を指定します。
- 実行時に作成する臨時タスクまたはタスク・テンプレートのプラグインを登録する。

  TTask クラスの setEventHandlerName メソッドを使用して、イベント・ハンドラーの名前を登録します。
- 実行時に、タスク・インスタンスの登録されたイベント・ハンドラーを変更する。

  update(Task task) メソッドを使用して、実行時にタスク・インスタンスの別のイベント・ハンドラーを使用します。呼び出し元には、このプロパティーを更新するタスク管理者権限が必要です。
- グローバル・レベルでプラグインを登録する。

  ヒューマン・タスク・コンテナーの「カスタム・プロパティー」ページの管理コンソールで、プラグインのカスタム・プロパティーを定義します。カスタム・プロパティーの値は、プラグイン名です。

第 2 部 アプリケーションのデプロイ

第 3 章 モジュールの準備とインストールの概要

モジュールのインストール (デプロイとも呼ばれる) では、モジュールをテスト環境または実稼働環境のいずれかで活動化します。この概要では、テストおよび実稼働環境と、モジュールのインストールに必要な手順の一部について簡単に説明します。

注: アプリケーションを実稼働環境でインストールするプロセスは、WebSphere Application Server Network Deployment バージョン 6 インフォメーション・センターの『アプリケーションの開発とデプロイ』の項で説明されているプロセスと同様です。これらのトピックをお読みになったことがない場合、まず目を通してください。

モジュールを実稼働環境にインストールする前に、必ずテスト環境での変更内容を確認してください。モジュールをテスト環境にインストールする場合は、WebSphere Integration Developer を使用します。詳しくは、WebSphere Integration Developer インフォメーション・センターを参照してください。モジュールを実稼働環境にインストールする場合は、WebSphere Process Server を使用します。

このトピックでは、モジュールの実稼働環境へのインストールおよびその準備に必要な概念と作業について説明します。モジュールで使用されるオブジェクトを収容するファイルや、モジュールをテスト環境から実稼働環境へ移行する方法について説明しているその他のトピックがあります。モジュールを正しくインストールするためには、これらのファイルとファイルに格納されている内容を理解することが大切です。

ライブラリーと JAR ファイルの概要

モジュールでは、ライブラリー内の成果物を使用することがよくあります。成果物およびライブラリーは、モジュールをデプロイするときに指定する Java アーカイブ (JAR) ファイルに含まれています。

モジュールの開発時に、そのモジュールのさまざまな部分で使用する特定のリソースまたはコンポーネントを指定する場合があります。これらのリソースまたはコンポーネントは、モジュールの開発時に作成したオブジェクトである場合と、既にサーバー上にデプロイされているライブラリー内のオブジェクトである場合があります。ここでは、アプリケーションのインストール時に必要となるライブラリーおよびファイルについて説明します。

ライブラリーの概要

ライブラリーには、WebSphere Integration Developer 内の複数のモジュールで使用されるオブジェクトおよびリソースが格納されています。これらの成果物は、JAR ファイル、リソース・アーカイブ (RAR) ファイル、または Web サービス・アーカイブ (WAR)ファイルに格納されています。これらの成果物の例としては、次のものがあります。

- インターフェースまたは Web サービス記述子 (拡張子 .wsdl のファイル)

- ビジネス・オブジェクトの XML スキーマ定義 (拡張子 .xsd のファイル)
- ビジネス・オブジェクト・マップ (拡張子 .map のファイル)
- リレーションシップ定義とロール定義 (拡張子 .rel および .rol のファイル)

ある成果物がモジュールで必要になると、EAR クラス・パスに基づいてサーバーがこれを探し出し、メモリーにまだロードされていない場合は、ロードします。それ以降は、成果物が置き換えられないかぎりは、その成果物に対する要求では、メモリーにロードしたコピーが使用されます。図 5 に、アプリケーションとコンポーネントおよび関連するライブラリーの包含関係を示します。

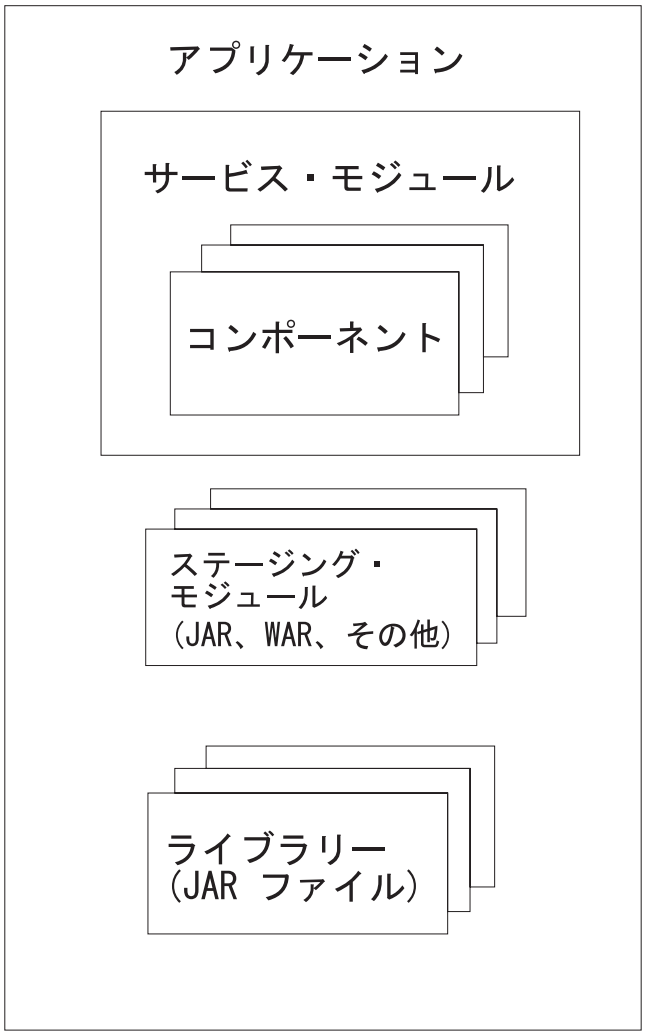


図 5. モジュール、コンポーネント、およびライブラリー間の関係

JAR、RAR、および WAR ファイルの概要

モジュールのコンポーネントを格納できるファイルは複数存在します。これらのファイルについては、Java プラットフォーム、Enterprise Edition 仕様に詳しい説明があります。 JAR ファイルの詳細については、JAR 仕様に説明があります。

WebSphere Process Server では、JAR ファイルにアプリケーション (モジュールで使用するほかのサービス・コンポーネントへの支援的な参照およびインターフェースをすべて含んだ、モジュールのアセンブル・バージョン) も格納されています。

アプリケーションを完全にインストールするには、この JAR ファイル、その他のすべてのライブラリー (JAR ファイル、Web サービス・アーカイブ (WAR) ファイル、リソース・アーカイブ (RAR) ファイル、ステージング・ライブラリー (Enterprise Java Beans - EJB) JAR ファイル、またはその他のアーカイブなど) が必要です。serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを作成します。

ステージング・モジュールの命名規則

ライブラリー内では、ステージング・モジュールの名前についての要件があります。ステージング・モジュールの名前は、それぞれのモジュールで固有でなければなりません。アプリケーションをデプロイするために必要なその他のモジュールには、ステージング・モジュールの名前との間に競合が発生しないような名前を付けてください。*myService* という名前のモジュールの場合、ステージング・モジュール名は、次のようになります。

- *myService*App
- *myService*EJB
- *myService*EJBClient
- *myService*Web

注: serviceDeploy コマンドは、サービスに WSDL ポート・タイプ・サービスが含まれている場合にのみ、*myService* Web ステージング・モジュールを作成します。

ライブラリー使用時の考慮事項

ライブラリーを使用することにより、ビジネス・オブジェクトの整合性およびモジュール間での処理の整合性が保証されます。なぜなら、それぞれの呼び出し側モジュールは、特定のコンポーネントの専用コピーを所有するからです。不整合や障害が発生しないようにするために、呼び出し側モジュールで使用するコンポーネントおよびビジネス・オブジェクトに対する変更は、すべての呼び出し側モジュール間で整合がとれている必要があります。呼び出し側モジュールを更新するには、次の手順を実行します。

1. モジュールおよびライブラリーの最新コピーを実動サーバーにコピーします。
2. serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを再作成します。
3. 呼び出し側モジュールを含む実行中のアプリケーションを停止し、再インストールします。
4. 呼び出し側モジュールを含むアプリケーションを再始動します。

関連資料


serviceDeploy コマンド

serviceDeploy コマンドを使用して、Service Component Architecture (SCA) 対応モジュールをサーバーにインストール可能な Java アプリケーションとしてパッケージします。このコマンドは、wsadmin を使用してバッチ・インストールを実行する場合に有効です。

EAR ファイルの概要

EAR ファイルは、サービス・アプリケーションの実動サーバーへのデプロイにおいて、重要な要素です。

エンタープライズ・アーカイブ (EAR) ファイルとは、アプリケーションがデプロイメントに必要とするライブラリー、エンタープライズ Bean、および JAR ファイルを格納した圧縮ファイルです。

アプリケーション・モジュールを WebSphere Integration Developer からエクスポートするときに、JAR ファイルを作成します。この JAR ファイルおよびその他の成果物ライブラリーまたはオブジェクトを、インストール・プロセスの入力として使用します。 serviceDeploy コマンドは、アプリケーションを構成する、コンポーネントの記述と Java コードを含む入力ファイルから EAR ファイルを作成します。

関連資料

serviceDeploy コマンド

serviceDeploy コマンドを使用して、Service Component Architecture (SCA) 対応モジュールをサーバーにインストール可能な Java アプリケーションとしてパッケージします。このコマンドは、wsadmin を使用してバッチ・インストールを実行する場合に有効です。

サーバーへのデプロイの準備

モジュールの開発およびテストが終了したら、モジュールをテスト・システムからエクスポートし、実稼働環境にデプロイする必要があります。アプリケーションをインストールするには、モジュールのエクスポート時に必要となるパスと、モジュールが必要とするライブラリーを認識している必要があります。

始める前に

このタスクを開始する前に、テスト・サーバーでモジュールの開発およびテストを完了し、各種問題およびパフォーマンス問題を解決しておく必要があります。

このタスクについて

このタスクでは、アプリケーションの必要な部分がすべて使用可能かどうか、および実動サーバーに移せるように正しくパッケージされているかどうか検証します。

注: WebSphere Integration Developer からエンタープライズ・アーカイブ (EAR) ファイルをエクスポートし、そのファイルを直接 WebSphere Process Server にインストールすることもできます。

重要: コンポーネント内部のサービスがデータベースを使用する場合、データベースに直接接続されたサーバーにアプリケーションをインストールします。

プロシージャー

1. デプロイするモジュール用のコンポーネントを含むフォルダーを見つけます。

コンポーネント・フォルダーの名前は *module-name* で、その中に *module*.module という名前のファイル (基本モジュール) があります。

2. モジュールに含まれるすべてのコンポーネントが、モジュール・フォルダーの下のコンポーネント・サブフォルダー内にあることを確認します。

   使いやすくするために、サブフォルダーには *module/component* のような名前を付けます。

3. 各コンポーネントを構成するすべてのファイルが適切なコンポーネント・サブフォルダーに格納されていて、*component-file-name*.component といった名前が付けられていることを確認します。

   コンポーネント・ファイルには、モジュール内の個々のコンポーネントの定義が記述されています。

4. 他のすべてのコンポーネントおよび成果物が、それらを必要とするコンポーネントのサブフォルダーに格納されていることを確認します。

   このステップで、コンポーネントが必要とする成果物への参照が使用可能であることを確認します。 serviceDeploy コマンドがステージング・モジュールに対して使用する名前と、これらのコンポーネントの名前が競合していてはなりません。『ステージング・モジュールの命名規則』を参照してください。

5. 参照ファイル (*module*.references) が、ステップ 1 (202 ページ) のモジュール・フォルダー内に存在することを確認します。

   参照ファイルは、モジュール内の参照およびインターフェースを定義します。

6. ワイヤー・ファイル (*module*.wires) がコンポーネント・フォルダーに存在することを確認します。

   ワイヤー・ファイルは、モジュール内の参照とインターフェース間の接続を完成させます。

7. マニフェスト・ファイル (*module*.manifest) がコンポーネント・フォルダーに存在することを確認します。

   マニフェストは、モジュールと、モジュールを構成するすべてのコンポーネントをリストします。マニフェストには、クラスパス・ステートメントも記述されています。これは、serviceDeploy コマンドが、モジュールが必要とする他のモジュールを検出できるようにするためです。

8. モジュールの圧縮ファイルまたは JAR ファイルを作成します。このファイルは serviceDeploy コマンドの入力として使用します。このコマンドは、実動サーバーにインストールするモジュールを準備してくれます。

デプロイメント前の MyValue モジュールのフォルダー構造の例

以下の例は、MyValue、CustomerInfo、および StockQuote というコンポーネントから構成される、MyValueModule モジュールのディレクトリー構造を示しています。

```
MyValueModule
   MyValueModule.manifest
   MyValueModule.references
   MyValueModule.wiring
   MyValueClient.jsp
process/myvalue
```

```
   MyValue.component
   MyValue.java
   MyValueImpl.java
service/customerinfo
   CustomerInfo.component
   CustomerInfo.java
   Customer.java
   CustomerInfoImpl.java
service/stockquote
   StockQuote.component
   StockQuote.java
   StockQuoteAsynch.java
   StockQuoteCallback.java
   StockQuoteImpl.java
```

次のタスク

『実動サーバーへのモジュールのインストール』の説明に従って、モジュールを実動システムにインストールします。

関連資料


serviceDeploy コマンド

serviceDeploy コマンドを使用して、Service Component Architecture (SCA) 対応モジュールをサーバーにインストール可能な Java アプリケーションとしてパッケージします。このコマンドは、wsadmin を使用してバッチ・インストールを実行する場合に有効です。

クラスター上のサービス・アプリケーションのインストールに関する考慮事項

クラスターにサービス・アプリケーションをインストールする場合、追加要件が発生します。クラスターにサービス・アプリケーションをインストールする際に、これらの考慮事項に注意することが重要です。

クラスターは、スケール・メリットを提供して、サーバー間の要求ワークロードのバランスを取り、アプリケーションのクライアントに対して一定レベルの可用性を提供できるようにすることで、処理環境に多くのメリットをもたらす可能性があります。クラスター上で、サービスを含むアプリケーションをインストールする前に、以下の事項を検討してください。

- アプリケーションのユーザーが、クラスタリングにより提供される処理能力と可用性を必要としているか。

  その場合、クラスタリングが正しい解決策です。クラスタリングにより、アプリケーションの可用性と能力が向上します。
- クラスターがサービス・アプリケーション用に正しく準備されているか。

  サービスを含む最初のアプリケーションをインストールして開始する前に、クラスターを正しく構成する必要があります。クラスターを正しく構成していない場合、アプリケーションは要求を正しく処理できません。
- クラスターのバックアップはあるか。

バックアップ・クラスターにもアプリケーションをインストールする必要があります。

第 4 章 実動サーバーへのモジュールのインストール

このトピックでは、テスト・サーバーからアプリケーションを取り出して実稼働環境にデプロイする際に行うステップについて説明します。

始める前に

サービス・アプリケーションを実動サーバーにデプロイする前に、テスト・サーバー上でアプリケーションをアセンブルし、テストします。テストが終わったら、「モジュールの開発とデプロイ」PDF の『サーバーへのデプロイの準備』の説明に従って必要なファイルをエクスポートします。次に、それらのファイルを実動システムでデプロイします。詳しくは、WebSphere Integration Developer および WebSphere Application Server Network Deployment のインフォメーション・センターを参照してください。

プロシージャー

1. モジュールおよびその他のファイルを実動サーバーにコピーします。

   アプリケーションに必要なモジュールおよびリソース (EAR、JAR、RAR、および WAR ファイル) が、実稼働環境に移動します。
2. serviceDeploy コマンドを実行して、インストール可能な EAR ファイルを作成します。

   このステップでは、アプリケーションを実稼働にインストールする準備として、サーバーにモジュールを定義します。
   a. デプロイするモジュールを含む JAR ファイルを見つけ出します。
   b. 前のステップで見つかった JAR ファイルを入力として使用して、 コマンドを実行します。
3. ステップ 2 で作成した EAR ファイルをインストールします。 アプリケーションのインストール方法は、アプリケーションをスタンドアロン・サーバーにインストールするか、セル内のサーバーにインストールするかによって異なります。

   注: 管理コンソールを使用しても、スクリプトを使用しても、アプリケーションをインストールすることができます。追加情報については、WebSphere Application Server インフォメーション・センターを参照してください。
4. 構成を保管します。 これで、モジュールがアプリケーションとしてインストールされました。
5. アプリケーションを開始します。

結果

これで、アプリケーションがアクティブになり、モジュールを通して処理が行われるようになります。

次のタスク

アプリケーションをモニターし、サーバーが要求を正しく処理していることを確認します。

関連資料


serviceDeploy コマンド

serviceDeploy コマンドを使用して、Service Component Architecture (SCA) 対応モジュールをサーバーにインストール可能な Java アプリケーションとしてパッケージします。このコマンドは、wsadmin を使用してバッチ・インストールを実行する場合に有効です。

serviceDeploy を使用したインストール可能な EAR ファイルの作成

アプリケーションを実稼働環境にインストールするには、実働サーバーにコピーされたファイルを取得して、インストール可能な EAR ファイルを作成します。

始める前に

このタスクを開始する前に、サーバーに対してデプロイしようとしているモジュールとサービスを含む JAR ファイルを用意する必要があります。詳しくは、『サーバーへのデプロイの準備』を参照してください。

このタスクについて

serviceDeploy コマンドは、JAR ファイルや、これに依存するそのほかの EAR、JAR、RAR、WAR、および ZIP ファイルを取得して、サーバーにインストールできる EAR ファイルを作成します。

プロシージャー

1. デプロイするモジュールを含む JAR ファイルを見つけ出します。
2. 前のステップで見つかった JAR ファイルを入力として使用して、 コマンドを実行します。

   このステップでは、EAR ファイルが作成されます。

   注: 管理コンソールで以下のステップを実行します。
3. サーバーの管理コンソールでインストールする EAR ファイルを選択します。
4. 「**保管**」をクリックして、EAR ファイルをインストールします。

関連資料


serviceDeploy コマンド

serviceDeploy コマンドを使用して、Service Component Architecture (SCA) 対応モジュールをサーバーにインストール可能な Java アプリケーションとしてパッケージします。このコマンドは、wsadmin を使用してバッチ・インストールを実行する場合に有効です。

Apache Ant タスクを使用したアプリケーションのデプロイ

このトピックでは、Apache™ Ant タスクを使用して、WebSphere Process Server に対するアプリケーションのデプロイメントを自動化する方法について説明します。Apache Ant タスクを使用すると、複数のアプリケーションのデプロイメントを定義し、サーバーへのそれらのアプリケーションの不在デプロイを実行できます。

始める前に

このタスクでは、以下が前提となります。
- デプロイしようとしているアプリケーションは、すでに開発およびテスト済みである。
- アプリケーションのインストール先が、同じ 1 つまたは複数のサーバーである。
- Apache Ant タスクに関する基本的な知識がある。
- デプロイメント・プロセスを理解している。

アプリケーションの開発およびテストについての詳細は、WebSphere Integration Developer インフォメーション・センターにあります。

WebSphere Application Server Network Deployment インフォメーション・センターのリファレンス部分には、アプリケーション・プログラミング・インターフェースに関するセクションがあります。Apache Ant タスクについては、パッケージ com.ibm.websphere.ant.tasks に説明があります。このトピックに関連するタスクとして、ServiceDeploy および InstallApplicationがあります。

このタスクについて

複数のアプリケーションを並行してインストールする必要がある場合は、デプロイメントの前に Apache Ant タスクを作成します。こうすると、プロセスに手動で介入しなくても、Apache Ant タスクによってアプリケーションをサーバーにデプロイおよびインストールできます。

プロシージャー

1. デプロイするアプリケーションを特定します。
2. 各アプリケーションごとに JAR ファイルを作成します。
3. JAR ファイルをターゲット・サーバーにコピーします。
4. 各サーバーの EAR ファイルを作成するために ServiceDeploy コマンドを実行する Apache Ant タスクを作成します。
5. ステップ 4 で作成した各 EAR ファイルに対して、InstallApplication コマンドをアプリケーション・サーバー上で実行する Apache Ant タスクを作成します。
6. ServiceDeploy Apache Ant タスクを実行して、アプリケーションの EAR ファイルを作成します。
7. InstallApplication Apache Ant タスクを実行して、ステップ 6 で作成した EAR ファイルをインストールします。

結果

アプリケーションは、ターゲット・サーバー上に正しくデプロイされます。

アプリケーションの無人デプロイの例

次に、myBuildScript.xml ファイルに含まれる Apache Ant タスクの例を示します。

```
<?xml version="1.0">

<project name="OwnTaskExample" default="main" basedir=".">
 <taskdef name="servicedeploy"
   classname="com.ibm.websphere.ant.tasks.ServiceDeployTask" />
 <target name="main" depends="main2">
  <servicedeploy scaModule="c:/synctest/SyncTargetJAR"
   ignoreErrors="true"
   outputApplication="c:/synctest/SyncTargetEAREAR"
   workingDirectory="c:/synctest"
   noJ2eeDeploy="true"
   cleanStagingModules="true"/>
 </target>
</project>
```

このステートメントは、Apache Ant タスクの呼び出し方法を指定します。

```
${WAS}/bin/ws_ant -f myBuildScript.xml
```

ヒント: このファイルにさらに別のプロジェクト・ステートメントを追加して、複数のアプリケーションを無人デプロイすることができます。

次のタスク

管理コンソールを使用して、新しくインストールしたアプリケーション・サーバーが始動され、ワークフローを正しく処理していることを確認してください。

関連資料

serviceDeploy コマンド

serviceDeploy コマンドを使用して、Service Component Architecture (SCA) 対応モジュールをサーバーにインストール可能な Java アプリケーションとしてパッケージします。このコマンドは、wsadmin を使用してバッチ・インストールを実行する場合に有効です。

第 5 章 ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール

ビジネス・プロセスまたはヒューマン・タスク、あるいはこの両方を含む Service Component Architecture (SCA) モジュールをデプロイメント・ターゲットに配布することができます。サーバーまたはクラスターをデプロイメント・ターゲットとすることができます。

始める前に

アプリケーションのインストール先のアプリケーション・サーバーまたはクラスターごとに、Business Flow Manager、Human Task Manager、またはその両方がインストールされ、構成されていることを確認します。

このタスクについて

ビジネス・プロセスおよびタスク・アプリケーションを、管理コンソールやコマンド行から、または管理スクリプトを実行してインストールすることができます。

結果

ビジネス・プロセス・アプリケーションまたはヒューマン・タスク・アプリケーションがインストールされると、すべてのビジネス・プロセス・テンプレートおよびヒューマン・タスク・テンプレートは開始状態になります。これらのテンプレートから、プロセス・インスタンスとタスク・インスタンスを作成できます。

次のタスク

プロセス・インスタンスまたはタスク・インスタンスを作成するには、アプリケーションを始動する必要があります。

関連概念

212 ページの『ビジネス・プロセスとヒューマン・タスクのデプロイメント』
WebSphere Integration Developer またはサービス・デプロイメントによりプロセスまたはタスクのデプロイメント・コードが生成されるときに、各プロセス・コンポーネントまたはタスク・コンポーネントが 1 つのセッション・エンタープライズ Bean にマップされます。すべてのデプロイメント・コードは、エンタープライズ・アプリケーション (EAR) ファイルにパッケージ化されます。また、エンタープライズ・アプリケーションのインストール中に、プロセスごとに、そのプロセスの Java コードを表す Java クラスが生成され、EAR ファイルに埋め込まれます。デプロイするモデルの新規バージョンごとに、新しいエンタープライズ・アプリケーションにパッケージ化する必要があります。

212 ページの『Network Deployment 環境へのビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール方法』
プロセス・テンプレートまたはヒューマン・タスク・テンプレートを Network Deployment 環境にインストールするときに、アプリケーションのインストール機能により以下のアクションが自動的に実行されます。

Network Deployment 環境へのビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール方法

プロセス・テンプレートまたはヒューマン・タスク・テンプレートを Network Deployment 環境にインストールするときに、アプリケーションのインストール機能により以下のアクションが自動的に実行されます。

アプリケーションのインストールは、非同期かつ段階別に実行されます。次の段階を開始する前に、前の段階が正常に完了している必要があります。

1. アプリケーションのインストールは Deployment Manager で開始されます。

   この段階では、ビジネス・プロセス・テンプレートとヒューマン・タスク・テンプレートが、WebSphere 構成リポジトリーで構成されます。アプリケーションの検証も行われます。エラーが発生した場合、エラーは System.out ファイルまたは System.err ファイルに報告されるか、あるいは Deployment Manager で FFDC エントリーとして報告されます。
2. アプリケーションのインストールはノード・エージェントで続行されます。

   この段階では、1 つのアプリケーション・サーバー・インスタンスでのアプリケーションのインストールが開始されます。このアプリケーション・サーバー・インスタンスは、デプロイメント・ターゲットであるか、またはその一部です。デプロイメント・ターゲットが、複数のクラスター・メンバーで構成されるクラスターの場合は、このクラスターのクラスター・メンバーからサーバー・インスタンスが任意に選択されます。この段階でエラーが発生した場合、エラーは SystemOut.log ファイル または SystemErr.log ファイルに報告されるか、あるいはノード・エージェントで FFDC エントリーとして報告されます。
3. サーバー・インスタンスでアプリケーションが実行されます。

   この段階では、プロセス・テンプレートとヒューマン・テンプレートがデプロイメント・ターゲットの Business Process Choreographer データベースに配置されます。エラーが発生した場合、System.out ファイルまたは SystemErr.log ファイルに報告されるか、あるいはこのサーバー・インスタンスで FFDC エントリーとして報告されます。

関連タスク

211 ページの『第 5 章 ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール』
ビジネス・プロセスまたはヒューマン・タスク、あるいはこの両方を含む Service Component Architecture (SCA) モジュールをデプロイメント・ターゲットに配布することができます。サーバーまたはクラスターをデプロイメント・ターゲットとすることができます。

ビジネス・プロセスとヒューマン・タスクのデプロイメント

WebSphere Integration Developer またはサービス・デプロイメントによりプロセスまたはタスクのデプロイメント・コードが生成されるときに、各プロセス・コンポーネントまたはタスク・コンポーネントが 1 つのセッション・エンタープライズ Bean にマップされます。すべてのデプロイメント・コードは、エンタープライズ・アプリケーション (EAR) ファイルにパッケージ化されます。また、エンタープライ

ズ・アプリケーションのインストール中に、プロセスごとに、そのプロセスの Java コードを表す Java クラスが生成され、EAR ファイルに埋め込まれます。デプロイするモデルの新規バージョンごとに、新しいエンタープライズ・アプリケーションにパッケージ化する必要があります。

ビジネス・プロセスまたはヒューマン・タスクが含まれているエンタープライズ・アプリケーションをインストールすると、これらのビジネス・プロセスまたはヒューマン・タスクは、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートとして Business Process Choreographer データベースに格納されます。デフォルトでは、新しくインストールされたテンプレートは、開始済み状態となります。ただし、新しくインストールされたエンタープライズ・アプリケーションの場合は、停止状態になります。インストール済みのエンタープライズ・アプリケーションは、個々に開始したり停止したりすることができます。

異なるエンタープライズ・アプリケーションそれぞれに、多数のバージョンのプロセス・テンプレートやタスク・テンプレートをデプロイできます。新規エンタープライズ・アプリケーションのインストール時に、インストールされるテンプレートのバージョンが次のように決定されます。

- テンプレートの名前とターゲット名前空間がまだ存在していない場合は、新規テンプレートがインストールされます。
- テンプレート名とターゲット名前空間が、既存のテンプレートと同じであるが、有効開始日が異なる場合は、既存のテンプレートの新規バージョンがインストールされます。

注: テンプレート名は、ビジネス・プロセスまたはヒューマン・タスクではなく、コンポーネントの名前から派生します。

有効開始日を指定しない場合、日付は次のように決定されます。

- WebSphere Integration Developer を使用する場合、有効開始日はヒューマン・タスクまたはビジネス・プロセスがモデル化された日付になります。
- サービス・デプロイメントを使用する場合、有効開始日は、serviceDeploy コマンドが実行された日付になります。アプリケーションがインストールされた日付を有効開始日として取得するのは、コラボレーション・タスクだけです。

関連タスク

211 ページの『第 5 章 ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール』

ビジネス・プロセスまたはヒューマン・タスク、あるいはこの両方を含む Service Component Architecture (SCA) モジュールをデプロイメント・ターゲットに配布することができます。サーバーまたはクラスターをデプロイメント・ターゲットとすることができます。

ビジネス・プロセス・アプリケーションおよびヒューマン・タスク・アプリケーションの対話式インストール

アプリケーションは、wsadmin ツールと installInteractive スクリプトを使用して、実行時に対話式でインストールできます。このスクリプトにより、管理コンソールを使用してアプリケーションをインストールする場合には変更することのできない設定を変更できます。

このタスクについて

ビジネス・プロセス・アプリケーションを対話式にインストールするには、次のステップを実行します。

プロシージャー

1. wsadmin ツールを開始します。

   profile_root/bin ディレクトリーで、`wsadmin` と入力します。
2. アプリケーションをインストールします。

   wsadmin コマンド行プロンプトで、次のコマンドを入力します。

   ```
   $AdminApp installInteractive application.ear
   ```

   ここで、*application.ear* は、ご使用のプロセス・アプリケーションを含むエンタープライズ・アーカイブ・ファイルの修飾名です。 一連のタスクにおいてプロンプトが出されるので、その時にアプリケーションの値を変更できます。
3. 構成の変更を保管します。

   wsadmin コマンド行プロンプトで、次のコマンドを入力します。

   ```
   $AdminConfig save
   ```

   変更を保管し、マスター構成リポジトリーへの更新を転送する必要があります。スクリプト・プロセスが終了し、変更を保存していなければ、変更は廃棄されます。

プロセス・アプリケーションのデータ・ソースと設定参照の設定値の構成

特定のデータベース・インフラストラクチャーで SQL ステートメントを実行するプロセス・アプリケーションは、構成する必要があります。これらの SQL ステートメントは、情報サービス・アクティビティーから発生したり、プロセスのインストールまたはインスタンスの開始時に実行するステートメントであったりします。

このタスクについて

このアプリケーションのインストール時には、次のタイプのデータ・ソースを指定できます。

- プロセス・インストール時に SQL ステートメントを実行するデータ・ソース
- プロセス・インスタンスの開始時に SQL ステートメントを実行するデータ・ソース
- SQL 断片アクティビティーを実行するデータ・ソース

SQL 断片アクティビティーを実行するために必要なデータ・ソースは、tDataSource タイプの BPEL 変数で定義されます。SQL 断片アクティビティーで必要とされるデータベース・スキーマ名とテーブル名は、tSetReference タイプの BPEL 変数で定義されます。これらの両方の変数の初期値は、構成することができます。

wsadmin ツールを使用してデータ・ソースを指定できます。

プロシージャー

1. wsadmin ツールを使用して、プロセス・アプリケーションを対話式にインストールします。
2. データ・ソースと設定参照を更新するタスクになるまで、タスクをステップスルーします。

   ご使用の環境でこれらの設定を構成します。次の例に、これらのタスクごとに変更可能な設定を示します。
3. 変更を保管します。

例: wsadmin ツールを使用したデータ・ソースと設定参照の更新

「**Updating data source**」タスクでは、プロセスのインストール時またはプロセスの開始時に使用される初期変数値およびステートメントのデータ・ソース値を変更できます。「**Updating set references**」タスクでは、データベース・スキーマとテーブル名に関連した設定を構成できます。

```
Task [24]: Updating data sources

//Change data source values for initial variable values at process start

Process name:  Test
// Name of the process template
Process start or installation time:  Process start
// Indicates whether the specified value is evaluated
//at process startup or process installation
Statement or variable:  Variable
// Indicates that a data source variable is to be changed
Data source name:  MyDataSource
// Name of the variable
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name to jdbc/newName
Task [25]: Updating set references

// Change set reference values that are used as initial values for BPEL variables

Process name:  Test
// Name of the process template
Variable:  SetRef
// The BPEL variable name
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name of the data source of the set reference to jdbc/newName
Schema name: [IISAMPLE]
// The name of the database schema
Schema prefix: []:
// The schema name prefix.
// This setting applies only if the schema name is generated.
Table name: [SETREFTAB]: NEWTABLE
// Sets the name of the database table to NEWTABLE
Table prefix: []:
// The table name prefix.
// This setting applies only if the prefix name is generated.
```

管理コンソールを使用した、ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのアンインストール

管理コンソールを使用すると、ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールできます。

始める前に

ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするには、以下の前提条件を満たしている必要があります。

- アプリケーションがスタンドアロン・サーバーにインストールされている場合は、そのサーバーが稼働しており、Business Process Choreographer データベースへのアクセス権限を持っている必要があります。
- アプリケーションがクラスターにインストールされている場合は、デプロイメント・マネージャーおよび少なくとも 1 つのクラスター・メンバーが稼働している必要があります。このクラスター・メンバーは、Business Process Choreographer データベースへのアクセス権限を持っています。
- アプリケーションが管理対象サーバーにインストールされている場合は、デプロイメント・マネージャーおよびこのサーバーが稼働している必要があります。このサーバーには Business Process Choreographer データベースへのアクセス権限が必要です。
- アプリケーションに属するすべてのビジネス・プロセス・テンプレートおよびヒューマン・タスク・テンプレートが停止状態にある必要があります。
- いずれの状態においてもビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートのインスタンスはありません。

開発環境および単体テスト環境として使用されているスタンドアロン・サーバー環境の場合、開発モードで実行するようサーバーを構成できます。この構成では、テンプレートを停止し、インスタンスが存在しないようにする必要はありません。ただし、この構成は実稼働環境に対しては無効です。

このタスクについて

ビジネス・プロセスまたはヒューマン・タスクが含まれるエンタープライズ・アプリケーションをアンインストールするには、以下のアクションを実行します。

プロシージャー

1. アプリケーションのすべてのプロセスおよびタスクのテンプレートを停止します。

   このアクションにより、プロセスおよびタスクのインスタンスの作成が回避されます。

   a. 管理コンソールのナビゲーション・ペインで、「**アプリケーション**」 → 「**SCA モジュール**」をクリックします。
   b. 停止するテンプレートを含んでいるモジュールを選択します。
   c. 「追加プロパティー」の下で、「**ビジネス・プロセス**」か「**ヒューマン・タスク**」、または両方を必要に応じてクリックします。

d. 適切なチェック・ボックスをクリックして、プロセスおよびタスクのテンプレートをすべて選択します。
e. 「**停止**」をクリックします。

ビジネス・プロセス・テンプレートまたはタスク・テンプレートが含まれるすべての EJB モジュールで、このステップを繰り返します。

2. データベース、クラスターごとに少なくとも 1 つのアプリケーション・サーバー、アプリケーションがデプロイされているスタンドアロン・サーバーが動作していることを確認してください。

ネットワーク・デプロイメント環境では、デプロイメント・マネージャー、管理対象のすべてのスタンドアロン・アプリケーション・サーバー、および少なくとも 1 つのアプリケーション・サーバーが、そのアプリケーションがインストールされているクラスターごとに稼働している必要があります。

3. アプリケーションにビジネス・プロセス・インスタンスまたはヒューマン・タスク・インスタンスがないことを確認します。

必要であれば、管理者は、Business Process Choreographer Explorer を使用して、残存するプロセス・インスタンスまたはタスク・インスタンスを削除することができます。

4. アプリケーションを停止してアンインストールするには、以下のようにします。
   a. 管理コンソールのナビゲーション・ペインで、「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」をクリックします。
   b. アンインストールするアプリケーションを選択し、「**停止**」をクリックします。

      アプリケーションでプロセス・インスタンスまたはタスク・インスタンスがまだ存在する場合は、このステップは失敗します。
   c. アンインストールするアプリケーションを再度選択し、「**アンインストール**」をクリックします。
   d. 「**保管**」をクリックして、変更を保管します。

結果

アプリケーションはアンインストールされます。

管理コマンドを使用した、ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのアンインストール

管理コマンドには、ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするための、管理コンソールの代替方法が用意されています。

始める前に

ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするには、以下の前提条件を満たしている必要があります。

- アプリケーションがスタンドアロン・サーバーにインストールされている場合は、そのサーバーが稼働しており、Business Process Choreographer データベースへのアクセス権限を持っている必要があります。
- アプリケーションがクラスターにインストールされている場合は、デプロイメント・マネージャーおよび少なくとも 1 つのクラスター・メンバーが稼働している必要があります。このクラスター・メンバーは、Business Process Choreographer データベースへのアクセス権限を持っています。
- アプリケーションが管理対象サーバーにインストールされている場合は、デプロイメント・マネージャーおよびこのサーバーが稼働している必要があります。このサーバーには Business Process Choreographer データベースへのアクセス権限が必要です。
- アプリケーションに属するすべてのビジネス・プロセス・テンプレートおよびヒューマン・タスク・テンプレートが停止状態にある必要があります。
- いずれの状態においてもビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートのインスタンスはありません。

開発環境および単体テスト環境として使用されているスタンドアロン・サーバー環境の場合、開発モードで実行するようサーバーを構成できます。この構成では、テンプレートを停止し、インスタンスが存在しないようにする必要はありません。ただし、この構成は実稼働環境に対しては無効です。

さらに、グローバル・セキュリティーが使用可能に設定されている場合は、使用するユーザー ID にオペレーター権限があることを確認します。

管理クライアントが接続しているサーバー・プロセスが動作していることを確認します。管理クライアントが自動的にサーバー・プロセスに接続できるよう、`-conntype NONE` オプションをコマンド・オプションとして使用しないでください。

このタスクについて

次の手順で、bpcTemplates.jacl スクリプトを使用して、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートを含むアプリケーションをアンインストールする方法を示します。また、アプリケーションをアンインストールするには、そのアプリケーションに属しているテンプレートを停止する必要があります。 bpcTemplates.jacl スクリプトを使用して、1 つの手順でテンプレートを停止およびアンインストールできます。

アプリケーションをアンインストールする前に、Business Process Choreographer Explorer を使用するなどして、アプリケーション内のテンプレートに関連付けられたプロセス・インスタンスまたはタスク・インスタンスを削除できます。また、bpcTemplates.jacl スクリプトで **-force** オプションを使用し、テンプレートに関連付けられたインスタンスの削除、テンプレートの停止、およびそれらのテンプレートのアンインストールを 1 度のステップで実行できます。

注意:
-force オプションはすべてのプロセス・インスタンスおよびタスク・インスタンスのデータを削除するため、注意して使用する必要があります。

プロシージャー

1. Business Process Choreographer サンプル・ディレクトリーに移動します。

Windows プラットフォームの場合は、次のように入力します。

```
cd install_root¥ProcessChoreographer¥admin
```

Linux、UNIX、および i5/OS プラットフォームの場合は、以下を入力します。

```
cd install_root/ProcessChoreographer/admin
```

2. テンプレートを停止して、対応するアプリケーションをアンインストールします。

Windows プラットフォームの場合は、次のように入力します。

```
install_root¥bin¥wsadmin -f bpcTemplates.jacl
                            [-user user_name]
                            [-password user password]
                            -uninstall application_name
                            [-force]
```

Linux、UNIX、および i5/OS プラットフォームの場合は、以下を入力します。

```
install_root/bin/wsadmin -f bpcTemplates.jacl
                            [-user user_name]
                            [-password user password]
                            -uninstall application_name
                            [-force]
```

各部の意味は、次のとおりです。

user_name
グローバル・セキュリティーが使用可能な場合は、認証用のユーザー ID を提供します。

user_password
グローバル・セキュリティーが使用可能な場合は、認証用のユーザー・パスワードを提供します。

application_name
グローバル・セキュリティーが使用可能な場合は、認証用のユーザー・パスワードを提供します。

結果

アプリケーションはアンインストールされます。

第 6 章 アダプターのインストール

アダプターを使用すると、開発アプリケーションとエンタープライズ情報システム内のその他のコンポーネントとの通信ができるようになります。

アダプターのインストールに使用するプロセスについては、WebSphere Integration Developer インフォメーション・センターの、「アダプターの構成と使用 (Configuring and using adapters)」に説明があります。

第 7 章 EIS アプリケーションのインストール

EIS アプリケーション・モジュールは、J2EE プラットフォームにデプロイできます。このデプロイメントでは、EAR ファイルとしてパッケージ化されたアプリケーションがサーバーにデプロイされます。J2EE のすべての成果物およびリソースが作成され、アプリケーションが構成されて、実行準備ができます。

このタスクについて

J2EE プラットフォームへのデプロイでは、以下の J2EE 成果物およびリソースが作成されます。

表 42. バインディングから J2EE 成果物への対応

SCA モジュールでのバインディング	生成される J2EE 成果物	作成される J2EE リソース
EIS インポート	モジュールのセッション EJB で生成されるリソース参照。	ConnectionFactory
EIS エクスポート	リソース・アダプターによってサポートされるリスナー・インターフェースに応じて生成またはデプロイされるメッセージ駆動型 Bean。	ActivationSpec
JMS インポート	ランタイムにより提供されるメッセージ駆動型 Bean (MDB) がデプロイされ、モジュールのセッション EJB でリソース参照が生成される。MDB はインポートに受信宛先が指定されている場合にのみ作成される。	• ConnectionFactory • ActivationSpec • Destination
JMS エクスポート	ランタイムにより提供されるメッセージ駆動型 Bean がデプロイされ、モジュールのセッション EJB でリソース参照が生成される。	• ActivationSpec • ConnectionFactory • Destination

インポートまたはエクスポートで、ConnectionFactory のようなリソースを定義している場合、リソース参照はモジュールのステートレス・セッション EJB のデプロイメント記述子内に生成されます。また、適切なバインディングが EJB バインディング・ファイル内に生成されます。リソース参照がバインドされる名前は、モジュール名とインポート名を基にして、ターゲット属性の値 (値が存在する場合)、またはリソースに与えられたデフォルトの JNDI 検索名のいずれかになります。

デプロイ時に、実装環境により、モジュールのセッション Bean が検出され、これを使用して、リソースが検索されます。

サーバーへのアプリケーションのデプロイ中に、EIS インストール・タスクにより、バインドされたエレメント・リソースが存在するかどうかの確認が行われま

す。このリソースが存在しない場合、SCDL ファイルで 1 つ以上のプロパティーを指定しているときは、EIS インストール・タスクによって、このリソースが作成され、構成されます。リソースが存在しない場合は、何も処置は行われず、リソースを作成してからアプリケーションを実行するものとみなされます。

JMS インポートが受信宛先を指定してデプロイされた場合、メッセージ駆動型 Bean (MDB) がデプロイされます。これにより、送信された要求に対する応答を listen します。MDB は、JMS メッセージの JMSreplyTo ヘッダー・フィールドにある、要求で送信された Destination に関連付けられます (この Destination で listen します)。MDB は、応答メッセージが到着すると、メッセージの相関 ID を使用して、コールバック Destination に保管されているコールバック情報を取得して、コールバック・オブジェクトを呼び出します。

インストール・タスクでは、インポート・ファイル内の情報から ConnectionFactory と 3 つの宛先が作成されます。これに加えて、ActivationSpec を作成して、ランタイム MDB が受信宛先で応答を listen できるようにします。ActivationSpec のプロパティーは、Destination/ConnectionFactory プロパティーから派生しています。JMS プロバイダーが SIBus リソース・アダプターである場合、JMS の Destination に対応する SIBus の Destination が作成されます。

JMS エクスポートをデプロイする場合、メッセージ駆動型 Bean (MDB) (JMS インポートの場合にデプロイされる MDB とは異なる MDB) がデプロイされます。MDB は、受信宛先で着信要求を listen して、その要求が SCA で処理されるようにディスパッチします。インストール・タスクでは、JMS インポートの場合と同様のリソース・セット、ActivationSpec、応答の送信に使用される ConnectionFactory、および 2 つの宛先が作成されます。これらのリソースのプロパティーはすべて、エクスポート・ファイルに指定されます。JMS プロバイダーが SIBus リソース・アダプターである場合、JMS の Destination に対応する SIBus の Destination が作成されます。

J2SE プラットフォームへの EIS アプリケーション・モジュールのデプロイ

EIS モジュールを J2SE プラットフォームにデプロイすることはできますが、EIS インポートのみがサポートされます。

始める前に

この作業を開始する前に、 WebSphere 統合開発環境で、JMS インポート・バインディングを使用して、EIS アプリケーション・モジュールを作成する必要があります。

このタスクについて

メッセージ・キューを使用して EIS システムに非同期にアクセスしたい場合は、EIS アプリケーション・モジュールに JMS インポート・バインディングを提供します。

J2SE プラットフォームにデプロイするのは、バインディングのインプリメンテーションを管理対象外モードで実行できる場合のみです。 JMS バインディングでは、

非同期および JNDI のサポートが必要です。このどちらも、基本の Service Component Architecture または J2SE では提供されません。J2EE Connector Architecture では、管理対象外のインバウンド通信がサポートされないため、EIS エクスポートは除去されます。

EIS インポートと共に EIS アプリケーション・モジュールが J2SE にデプロイされると、モジュールの依存関係に加えて、インポートに使用される WebSphere Adapter も、マニフェスト内か、または SCA でサポートされる他のなんらかの形式で、依存関係として指定されなければなりません。

J2EE プラットフォームへの EIS アプリケーション・モジュールのデプロイ

EIS モジュールを J2EE プラットフォームにデプロイすると、EAR ファイルとしてパッケージ化されたアプリケーションがサーバーにデプロイされます。J2EE のすべての成果物およびリソースが作成され、アプリケーションが構成されて、実行準備ができます。

始める前に

この作業を開始する前に、 WebSphere 統合開発環境で、JMS インポート・バインディングを使用して、EIS モジュールを作成する必要があります。

このタスクについて

J2EE プラットフォームへのデプロイでは、以下の J2EE 成果物およびリソースが作成されます。

表 43. バインディングから J2EE 成果物への対応

SCA モジュールでのバインディング	生成される J2EE 成果物	作成される J2EE リソース
EIS インポート	モジュールのセッション EJB で生成されるリソース参照。	ConnectionFactory
EIS エクスポート	リソース・アダプターによってサポートされるリスナー・インターフェースに応じて生成またはデプロイされるメッセージ駆動型 Bean。	ActivationSpec
JMS インポート	ランタイムにより提供されるメッセージ駆動型 Bean (MDB) がデプロイされ、モジュールのセッション EJB でリソース参照が生成される。MDB はインポートに受信宛先が指定されている場合にのみ作成される。	• ConnectionFactory • ActivationSpec • Destination

表 43. バインディングから J2EE 成果物への対応 (続き)

SCA モジュールでのバインディング	生成される J2EE 成果物	作成される J2EE リソース
JMS エクスポート	ランタイムにより提供されるメッセージ駆動型 Bean がデプロイされ、モジュールのセッション EJB でリソース参照が生成される。	• ActivationSpec • ConnectionFactory • Destination

インポートまたはエクスポートで、ConnectionFactory のようなリソースを定義している場合、リソース参照はモジュールのステートレス・セッション EJB のデプロイメント記述子内に生成されます。また、適切なバインディングが EJB バインディング・ファイル内に生成されます。リソース参照がバインドされる名前は、モジュール名とインポート名を基にして、ターゲット属性の値 (値が存在する場合)、またはリソースに与えられたデフォルトの JNDI 検索名のいずれかになります。

デプロイ時に、実装環境により、モジュールのセッション Bean が検出され、これを使用して、リソースが検索されます。

サーバーへのアプリケーションのデプロイ中に、EIS インストール・タスクにより、バインドされたエレメント・リソースが存在するかどうかの確認が行われます。このリソースが存在しない場合、SCDL ファイルで 1 つ以上のプロパティーを指定しているときは、EIS インストール・タスクによって、このリソースが作成され、構成されます。リソースが存在しない場合は、何も処置は行われず、リソースを作成してからアプリケーションを実行するものとみなされます。

JMS インポートが受信宛先を指定してデプロイされた場合、メッセージ駆動型 Bean (MDB) がデプロイされます。これにより、送信された要求に対する応答を listen します。MDB は、JMS メッセージの JMSreplyTo ヘッダー・フィールドにある、要求で送信された Destination に関連付けられます (この Destination で listen します)。MDB は、応答メッセージが到着すると、メッセージの相関 ID を使用して、コールバック Destination に保管されているコールバック情報を取得して、コールバック・オブジェクトを呼び出します。

インストール・タスクでは、インポート・ファイル内の情報から ConnectionFactory と 3 つの宛先が作成されます。これに加えて、ActivationSpec を作成して、ランタイム MDB が受信宛先で応答を listen できるようにします。ActivationSpec のプロパティーは、Destination/ConnectionFactory プロパティーから派生しています。JMS プロバイダーが SIBus リソース・アダプターである場合、JMS の Destination に対応する SIBus の Destination が作成されます。

JMS エクスポートをデプロイする場合、メッセージ駆動型 Bean (MDB) (JMS インポートの場合にデプロイされる MDB とは異なる MDB) がデプロイされます。MDB は、受信宛先で着信要求を listen して、その要求が SCA で処理されるようにディスパッチします。インストール・タスクでは、JMS インポートの場合と同様のリソース・セット、ActivationSpec、応答の送信に使用される ConnectionFactory、および 2 つの宛先が作成されます。これらのリソースのプロパティーはすべて、エクスポート・ファイルに指定されます。JMS プロバイダーが SIBus リソース・アダプターである場合、JMS の Destination に対応する SIBus の Destination が作成されます。

第 8 章 失敗したデプロイメントのトラブルシューティング

このトピックでは、アプリケーションのデプロイ時の問題の原因を判別するために行うステップについて説明します。また、参考になるいくつかのソリューションも示されています。

始める前に

このトピックは、以下の事項を前提としています。
- モジュールのデバッグの基本について理解している。
- モジュールのデプロイ中にロギングおよびトレースがアクティブになっている。

このタスクについて

デプロイメントのトラブルシューティングのタスクは、エラーの通知を受け取った後に開始します。失敗したデプロイメントには、アクションをとる前に検査する必要のあるさまざまな症状があります。

プロシージャー

1. アプリケーションのインストールが失敗したかどうか判別します。

   SystemOut.log ファイルを調べて、失敗の原因を示すメッセージを探します。アプリケーションをインストールできない理由には、以下のようなものがあります。
   - 同一の Network Deployment セル内の複数のサーバーにアプリケーションをインストールしようとしている。
   - アプリケーションの名前が、アプリケーションをインストールする Network Deployment セル上の既存のモジュールの名前と同じである。
   - EAR ファイル内部の J2EE モジュールを異なるターゲット・サーバーにデプロイしようとしている。

   重要: インストールが失敗し、アプリケーションにサービスが含まれる場合、アプリケーションの再インストールを試みる前に、失敗の前に作成された SIBus 宛先または J2C 活動化仕様を除去する必要があります。これらの成果物を除去する最も簡単な方法は、失敗後に「**保管**」>「**すべて廃棄 (Discard all)**」をクリックする方法です。不注意で変更を保存した場合、SIBus 宛先および J2C 活動化仕様を手動で除去する必要があります (『管理』セクションの『SIBus 宛先の削除』および『J2C 活動化仕様の削除』を参照)。
2. アプリケーションが正しくインストールされている場合は、プログラムが正常に開始したかどうかを確認します。

   アプリケーションが正常に開始していない場合は、サーバーがアプリケーションのリソースを初期化しようとしたときに障害が起きています。
   a. SystemOut.log ファイルを調べて、対処法を指示するメッセージを探します。
   b. アプリケーションで必要なリソースが使用可能か、また、それらのリソースが正常に開始されたかどうかを確認します。

開始されないリソースがあると、アプリケーションは実行されません。これは、情報が失われるのを防ぐためです。リソースが開始しない理由には次のものがあります。
- 指定されたバインディングが正しくない。
- リソースが正しく構成されていない。
- リソースがリソース・アーカイブ (RAR) ファイルに含まれていない。
- Web リソースが Web サービス・アーカイブ (WAR) ファイルに含まれていない。

c. コンポーネントが欠落していないかどうか判別します。

コンポーネント欠落の原因は、エンタープライズ・アーカイブ (EAR) ファイルが正しく作成されなかったことにあります。モジュールが必要とするすべてのコンポーネントが、Java アーカイブ (JAR) ファイルをビルドするテスト・システムの正しいフォルダーにあることを確認してください。『サーバーへのデプロイの準備』で追加情報について説明します。

3. アプリケーションで情報が処理されているかどうかを調べます。

実行中のアプリケーションでも、情報の処理に失敗することがあります。この理由は、ステップ 2b (227 ページ) で示した理由と同様です。

a. アプリケーションが、別のアプリケーションに含まれるサービスを使用するかどうかを判別します。 その別のアプリケーションがインストール済みで、正常に開始されていることを確認します。

b. 失敗したアプリケーションが使用する別のアプリケーションに含まれる、各種デバイス用のインポート・バインディングおよびエクスポート・バインディングが正しく構成されていることを確認します。 管理コンソールを使用して、バインディングを調べ、訂正してください。

4. 問題を解決してから、アプリケーションを再始動します。

J2C 活動化仕様の削除

システムは、サービスを含むアプリケーションをインストールする際に、J2C アクティベーション・スペックを作成します。アプリケーションを再インストールする前に、これらの仕様を削除しなければならない場合があります。

始める前に

アプリケーションのインストールが失敗したために仕様を削除する場合は、Java Naming and Directory Interface (JNDI) 名のモジュールが、インストールに失敗したモジュールの名前と一致することを確認してください。JNDI 名の 2 番目の部分は、宛先をインプリメントしたモジュールの名前です。例えば sca/SimpleBOCrsmA/ActivationSpec では、**SimpleBOCrsmA** がモジュール名です。

このタスクに必要なセキュリティーのロール: セキュリティー権限および役割ベースの権限を使用可能にした場合、このタスクを実行するには、管理者またはコンフィギュレーターとしてログインする必要があります。

このタスクについて

サービスを含むアプリケーションのインストール後に不注意で構成を保管した場合、J2C 活動化仕様を必要としなければ、活動化仕様を削除します。

プロシージャー

1. 削除する活動化仕様を見つけます。

   仕様はリソース・アダプター・パネルに含まれています。「**リソース**」>「**リソース・アダプター (Resource adapters)**」をクリックして、このパネルにナビゲートします。

   a. **Platform Messaging Component SPI Resource Adapter** を見つけます。

      このアダプターを見つけるには、スタンドアロン・サーバーの**ノード**・スコープまたはデプロイメント環境の**サーバー**・スコープにいる必要があります。

2. Platform Messaging Component SPI Resource Adapter に関連付けられた J2C 活動化仕様を表示します。

   リソース・アダプター名をクリックすると、関連する仕様が次のパネルに表示されます。

3. 削除するモジュール名に一致する **JNDI 名**の仕様をすべて削除します。

   a. 該当する仕様の横のチェック・ボックスをクリックします。

   b. 「**削除**」をクリックします。

結果

システムが、選択された仕様を表示から除去します。

次のタスク

変更を保管します。

SIBus 宛先の削除

SIBus 宛先とは、アプリケーションに対してサービスを使用可能にする接続です。宛先を除去しなければならない場合もあります。

始める前に

アプリケーションのインストールが失敗したために宛先を削除する場合は、宛先名のモジュールが、インストールに失敗したモジュールの名前と一致することを確認してください。宛先の 2 番目の部分は、宛先をインプリメントしたモジュールの名前です。例えば sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer では、**SimpleBOCrsmA** がモジュール名です。

このタスクに必要なセキュリティーのロール: セキュリティー権限および役割ベースの権限を使用可能にした場合、このタスクを実行するには、管理者またはコンフィギュレーターとしてログインする必要があります。

このタスクについて

サービスを含むアプリケーションのインストール後に不注意で構成を保管した場合、または SIBus 宛先を必要としなくなった場合、その宛先を削除します。

注: このタスクは、SCA システム・バスからのみ宛先を削除します。また、サービスを含むアプリケーションを再インストールする前にも、アプリケーション・バスからエントリーを除去する必要があります。このインフォメーション・センターの『管理』セクションの『J2C 活動化仕様の削除』を参照してください。

プロシージャー

1. 管理コンソールにログインします。
2. SCA システム・バスの宛先を表示します。

   「**サービス統合**」>「**バス**」をクリックしてパネルにナビゲートします。
3. SCA システム・バス宛先を選択します。

   表示の中で、**SCA.SYSTEM.*cellname*.Bus** をクリックします。ここで *cellname* は、削除する宛先を含むモジュールが含まれているセルの名前です。
4. 除去するモジュールに一致するモジュール名を含む宛先を削除します。
   a. 該当する宛先の横のチェック・ボックスをクリックします。
   b. 「**削除**」をクリックします。

結果

パネルには残った宛先のみが表示されます。

次のタスク

これらの宛先を作成したモジュールに関連する J2C 活動化仕様を削除してください。

第 3 部 付録

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711
東京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
577 Airport Blvd., Suite 800
Burlingame, CA 94010
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生した創作物にも、次のように、著作権表示を入れていただく必要があります。「(c) (お客様の会社名) (西暦年)」このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 (c) Copyright IBM Corp. _年を入れる_. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

プログラミング・インターフェース情報がある場合、それらはこのプログラムを使用してアプリケーション・ソフトウェアを作成する際に役立つよう提供されています。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

IBM、IBM logo、developerWorks、WebSphere、z/OS は、International Business Machines Corporation の米国およびその他の国における商標です。

Adobe は、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

この製品には、Eclipse Project (http://www.eclipse.org) により開発されたソフトウェアが含まれています。

IBM WebSphere Process Server for Multiplatforms バージョン 6.1.0

Printed in Japan

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12