



TIBCO ActiveMatrix® BPM Java Component Development

Software Release 4.3

April 2019

Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

ANY SOFTWARE ITEM IDENTIFIED AS THIRD PARTY LIBRARY IS AVAILABLE UNDER SEPARATE SOFTWARE LICENSE TERMS AND IS NOT PART OF A TIBCO PRODUCT. AS SUCH, THESE SOFTWARE ITEMS ARE NOT COVERED BY THE TERMS OF YOUR AGREEMENT WITH TIBCO, INCLUDING ANY TERMS CONCERNING SUPPORT, MAINTENANCE, WARRANTIES, AND INDEMNITIES. DOWNLOAD AND USE OF THESE ITEMS IS SOLELY AT YOUR OWN DISCRETION AND SUBJECT TO THE LICENSE TERMS APPLICABLE TO THEM. BY PROCEEDING TO DOWNLOAD, INSTALL OR USE ANY OF THESE ITEMS, YOU ACKNOWLEDGE THE FOREGOING DISTINCTIONS BETWEEN THESE ITEMS AND TIBCO PRODUCTS.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, the TIBCO logo, Two-Second Advantage, TIB, Information Bus, ActiveMatrix, Business Studio, Enterprise Message Service, Hawk, and Rendezvous are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

This and other products of TIBCO Software Inc. may be covered by registered patents. Please refer to TIBCO's Virtual Patent Marking document (<https://www.tibco.com/patents>) for details.

Copyright © 2010-2019. TIBCO Software Inc. All Rights Reserved.

Contents

Figures	6
TIBCO Documentation and Support Services	7
Java Components	9
Creating a Java Component	9
Configuring a Java Components Implementation	10
Updating a Java Component	10
Configuring a Java Components Custom Feature	11
Upgrading a Java Component	11
Component Feature Dependencies	12
Java Component Reference	13
Java Component Implementations	16
Data Binding	17
Generating XML Data Binding Classes	18
Data Binding Classes for Abstract and Concrete WSDL Files	18
XML Data Binding Reference	19
Opening a Java Component Implementation	21
Generating a Java Component Implementation	21
Generate Java Component Implementation Reference	22
Regenerating a Java Component Implementation	25
Upgrading a Java Component Implementation	26
Life Cycle Events	27
Component Context	28
Accessing a Property	28
Accessing a Resource	29
Accessing a Hibernate Resource	29
Accessing a JDBC Resource	30
Accessing JMS Resources	31
Accessing LDAP Connections	32
Accessing SMTP Connections	34
Accessing a Teneo Resource	34
Invoking an HTTP Request	36
Invoking a Reference Operation	41
Error Handling	41
SOAPException Reference	46
Context Parameters	47
Working with Context Parameters	50

Retrieving a Context Parameter from a Request	50
Setting a Context Parameter in a Request	50
Retrieving a Context Parameter from a Response	50
Setting a Context Parameter in a Response	51
.....	51
.....	52
Endpoint References	52
Retrieving an Endpoint Reference	52
Creating an Endpoint Reference	53
Custom Features	55
Bundles and Plug-in Projects	56
Configuring Dependencies on External Java Classes	57
Versions	58
Converting Migrated Java Component Implementations	60
Creating an Abstract Class	60
Editing a Manifest	64
Regenerating a Component Implementation	65
Removing 2.x Data Binding JAR Files	65
Correcting Custom Feature File	66
Default XML to Java Mapping	67

Figures

Dependencies	13
Relaxed Feature Dependency	13

TIBCO Documentation and Support Services

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit <https://docs.tibco.com>.

Product-Specific Documentation

Documentation for TIBCO ActiveMatrix® Service Grid is available on the <https://docs.tibco.com/products/tibco-activematrix-service-grid> page.

Use of the following features, installation profiles and development tools requires a TIBCO ActiveMatrix Service Grid license:

- TIBCO ActiveMatrix Policy Director Governance, TIBCO ActiveMatrix SPM Dashboard, and TIBCO ActiveMatrix SPM Runtime Server profiles; and
- TIBCO ActiveMatrix Service Grid development tools for Java, Webapp and Spring components.

Customers with only a TIBCO ActiveMatrix Service Bus license are not licensed to use these features, tools or profiles.

The following documents form the documentation set:

- *TIBCO ActiveMatrix Service Grid Concepts*: Read this manual before reading any other manual in the documentation set. This manual describes terminology and concepts of the platform. The other manuals in the documentation set assume you are familiar with the information in this manual.
- *TIBCO ActiveMatrix Service Grid Development Tutorials*: Read this manual for a step-by-step introduction to the process of creating, packaging, and running composites in TIBCO Business Studio.
- *TIBCO ActiveMatrix Service Grid Composite Development*: Read this manual to learn how to develop and package composites.
- *TIBCO ActiveMatrix Service Grid Java Component Development*: Read this manual to learn how to configure and implement Java components.
- *TIBCO ActiveMatrix Service Grid Mediation Component Development*: Read this manual to learn how to configure and implement Mediation components.
- *TIBCO ActiveMatrix Service Grid Mediation API Reference*: Read this manual to learn how to develop custom Mediation tasks.
- *TIBCO ActiveMatrix Service Grid Spring Component Development*: Read this manual to learn how to configure and implement Spring components.
- *TIBCO ActiveMatrix Service Grid WebApp Component Development*: Read this manual to learn how to configure and implement Web Application components.
- *TIBCO ActiveMatrix Service Grid REST Binding Development*: Read this manual to learn how to configure and implement REST components.
- *TIBCO ActiveMatrix Service Grid Administration Tutorials*: Read this manual for a step-by-step introduction to the process of creating and starting the runtime version of the product, starting TIBCO ActiveMatrix servers, and deploying applications to the runtime.
- *TIBCO ActiveMatrix Service Grid Administration*: Read this manual to learn how to manage the runtime and deploy and manage applications.

- *TIBCO ActiveMatrix Service Grid Hawk ActiveMatrix Plug-in*: Read this manual to learn about the Hawk plug-in and its optional configurations.
- *TIBCO ActiveMatrix Service Grid Policy Director Governance Custom Actions*: Read this manual to learn how you can configure and enforce policies for ActiveMatrix and external services hosted in third party containers, using TIBCO ActiveMatrix Policy Director Governance.
- *TIBCO ActiveMatrix Service Grid Service Performance Manager API Reference*: Read this manual to learn how to use the SPM APIs.
- *TIBCO ActiveMatrix Service Grid Error Codes*: Read this manual to know more about the error messages and how you could use them to troubleshoot a problem.
- *TIBCO ActiveMatrix Service Grid Installation and Configuration*: Read this manual to learn how to install and configure the software.
- *TIBCO ActiveMatrix Service Grid Security Guidelines*: Read this manual to learn more about security guidelines and recommendations for TIBCO ActiveMatrix Service Grid.
- *TIBCO ActiveMatrix Service Grid Release Notes*: Read this manual for a list of new and changed features, steps for migrating from a previous release, and lists of known issues and closed issues for the release.

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit <http://www.tibco.com/services/support>.
- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support portal at <https://support.tibco.com>.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to <https://support.tibco.com>. If you do not have a user name, you can request one by clicking Register on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the [TIBCO Ideas Portal](#). For a free registration, go to <https://community.tibco.com>.

Java Components

Java components integrate Java classes into the TIBCO ActiveMatrix platform.

The integration conforms to [SCA-J](#) specifications. Java components support service implementation using the flexibility and power of a general purpose programming language.

TIBCO Business Studio facilitates Java component implementation by providing a rich set of automatic code generation and synchronization features. TIBCO Business Studio supports both WSDL-first and code-first development.

You can develop Java components and generate classes that conform to the WSDL interface specification of the component's services and references. When you add a service, reference, or property to a Java component and regenerate the implementation, TIBCO Business Studio adds fields and methods that represent the service, reference, or property to the component's implementation class.

You can also configure an existing Java class as the implementation of a Java component and update component properties to match the implementation.

For information on Java components, see *Java Component Development*.

Creating a Java Component

You can create a Java component by starting with a WSDL file and generating the component implementation, or by creating the component and configuring it with an existing implementation.

Procedure

- Choose an option and follow the relevant procedure.

Option	Description
Wizard	<ol style="list-style-type: none"> Create an SOA project selecting the SOA Project from WSDL project type. <div style="display: flex; align-items: center; margin-top: 10px;">  <div style="border-left: 1px solid black; padding-left: 10px;"> <p>You can generate Java component with two or more WSDLs containing the same operation name.</p> </div> </div> In the Component Details page of the wizard, specify Java for the component implementation type. Specify code generation details as described in Generate Java Component Implementation Reference.
Manual	<ol style="list-style-type: none"> Create an SOA project of any type. Open the composite created in the project. Do one of the following: <ul style="list-style-type: none"> Click the Java icon  in the Palette and click the canvas. Click the canvas and click the Java icon  in the pop-up toolbar. Generate the Java implementation as described in Generating a Java Component Implementation or configure an existing implementation as described in Configuring a Java Components Implementation.
Command-Line	<ol style="list-style-type: none"> Create a command-line build file and specify an <code>sds.createComponent</code> task. Use <code><param name="..." value="..." /></code> subelements to specify details about the

Option	Description
	<p>containing the plug-in that contains the component implementation. For example:</p> <pre data-bbox="523 283 1417 583"> <sds.createComponent projectName="NewSoaProject" compositeName="MyComposite" componentName="MyJavaComponent" implementationLoc="/ MyJavaProject/src/com/example/impl/MyJavaComponentImpl.java"> <param name="feature.id" value="my.custom.feature"/> <param name="feature.file.path" value="/NewSoaProject/Deployment Artifacts/ MyCustomFeature.customfeature"/> <param name="create.new.feature" value="false"/> <param name="use.existing.feature" value="true"/> <param name="feature.version" value="1.0.0.qualifier"/> </sds.createComponent> </pre> <p>2. Run the command-line with the build file.</p>

A Java component is added to the composite and its implementation is configured.

Configuring a Java Components Implementation

When you generate a Java component implementation or create an SOA project from a Java implementation, the component's Implementation field is configured automatically.

You can also manually configure an existing Java class in the workspace as the implementation of a Java component. The class must be contained within a Java plug-in project. If you configure a class that is not contained within a plug-in project, TIBCO Business Studio will convert the project to a plug-in project when you configure the implementation.



The Java class must conform to 3.x format as described in [Converting Migrated Java Component Implementations](#) to be able to regenerate the class after adding references, services, or properties. If your class does not conform to 3.x format, follow the procedures in the section to migrate the implementation class to 3.x format.

Procedure

1. Click the component.
2. In the Properties view, click the **Implementation** tab.
3. Click the **Browse...** button at the right of the Class field.
The Select Implementation Class displays.
4. In the Select Entries field, type a partial class name.
The classes that match the name display in the Matching types list.
5. Click a class and click **OK**.
The Class and Location fields are filled in. An error badge is added to the component. To resolve the error, configure the component's custom feature and update the component.

Updating a Java Component

You typically update a component after you have configured its implementation. You can perform the update from the canvas or from the Problems view.

Procedure

- The procedure depends on the control you want to use.

Control	Procedure
Canvas	1. Right-click the component and select Refresh from Implementation .
Canvas	1. Right-click a component and select Quick Fixes > Update Component from Implementation .
Problems View	<ol style="list-style-type: none"> 1. In the Problems view, right-click an error of the form The component "<i>ComponentName</i>" is out of sync with its implementation and select Quick Fix. 2. In the Quick Fix dialog select Update Component from Implementation. 3. Click Finish.

All the changes made to the component since the implementation was generated are discarded and the component is refreshed from the implementation.

Configuring a Java Components Custom Feature

When you generate a Java component implementation or create an SOA project from a Java implementation, the component's custom feature field is automatically created and configured. If you manually configure the component's implementation, you must manually create and configure the custom feature. If the component implementation uses a library, add the custom feature containing the library in the Properties view.

Procedure

1. Choose an initial control and follow the relevant procedure.

Initial Control	Procedure
Properties View	<ol style="list-style-type: none"> 1. Click the component. 2. In the Properties view, click the Implementation tab. 3. Click the  button to the right of the Features table.
Quick Fix	1. Right-click the component and select Quick Fixes > Select Custom Feature

The Select a Feature dialog displays.

2. In the Select an item to open field, type a partial feature name.
The feature that matches the name displays in the Matching items list.
3. Click a feature and click **OK**.
The feature is added to the Features list.

Upgrading a Java Component

After you modify a Java component and its implementation you should update the relevant plug-in, feature, and component versions. Observe the following guidelines for updating version components according to the type of modification you make:

- Major - Deleting a property, service, reference, or component and any code changes that are not backward compatible.
- Minor - Adding a property, service, reference or component and any changes that are backward compatible.
- Service - Modifying a property, service, reference, or components that is backward compatible. For example, if you make a minor change to the implementation.

Procedure

1. Open the plug-in manifest of the plug-in containing the component implementation.
 - a) In the Overview tab, increment the appropriate version component of the plug-in. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - b) In the Runtime tab, increment the appropriate version component of the exported package. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - c) Save the manifest.
2. Open the custom feature containing the plug-in.
 - a) In the Overview tab, increment the appropriate version component of the feature. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - b) In the Plug-ins tab, increment the appropriate version component of the included plug-in. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - c) Save the feature.
3. Click the modified component.
 - a) In the General tab, increment the appropriate version component of the component. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - b) In the Implementation tab, if the Compute Component Dependencies and Feature Dependencies checkboxes are unchecked, update the version ranges for the component implementation bundle or package and feature as appropriate. For example, if you add a property, change the version range from [1.0.0, 2.0.0) to [1.1.0, 2.0.0). If the checkboxes are checked, TIBCO Business Studio automatically updates the applicable version ranges.
4. Save the composite.
5. Create a DAA containing the upgraded composite and feature.

Component Feature Dependencies

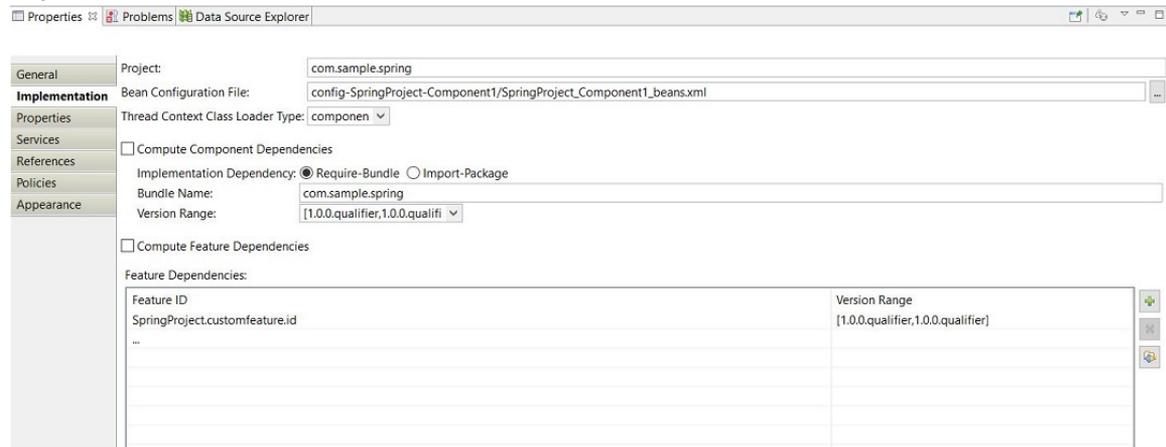
When a component implementation is dependent on a shared library, the feature containing the dependency must be specified in the component's Feature Dependencies table.

By default, a component is configured to depend on the custom features containing:

- The component implementation
- External libraries reference by the component implementation

In both cases, the default version range is set to "[1.0.0.qualifier,1.0.0.qualifier)".

Dependencies



If the *qualifier* component of a version is set to "qualifier" when you create a DAA, TIBCO Business Studio replaces "qualifier" with a generated qualifier that defaults to a timestamp. The effect is that the application requires that the version of the features installed on a node be a perfect match to a version that includes a timestamp.

External Library Dependencies

It is not possible to know the value of the version's qualifier component for the feature containing an external library when you package the composite. Therefore, if you are using an external library, you should "relax" the version range of the feature containing the library. For example, change the range from "[1.0.0.qualifier,1.0.0.qualifier]" to "[1.0.0,2.0.0)" as shown in the following screen shot.

Relaxed Feature Dependency

Feature ID	Version Range
jv.helloworld4.svcs.feature	[1.0.0,2.0.0)
jv.helloworld4.soa.customfeature.id	[1.0.0.qualifier,1.0.0.qualifier]
...	

Java Component Reference

Field	Description
Class	Fully-qualified name of the class that implements the component.
Location	Location of the class in the workspace.
Thread Context Class Loader Type	<p>Configures the value returned by the call <code>Thread.currentThread().getContextClassLoader()</code> inside a Java implementation class (once it is instantiated):</p> <ul style="list-style-type: none"> • component - The class loader of the component configured through the component requirements • bundle - The class loader of the bundle (that is, plug-in) that contains the Java implementation class. • none - A null thread context class loader.

Field	Description
Compute Component Dependencies	<p>Indicate whether to TIBCO Business Studio should compute the component bundle dependencies. When a component is deployed on a node, ActiveMatrix generates a component bundle. When checked, the component implementation bundles required by the component bundle are computed and identified when you package the composite. When unchecked, the Implementation Dependency and Compute Feature Dependencies fields display and you can manually specify the dependencies.</p> <p>Default:</p> <ul style="list-style-type: none"> • New projects - checked. • Legacy projects - unchecked.
Implementation Dependency	<p>Type of the dependency of the component bundle on the component implementation.</p> <ul style="list-style-type: none"> • Require Bundle - The bundle containing the component implementation is declared as a required bundle. When selected, the Bundle Name field appears. • Import Package - The package exported by the component implementation is declared as an imported package. When selected, the Import Package field displays. <p>Default:</p> <ul style="list-style-type: none"> • New projects - Require Bundle. • Legacy projects - Import Package.
Bundle Name	<p>Symbolic name of the bundle containing the component implementation.</p> <p>Default: The bundle in which the component implementation class is present.</p>
Package Name	<p>Name of the package containing the component implementation.</p> <p>Default: The package in which the component implementation class is present.</p>
Version Range	<p>Versions of the bundle or package that satisfy the component bundle's dependency. When specifying a range for a bundle, you can require an exact match to a version that includes a build qualifier. In contrast, the range for a package is inexact.</p> <p>Default:</p> <ul style="list-style-type: none"> • Bundle - [1.0.0.qualifier,1.0.0.qualifier]. • Package - [1.0.0, 2.0.0).

Field	Description
Compute Feature Dependencies	Indicate whether to compute the features on which the component bundle depends. When unchecked the Feature Dependencies table displays. Default: <ul style="list-style-type: none"> • New projects - checked. • Legacy projects - unchecked.
Preview	A link that when clicked displays a dialog containing a list of the features on which the component bundle depends.

Features Dependencies

The features on which the component bundle depends.

Column	Description
Feature ID	ID of the feature.
Version Range	Range of feature versions.

By default the table contains the automatically generated feature containing the component implementation bundle.

sds.CreateComponent Command-Line Task

Element or Parameter	Description
<code>implementationLoc</code>	Workspace relative path to the implementation class.
<code>feature.id</code>	ID of the feature.
<code>feature.file.path</code>	Workspace relative path to the feature file.
<code>create.new.feature</code>	Indicate whether to create a new feature.
<code>use.existing.feature</code>	Indicate whether to use an existing feature.
<code>feature.version</code>	Feature version.

Java Component Implementations

A Java component implementation consists of the abstract and concrete classes that represent the component. The abstract class defines service method signatures, reference fields and accessor methods, and property fields and accessor methods. The concrete class contains the implementations of the service methods. Java component implementations are stored in Java plug-in projects.

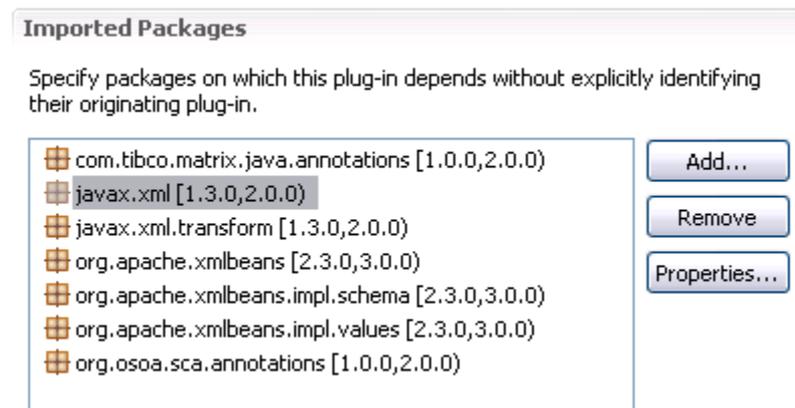
Declaring Dependencies on Packages

Normally, if you import packages and do not add them to the manifest, TIBCO Business Studio displays an error. However, if you import any of the `javax.xml.*` or `org.ietf.jgss` packages and do not declare the import in the manifest, TIBCO Business Studio does not display an error because TIBCO Business Studio resolves those packages from the configured JRE. If you then deploy the application without the declaration in the manifest, the application will not run. Hence, you must ensure that you import `javax.xml` or `org.ietf.jgss` packages in the manifest file.

For example, if you imported the following classes:

```
import javax.xml.XMLConstants;
import javax.xml.transform.TransformerFactory;
```

The corresponding import packages in the manifest should be:



Each subpackage of javax.xml may have a different version:



```

  javax.xml (1.3.0)
  javax.xml.bind (2.1.0)
  javax.xml.bind.annotation (2.1.0)
  javax.xml.bind.annotation.adapters (2.1.0)
  javax.xml.bind.attachment (2.1.0)
  javax.xml.bind.helpers (2.1.0)
  javax.xml.bind.util (2.1.0)
  javax.xml.crypto
  javax.xml.crypto.dom
  javax.xml.crypto.dsig
  javax.xml.crypto.dsig.dom
  javax.xml.crypto.dsig.keyinfo
  javax.xml.crypto.dsig.spec
  javax.xml.datatype (1.3.0)
  javax.xml.namespace (1.1.0)
  javax.xml.parsers (1.3.0)
  
```

Data Binding

Data binding is the process of converting objects described in an XML document to Java objects and vice versa. You can generate data binding classes directly from a WSDL or schema document or while generating a Java or Spring component implementation.

TIBCO Business Studio supports two data binding technologies: [JAXB](#) and [XMLBeans](#). The default mapping of WSDL and XSD schema elements to Java programming language elements is described in [Default XML to Java Mapping](#).

Data Binding Configuration Files

You can change the mapping of XML elements to Java objects by specifying mapping constraints in a data binding configuration file. Each data binding technology has its own configuration file format: XMLBeans XSDCONFIG or JAXB XJB. See the XMLBeans and JAXB specifications for the formats of their configuration files.

For example, the following XMLBeans configuration file maps the target namespace `http://ns.tibco.com/Hello/` to the package `com.sample.othername.hello` and specifies suffixes, prefixes, and name replacements for generated classes.

```

<xb:config xmlns:xb="http://xml.apache.org/xmlbeans/2004/02/xbean/config"
           xmlns:ct="http://ns.tibco.com/Hello/">

  <xb:namespace uri="http://ns.tibco.com/Hello/ http://someurihere.com">
    <xb:package>com.sample.othername.hello</xb:package>
  </xb:namespace>

  <!--
    The ##any value is used to indicate "all URIs".
    The <prefix> tag is used to prepend to top-level Java type names generated in
    the specified namespace.
    The <suffix> tag is used to append to top-level Java types names generated in
    the specified namespace.
    The <prefix> and <suffix> are not used for inner Java types.
  -->

  <xb:namespace uri="##any">
    <xb:prefix>Xml</xb:prefix>
    <xb:suffix>BeanClass</xb:suffix>
  
```

```

</xb:namespace>

<!-- The <qname> tag specifies a Java class name for a qualified name -->
<xb:qname name="ct:HelloResponse" javaname="TheHelloResponse" />

</xb:config>

```

Generating XML Data Binding Classes

You can generate XML data binding classes for you SOA project from the Project Explorer. When you complete the process, a JAR file and optional schema document are created.

Steps 1 through 3 are optional. They are recommended if you want to share the data binding classes between more than one SOA project.

Procedure

1. Create a SOA project. In the Asset Type Selection screen, uncheck the **TIBCO SOA Platform** checkbox.
2. Click **Finish**.
In the Project Explorer, an SOA project is created containing only the `Service Descriptors` special folder.
3. Import or create WSDL and schema documents in the `Service Descriptors` folder.
4. In the Project Explorer, right-click a WSDL or schema document and select **Generate XML Data Bindings**.
The XML Data Binding Classes dialog displays.
5. Configure the XML data binding type, Beans, and interface JAR file properties as described in [XML Data Binding Reference](#).
6. Click **Finish**.
A JAR file containing the XML data binding classes for each WSDL and schema document is created in the specified location unless you click the Use this JAR for All Data Bindings link.

Data Binding Classes for Abstract and Concrete WSDL Files

Using an abstract WSDL and its generated concrete WSDL in services and references of the same component requires special consideration if the WSDL contains an embedded schema. When a concrete WSDL file is generated from an abstract WSDL file that has an embedded schema, the resulting concrete WSDL file will also contain the same embedded schema. When you generate data binding classes for both WSDL files, the code generator generates duplicate classes for the embedded schema.

The impact of this is two-fold:

- Generating the data binding classes for the abstract and concrete WSDL files into a single JAR is not supported.
- When you generate the data binding classes for the two WSDL files into different bean JARs, both JARs will contain the same classes.

For correct operation, you must manually remove one of the resulting bean JARs from the bundle containing the bean JARs as follows:

1. Open the component implementation bundle's manifest in the Manifest Editor.
2. Click the Runtime tab, and delete one of the JARs containing the duplicate bean classes from the Classpath area.
3. Save the manifest.

To avoid having to manually edit the manifest, the recommended method for using abstract and concrete WSDL files in the same composite is to use only abstract WSDL files for the component

services and references and use the concrete WSDL only for the corresponding promoted references as follows:

1. Delete the wire between component references and promoted references.
2. Configure the component reference with the abstract WSDL.
3. Configure the promoted reference with the concrete WSDL.
4. Wire the component reference to the promoted reference using the Wire tool in the Palette.

XML Data Binding Reference

Field	Description
Type	<p>The type of the data binding being generated: XMLBeans or JAXB.</p> <p>If a JAR file already exists for the contract selected in the Contracts list, and you choose a binding type different than the one that exists in the JAR file or the contract has changed since the JAR file was generated, the Overwrite Existing JAR checkbox will be checked.</p> <p>Default: XMLBeans.</p> <p> Generating implementations for two or more components in the same Java plug-in project using different binding types is not supported.</p>
Contracts	A list of WSDL and schema files for which XML data binding classes will be generated.
JAR Type	The type of JAR file being generated: Beans or Interface.
Source File	The path to the source file containing the selected contract.
JAR File	<p>The path to the generated JAR file.</p> <p>Default: When generating a component implementation:</p> <ul style="list-style-type: none"> • Beans - <i>projectName</i>/libs/<i>contractFileName</i>.wsdl.jar • Interface - <i>projectName</i>/libs/<i>contractFileName</i>.wsdl_interface.jar <p>where <i>contractFileName</i> is the name of the file containing the contract selected in the Contracts list and <i>projectName</i> is the name of the project containing the component implementation.</p> <p>When generating from a contract file:</p> <ul style="list-style-type: none"> • Beans - <i>projectName</i>.libs/libs/<i>contractFileName</i>.wsdl.jar • Interface - <i>projectName</i>.libs/libs/<i>contractFileName</i>.wsdl_interface.jar <p>where <i>contractFileName</i> is the name of the file containing the contract selected in the Contracts list and <i>projectName</i> is the name of the project containing the contract file.</p>
Use this JAR for All Data Bindings	Indicate that all data binding classes should be generated into the JAR file specified in the JAR File field. You must generate all data binding classes into a single JAR file whenever there are cyclical references between schema files.

Field	Description
Set JAR Folder	<p>Invokes a dialog where you can set the folder to contain generated JAR files:</p> <ul style="list-style-type: none"> All Generated JARs - All JAR files will be generated in the same folder as the destination of the currently selected JAR. New Generated JARs - Only newly generated JAR files will be generated in the same folder as the destination of the currently selected JAR file. <p> Setting the JAR folder affects only the JAR files generated by the wizard. It has no effect outside the wizard nor on subsequent wizard runs.</p> <p>Default: All Generated JARs.</p>
JAR Status	<p>The status of the JAR file containing the classes generated for the selected contract:</p> <ul style="list-style-type: none"> JAR is non-existent and will be generated. - The JAR file does not exist. Different binding type. JAR must be overwritten. - The value of the Type field is different than the type of the data binding classes in the JAR file. JAR exists and will be overwritten. - The JAR file exists and the Overwrite Existing JAR checkbox is checked. JAR exists and will be preserved. - The JAR file exists and the Overwrite Existing JAR checkbox is unchecked. JAR is outdated and will be overwritten. - The selected contract has changed since the JAR file was generated and the Overwrite Existing JAR checkbox is checked, so the JAR file will be generated. JAR is outdated and will be preserved. - The selected contract has changed since the JAR file was generated and the Overwrite Existing JAR checkbox is unchecked, so the JAR file will not be generated.
Overwrite Existing JAR	<p>Enabled only when the JAR file exists. When checked, the JAR file will be regenerated. When unchecked, the existing file will be reused and will not be modified.</p>
Advanced	
Export Data Binding Packages	<p>Indicate that all packages of classes generated into the same plug-in as the component implementation should be exported in the component's implementation plug-in manifest using the Export-Package directive. This allows you to reuse data binding JAR files generated into the same plug-in as the component implementation. However, this is not the recommended approach for data binding library sharing. Instead, you should generate data binding JAR files into a separate plug-in project.</p> <p>Default: Unchecked.</p>
Use Configuration File	<p>Indicate that the specified data binding configuration file should be used when generating JAR files. When you check the checkbox, the text field is enabled.</p> <p>Default: Unchecked.</p>

Opening a Java Component Implementation

You can open a Java component implementation from the canvas, the Properties view, or the project explorer.

Procedure

- Choose an initial control and follow the relevant procedure.

Control	Procedure
Canvas	Double-click the component.
Properties View	<ol style="list-style-type: none"> Click the Implementation tab. Click the Class field label.
Project Explorer	Right-click the implementation file and select Open With > Java Editor .
Canvas	Right-click the component and select Open Implementation .

The implementation file opens in the Java editor.

Generating a Java Component Implementation

You can start the wizard for generating a Java component implementation from the canvase, from the Property view, or from the Problems view.

Procedure

- Choose an initial control and follow the relevant procedure.

Control	Procedure
Properties View	<ol style="list-style-type: none"> In the Validation Report area on the General tab of the component's Property View, click the fix... link. Select Generate Java Implementation.
Canvas	<ol style="list-style-type: none"> Right-click the component and select Quick Fixes > Generate Java Implementation.
Canvas	<ol style="list-style-type: none"> Right-click the component and select Generate Java Implementation.
Problems View	<ol style="list-style-type: none"> In the Problems view, right-click an error of the form Component "<i>ComponentName</i>" is not configured and select Quick Fixes. In the Quick Fix dialog, click Generate Java Implementation. Click Finish.

The Generate Java Implementation dialog displays.

- Configure the project, package, and class details in the [Implementation Classes](#) screen.
- If the component has a service or reference, choose a data binding option and follow the appropriate procedure:

Data Binding Option	Procedure
Accept Defaults	A JAR containing XMLBeans data binding classes with the default options is generated.
Configure	<ol style="list-style-type: none"> 1. Click Next. 2. Configure the data binding type and Beans and interface JAR properties as described in XML Data Binding Reference.

4. Click **Finish**.

The following objects are generated:

- A Java plug-in project containing abstract and concrete implementation classes. If the component has a service or reference, interface and data binding classes are also generated.
- A custom feature file that references the Java plug-in in the `Deployment Artifacts` special folder in the SOA project. The component is configured to depend on the generated custom feature. If the component implementation depends on a data binding library plug-in, the component is configured to depend on the custom feature containing the data binding library.
- Objects that map to the following component elements:
 - Service - An interface. If the port type is named *PortType*, the interface is named *PortType*. The clause `implements PortType` is added to the abstract class.
 - Reference - Field and accessor methods are added to the abstract class.
 - Property - Field and accessor methods are added to the abstract class.

Generate Java Component Implementation Reference

When you generate a Java implementation you specify the project location, package, and names of the implementation classes and the type of the XML data binding classes and location of the JAR file containing the classes.

Implementation Classes

Field	Description
Project	The name of the plug-in project to contain the implementation. Default: <code>com.sample.soaprojectname</code> .
Source Folder	The name of the source folder in the plug-in project. Default: <code>src</code> .
Package	The name of the package of the implementation class. Default: <code>com.sample.soaprojectname</code> .
Class	The name of the implementation class. Default: <code>ComponentName</code> , where <i>ComponentName</i> is the name of the component.
Overwrite Concrete Class	Indicate whether to overwrite the implementation class if it already exists. Default: Unchecked.

Field	Description
Use Default Location for Superclass	Indicate whether to generate the superclass of the implementation class in the same package as the implementation class and name the class <i>AbstractComponentName</i> . When unchecked, the Superclass Package and Superclass fields are enabled. Default: Checked.
Superclass Package	The name of the package of the superclass of the implementation class. Default: <i>com.sample.soaprojectname</i> .
Superclass	The name of the superclass of the implementation class. Default: <i>AbstractComponentName</i> .

XML Data Binding Classes



If the component does not have any services or references, this screen does not display.

Field	Description
Type	The type of the data binding being generated: XMLBeans or JAXB. If a JAR file already exists for the contract selected in the Contracts list, and you choose a binding type different than the one that exists in the JAR file or the contract has changed since the JAR file was generated, the Overwrite Existing JAR checkbox will be checked. Default: XMLBeans.  Generating implementations for two or more components in the same Java plug-in project using different binding types is not supported.
Contracts	A list of WSDL and schema files for which XML data binding classes will be generated.
JAR Type	The type of JAR file being generated: Beans or Interface.
Source File	The path to the source file containing the selected contract.

Field	Description
JAR File	<p>The path to the generated JAR file.</p> <p>Default: When generating a component implementation:</p> <ul style="list-style-type: none"> • Beans - <i>projectName</i>/libs/<i>contractFileName</i>.wsdl.jar • Interface - <i>projectName</i>/libs/<i>contractFileName</i>.wsdl_interface.jar <p>where <i>contractFileName</i> is the name of the file containing the contract selected in the Contracts list and <i>projectName</i> is the name of the project containing the component implementation.</p> <p>When generating from a contract file:</p> <ul style="list-style-type: none"> • Beans - <i>projectName</i>.libs/libs/<i>contractFileName</i>.wsdl.jar • Interface - <i>projectName</i>.libs/libs/<i>contractFileName</i>.wsdl_interface.jar <p>where <i>contractFileName</i> is the name of the file containing the contract selected in the Contracts list and <i>projectName</i> is the name of the project containing the contract file.</p>
Use this JAR for All Data Bindings	<p>Indicate that all data binding classes should be generated into the JAR file specified in the JAR File field. You must generate all data binding classes into a single JAR file whenever there are cyclical references between schema files.</p>
Set JAR Folder	<p>Invokes a dialog where you can set the folder to contain generated JAR files:</p> <ul style="list-style-type: none"> • All Generated JARs - All JAR files will be generated in the same folder as the destination of the currently selected JAR. • New Generated JARs - Only newly generated JAR files will be generated in the same folder as the destination of the currently selected JAR file. <div style="border-left: 1px solid black; padding-left: 10px; margin-top: 10px;">  <p>Setting the JAR folder affects only the JAR files generated by the wizard. It has no effect outside the wizard nor on subsequent wizard runs.</p> </div> <p>Default: All Generated JARs.</p>

Field	Description
JAR Status	<p>The status of the JAR file containing the classes generated for the selected contract:</p> <ul style="list-style-type: none"> • JAR is non-existent and will be generated. - The JAR file does not exist. • Different binding type. JAR must be overwritten. - The value of the Type field is different than the type of the data binding classes in the JAR file. • JAR exists and will be overwritten. - The JAR file exists and the Overwrite Existing JAR checkbox is checked. • JAR exists and will be preserved. - The JAR file exists and the Overwrite Existing JAR checkbox is unchecked. • JAR is outdated and will be overwritten. - The selected contract has changed since the JAR file was generated and the Overwrite Existing JAR checkbox is checked, so the JAR file will be generated. • JAR is outdated and will be preserved. - The selected contract has changed since the JAR file was generated and the Overwrite Existing JAR checkbox is unchecked, so the JAR file will not be generated.
Overwrite Existing JAR	Enabled only when the JAR file exists. When checked, the JAR file will be regenerated. When unchecked, the existing file will be reused and will not be modified.
Advanced	
Export Data Binding Packages	<p>Indicate that all packages of classes generated into the same plug-in as the component implementation should be exported in the component's implementation plug-in manifest using the Export-Package directive. This allows you to reuse data binding JAR files generated into the same plug-in as the component implementation. However, this is not the recommended approach for data binding library sharing. Instead, you should generate data binding JAR files into a separate plug-in project.</p> <p>Default: Unchecked.</p>
Use Configuration File	<p>Indicate that the specified data binding configuration file should be used when generating JAR files. When you check the checkbox, the text field is enabled.</p> <p>Default: Unchecked.</p>

Regenerating a Java Component Implementation

You should regenerate the component implementation after you add a service, reference, or property to the component or to recreate the data binding classes created by a previous version of TIBCO Business Studio. The regeneration will regenerate the abstract class, but it will not change or remove any code from the implementation class.

Prerequisites

The implementation must have been originally generated by TIBCO Business Studio.

It is possible that the implementation class will have errors after regeneration (for example when the reference has been removed and the implementation is using the reference).

If the implementation was generated with a previous version of TIBCO Business Studio, a dialog displays asking if you want to delete legacy JARs. Legacy JARs are named *PortType-timestamp-service-beans.jar* and *PortType-timestamp-service-interface.jar*.

Procedure

1. Choose an initial control and follow the relevant procedure.

Control	Procedure
Canvas	1. Right-click the component and select Regenerate Java Implementation .
Problems View	<ol style="list-style-type: none"> 1. In the Problems view, right-click an error of the form The component "<i>ComponentName</i>" is out of sync with its implementation and select Quick Fix. 2. In the Quick Fix dialog select Update Implementation Class. 3. Click Finish.
Command Line	<ol style="list-style-type: none"> 1. Create a command-line build file and specify an <code>sds.javait.regenerateComponent</code> task. For example: <pre><sds.javait.regenerateComponent projectName="MyProject" compositeName="MyComposite" componentName="MyJavaComponent" /></pre> 2. Run the command-line with the build file.

2. Decide how you want to handle legacy JARs and follow the appropriate procedure.

Legacy JARs	Procedure
Delete	1. Click Yes .
Retain	<ol style="list-style-type: none"> 1. Open the <code>META-INF/Manifest.MF</code> file in the component implementation's Java plug-in project. 2. Delete the legacy JARs from the <code>Bundle-ClassPath</code> property. 3. If the preceding step failed, right-click the Java project and select PDE Tools > Update classpath. 4. If necessary, remove legacy JARs from the build properties file in the component implementation's Java plug-in project. 5. Delete the legacy JARs from the project. 6. Fix compilation errors if any.

The implementation is updated to match the component.

Upgrading a Java Component Implementation

Perform these steps after you modify a component implementation.

Procedure

1. Open the plug-in manifest of the plug-in containing the component implementation.

- a) In the Overview tab, increment the appropriate version component of the plug-in. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - b) In the Runtime tab, increment the appropriate version component of the exported package. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - c) Save the manifest.
2. Open the custom feature containing the plug-in.
 - a) In the Overview tab, increment the appropriate version component of the feature. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - b) In the Plug-ins tab, increment the appropriate version component of the included plug-in. For example, if you add a property, change 1.0.0.qualifier to 1.1.0.qualifier.
 - c) Save the feature.

Life Cycle Events

The ActiveMatrix runtime exposes component life cycle events—Init and Destroy—to component implementations.

Methods annotated with @Init and @Destroy are invoked when the life cycle events trigger. [Life Cycle Events](#) describes the meaning of each event and how component implementations can handle each event.

Life Cycle Events

Event	When Invoked
Init	<p>When the application containing the component or the component is started.</p> <p>When this event is triggered all the component's properties, references, and resources have been initialized.</p> <p>The method invoked when this event is triggered is typically used to validate component configuration and open connection to resources.</p>
Destroy	<p>When the application containing the component or the component is stopped.</p> <div style="display: flex; align-items: center;">  <div style="border-left: 1px solid black; padding-left: 10px;"> <p>If you open connections to resources in a method that is invoked by an Init event you must close the connections to the resources in the method that is invoked by a Destroy event.</p> </div> </div>

When TIBCO Business Studio generates a Java or Spring component implementation, it automatically adds the appropriately annotated initialization and destruction methods:

```
org.osoa.sca.annotations.Init;
org.osoa.sca.annotations.Destroy;
@Init
public void init()
{
    // Component initialization code.
    // All properties are initialized and references are injected.
}
@Destroy
public void destroy()
{
    // Component disposal code.
    // All properties are disposed.
}
```

You can customize these methods to perform application-specific initialization and cleanup.

Component Context

A *component context* provides access to the context in which a component executes. The context includes the component's name and containing application name, the node on which it executes, the host managing the node, context parameters available to the component, the component's work area, and so on.

To access the component context, add the following declarations to a Java or Spring component implementation:

```
import org.osoa.sca.annotations.Context;
import com.tibco.amf.platform.runtime.extension.context.ComponentContext;
@Context
public ComponentContext componentContext;
```

These declarations are automatically added to the abstract component implementation when a context parameter is defined for the component. The TIBCO ActiveMatrix platform injects the component context object into the component implementation.

If a component implementation wants to create a file, it should do so in the work area assigned to each component. The TIBCO ActiveMatrix platform ensures that these files are deleted when the component is undeployed. Work areas are backed up during node upgrade. A component implementation can retrieve its work area through the component context's `getWorkArea()` method which returns the `java.io.File` object that represent the work area folder for that component.

Accessing a Property

When you generate a Java or Spring component implementation after adding a property to the component, TIBCO Business Studio adds properties and methods to the component's abstract implementation class:

The following items are added.

- SCA property annotation import
- A field that represents the property
- Accessor methods

The TIBCO ActiveMatrix platform injects the property object into the component implementation.

For example, if you add a property named `greeting` of type `String` to a component, the following code is added:

```
org.osoa.sca.annotations.Property;
private String greeting;

@Property(name = "greeting")
public void setGreeting(String greeting)
{
    this.greeting = greeting;
}

public String getGreeting()
{
    return greeting;
}
```

To reference the property invoke the accessor methods. For example:

```
resp.setHelloResponse(getGreeting() + " " + name + "! "
+ "The current time is " + time + ".");
```

Accessing a Resource

You can access a resource with an accessor method that is part of the component implementation.

Procedure

1. Add a property of the resource type to the component.
2. Generate or regenerate the component implementation. TIBCO Business Studio adds imports, fields, and resource accessor methods to the component's abstract implementation class. When the component is instantiated, the TIBCO ActiveMatrix platform injects the resource object into the component implementation.
3. Access the resource using the generated accessor methods.
4. After you are finished with the resource and any objects retrieved from the resource, close the objects.

Accessing a Hibernate Resource

You can access a Hibernate resource from the Hibernate session that is associated with the session factory

If you create a property named `sessionFactory` of type `Hibernate Resource Template`, TIBCO Business Studio adds the following to the abstract implementation class:

```
import org.osoa.sca.annotations.Property;
import
com.tibco.amf.sharedresource.runtime.core.hibernate.sharedresource.ProxySessionFacto
ry;

private ProxySessionFactory sessionFactory;

    @Property(name = "sessionFactory")
    public void setSessionFactory(ProxySessionFactory sessionFactory)
    {
        this.sessionFactory = sessionFactory;
    }

    public ProxySessionFactory getSessionFactory()
    {
        return sessionFactory;
    }
```

Procedure

1. Retrieve the proxy session factory using the generated `getSessionFactory` method.
2. Register the model class using the session factory `addClass` method.
3. Retrieve the Hibernate session from the session factory using the `openSession` method.
4. Retrieve a transaction from the session.
5. Create a query.
6. Execute the query.
7. Save the session.
8. Commit the transaction.
9. Close the session.

10. When the component is destroyed, unregister the model class using the `removeClass` method.

```
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
...
    final Session session = getSessionFactory().openSession();
    try
    {
        /**
         * Begin a transaction before performing any queries.
         * Closing the session cleans up the transaction.
         */
        Transaction tx = session.beginTransaction();
        final Query query = session.createQuery("UPDATE ...");
        ...
        int result = query.executeUpdate();
        if (result == 0)
        {
            ...
            session.save(report);
        }
        tx.commit();
    } finally
    {
        session.close();
    }
    ...

    @Init
    public void init()
    {
        if (getSessionFactory() == null)
        {
            throw new IllegalStateException("Failed to inject
ProxySessionFactory");
        }

        /** Register the ModelClass model class on SessionFactory */
        getSessionFactory().addClass(ModelClass.class);

        try
        {
            // Initializes database data.
            initializeDBData();
        } catch (Throwable th) {
            ...
        }
    }
    ...
    @Destroy
    public void destroy()
    {
        if (getSessionFactory() != null)
        {
            /**
             * Unregister the ModelClass model class from SessionFactory
             */
            getSessionFactory().removeClass(ModelClass.class);
        }
    }
}
```

Accessing a JDBC Resource

If you create a property named `jdbcr` of type JDBC Resource Template, TIBCO Business Studio adds the following to the abstract implementation class:

```
import org.osoa.sca.annotations.Property;
import javax.sql.DataSource;
```

```
private DataSource jdbcr;

    @Property(name = "jdbcr")
    public void setDbr(DataSource jdbcr) {
        this.jdbcr = jdbcr;
    }

    public DataSource getJdbc() {
        return jdbcr;
    }
}
```

Procedure

- Invoke the accessor methods in your component implementation.

```
import javax.sql.DataSource;
DataSource ds = getJdbc();
Connection connection = null;

try {
    connection = ds.getConnection();
    ensureTablesExist(connection);

    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(sqlString);
    PhoneEntryType entry = null;

    while(rs.next()) {
        entry = resp.addNewOut();
        entry.setEntryId(rs.getString("id"));
        entry.setFirstName(rs.getString("firstName"));
        entry.setLastName(rs.getString("lastName"));
        entry.setPhone(rs.getString("phone"));
    }

} catch(SQLException e) {
    e.printStackTrace();
} finally {
    try{
        connection.close();
    }catch(Exception e){};
    ...
}
}
```

Accessing JMS Resources

To access JMS resources, create JMS Connection Factory and JMS Destination properties. If you create a property named connectionFactory of type JMS Connection Factory and a property named destination of type JMS Destination, TIBCO Business Studio adds the following to the abstract implementation class:

```
import javax.jms.ConnectionFactory;
import javax.jms.Destination;

private ConnectionFactory connectionFactory;

    @Property(name = "connectionFactory")
    public void setConnectionFactory(ConnectionFactory connectionFactory) {
        this.connectionFactory = connectionFactory;
    }

    public ConnectionFactory getConnectionFactory() {
        return connectionFactory;
    }

private Destination destination;
```

```

@property(name = "destination")
public void setDestination(Destination destination) {
    this.destination = destination;
}

public Destination getDestination() {
    return destination;
}

```

Procedure

- Invoke the accessor methods in your component implementation.

```

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSEException;

import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

@Init
public void init() throws JMSEException {
    connection = getConnectionFactory().createConnection();
    connection.start();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    System.out.println(session);
}

@Destroy
public void destroy() throws JMSEException {
    session.close();
    connection.stop();
    connection.close();
}

private String doPublish(String input ) throws JMSEException {
    MessageProducer producer = session.createProducer(getDestination());
    TextMessage message = session.createTextMessage("Input from SOAP
Request : " + input );
    producer.send(message);
    String msg_id = message.getJMSMessageID();
    return msg_id;
}

```

Accessing LDAP Connections

If you create a property named `ldapr` of type LDAP Connection Resource Template, TIBCO Business Studio adds the following to the abstract implementation class:

```

import org.osoa.sca.annotations.Property;
import javax.naming.ldap.LdapContext;

private LdapContext ldapr;

@property(name = "ldapr")
public void setLdap(LdapContext ldapr) {
    this.ldapr = ldapr;
}

public LdapContext getLdap() {
    return ldapr;
}

```

Procedure

1. To update the resource:

```

...
Attributes attr = new BasicAttributes(true);
Attribute objFact = new BasicAttribute("objectclass");
objFact.add("top");
objFact.add("person");
objFact.add("uidObject");
objFact.add("organizationalPerson");

attr.put(objFact);
attr.put("uid", uid);
attr.put("cn", commonName);
attr.put("sn", surname);
attr.put("userPassword", password);
Name name = new LdapName("uid=" + uid + ",ou=People,dc=tibco,dc=com");
getLdapr().createSubcontext(name, attr);
...
public void destroy() {
    try {
        getLdapContext().close();
    } catch (NamingException e) {
        e.printStackTrace();
    }
}
}

```

2. To query the resource:

```

...
StringBuffer sb = new StringBuffer();
NamingEnumeration<SearchResult> results = null;
try {
    SearchControls controls = new SearchControls();
    controls.setSearchScope(SearchControls.SUBTREE_SCOPE);
    MessageFormat format = new MessageFormat("&(uid={0})(objectclass=*)");
    String format2 = format.format(new Object[] { uid });
    results = getLdapr().search("uid=" + uid + ",ou=People,dc=tibco,dc=com",
        format2,controls);

    while (results.hasMore()) {
        SearchResult searchResult = (SearchResult) results.next();
        Attributes attributes = searchResult.getAttributes();
        NamingEnumeration<? extends Attribute> enumeration =
attributes.getAll();
        for (; enumeration.hasMoreElements();) {
            Attribute object = (Attribute) enumeration.next();
            sb.append(object.toString());
            sb.append('\n');
            if (logger.isInfoEnabled()) {
                logger.info(object.toString() );
            }
        }
    }

} catch (NameNotFoundException e) {
    ...
} catch (NamingException e) {
    ...
}

return sb.toString();
...

```

Accessing SMTP Connections

If you create a property named `smtpr` of type SMTP Resource Template, TIBCO Business Studio adds the following to the abstract implementation class:

```
import org.osoa.sca.annotations.Property;
import javax.mail.Session;

private Session smtpr;

    @Property(name = "smtpr")
    public void setSmtpr(Session smtpr) {
        this.smtpr = smtpr;
    }

    public Session getSmtpr() {
        return smtpr;
    }
}
```

Procedure

- Invoke the accessor methods in your component implementation.

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
...
    Transport transport = null;
    try{
        Session session = getSmtpr();
        transport = session.getTransport();
        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress(mailFrom));
        InternetAddress dests[] = new InternetAddress[]{ new
InternetAddress(mailTo) };
        message.setRecipients(Message.RecipientType.TO, dests);
        message.setSubject(subject);

        message.setDataHandler(new DataHandler(new ByteArrayDataSource(
            requestContent, "text/plain")));
        transport.connect();
        transport.sendMessage(message, dests);
    } catch(Exception exp){
        ...
    }
    return false;
} finally {
    if (transport != null)
        try {
            transport.close();
        } catch (MessagingException e) {
            e.printStackTrace();
        }
}
return true;
...
}
```

Accessing a Teneo Resource

If you create a property named `sessionFactory` of type Teneo Resource Template, TIBCO Business Studio adds the following to the abstract implementation class:

```
import org.osoa.sca.annotations.Property;
import
com.tibco.amf.sharedresource.runtime.core.teneo.sharedresource.TeneoSessionFactory;
```

```

private TeneoSessionFactory sessionFactory;

@property(name = "sessionFactory")
public void setSessionFactory(TeneoSessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}

public TeneoSessionFactory getSessionFactory() {
    return sessionFactory;
}

```

Procedure

- Invoke the accessor methods in your component implementation.

```

...
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
    Session session = null;
    Transaction tx = null;
    try {
        session = getSessionFactory().openSession();
        tx = session.beginTransaction();
        Query q = session.createQuery("...");
        User user = (User)q.uniqueResult();
        Trip trip = ...;
        ...
        id = (Long) session.save(trip);
        user.getTrips().add(trip);
        session.save(user);
    }
    catch (Throwable th) {
        error = true;
        result = "failed: " + th.getMessage();
        th.printStackTrace();
    }
    finally {
        if (tx != null) {
            if (error) {
                try {
                    tx.rollback();
                }
                catch (Throwable th) {
                    th.printStackTrace();
                }
            }
            else {
                try {
                    tx.commit();
                }
                catch (Throwable th) {
                    th.printStackTrace();
                }
            }
        }
        if (session != null) {
            try {
                session.close();
            }
            catch (Throwable th) {
                th.printStackTrace();
            }
        }
    }
    ...

```

Invoking an HTTP Request

You can use an HTTP client resource to invoke HTTP requests from component implementations. A POST example illustrates this.

Procedure

1. Add a property of type HTTP Client Resource Template to the component.

When you regenerate the implementation, TIBCO Business Studio adds an HTTP client connection factory property to the abstraction implementation class. For a property named `httpClientConnectionFactory`, TIBCO Business Studio adds the following:

```
import
com.tibco.amf.sharedresource.runtime.core.http.httpClient.HttpClientConnectionFac
tory;

private HttpClientConnectionFactory httpClientConnectionFactory;

    public void setHttpClientConnectionFactory(
        HttpClientConnectionFactory httpClientConnectionFactory) {
        this.httpClientConnectionFactory = httpClientConnectionFactory;
    }

    /**
     * @return Returns the HttpClientConnectionFactory
     */
    public HttpClientConnectionFactory getHttpClientConnectionFactory() {
        return httpClientConnectionFactory;
    }
```

2. Retrieve an HTTP client object from the connection factory.
3. Invoke HTTP methods on the HTTP client object.

Post Example

The following example demonstrates how to invoke an HTTP Post request using an HTTP client object retrieved from the HTTP client connection factory or using an HTTP connection:

```
import org.apache.http.HttpClientConnection;
import org.apache.http.HttpEntity;
import org.apache.http.HttpException;
import org.apache.http.HttpHost;
import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.InputStreamEntity;
import org.apache.http.protocol.BasicHttpContext;
import org.apache.http.protocol.BasicHttpProcessor;
import org.apache.http.protocol.ExecutionContext;
import org.apache.http.protocol.HttpRequestExecutor;
import org.apache.http.protocol.RequestConnControl;
import org.apache.http.protocol.RequestContent;
import org.apache.http.protocol.RequestExpectContinue;
import org.apache.http.protocol.RequestTargetHost;
import org.apache.http.protocol.RequestUserAgent;

    public GetQuoteResponseDocument getQuote(GetQuoteDocument parameters) {
        String symbol = parameters.getGetQuote().getSymbol();
        String value = ERROR_MSG;
        try {
            /**
             * Two ways of using HTTP client API, randomly selected at
runtime:
             * a) HTTP Client
             * b) HTTP Connection
             */
            if(random.nextBoolean()){
                value =
getQuoteUsingHttpClient(getHttpClientConnectionFactory(), symbol.trim());
            }else{
                value = getQuoteUsingHttpConnection(getHttpClientConnectionFactory(),
symbol.trim());
            }
        } catch (Exception e) {
            if(logger.isErrorEnabled()){
                logger.error(ERROR_MSG,e);
            }
        }

        GetQuoteResponseDocument responseDoc =
GetQuoteResponseDocument.Factory.newInstance();
        responseDoc.addNewGetQuoteResponse().setGetQuoteResult(value);
        return responseDoc;
    }

    /**
     * Processes the request using HTTPClient API
     */
    private String getQuoteUsingHttpClient(HttpClientConnectionFactory
connectionFactory,
        String symbol) throws HttpException, ClientProtocolException,
IOException {
        String responseString = ERROR_MSG;
        String message = getContent(symbol);
        byte[] bytes = message.getBytes();

        /** HTTPClient provides a facade to a number of special purpose
handler or strategy
         * implementations responsible for handling of a particular aspect of
         * the HTTP protocol such as redirect or authentication handling or
         * making decision about connection persistence and keep alive
duration.
         * This allows you to selectively replace the default implementation
         * of those aspects with custom, application-specific ones.

```

```

    */
    HttpClientWrapper httpClient = connectionFactory.getHttpClient();
    Host configuration = connectionFactory.getHostConfiguration();

    /**
     * Construct the request URL
     */
    String url = configuration.getSchemeName() + "://" +
configuration.getHostName()+":"+
        configuration.getPort() + "/stockquote.asmx";

    /**
     * Prepare request object and its header for HTTP Post request
     */
    HttpPost httpPost = new HttpPost(url);
    httpPost.setHeader("Content-Type", "text/xml; charset=utf-8");
    httpPost.setHeader("SOAPAction", "http://www.webserviceX.NET/
GetQuote");

    /**
     * Sets the Entity to handle content management.
     */
    ByteArrayInputStream instream = new ByteArrayInputStream(bytes);
    InputStreamEntity e2 = new InputStreamEntity(instream, -1);
    httpPost.setEntity(e2);

    /**
     * Execute the POST URL using HttpClientWrapper which takes care of
     * connection management other functionality internally.
     */
    HttpResponse response = httpClient.execute(httpPost);
    /**
     * Get the response Entity which holds the response content from
    HttpResponse.
     */
    HttpEntity resEntity = response.getEntity();

    //Reads the response
    responseString = getResponseString(resEntity);
    if (resEntity != null) {
        /**
         * The Method consumeContent() is called to indicate that the
content of this entity
         * is no longer required. All entity implementations are expected
to
         * release all allocated resources as a result of this method
         * invocation.
         */
        resEntity.consumeContent();
    }

    return responseString;
}

/**
 * Processes the request using HTTPConnection API
 */
private String getQuoteUsingHttpConnection(HttpClientConnectionFactory
connectionFactory,
    String symbol) throws HttpException, IOException {

    String responseString = ERROR_MSG;
    String message = getContent(symbol);
    byte[] bytes = message.getBytes();

    HttpClientConnection httpClientConnection =
connectionFactory.getHttpClientConnection();
    Host configuration = connectionFactory.getHostConfiguration();

    /**
     * Construct the request URL

```

```

        */
        String url = configuration.getSchemeName() + "://" +
configuration.getHostName() + ":" +
        configuration.getPort() + "/stockquote.asmx";

        try {

            /**
             * Prepare request object and its header for HTTP Post request
             */
            HttpPost httpPost = new HttpPost(url);
            httpPost.setHeader("Content-Type", "text/xml; charset=utf-8");
            httpPost.setHeader("SOAPAction", "http://www.webserviceX.NET/
GetQuote");

            /**
             * Sets the Entity to handle content management.
             */
            ByteArrayInputStream instream = new ByteArrayInputStream(bytes);
            InputStreamEntity e2 = new InputStreamEntity(instream, -1);
            httpPost.setEntity(e2);

            /**
             * Set HTTP params on Post request object
             */
            httpPost.setParams(connectionFactory.getHttpParams());

            /** HttpContext represents execution state of an HTTP process.
             * It is a structure that can be used to map an attribute name
             * to an attribute value. Internally HTTP context
implementations
             * are usually backed by a HashMap.
             */
            BasicHttpContext basicHttpContext = new BasicHttpContext(null);
            // Populate the execution context

            basicHttpContext.setAttribute(ExecutionContext.HTTP_CONNECTION,httpClientConne
ction);

            basicHttpContext.setAttribute(ExecutionContext.HTTP_TARGET_HOST,connectionFact
ory.
                getHostConfiguration());

            basicHttpContext.setAttribute(ExecutionContext.HTTP_REQUEST,httpPost);

            /** HTTP protocol processor is a collection of protocol
interceptors that
             * implements the Chain of Responsibility pattern, where each
individual
             * protocol interceptor is expected to work on a particular
aspect of the HTTP
             * protocol for which the interceptor is responsible.
             */
            BasicHttpProcessor httpProcessor = new BasicHttpProcessor();
            // Required request interceptors
            httpProcessor.addInterceptor(new RequestContent());
            httpProcessor.addInterceptor(new RequestTargetHost());
            // Recommended request interceptors
            httpProcessor.addInterceptor(new RequestConnControl());
            httpProcessor.addInterceptor(new RequestUserAgent());
            httpProcessor.addInterceptor(new RequestExpectContinue());

            /** HttpRequestExecutor is a client side HTTP protocol handler
based on the
             * blocking I/O model that implements the essential requirements
of the HTTP
             * protocol for the client side message processing
             */
            HttpRequestExecutor httpexecutor = new HttpRequestExecutor();

            // Prepare request

```

```

        httpexecutor.preProcess(httpPost, httpProcessor,
basicHttpContext);
        // Execute request and get a response
        HttpResponse response =
httpexecutor.execute(httpPost,httpClientConnection,
        basicHttpContext);
        // Finalize response
        httpexecutor.postProcess(response, httpProcessor,
basicHttpContext);
        HttpEntity resEntity = response.getEntity();

        //Reads the response
        responseString = getResponseString(resEntity);
        if (resEntity != null) {
/**
 * The Method consumeContent() is called to indicate that the
content of this entity
to
 * is no longer required. All entity implementations are expected
to
 * release all allocated resources as a result of this method
 * invocation.
 */
        resEntity.consumeContent();
        }
    } finally {
        httpClientConnection.close();
    }
    return responseString;
}

/**
 * Reads and returns the string content from response Entity
 */
private String getResponseString(HttpEntity resEntity)
    throws IOException {
    if (resEntity != null) {
        InputStream content = resEntity.getContent();
        byte[] cbytes = new byte[new Long(1000).intValue()];
        int x = -1;
        StringBuilder sb = new StringBuilder();
        while ((x = content.read(cbytes)) != -1) {
            String reponseContent = new String(cbytes);
            sb.append(reponseContent);
        }
        return getValue(sb.toString().trim());
    }
    return ERROR_MSG;
}

/**
 * Returns the request content.
 * @param symbol
 * @return
 */
private String getContent(String symbol) {
    return "<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/
soap/envelope/\"
        + \"xmlns:web=\"http://www.webserviceX.NET/\">\"
        + \"<soapenv:Header/>\"
        + \"<soapenv:Body>\"
        + \"<web:GetQuote>\"
        + \"<web:symbol>\"+symbol+\"</web:symbol>\"
        + \"</web:GetQuote>\"
        + \"</soapenv:Body>\"
        + \"</soapenv:Envelope>\";
}

```

Invoking a Reference Operation

When you add a reference to a Spring component, TIBCO Business Studio adds following a field and accessor methods to the abstract component implementation. TIBCO ActiveMatrix injects the referenced object into the component implementation.

TIBCO Business Studio adds the following elements to the abstract component implementation:

- SCA reference annotation import
- A field that declares the referenced object
- Accessor methods

The TIBCO ActiveMatrix platform injects the referenced object into the component implementation. For example, if you add a reference to port type `DateManagerPT`, the following code is added:

```
import org.osoa.sca.annotations.Reference;
@Reference(name = "DateManagerPT")
public void setDateManagerPT(DateManagerPT DateManagerPT) {
    this.DateManagerPT = DateManagerPT;
}

public DateManagerPT getDateManagerPT() {
    return this.DateManagerPT;
}
```

When you pass an XMLBeans document object to a reference invocation, the object is passed by reference. Since the state of an object is not guaranteed across reference invocations, you cannot access the object after the reference invocation. If you need to access the object after the invocation, make a deep copy of the object using the `copy` method before you invoke the reference. For example, if you needed to access the `req` object after the call to `getCurrentTime`, make a deep copy of `req` as follows:



```
TimeRequestDocument req = TimeRequestDocument.Factory.newInstance();
req.setTimeRequest("America/Los_Angeles");
TimeRequestDocument reqcopy = (TimeRequestDocument)req.copy();
TimeResponseDocument time = getDateManagerPT().getCurrentTime(req);
System.out.println("The time in " + reqcopy.getTimeRequest() + " is " +
    time.getTimeResponse());
```

Procedure

- Add the statement `getportType().operation`. If the reference is configured for dynamic wiring, you must define a method to create an endpoint reference (see [Creating an Endpoint Reference](#)) and call the method before invoking the reference object. For information on wiring, see *Static and Dynamic Wiring in Composite Development*.

The following code snippet demonstrates how to invoke the `getCurrentTime` operation on the reference configured for dynamic wiring with port type `DateManagerPT`:

```
setEPR(targetURI);
String time = currentTime.getTimeResponse();

resp.setHelloResponse(getJavaGreeting() + " " + name + "! "
    + "The current time is " + time + ".");
return resp;
```

Error Handling

In service-oriented applications, SOAP clients expect a fault message to be returned when an error occurs during processing. A *fault message* is a SOAP message

A fault message has the following subelements:

Subelement	Description
<code>faultcode</code>	A code that identifies the fault.
<code>faultstring</code>	An explanation of the fault.
<code>faultactor</code>	Information about what caused the fault to occur.
<code>detail</code>	Application-specific information about the fault.

Fault messages defined in the WSDL file are called *declared faults*. Fault messages that are not defined in the WSDL file are called *undeclared faults*. The process for generating a fault message is implementation dependent and typically depends on whether the fault is declared or not.

Declared Faults

When you add a service to a Spring component TIBCO Business Studio generates a fault exception class for each fault declared in the WSDL file that defines the service's port type.

Example WSDL File

The following WSDL fragment shows the `getWeather` operation with two declared faults: `orderFault` and `orderFault2`. The detail element for the `orderFault` message contains a `ZipCodeFault` element. The detail element for the `orderFault2` message contains a `CityFault` element.

```
<wsdl:types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.example.org/weatherschema"
    targetNamespace="http://www.example.org/weatherschema"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
    <complexType name="WeatherRequestType">
      <sequence>
        <element name="city" type="string"/>
        <element name="state" type="string"/>
        <element name="zip" type="string"/>
      </sequence>
    </complexType>
    <complexType name="WeatherResponseType">
      <sequence>
        <element name="high" type="float"/>
        <element name="low" type="float"/>
        <element name="forecast" type="string"/>
      </sequence>
    </complexType>

    <element name="WeatherRequest" type="tns:WeatherRequestType"/>
    <element name="WeatherResponse" type="tns:WeatherResponseType"/>
    <element name="ZipCodeFault" type="string"/>
    <element name="CityFault" type="string" />
  </schema>
</wsdl:types>
<wsdl:message name="invalidZipCodeFault">
  <wsdl:part name="error" element="ns0:ZipCodeFault"/>
</wsdl:message>
<wsdl:message name="invalidCityFault">
  <wsdl:part name="error" element="ns0:CityFault" />
</wsdl:message>
<wsdl:portType name="WeatherReportPT">
  <wsdl:operation name="GetWeather">
    <wsdl:input message="tns:GetWeatherRequest"/>
    <wsdl:output message="tns:GetWeatherResponse"/>
    <wsdl:fault name="orderFault" message="tns:invalidZipCodeFault"/>
    <wsdl:fault name="orderFault2" message="tns:invalidCityFault" />
  </wsdl:operation>
</wsdl:portType>
```

Code Generation

When TIBCO Business Studio generates the Spring component implementation `invalidCityFault` and `invalidZipCodeFault` are mapped to the exceptions:

```
public class InvalidCityFaultException extends java.lang.Exception
public class InvalidZipCodeFaultException extends java.lang.Exception
```

and a `throws` clause containing the generated exceptions is added to the `getWeather` method.

To generate the `invalidCityFault` fault message while processing the `getWeather` method, throw `InvalidCityFaultException`. The `faultcode` and `faultactor` subelements of the SOAP fault element are predefined as:

Subelement	Content
<code>faultcode</code>	SOAP-ENV:Server

Subelement	Content
faultactor	DefaultRole

Setting Fault Message Subelements

To customize the values of the `faultstring` and `detail` subelements:

1. Create a string object and set to an informative message. The message is mapped to the SOAP message's `faultstring` element.
2. Create a fault document and set the appropriate fault property of the document to the reason for the error. The reason is mapped to the SOAP message `detail` element.
3. Create a fault message exception that contains the message and fault document.

For example, if the `city` element of the request is not set correctly, configure and throw the fault message exception as follows:

```
public WeatherResponseDocument getWeather(WeatherRequestDocument getWeatherRequest)
    throws org.example.www.WeatherService.InvalidCityFaultException,
           org.example.www.WeatherService.InvalidCityFaultException {
    WeatherRequestType weatherRequest = getWeatherRequest.getWeather();
    if (weatherRequest.getCity()==null||weatherRequest.getCity().equals("")) {
        CityFaultDocument cityFaultDocument =
            CityFaultDocument.Factory.newInstance();

        XmlString msg = XmlString.Factory.newInstance();
        msg.setStringValue("Error processing getWeather.");
        // set detail
        cityFaultDocument.setCityFault("Invalid city for zipcode " +
            weatherRequest.getZip());
        // set faultstring
        InvalidCityFaultException invalidCityFaultException =
            new InvalidCityFaultException(msg.getStringValue(), cityFaultDocument);
        throw invalidCityFaultException;
    }
    ...
}
```

which would generate the following SOAP fault message if `getCity` does not return a valid value:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Error processing getWeather.</faultstring>
<faultactor>DefaultRole</faultactor>
<detail>
<CityFault xmlns="http://www.example.org/weatherschema">
  Invalid city for zipcode 95070.</CityFault>
</detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A consumer invoking `getWeather` should handle the exception as follows:

```
public WeatherResponseDocument getWeather(WeatherRequestDocument getWeatherRequest)
{
    try {
        return getWeatherReportPT().getWeather(getWeatherRequest);
    }
    catch(Exception e) {
        if (e instanceof InvalidCityFaultException)
            throw (InvalidCityFaultException)e;
        else{
            System.out.println("Error occurred.");
            throw new RuntimeException(e.getMessage(),e);
        }
    }
}
```

```
...
}
```

Undeclared Faults

When an undeclared fault occurs, the TIBCO ActiveMatrix runtime returns a SOAP fault with the following subelements:

Subelement	Content
faultcode	SOAP-ENV:Server
faultstring	java.lang.RuntimeException
faultactor	DefaultRole
detail	A stack trace indicating where the exception occurred.

If a runtime exception occurs while processing `getWeather`, the following SOAP fault would be generated:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
  <SOAP-ENV:Fault>
    <faultcode>SOAP-ENV:Server</faultcode>
    <faultstring>java.lang.RuntimeException: Undeclared fault...</faultstring>
    <faultactor>DefaultRole</faultactor>
    <detail>
      <tibco:myFaultDetail xmlns:tibco="http://tibcouri/">
        org.osoa.sca.ServiceRuntimeException: java.lang.RuntimeException: Undeclared
fault...
        at com.tibco.amf.platform.runtime.componentframework.internal.proxies.
ProxyInvocationHandler.invoke(ProxyInvocationHandler.java:473)
        at $Proxy21.invoke(Unknown Source)
        at com.tibco.amf.binding.soap.runtime.transport.http.SoapHttpInboundEndpoint.
processHttpPost(SoapHttpInboundEndpoint.java:250)
        at com.tibco.amf.binding.soap.runtime.transport.http.SoapHttpServer.doPost(
SoapHttpServer.java:103)
        ...
        Caused by: java.lang.RuntimeException: Undeclared fault...
        at com.sample.faultservice.Component1.getWeather(Component1.java:50)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:
39)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
25)
        at java.lang.reflect.Method.invoke(Method.java:585)
        at com.tibco.amf.platform.runtime.componentframework.internal.proxies.
ProxyInvocationHandler.invoke(ProxyInvocationHandler.java:426)
        ... 20 more
      </tibco:myFaultDetail>
    </detail>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

To specify SOAP fault subelements for undeclared faults, convert the runtime exception into a `com.tibco.amf.platform.runtime.extension.SOAPException` class. The following code fragment illustrates how to modify the subelements of the SOAP fault:

```
import com.tibco.amf.platform.runtime.extension.exception.SOAPException;
import com.tibco.amf.platform.runtime.extension.exception.SOAPDetail;
import com.tibco.amf.platform.runtime.extension.exception.SOAPCode;
import java.net.URI;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
...
```

```

URI role = null;
try {
    role = new URI("http://soapexception.role");
} catch (URISyntaxException e) {
    e.printStackTrace();
}
WeatherRequestType weatherRequest = getWeatherRequest.getWeatherRequest();
Node domNode = weatherRequest.getDomNode();
//Set the original request as the fault detail
SOAPDetail<Element> soapDetail = new SOAPDetail<Element>(Element.class,
(Element)domNode);
SOAPCode soapCode = new SOAPCode(new QName("fault code"));

SOAPException soapException = new SOAPException(soapCode, "reason", role,
soapDetail);
throw soapException;
...

```

The following example illustrates how to catch the SOAPException exception returned by an invocation of a referenced service:

```

public WeatherResponseDocument getWeather(WeatherRequestDocument getWeatherRequest)
throws org.example.www.WeatherService.InvalidZipCodeFaultException {
    try {
        return getWeatherReportPT().getWeather(getWeatherRequest);
    } catch (InvalidZipCodeFaultException e) {
        System.out.println("====InvalidZipcodeFaultException====");
        throw e;
    } catch(Exception e) {
        System.out.println("====Runtime Exception====");
        if(e instanceof ServiceRuntimeException) {
            ServiceRuntimeException sre = (ServiceRuntimeException) e;
            Throwable cause = sre.getCause();
            if(cause instanceof SOAPException) {
                SOAPException soapex = (SOAPException) cause;
                if (soapex.getCode()!=null) {
                    System.out("Fault code: " + soapex.getCode().toString());
                }
                if (soapex.getRole()!=null) {
                    System.out("Fault role: " + soapex.getRole().toString());
                }
                throw soapex;
            }
        }
        throw new RuntimeException(e);
    }
    ...
    return null;
}

```

SOAPException Reference

SOAPException has SOAPCode and SOAPDetail in its parameter list. All three are discussed below.

public SOAPException(final SOAPCode code, final String[] reason, final URI node, final URI role, final SOAPDetail<T> detail)

The parameters of type SOAPXXX map to subelements of SOAP faults. Some of the parameters are for future use.

Parameter	Description
code	Intended for use by software to provide an algorithmic mechanism for identifying the fault.
reason	Provides a human readable explanation of the fault.
node	A URI that identifies the SOAP node that generated the fault. Its absence implies that the first recipient of the message generated the fault.

Parameter	Description
role	A URI identifying the source of the fault within the message path.
detail	Carries application-specific error information related to the Body element. It must not be used to carry information about error information belonging to header entries. Detailed error information belonging to header entries must be carried within header entries.

public SOAPCode(final QName codeValue, final SOAPCode subcode)

The type of the **code** parameter of SOAPException. The structure of the **code** parameter is hierarchical. The top-level code is a base fault and must be understood by all SOAP nodes. Nested codeValues are application-specific, and represent further elaboration or refinement of the base fault.

Parameter	Description
codeValue	The QName that identifies the code for the fault.
subcode	An optional subcode. Each child subcode element has a mandatory codeValue and an optional subcode subelement to support application-specific codes.

public SOAPDetail(final Class<T> detailType, final T detail)

The type of the **detail** parameter of SOAPException.

Parameter	Description
detailType	Type of the detail data provided to the exception.
detail	Detail data provided to the exception. For example: <code>SOAPDetail<Element> soapDetail = new SOAPDetail<Element>(Element.class, (Element)domNode);</code>

Context Parameters

A *context parameter* is a key-value pair passed to a service operation invocation. The values are populated by bindings, which map transport and binding headers to context parameters. Context parameters allow component developers to access transport and binding metadata that could affect the execution of an operation but which is not passed in the input, output, and fault messages defined by an abstract WSDL file.

A service may be supported on multiple types of transport bindings; each binding protocol specifies its own headers. For example, HTTP supports a body and headers that specify metadata that can be mapped to context parameters. The SOAP similarly defines a body and headers that are different than HTTP headers. The JMS protocol defines headers and allows developers to define application-specific properties. Typically, a client invoking a service will set some headers. For example, browsers usually set the HTTP Locale and Referrer headers.

Component implementations can read and set the values of context parameters and the values can be used to control service operation logic. The operation behavior thus changes according to the metadata. For example, consider a service that pays the agent that referred a customer to a website. To track the referrer on a SOAP/HTTP binding, you would specify a mapping from the HTTP Referrer header to a context parameter. If the service has a JMS binding, you would specify a mapping from a JMS message property named `referencedBy` to the same context parameter. When the incoming message is SOAP/

HTTP, the HTTP Referrer header is copied into the context parameter and when a JMS message arrives, the `referencedBy` property is copied into the context parameter. The following table lists the header sources for each binding type.

Header Source

Binding Type	Header Source
SOAP/HTTP	HTTP Transport Header, HTTP Context, TCP Context, SOAP Header, SOAP Fault
SOAP/JMS	JMS Header, JMS Application Properties, SOAP Fault, SOAP Header
JMS	JMS Header, JMS Application Properties



In the TIBCO ActiveMatrix platform, the context parameter key `com.tibco.security.userinformation` is used to communicate security context. It can be retrieved by a component from `requestContext`. However, when invoking a reference this context parameter may be overwritten by a policy agent before the SOAP binding maps it to a HTTP Transport Header or JMS application property. Therefore, you cannot set this context parameter in a component before invoking a reference.

The following sections list the headers available in each header source. The tables in each section list which headers are available in service or reference bindings.

- From: XXX Binding To: Context applies to inbound messages received on either a service ("in" part of "in-out" MEP) or a reference ("out|fault" part of "in-out" MEP)
- From: Context To: XXX Binding applies to outbound messages sent from either a service ("out|fault" part of "in-out" MEP) or a reference ("in" part of "in-out" MEP)

HTTP Context

From: SOAP/HTTP Binding (WS-A = OFF)	To: Context
Service	HTTP-Method, HTTP-FileSpec, HTTP-Version
Reference	HTTP-Status-Code, HTTP-Status-Message

From: SOAP/HTTP Binding (WS-A = ON)	To: Context
Service	HTTP-Method, HTTP-FileSpec, HTTP-Version
Reference	None

TCP Context

From: SOAP/HTTP Binding	To: Context
Service	Local-TCP-Host, Local-TCP-Port, Remote-TCP-Host, Remote-TCP-Port
Reference	None

SOAP Fault

From: SOAP Binding (For Declared Faults)	To: Context
Service	None
Reference	Role, Code

From: Context	To: SOAP Binding (For Declared Faults)
Service	Role, Code
Reference	None

JMS Header

From: SOAP/JMS Binding	To: Context
Service	JMSCorrelationID, JMSDeliveryMode, JMSMessageID, JMSType
Reference	None

From: Context	To: SOAP/JMS Binding
Service	None
Reference	JMSCorrelationID, JMSDeliveryMode, JMSType

For information on how to create context parameters, see *Composite Development*.

The first time you add a context parameter to a service or reference wired to a Spring component, an error badge will be added to the Spring component and the error will be reported in the Problems view. When you resolve the error by updating the implementation, the following is added to the abstract component implementation:

```
import org.osoa.sca.annotations.Context;
import com.tibco.amf.platform.runtime.extension.context.ComponentContext;
/**
 * Use this property to access the context parameters.
 * Context parameters for this component are:
 * parameterName : DIRECTION
 */
@Context
public ComponentContext componentContext;
```

For each successive parameter, no error badge is added to the component. To update the list of context parameters in the comment, update the component implementation.

Methods defined on `ComponentContext` allow you to retrieve and set a `RequestContext`, which in turn has methods for retrieving and setting the context parameters.

Working with Context Parameters

You can retrieve a context parameter from a request or from a response, and set a context parameter in a request or a response.

Retrieving a Context Parameter from a Request

Procedure

1. Retrieve the request context:

```
import com.tibco.amf.platform.runtime.extension.context.RequestContext;
RequestContext requestContext =
    (RequestContext)componentContext.getRequestContext();
```

2. Retrieve the parameter from the request context:

```
requestContext.getParameter(parameterName, Type.class);
```

where *Type* can take the values String, Integer, Long, Boolean, Map, QName, and URI.

Setting a Context Parameter in a Request

Procedure

1. Create a mutable request context:

```
import com.tibco.amf.platform.runtime.extension.context.MutableRequestContext;
MutableRequestContext mutableRequestContext = componentContext.
    createMutableRequestContext();
```

2. Set a parameter on the mutable request context:

```
mutableRequestContext.setParameter(parameterName, Type.class,
    parameterValue);
```

3. Set the request context on the component's context to the mutable request context:

```
componentContext.setRequestContext(mutableRequestContext);
```

4. Invoke a reference.



The `componentContext.getRequestContext()` function returns the request context that corresponds to the last remotable service invocation whereas `componentContext.setContext()` assigns the context that gets used for the next downstream service invocation. For example, in the following case, `curCtx` does not equal to `newCtx` but equals to `oldCtx`.

```
RequestContext oldCtx =
    (RequestContext)componentContext.getRequestContext();
MutableRequestContext newCtx =
    componentContext.createMutableRequestContext();
componentContext.setRequestContext(newCtx);
RequestContext curCtx =
    (RequestContext)componentContext.getRequestContext();
```

Retrieving a Context Parameter from a Response

Procedure

1. Retrieve a callback context from the mutable request context:

```
import com.tibco.amf.platform.runtime.extension.context.CallbackContext;
CallbackContext callbackContext = mutableRequestContext.getCallbackContext();
```

2. Retrieve a parameter from the callback context:

```
callbackContext.getParameter(parameterName, Type.class);
```

Setting a Context Parameter in a Response

Procedure

1. Retrieve a mutable callback context from the original request context:

```
import com.tibco.amf.platform.runtime.extension.context.MutableCallbackContext;
MutableCallbackContext mutableCallbackContext =
(MutableCallbackContext)requestContext.
getCallbackContext();
```

2. Set a parameter on the mutable callback context:

```
mutableCallbackContext.setParameter(parameterName,
Type.class, parameterValue);
```

Distributed File System Example

A distributed file system component manages files based on the host address of the machine on which the file was created. The address is tracked in a context parameter named `httpHost`:

Name	Operations	Direction	Type	Definition
httpHost	prepareFragmentedWrite,write	Input	Basic	string

The file system component is invoked by SOAP clients through a service with a SOAP binding and by a web application component.

- If a request comes through the SOAP binding, the context parameter is mapped to the TCP remote host header by the SOAP binding:

SOAP Binding Context Parameter Mapping

Context Parameter	Direction	Header Source	Header Name
httpHost	INPUT	TCP	remoteHost

- If the request originates from the web application, the parameter value is retrieved from the HTTP request and manually set by the servlet implementing the web application component:

```
String host = req.getRemoteHost();
MutableRequestContext mutableRequestContext =
componentContext.createMutableRequestContext();
mutableRequestContext.setParameter("httpHost", String.class, host);
componentContext.setRequestContext(mutableRequestContext);
```

The file system component retrieves the value of the context parameter as follows:

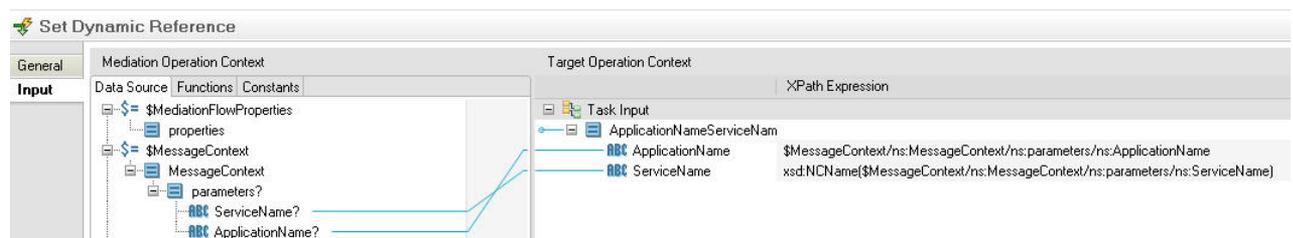
```
RequestContext requestContext = componentContext.getRequestContext();
String host requestContext.getParameter("httpHost", String.class);
```

Dynamic Binding Example

Application logic can depend on the value of the application and service name. In particular, the application logic may be used to dynamically determine the target of a reference invocation (also referred to as wire by implementation) in a mediation flow. The following example illustrates how to retrieve the application and service name in a Spring component that invokes a mediation component service, and set context parameters with that data:

```
String appName = componentContext.getApplicationName();
String svcname = componentContext.getRequestContext().getServiceName();
MutableRequestContext mutableRequestContext =
    componentContext.createMutableRequestContext();
mutableRequestContext.setParameter("ServiceName", java.lang.String.class,
    svcname);
mutableRequestContext.setParameter("ApplicationName",
    java.lang.String.class, appName);
componentContext.setRequestContext(mutableRequestContext);
```

The context parameters are then mapped in the mediation flow's Set Dynamic Reference task property sheet as follows:



Endpoint References

If WS-Addressing is enabled on a SOAP binding, endpoint references are accessible from within Spring component implementations. For information on enabling WS-Addressing and endpoint references, see WS-Addressing in *Composite Development*.

Retrieving an Endpoint Reference

Prerequisites

Enable the SOAP binding that delivers the message to the Spring component for WS-Addressing to use reference parameters.

Procedure

1. Import context, endpoint reference, request context, wildcard extension, and URI class definitions:

```
import com.tibco.amf.platform.runtime.extension.context.ComponentContext;
import com.tibco.amf.platform.runtime.extension.context.EndpointReference;
import com.tibco.amf.platform.runtime.extension.context.RequestContext;
import com.tibco.amf.platform.runtime.extension.context.WildCardExtension;
import java.net.URI;
```

2. Declare the component context:

```
@Context
public ComponentContext componentContext;
```

3. Retrieve the request context:

```
RequestContext requestContext =
    (RequestContext)componentContext.getRequestContext();
```

4. Retrieve the endpoint reference:

```
EndpointReference<Element> endpointReference =
    requestContext.getEndpointReference(Element.class);
```

5. Optionally retrieve the URI and reference parameters from the endpoint reference:

```
URI uri = endpointReference.getAddress().getURI();
WildcardExtension<Element> refElements =
    endpointReference.getReferenceParameters();
```

Creating an Endpoint Reference

Prerequisites

Enable the SOAP binding that delivers the message to the Spring component for WS-Addressing to use reference parameters.

You must set an endpoint reference object before invoking a reference that is dynamically wired to a service.

Procedure

1. Import context, endpoint reference, and URI class definitions:

```
import org.osoa.sca.annotations.Context;
import com.tibco.amf.platform.runtime.extension.context.ComponentContext;
import com.tibco.amf.platform.runtime.extension.context.EndpointReference;
import com.tibco.amf.platform.runtime.extension.context.MutableRequestContext;
import com.tibco.amf.platform.runtime.extension.support.ElementEndpointReference;
import com.tibco.amf.platform.runtime.extension.support.ElementWildcardExtension;
```

2. Declare the component context:

```
@Context
public ComponentContext context;
```

3. Create an endpoint reference:

```
EndpointReference<Element> epr = new ElementEndpointReference(targetURI);
```

4. Optionally create and set a list of parameters. For example:

```
String property1= "<property1 " + "xmlns=\"" + WSQName + "\">value1</property1>";
String property2= "<property2 " + "xmlns=\"" + WSQName + "\">value2</property2>";
List<Element> elements = Arrays.asList(
    DOMUtils.getDOMNode(property1).getDocumentElement(),
    DOMUtils.getDOMNode(property2).getDocumentElement());
ElementWildcardExtension refParams = new ElementWildcardExtension(null,
elements);
epr.setReferenceParameters(refParams);
```

5. Create a mutable request context object:

```
MutableRequestContext mctxt =
    (MutableRequestContext)context.createMutableRequestContext();
```

6. Set the endpoint reference of the mutable context object:

```
mctxt.setEndpointReference(epr);
```

7. Set the request context of the component context to the mutable request context:

```

context.setRequestContext(mctx);

import org.osoa.sca.annotations.Context;
import com.tibco.amf.platform.runtime.extension.context.ComponentContext;
import com.tibco.amf.platform.runtime.extension.context.EndpointReference;
import com.tibco.amf.platform.runtime.extension.context.MutableRequestContext;
import
com.tibco.amf.platform.runtime.extension.support.ElementEndpointReference;
import
com.tibco.amf.platform.runtime.extension.support.ElementWildcardExtension;
import java.net.URI;
import java.util.List;
import java.util.Arrays;
@Context
public ComponentContext context;
public static final String WSQName = "com.ws.base";
private void setEPR(URI targetURI) {
    EndpointReference<Element> epr = new
ElementEndpointReference(targetURI);

    String property1= "<property1 " + "xmlns=\"" + WSQName + "\">value1</
property1>";
    String property2= "<property2 " + "xmlns=\"" + WSQName + "\">value2</
property2>";
    List<Element> elements = Arrays.asList(
        DOMUtils.getDOMNode(property1).getDocumentElement(),
        DOMUtils.getDOMNode(property2).getDocumentElement());
    ElementWildcardExtension refParams = new
ElementWildcardExtension(null, elements);
    epr.setReferenceParameters(refParams);

    MutableRequestContext mctx = (MutableRequestContext)
context.createMutableRequestContext();
    mctx.setEndpointReference(epr);
    context.setRequestContext(mctx);
}

```

Custom Features

A *feature* is a software package that contains plug-ins, which in turn contain component implementations and libraries. A feature is identified by an ID, a multi-part version, and its dependencies on other features. There are two types of features: system and shared library.

System features are part of a TIBCO ActiveMatrix product or contain the drivers that are installed using TIBCO Configuration Tool. Shared library features contain component implementations and libraries. When you create a distributed application archive containing a composite, you can package the composite's required features in the application archive or you can package the features as a standalone distributed application archive.

When you upload a distributed application archive containing a composite in Administrator you can optionally import the features contained in the archive into the Administrator software repository. When you deploy an application, Administrator automatically distributes the features (and any features that it depends on) to the host that manages the nodes on which the application is distributed and installs the features on those nodes. You can also manually install features on the other nodes managed by that host.

Version Numbers

A version number is a multicomponent number of the form *major.minor.service.qualifier*. Changes in the value of each component reflect different types of changes in the versioned object:

- *major* - Reflects breaking changes to the interface.
- *minor* - Reflects non-breaking changes in an externally visible way. Examples of externally visible changes include binary compatible changes, significant performance changes, major code rework, and so on.
- *service* - Reflects changes that are not visible in the interface. For example, a bug has been fixed in the code, documentation has changed, compiler settings have changed, and so on.
- *qualifier* - Identifies when and where the object was built or packaged.

When you create an object in TIBCO Business Studio, the version is set to "1.0.0.qualifier". If the *qualifier* component of a version is set to "qualifier" when you create a DAA, TIBCO Business Studio replaces "qualifier" with a generated qualifier that defaults to a timestamp. You can customize the format of the generated qualifier by specifying a qualifier replacement.

Version Ranges

Some fields require you to specify a version range. For example, a feature may have a dependency on a range of versions of another feature. A *version range* is an interval specified as: *bracket lower limit, upper limit bracket*, where *bracket* can be "[", "]", "(", or ")", which denotes an inclusive end of the range or "(", ")", which denotes an exclusive end of the range. If one end of the range is to be included and the other excluded, the range can contain a square bracket with a round bracket.

There are three common use cases:

- A strict version range, such as [1.0.0,1.0.0], denotes version 1.0.0 and only that version.
- A half-open range, such as [1.0.0,2.0.0), which has an inclusive lower limit and an exclusive upper limit, denotes version 1.0.0 and any later version, up to, but not including, version 2.0.0.
- An unbounded open range expressed as a single number such as 2.0.0, which is equivalent to the range [2.0.0, *infinity*), denotes version 2.0.0 and any later version.

A custom feature named *compositeName.customfeature.id* containing the component implementation plug-in is created automatically when you generate a component implementation. The custom feature file is stored in the `Deployment Artifacts` folder of the SOA project.

Bundles and Plug-in Projects

A *bundle* is an OSGi mechanism for grouping Java classes into a modular, sharable unit. In TIBCO Business Studio, a plug-in project implements a bundle.

Plug-in properties, including the packages it exports and the objects on which it depends, are described in the plug-in manifest. The manifest file is located in *plug-inFolderName*META-INF/MANIFEST.MF. The default editor for the file is a manifest editor which displays OSGi headers in property sheets and in the MANIFEST.MF source view. [Plug-in Project Best Practices](#) summarizes the best practices you should follow when configuring plug-ins.

Plug-in Project Best Practices

Property	Manifest Editor UI	OSGi Header in Source View	Best Practice
Unique Identifier	Overview > ID	Bundle-SymbolicName	Give the plug-in a symbolic name that follows Java package name conventions. That is, <i>com.companyName.plug-inName</i> .
Version	Overview > Version	Bundle-Version	Follow the recommendations in Versions .
Display Name	Overview > Name	Bundle-Name	Give the plug-in an appropriate, descriptive display name.
Dependencies	Dependencies > Imported Packages Required Plug-ins	Import-Package Require-Bundle	<ul style="list-style-type: none"> Express dependencies based on the contents of the plug-in: <ul style="list-style-type: none"> For plug-ins that you create or if you want tight control of the dependency, specify the dependency as a required bundle. You can (but are not required to) indicate a perfect match to a specific plug-in and build. For example, when TIBCO Business Studio generates a component implementation, the component's dependency on the plug-in containing the component implementation is expressed as <code>[1.0.0.qualifier,1.0.0.qualifier]</code>. For third-party plug-ins, specify the dependency as an imported package. To allow packages to be upgraded without requiring plug-ins dependent on those packages to be upgraded, specify package dependency ranges of the form <code>[x.y.z,x+1.0.0)</code>. That is, up to, but not including the next major version. For example, <code>[2.3.0, 3.0.0)</code>. Minimize or eliminate optional imports.

Property	Manifest Editor UI	OSGi Header in Source View	Best Practice
Exported Packages	Runtime > Exported Packages	Export-Package	<ul style="list-style-type: none"> • Export only those packages imported by other plug-ins. • Put classes that are not exported in packages that are not exported. • Specify versions of all exported packages. • Import all exported packages, with a range floor of the exported package version, and a range ceiling of the next major version exclusive, per the range definition above. • If the classes in the exported packages use other classes (for example, they extend classes from another package or the classes appear in method signatures) add the <code>uses</code> directive to the export package definition.

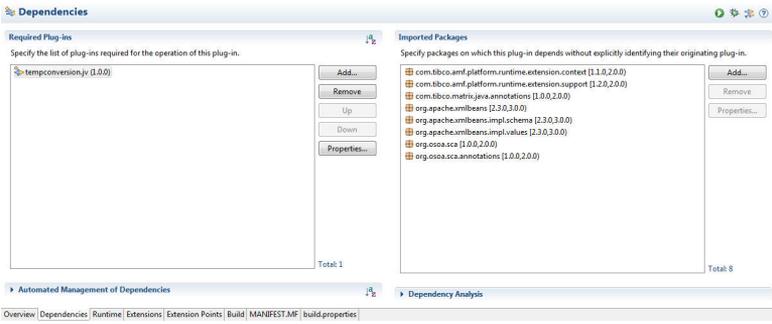
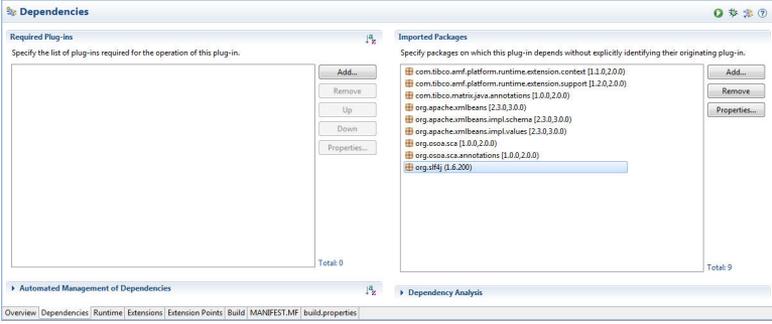
Configuring Dependencies on External Java Classes

Java and Spring component implementations can use Java classes contained in a library plug-in project in the same workspace as the component implementation. You must configure the dependency in both the component implementation and component.

Procedure

1. Expand the `META-INF` directory of plug-in project containing the component implementation.
2. Double-click `MANIFEST.MF`.
The manifest opens in the manifest editor.
3. Click the **Dependencies** tab.
4. Follow the appropriate procedure based on the [dependency](#) type.

Dependency Type	Procedure
Plug-in	<ol style="list-style-type: none"> 1. Click Add... to the right of the Required Plug-ins table. 2. Select the plug-in containing the referenced class. 3. Click OK. <p>For example, if you reference a temperature conversion plug-in in a Spring component implementation, add <code>tempconversion.jv</code> as a required plug-in:</p>

Dependency Type	Procedure
	
Package	<ol style="list-style-type: none"> 1. Click Add... to the right of the Imported Packages table. 2. Click the referenced package. 3. Click OK. <p>For example, if you reference a logging class in a Spring component implementation, add <code>org.slf4j</code> as an imported package:</p> 

5. If the library plug-in will be packaged and deployed separately from the component implementation:
 - a) Open the composite.
 - b) Click the Spring component.
 - c) In the Properties view, click the **Implementation** tab.
 - d) Uncheck the **Package Implementation Bundle with Application** and **Compute Feature Dependencies** checkboxes.
The Features Dependencies table displays.
 - e) In the Feature Dependencies table, click the feature that contains the library plug-in.
 - f) Relax value of the version as described in [External Library Dependencies](#).

Versions

A *version* is a property that controls how an object is treated at installation or deployment. Versions are specified in TIBCO Business Studio and cannot be modified in Administrator.

The following objects have versions:

- Composites and application templates.
- Components - During application upgrade, Administrator compares component versions to determine whether the component needs to be upgraded.

- Features
- Plug-ins
- Packages

Version Numbers

A version number is a multicomponent number of the form *major.minor.service.qualifier*. Changes in the value of each component reflect different types of changes in the versioned object:

- *major* - Reflects breaking changes to the interface.
- *minor* - Reflects non-breaking changes in an externally visible way. Examples of externally visible changes include binary compatible changes, significant performance changes, major code rework, and so on.
- *service* - Reflects changes that are not visible in the interface. For example, a bug has been fixed in the code, documentation has changed, compiler settings have changed, and so on.
- *qualifier* - Identifies when and where the object was built or packaged.

When you create an object in TIBCO Business Studio, the version is set to "1.0.0.qualifier". If the *qualifier* component of a version is set to "qualifier" when you create a DAA, TIBCO Business Studio replaces "qualifier" with a generated qualifier that defaults to a timestamp. You can customize the format of the generated qualifier by specifying a qualifier replacement.

Version Ranges

Some fields require you to specify a version range. For example, a feature may have a dependency on a range of versions of another feature. A *version range* is an interval specified as: *bracket lower limit, upper limit bracket*, where *bracket* can be "[" or "]", which denotes an inclusive end of the range or "(" or ")", which denotes an exclusive end of the range. If one end of the range is to be included and the other excluded, the range can contain a square bracket with a round bracket.

There are three common use cases:

- A strict version range, such as [1.0.0,1.0.0], denotes version 1.0.0 and only that version.
- A half-open range, such as [1.0.0,2.0.0), which has an inclusive lower limit and an exclusive upper limit, denotes version 1.0.0 and any later version, up to, but not including, version 2.0.0.
- An unbounded open range expressed as a single number such as 2.0.0, which is equivalent to the range [2.0.0, *infinity*), denotes version 2.0.0 and any later version.

Converting Migrated Java Component Implementations

To regenerate a Java component implementation that has been migrated from 2.x to 3.x, you must manually convert the implementation to the 3.x structure.

In TIBCO ActiveMatrix 2.x, Java component implementations are generated into a single class. In TIBCO ActiveMatrix 3.x, Java component implementations are generated into two classes: an abstract class and a concrete class. The abstract class contains all the service method declarations, references, and properties and their respective accessor methods. The concrete class contains stubs of the service methods, which you complete with business logic. When you regenerate a 3.x Java component implementation (for example, after adding a property or service), only the abstract class is overwritten unless you specify that the concrete class should be overwritten by checking the Overwrite Concrete Class checkbox.

When a Java component implementation is migrated from 2.x to 3.x, the 2.x class is not converted to the 3.x structure. If you want to be able to regenerate a Java component implementation that has been migrated from 2.x to 3.x, you must manually convert the implementation to the 3.x structure following the procedures in this section.

Creating an Abstract Class

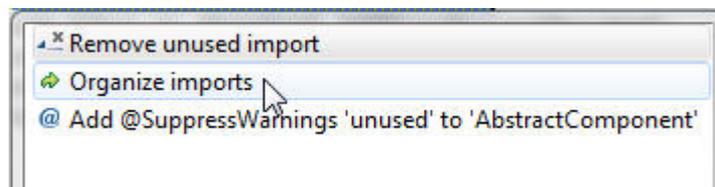
You can create an abstract class in the Project Explorer view.

Procedure

1. In the Project Explorer view, copy the component implementation class and add the prefix `Abstract` to the class name of the copy. For example, if the component class is named `ClassName`, then rename the copy to `AbstractClassName`.
2. Edit the abstract component implementation class `AbstractClassName`.
 - a) Delete the service method implementations and the life cycle methods (`@Init` and `@Destroy`). Errors will display.
 - b) Add the abstract declaration to `AbstractClassName`.
 - c) Add the abstract declaration to each service method.
 - d) Set the access level of properties (`@Property`) and references (`@Reference`) to `private`.
 - e) Add the javadoc annotation `@Generated` `TEMPL003` at the class level:

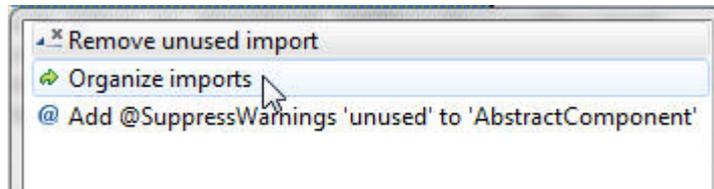
```
/**
 * @Generated TEMPL003
 */
public abstract class AbstractClassName
```

- f) Move custom user members and methods from `AbstractClassName` to `ClassName`.
- g) If any of the import statements displays a warning icon , left-click the icon and click **Organize imports**: in the pop-up that displays.



- h) Save the abstract class.
3. Edit the component implementation class `ClassName`.
 - a) `ClassName` implements one or more port types. Replace all the implements declarations with `extends AbstractClassName`.

- b) Remove all reference and property declarations and all accessor methods from the component implementation class.
- c) Edit method implementations to use accessor methods for property and reference objects.
- d) If any of the import statements displays a warning icon , left-click the icon and click **Organize imports** in the pop-up that displays.



- e) Save the concrete class.

Before Concrete Class

```

public class Foo implements Calculator, AreaService {

    public CalculatorOutputDocument add(CalculatorInputDocument input)
    {

        return CalculatorOutputDocument.Factory.newInstance();

    }

    public CalculatorOutputDocument divide(CalculatorInputDocument input)
    {

        return CalculatorOutputDocument.Factory.newInstance();

    }

    public CalculatorOutputDocument multiply(CalculatorInputDocument
input)
    {

        return CalculatorOutputDocument.Factory.newInstance();

    }

    public CalculatorOutputDocument subtract(CalculatorInputDocument
input)
    {

        return CalculatorOutputDocument.Factory.newInstance();

    }

    @Reference
    Phonebook Reference1;

    public Phonebook getReference1()
    {

        return Reference1;

    }

    public void setReference1(Phonebook Reference1)
    {

        this.Reference1 = Reference1;

    }

    @Reference
    AreaService Reference2;

    public AreaService getReference2()
    {

        return Reference2;

    }

    public void setReference2(AreaService Reference2)
    {

        this.Reference2 = Reference2;

    }

    public AreaDocument calculateRectArea(ParametersDocument parameters)
    {

        return AreaDocument.Factory.newInstance();

    }

    @Property
    String MyProperty1;

    public String getMyProperty1()
    {

        return MyProperty1;

    }

    public void setMyProperty1(String MyProperty1)

```

```

    {
        this.MyProperty1 = MyProperty1;
    }

    @Property
    String MyProperty2;

    public String getMyProperty2()
    {
        return MyProperty2;
    }

    public void setMyProperty2(String MyProperty2)
    {
        this.MyProperty2 = MyProperty2;
    }
}

```

After Abstract Class

```

/**
 * @Generated TEMPL003
 */
public abstract class AbstractFoo implements Calculator, AreaService
{
    @Reference
    private Phonebook Reference1;

    public Phonebook getReference1()
    {
        return Reference1;
    }
    public void setReference1(Phonebook Reference1)
    {
        this.Reference1 = Reference1;
    }

    @Reference
    private AreaService Reference2;

    public AreaService getReference2()
    {
        return Reference2;
    }
    public void setReference2(AreaService Reference2)
    {
        this.Reference2 = Reference2;
    }

    @Property
    private String MyProperty1;

    public String getMyProperty1()
    {
        return MyProperty1;
    }
    public void setMyProperty1(String MyProperty1)
    {
        this.MyProperty1 = MyProperty1;
    }

    @Property
    private String MyProperty2;

    public String getMyProperty2()
    {
        return MyProperty2;
    }
    public void setMyProperty2(String MyProperty2)
    {
        this.MyProperty2 = MyProperty2;
    }
}

```

```

        public abstract CalculatorOutputDocument add(CalculatorInputDocument
input);
        public abstract CalculatorOutputDocument
divide(CalculatorInputDocument input);
        public abstract CalculatorOutputDocument
multiply(CalculatorInputDocument input);
        public abstract CalculatorOutputDocument
substract(CalculatorInputDocument input);
        public abstract AreaDocument calculateRectArea(ParametersDocument
parameters);
    }

```

After Concrete Class

```

public class Foo extends AbstractFoo
{
    public CalculatorOutputDocument add(CalculatorInputDocument input)
    {
        return CalculatorOutputDocument.Factory.newInstance();
    }
    public CalculatorOutputDocument divide(CalculatorInputDocument input)
    {
        return CalculatorOutputDocument.Factory.newInstance();
    }
    public CalculatorOutputDocument multiply(CalculatorInputDocument
input)
    {
        return CalculatorOutputDocument.Factory.newInstance();
    }
    public CalculatorOutputDocument substract(CalculatorInputDocument
input)
    {
        return CalculatorOutputDocument.Factory.newInstance();
    }
    public AreaDocument calculateRectArea(ParametersDocument parameters)
    {
        return AreaDocument.Factory.newInstance();
    }
}

```

Editing a Manifest

You can edit a manifest from the Project Explorer.

Procedure

1. In the Project Explorer view, double-click the META-INF/MANIFEST.MF file of the Java component implementation's plug-in project.
The manifest file opens in the manifest editor.
2. Click the **Runtime** tab.
3. In the Exported Packages pane, select all the packages and click **Remove**.
4. Click **Add...**
The Exported Packages dialog displays.

5. In the Exported Packages dialog, click the package containing the component implementation's classes and click **OK**.
The package is added to the Exported Packages list.
6. Click **Properties**.
7. In the Version field, set the version to the same value as that of the Java component implementation plug-in, without the literal "qualifier" component.
8. Click the **MANIFEST.MF** tab. If there is a Bundle-Localization attribute, click the **Build** tab and ensure that the plugin.properties checkbox is checked.
9. Save the manifest file.

Regenerating a Component Implementation

You can regenerate a component implementation from the Project Explorer.

Procedure

1. In the Project Explorer view, double-click the composite containing the Java component to regenerate.
2. Right-click the Java component and select **Regenerate Java Component**.
If a dialog displays with the message "The Java file corresponding to this component was not generated by studio and is not supported. Proceeding with code generation will overwrite this Java class. Do you want to continue?", click **No** and ensure that the abstract class has a javadoc comment containing the tag @Generated TEMPL003.

If a dialog displays with the message "There are multiple elements with the same QName (namespace + name combination) in the WSDL files or schemas referenced by the component. This will result in duplicated classes, an invalid Java project, and an out-of-sync component.", click **Continue**.

Removing 2.x Data Binding JAR Files

You can remove 2.x data binding JAR files from the Project Explorer.

Prerequisites

Identify the JARs generated for the 2.x project. New JARs contain a hash number in the name (for example, AreaService-b58a5fc-service-beans.jar), so JARs that don't have it were generated by a 2.x product.

Procedure

1. In the Project Explorer view, right-click the Java component implementation project and select **Properties**.
2. Click the **Libraries** tab.
3. Select the 2.x JAR files and click **Remove**.
4. Click **OK**.
5. In the Project Explorer view, expand the lib folder in the Java component implementation project.
6. Select the 2.x JAR files, then right-click and select **Delete**.
A confirmation dialog displays.
7. Click **OK**.
8. In the Project Explorer view, double-click the META-INF/MANIFEST.MF file of the Java component implementation's plug-in project.

9. Click the **Runtime** tab.
10. In the Classpath pane, select the deleted JARs and click **Remove**.
11. Save the manifest file.

Correcting Custom Feature File

You can correct custom feature files from the Project Explorer.

Procedure

1. In the Project Explorer view, expand the `Deployment Artifacts` folder in the SOA project and double-click the component implementation custom feature. The custom feature opens in a custom feature editor.
2. Click the **Plug-ins** tab. If the Java component implementation plug-in is listed more than once, delete the plug-in duplicates. Ensure that the version of the plug-in contained in the custom feature exactly matches the version of the plug-in it refers to.
3. Ensure the custom feature file does not appear in more than one subfolder of the SOA project. Delete any duplicates so that each Java component implementation plug-in is contained in only one custom feature in the workspace.
4. Save the custom feature file.

Default XML to Java Mapping

When you generate a Spring component implementation or XML data binding classes, TIBCO Business Studio maps WSDL and XSD schema elements to Java programming language elements.

The following sections describes the default mappings of WSDL `definitions`, `types`, `portType`, `operation`, `message`, `part`, and `fault` elements to Java.



Generating implementations for two or more components in the same Java plug-in project using different binding types is not supported.



The payload for a `xsd:gMonth` datatype is converted incorrectly if you use JAXB data binding.

wSDL:definitions

The `wSDL:definitions` element's `targetNamespace` attribute is mapped to a Java package. By default, for a target namespace whose structure is: `http://rootPart/subPart`, the order of the elements in the root part of the target namespace are reversed in the package name. Subparts appearing after the root part separated by slashes are appended to the root part with a period (.). For example, the namespace `http://ns.tibco.com/StockQuote` becomes the package `com.tibco.ns.stockQuote`. If the first character of a namespace identifier is invalid, the preprocessor prepends an underscore `_` in front of the identifier.

wSDL:portType

A `wSDL:portType` element is mapped to a Java interface. The name of the interface is the value of the `name` attribute of the corresponding `wSDL:portType` element.

The generated interface contains Java methods mapped from the `wSDL:operation` subelements of the `wSDL:portType` element. Since WSDL 1.1 does not support port type inheritance, each generated interface contains methods for all the operations in the corresponding port type.

wSDL:operation

Each `wSDL:operation` element is mapped to a Java method in the corresponding Java interface. The `name` attribute of the `wSDL:operation` element determines the name of the generated method. If the `wSDL:operation` element contains a `wSDL:fault` message, the fault is mapped to a Java exception that appears in the `throws` clause of the generated method. See also [wSDL:fault](#).

wSDL:output, wSDL:input, and wSDL:part

The `name` attribute of the `part` element of the `wSDL:output` message is mapped to the return type of the generated Java method according to the XML data binding type as follows:

- JAXB - `name`
- XMLBeans - `nameDocument`

The method for accessing components of request parameters and defining response objects depends on the type of data binding you choose.

JAXB

The `type` or `element` attribute of the `part` element of the `wSDL:input` message is mapped to the type of the input parameter of the generated Java method. The `name` attribute of the `part` element of the `wSDL:input` message is mapped to the name of an input parameter of the generated Java method. You can directly access components of a request parameter as follows:

```
public AddPhoneResponse addPhone(AddPhoneRequest addPhoneParameters) {
    ...
    String firstName = addPhoneParameters.getFirstName();
    String lastName = addPhoneParameters.getLastName();
    String phone = addPhoneParameters.getPhone();
}
```

To create a response object or a complex object defined in the WSDL document:

1. Import `packageName.ObjectFactory`, where `packageName` is the package name generated from the WSDL document namespace.
2. Create an instance of `ObjectFactory`.
3. Create an object of type `Type` with the `createType` method.

For example:

```
import com.tibco.ns.hello.phonebook.ObjectFactory;
import com.tibco.ns.hello.phonebook.GetPhoneResponse;
...
ObjectFactory objectFactory = new ObjectFactory();
GetPhoneResponse phoneResponse = objectFactory.createGetPhoneResponse();

try{
    ...
    PhoneEntryType entry = objectFactory.createPhoneEntryType();
    while(rs.next()){
        entry.setEntryId(rs.getString("id"));
        entry.setFirstName(rs.getString("firstName"));
        entry.setLastName(rs.getString("lastName"));
        entry.setPhone(rs.getString("phone"));
    }
}catch(SQLException e){
    ...
}
return phoneResponse;
```



When implementing a JAXB-based Spring component service, users typically form a response object in their service method, populate it with some response data, and return it from the method. Such a returned object is then marshalled into an XML (DOM) payload by the platform. While the platform code is marshalling this payload, if the user code manipulates the contents of the same object, the JAXB marshaller throws a `java.util.ConcurrentModificationException`. Make sure the contents of the response object returned from the service method are not modified by multiple threads.

XMLBeans

There are two ways to specify the type of a message part: indirectly through an `element` attribute that is defined in the `wsdl:types` element or directly with a `type` attribute. If you use XMLBeans binding, the generated Java code depends on how you specify the types of message parts.

When you define the types of the parts through the `element`, attribute classes named `ElementNameDocument`, where `ElementName` is the input and output message type element name with the first letter capitalized, are generated. The generated Java method accepts a document type named `ElementNameDocument`. The generated method returns a document type similarly named according to the element that specifies the type of the output message part.

In the following WSDL document, the types of the message parts are defined through an `element` attribute:

```
<wsdl:definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://ns.tibco.com/StockQuote/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://ns.tibco.com/StockQuote/">
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      attributeFormDefault="unqualified" elementFormDefault="qualified"
      targetNamespace="http://ns.tibco.com/StockQuote/">
      <xs:element name="symbol" type="xs:string"/>
      <xs:element name="quote" type="xs:float"/>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="OperationRequest">
    <wsdl:part name="stockQuoteRequest" element="tns:symbol"/>
  </wsdl:message>
```

```

<wsdl:message name="OperationResponse">
  <wsdl:part name="stockQuoteResponse" element="tns:quote"/>
</wsdl:message>
<wsdl:portType name="StockPT">
  <wsdl:operation name="getQuote">
    <wsdl:input message="tns:OperationRequest"/>
    <wsdl:output message="tns:OperationResponse"/>
  </wsdl:operation>
</wsdl:portType>
</wsdl:definitions>

```

The following code fragment shows the generated Java class implementation:

```

import com.tibco.stockQuote.SymbolDocument;
import com.tibco.stockQuote.QuoteDocument;

public class StockQuoteServiceImpl extends AbstractStockQuoteServiceImpl {
  public QuoteDocument getQuote(SymbolDocument stockQuoteRequest)
  {
    String sym = stockQuoteRequest.getSymbol();
    float quote = quoteLookup(sym);
    QuoteDocument response = QuoteDocument.Factory.newInstance();
    response.setQuote(quote);
    return response;
  }
}

```

The relationships between the message part, message part type, message type element, and document type are:

Message Part	Type	Element	Document Type
stockQuoteRequest	xs:string	tns:symbol	SymbolDocument
stockQuoteResponse	xs:float	tns:quote	QuoteDocument

The value of the request message part is retrieved from the document using bean-style accessors. In the example, the stock symbol is retrieved from the SymbolDocument object with the getSymbol method.

You create a response document, of type QuoteDocument, by calling the newInstance method of the document factory class. Finally, you set the value of the response message part by calling the setQuote method on the response document.

In the following WSDL document, the types of the message parts are specified through a type attribute:

```

<wsdl:definitions
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://ns.tibco.com/StockQuote/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
name="Untitled" targetNamespace="http://ns.tibco.com/StockQuote/">
  <wsdl:message name="OperationRequest">
    <wsdl:part name="symbol" type="xs:string"/>
  </wsdl:message>
  <wsdl:message name="OperationResponse">
    <wsdl:part name="quote" type="xs:float"/>
  </wsdl:message>
  <wsdl:portType name="StockPT">
    <wsdl:operation name="getQuote">
      <wsdl:input message="tns:OperationRequest"/>
      <wsdl:output message="tns:OperationResponse"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>

```

For this WSDL document, the generated Java code references the message parts directly, instead of through documents. However, the types of the message parts are XMLBeans types, which means that you must use the XMLBeans API to access the XML data bound to Java objects and convert between XMLBeans types and native Java types in your method implementation. To perform this conversion,

you use [get|set]*Type*Value methods, where *Type* is the native Java type. Like the document types described earlier, you create XMLBeans objects by calling the newInstance method of the type's Factory class.

```
import org.apache.xmlbeans.XmlFloat;
import org.apache.xmlbeans.XmlString;

public class StockQuoteServiceImpl extends AbstractStockQuoteServiceImpl {

    public XmlFloat getQuote(XmlString symbol){
        float quote = quoteLookup(symbol.getStringValue());
        XmlFloat resp = XmlFloat.Factory.newInstance();
        resp.setFloatValue(quote);
        return resp;
    }
}
```

wsdl:fault

A wsdl:fault element is mapped to a Java exception. The generated exception class extends the class java.lang.Exception. The name of the exception is formed by concatenating the name attribute of the wsdl:message referenced by the wsdl:fault element with Exception. For the following WSDL fragment, the exception class would be named GetCurrentTimeFaultMsgException.

```
<schema>
...
<element name="CurrentTimeFault" type="string"/>
...
</schema>
<wsdl:message name="getCurrentTimeFaultMsg">
  <wsdl:part element="tns:getCurrentTimeFault" name="faultInfo"/>
</wsdl:message>
<wsdl:portType name="DateManagerPT">
  <wsdl:operation name="getCurrentTime">
    <wsdl:input message="tns:OperationRequest"/>
    <wsdl:output message="tns:OperationResponse"/>
    <wsdl:fault message="ns0:getCurrentTimeFaultMsg" name="faultMsg"/>
  </wsdl:operation>
</wsdl:portType>
```

XMLBeans

A fault object named *fault*Document is generated, where *fault* is the type of the fault message's part. For the preceding WSDL fragment, the fault object would be named GetCurrentTimeFaultDocument.