BEA WebLogic Server™®

Beehive Integration in BEA WebLogic Server 9.0

Version 9.0
Document Revised: July 20, 2005

5. Beehive Tutorial

About This Document

This document describes the implementation of the Apache Beehive open-source software suite in BEA WebLogic Server. Specifically, it describes the additional functionality available in WebLogic Server that is not available in the open-source version of Beehive.

For more information about the version of the Apache Beehive open-source project installed on your system, please see Apache Beehive Documentation in Chapter 1, "Beehive Applications."

This document covers the following topics:

- Chapter 1, "Beehive Applications"

  An overview of Beehive.

- Chapter 2, "Beehive System Controls"

  An overview of the controls section of Beehive.

- Chapter 3, "Building Beehive Applications"

  Detailed instructions for building Beehive applications under WebLogic Server.

- Chapter 4, "Annotation for Rowsets"

  Describes the metadata annotations available in WebLogic Server for handling Java RowSet objects.

- Chapter 5, "Beehive Tutorial"

  Walks the reader through building three types of Beehive applications.

What You Need to Know

This document is intended for developers who want to develop Beehive applications under WebLogic Server.

Product Documentation on the dev2dev Web Site

BEA product documentation, along with other information about BEA software, is available from the BEA dev2dev Web site:

`http://dev2dev.bea.com`

To view the documentation for a particular product, select that product from the list on the dev2dev page; the home page for the specified product is displayed. From the menu on the left side of the screen, select Documentation for the appropriate release. The home page for the complete documentation set for the product and release you have selected is displayed.

Related Information

Readers of this document might find the documentation of open-source projects at the Apache Software Foundation site, http://www.apache.org, especially useful.

Contact Us!

Your feedback on the BEA WebLogic Server documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Server documentation.

In your e-mail message, please indicate that you are using the documentation for BEA WebLogic Server 9.0.

If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support at **http://support.bea.com**. You can also contact Customer Support by using the contact information provided on the quick reference sheet titled “BEA Customer Support,” which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.
`monospace text`	Indicates *user input*, as shown in the following examples: • Filenames: `config.xml` • Pathnames: *`BEAHOME`*`/config/examples` • Commands: `java -Dbea.home=BEA_HOME` • Code: `public TextMsg createTextMsg(`
	Indicates *computer output*, such as error messages, as shown in the following example: `Exception occurred during event dispatching:java.lang.ArrayIndexOutOfBoundsException: No such child: 0`
`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `java utils.MulticastTest -n` *`name`* `[-p` *`portnumber`*`]`

Convention	Item
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed. *Example*: `java weblogic.deploy [list|deploy|update]`
`. . .`	Indicates one of the following in a command line: • That an argument can be repeated several times in a command line • That the statement omits additional optional arguments • That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o` *`name`*`] [-f "file1.cpp file2.cpp file3.cpp . . ."`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER 1

Beehive Applications

What Is a Beehive Application?

Beehive is an open source J2EE programming framework designed to make the most difficult J2EE programming tasks easier.

Where the traditional J2EE programmer would use complex APIs and configuration files, the Beehive programmer uses “metadata annotations,” also known simply as “annotations.” Annotations dramatically simplify the J2EE code and allow non-experts to create J2EE applications. Annotations stand in for common, boilerplate coding tasks. Instead of learning the complex J2EE API, developers can set properties on their code using metadata annotations.

Annotations are a new feature in Java, added in the J2SE 5.0 version of the language. For background information about annotations, please see http://java.sun.com/j2se/1.5.0/index.jsp and http://jcp.org/en/jsr/detail?id=175.

Beehive ships with three main components:

- NetUI Page Flow

  A web application framework built on Apache Struts. Page Flows centralize application logic and state in Java “controller” classes. An integrated set of JSP tags is provided, as well as integration with JavaServer Faces and with raw Struts applications.

- Java Controls

  A lightweight component framework that helps programmers encapsulate application logic and leverage metadata annotations into their programming model. Developers can create

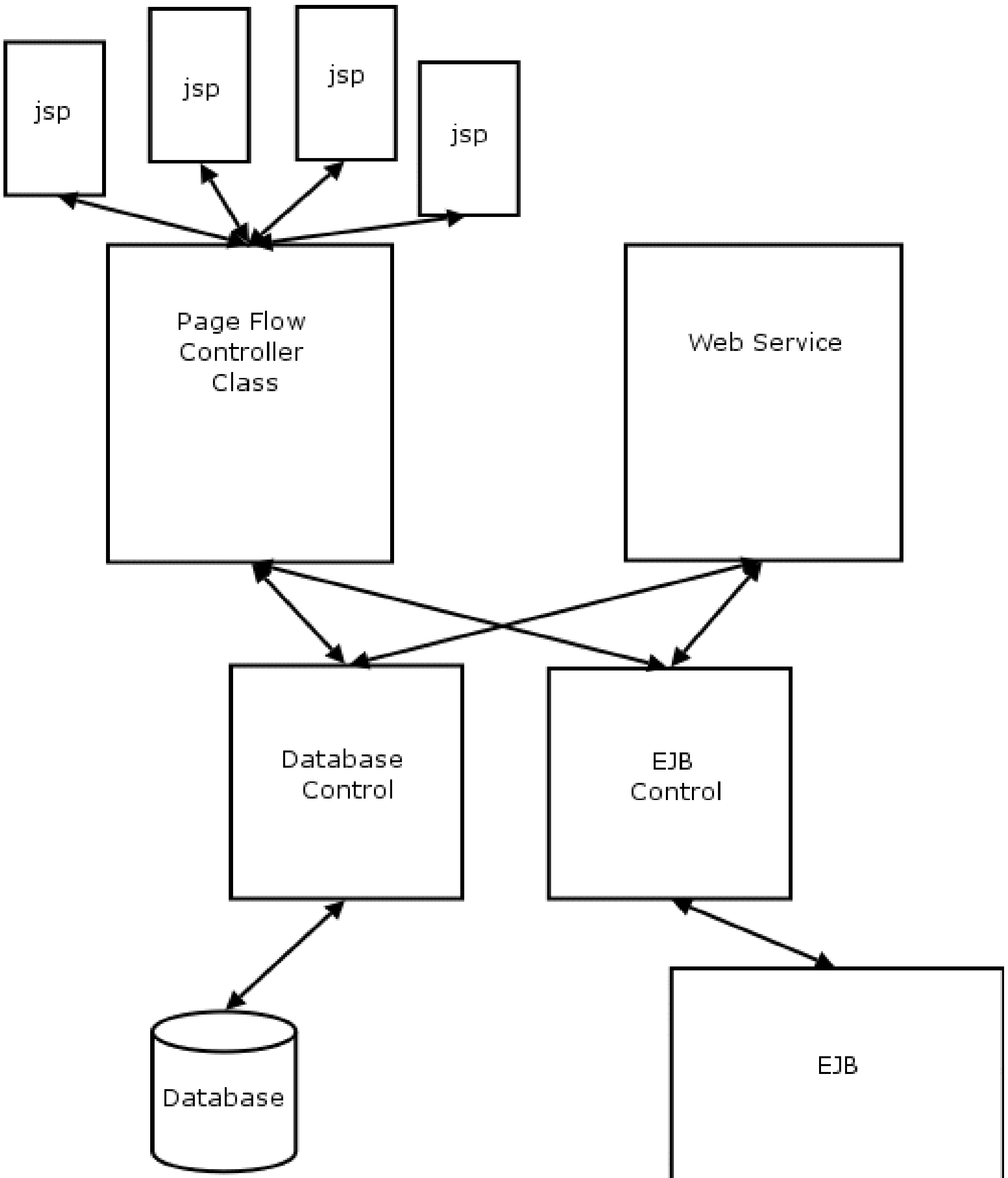
custom controls or use the pre-built **system controls**. The Beehive system controls provide access to these common J2EE components: database, EJB, JMS, and web services.

- Web Services

  An implementation of JSR 181, an annotation-driven programming model for web services. For more information about the JSR 181 specification, please see http://www.jcp.org/en/jsr/detail?id=181.

The diagram below shows a typical Beehive application. The application has three basic components:

- Apage flow web application providing access for end users over the internet
- Aweb service interface providing in-machine access to application resources
- Two controls (database and EJB) providing access to backend resources


jsp
jsp
jsp
jsp
Page Flow Controller Class
Web Service
Database Control
EJB Control
Database
EJB

Apache Beehive Documentation

Beehive is a project of the Apache Software Foundation. The version of Beehive that is installed on your computer during the installation of WebLogic Server does not necessarily correspond to the version currently documented on the Apache website. To see the full, current Apache Beehive documentation, refer to http://beehive.apache.org.

A copy of the Beehive documentation from the Apache site is installed on your computer when WebLogic Server is installed. This documentation corresponds to the version of Beehive on your computer. To read this documentation, navigate to

`<WEBLOGIC_HOME>\beehive\apache-beehive-incubating-1.0m1\docs\docs`

and double-click on the file `index.html` in that folder.

Note: The version of Beehive shipped with WebLogic Server 9.0 does not include the web service metadata sub-project. Consequently, the web service metadata documentation has not been included and hyperlinks to the removed documentation content are broken.

Running the WebLogic Server/Beehive Integration Samples

Samples Architecture

WebLogic Server contains three interconnected Beehive samples. These samples demonstrate how to build progressively more complex applications based on a simple web service. The simple web service, called creditRatingApp, at the core of these samples takes a nine-digit Social Security number and responds with a credit rating. The remaining two samples are more complex client applications of this basic web service.

These three samples, named creditRatingApp, bankLoanApp, and customerLoanApp, are described in "creditRatingApp" on page 1-5, "bankLoanApp" on page 1-5, and "customerLoanApp" on page 1-5, respectively.

Locating the Samples

The samples documented here are located on your system in

`<WEBLOGIC_HOME>\beehive\weblogic-beehive\samples.`

creditRatingApp

This sample is a stateless web service (built using JSR 181 web service annotations) that takes a nine-digit Social Security number and responds with a credit worthiness rating. The following two samples are clients of this web service.

bankLoanApp

This sample is an application that evaluates a loan seeking customer. Given the customer's Social Security number, the application will (1) return an interest rate appropriate for that customer's credit rating and (2) decide whether or not a customer should be given a loan of a specified amount.

The application consists of a conversational (stateful) web service consisting of a start and a finish method. The start method takes a Social Security number and returns an interest rate based on the customer's credit rating. The start method acquires the customer's credit rating by calling the creditRatingApp web service described in "creditRatingApp" on page 1-5. The credit rating is then passed to a local EJB, which calculates the appropriate interest rate.

The finish method determines whether the customer may borrow a specified amount. This method takes a float value and returns a boolean value. This method also checks a database to see if the customer has borrowed money in the past.

Because this application relies on a database connection, to run this application, you must first create the necessary database. Details on creating the necessary database are described in step 3 in "Running the bankLoanApp Application" on page 1-6.

The Java source for this web service is located at

`bankLoanApp\src\services\pkg\bankLoanConversation.java.`

customerLoanApp

This sample is a web application interface on the bankLoanApp sample described in "bankLoanApp" on page 1-5. It provides JSPs from which users can make requests of and view responses from the bankLoanApp.

Running the Beehive Samples

To run the samples, you must first create a Beehive-enabled server domain, then you must build and deploy the three samples in the following order: creditRatingApp, bankLoanApp, customerLoanApp.

Creating a Beehive-enabled Domain

To create a Beehive-enabled server domain, follow these steps:

1. Start the domain configuration wizard: Start > BEA Products > Tools > Configuration Wizard.

   Click Next on each page of the wizard without changing any of the default values, except for the following changes:

   On the second page (labeled **Select Domain Source**), place a check in the checkbox next to “Apache Beehive.”

   On the third page (labeled **Configure Administrator Username and Password**), in the **User password** field, enter `weblogic`.

2. Start an instance of the new server by running

   ```
   <BEA_HOME>\user_projects\domains\base_domain\startWebLogic.cmd
   ```

   on Windows or

   ```
   <BEA_HOME>/user_projects/domains/base_domain/startWebLogic.sh
   ```

   on UNIX.

3. Set up the build environment by opening a command shell and running

   ```
   <BEA_HOME>\user_projects\domains\base_domain\bin\setDomainEnv.cmd
   ```

   on Windows or

   ```
   <BEA_HOME>/user_projects/domains/base_domain/bin/setDomainEnv.sh
   ```

   on UNIX.

Running the creditRatingApp Web Service

To build and deploy the **creditRatingApp**:

1. Navigate to `creditRatingApp\src`.
2. Run the following Ant command:

   ```
   ant clean build pkg.exploded deploy.exploded
   ```

Running the bankLoanApp Application

1. Navigate to `bankLoanApp\src`.
2. Run the following Ant command:

```
ant clean build pkg.exploded deploy.exploded
```

3. To create the necessary backend database, visit the URL http://localhost:7001/bankLoanWeb and click the link **Create the Table**. After the table has been created, click **Return to index.**
4. To test the bankLoanApp, enter a nine-digit number in the field marked **Get the amount borrowed by SSN** and click **Submit**. The amount previously borrowed by the applicant will be displayed in the server console.

Running the customerLoanApp

To build and deploy the **customerLoanApp**:

1. Navigate to `customerLoanApp\src`.
2. Run the following Ant command:

   ```
   ant clean build pkg.exploded deploy.exploded
   ```

3. To test the web application, visit the URL http://localhost:7001/customerLoanWeb.
4. Enter a nine-digit Social Security number to retrieve an interest rate; then request a loan of a specific amount.

Beehive System Controls

Beehive system controls in WebLogic Server are an implementation of the system controls in the Apache Beehive project. These sections describe the additional functionality provided by the WebLogic Server implementation.

For a detailed description of Beehive controls and their functionality, please see the Apache Beehive documentation at http://beehive.apache.org.

Note: Beehive system controls are installed on your computer automatically during the BEA Products installation process. Because the system controls are pre-installed, the sections of the Apache Beehive documentation that describe how to download and install Beehive do not apply to WebLogic Server users.

JDBC Control

JDBC controls make it easy to access a relational database from your Java code using SQL commands. The JDBC controls automatically perform the translation from database queries to Java objects so that you can easily access query results.

A JDBC control can operate on any database for which an appropriate JDBC driver is available and for which a data source has been configured. When you add a new JDBC control to your application, you specify a data source for that control. The data source indicates which database the control is bound to.

The full functionality of Apache Beehive JDBC controls is available to you in WebLogic Server. In addition, WebLogic Server provides you with a special annotation, SQLRowset, that supports retrieving of data as a RowSet. For information about this annotation, see "SQLRowSet" on page 4-1.

Web Service Control

The web service control gives your application easy access to web services. You use the web service control in an application just like any other Beehive control: (1) you declare the control using the `@Control` annotation on a class member field, and (2) you call the methods on the control. For code examples, please see "Beehive Tutorial" on page 5-1.

Web service controls are generated on a per-service basis. WebLogic Server provides a service control generation tool based on the target service's WSDL file. For details on generating a web service control, see "Building Web Service Controls" on page 3-7.

Note: WebLogic Server does not ship with the Beehive implementation of the web service control. Instead, it ships with its own implementation of a web service control. In other words, generated service controls extend the class com.bea.control.ServiceControl, *not* the class org.apache.beehive.controls.system.webservice.ServiceControl.

Compiling a web service control is a multi-step process described in "Building Web Service Controls" on page 3-7.

EJB and JMS Controls

The EJB and JMS controls shipped with WebLogic Server 9.0 are identical to the Beehive versions. For detailed information on these controls, see the Apache Beehive documentation at http://beehive.apache.org/.

Control Security

For background information about the terms and concepts referred to in this section, please see the Java Authentication and Authorization Service documentation on the Sun website and the BEA WebLogic Server Security documentation.

WebLogic Server provides security for Beehive controls through the `@Security` annotation.

The com.bea.controls.Security annotation is used to decorate control methods. The `@Security` annotation interface has the following optional attributes:

@Security Attributes

Attribute Name	Value	Description
rolesAllowed	String[]	Optional. An Array of role-names allowed to run the annotated method. If the current principal calling the method has been granted one of the specified roles, then the method invocation will be granted. Otherwise, a javax.security.SecurityException will be thrown.
runAs	String	Optional. A role-name. The method will be run with the current principal being set to the role specified, unless runAsPrincipal is given.
runAsPrincipal	String	Optional. A principal name. The method will run with the current principal as specified. Must be used in conjunction with runAs.

The `@Security` annotation can be placed on a method of a control interface. `@Security` is not allowed on `@Control` fields or control class declarations.

To use the `@Security` annotation in your control file, you must import the class com.bea.control.annotations.Security.

```
import com.bea.control.annotations.Security;
```

You must also (1) define the referenced roles at the application-level using <security-role> in application.xml and (2) provide role mappings using <security-role-assignment> weblogic-application.xml.

For example, if you reference the `manager` role in the `@Security` annotation:

```
@Security( rolesAllowed={"manager"} )
public String getSalary();
```

then you must define the manager role using <security-role> in application.xml:

application.xml

```
<security-role>
   <role-name>manager</role-name>
</security-role>
```

and you must provide role/principal mappings using <security-role-assignment> in weblogic-application.xml:

weblogic-application.xml

```
<security>
   <security-role-assignment>
      <role-name>manager</role-name>
      <principal-name>manager</principal-name>
   </security-role-assignment>
</security>
```

Supported Role Scopes

Only the "application" role scope is supported. Global, web-app, and EJB scopes are not supported. The application scoping is defined by the <security-role> and <security-role-assignment> elements in the application.xml and weblogic-application.xml files, respectively (see implementation details in "Control Security" on page 2-2).

The runAs attribute value (a role) is mapped to a principal as follows:

1. If a <security-role-assignment> element with the <role-name> having the value `id` is defined in the weblogic-application.xml file, then the value of the first <principal-name> element is used as the principal if given.
2. Otherwise, the `id` is assumed to be the name of a principal.

Building Beehive Applications

Build Resources

WebLogic Server ships with the following Ant files for building and deploying Beehive applications:

```
<WEBLOGIC_HOME>\beehive\weblogic-beehive\ant\weblogic-beehive-buildm
odules.xml
```

```
<WEBLOGIC_HOME>\beehive\weblogic-beehive\ant\weblogic-beehive-import
s.xml
```

```
<WEBLOGIC_HOME>\beehive\weblogic-beehive\ant\weblogic-beehive-tools.
xml
```

`weblogic-beehive-buildmodules.xml` contains build macros (`<macrodef>` elements) for building a web app module.

`weblogic-beehive-imports.xml` is a utility file that defines paths to build resource JARs required by weblogic-beehive-tools.xml.

`weblogic-beehive-tools.xml` contains build macros for building other Beehive-related source artifacts, including XMLBean schemas and the three Beehive components: Java Controls, NetUI Page Flow and Web Services.

To use these build resources, import these three files into your Ant build file:

```
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-imports.xml
"/>
```

```
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-tools.xml"/
>

<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-buildmodule
s.xml"/>
```

Then call the build macros to build your Beehive application. For example, the following Ant target calls the `build-webapp` macro (`<macrodef name="build-webapp"/>`) located in `weblogic-beehive-buildmodules.xml`.

```
<build-webapp
        webapp.src.dir="${src.dir}/myWebApp"
        webapp.build.dir="${dest.dir}/myWebApp"
        app.build.classpath="myWebApp.build.classpath"/>
</target>
```

A template Ant build file is provided in "Template Build File" on page 3-9. The "Beehive Tutorial" on page 5-1 puts this template Ant build file into practice.

Note that the template build file assumes that your Beehive application contains the following library reference in `myBeehiveApp\META-INF\weblogic-application.xml`:

```
<weblogic-application xmlns="http://www.bea.com/ns/weblogic/90">
   <library-ref>
      <library-name>weblogic-beehive-1.0</library-name>
   </library-ref>
</weblogic-application>
```

If your Beehive application contains a web module, the template build file assumes the following library references in `myBeehiveApp\myWebApp\WEB-INF\weblogic.xml`:

```
<weblogic-web-app
    xmlns="http://www.bea.com/ns/weblogic/90"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.bea.com/ns/weblogic/90
    http://www.bea.com/ns/weblogic/90/weblogic-web-app.xsd">
    <library-ref>
        <library-name>beehive-netui-1.0</library-name>
    </library-ref>
    <library-ref>
        <library-name>struts-1.2</library-name>
    </library-ref>
    <library-ref>
        <library-name>jstl-1.1</library-name>
    </library-ref>
</weblogic-web-app>
```

On disk, these referenced libraries reside in the EAR and WAR files at `<WEBLOGIC_HOME>\common\deployable-libraries`. Note that the Ant task `<libclasspath>` is used to unpack the referenced EAR and WAR files. See Using the libclasspath Ant Task in *Developing Applications with WebLogic Server*.

Ant Tasks

The following Ant macros describe the most important Ant tasks included in the build resource files.

weblogic-beehive-buildmodules.xml

build-webapp-module

Builds and assembles a web application module. This Ant task will compile any page flows and Java Controls inside the web application.

This task takes the following parameters:

Name	Definition
`webapp.src.dir`	Required. The base directory of the web application
`webapp.build.dir`	Required. The *web application* build directory
`app.build.dir`	Required. The *application* build directory
`app.build.classpath`	Required. The classpath used for compilation
`temp.dir`	Required. A temporary directory

Control source Java files are assumed to be under [webAppDir]\WEB-INF\src.

Page flow source Java files are assumed to be under [webAppDir].

build-control-module

Builds a distributable control JAR from a directory of Java control source files.

The task takes the following parameters:

Name	Definition
`srcdir`	Required. The directory containing the Java sources
`destjar`	Required. The name of the JAR produced
`tempdir`	Required. A temporary directory containing the generated sources and classes
`classpathref`	Required. The classpath used for compilation

build-ejb-module

Builds an EJB JAR from one or more EJB controls.

The task takes the following parameters:

Name	Definition
`srcdir`	Required. The directory containing the EJB sources
`destjar`	Required. The name of the JAR produced containing the EJB controls
`tempdir`	Required. A temporary directory containing the generated sources and classes
`classpathref`	Required. The classpath used for compilation

weblogic-beehive-tools.xml

assemble-controls

Assemble the controls found in a directory. Assembly is required as part of the process of compiling web service, JMS, and EJB controls. For details on the control build process, see "Building JMS and EJB Controls" on page 3-9.

The task takes the following parameters:

Name	Definition
`moduledir`	Required. The directory containing the exploded contents of the module
`destdir`	Required. The destination directory for compiled class files
`classpathref`	Required. The classpath used for compilation

`assemblerclass`	Required.
	For EJB modules the value must be
	`org.apache.beehive.controls.runtime.assembly.EJBAssemblyContext$Factory`
	For web app modules the value must be
	`org.apache.beehive.controls.runtime.assembly.WebAppAssemblyContext$Factory`

generate-type-library

Generates a Java type library from a WSDL file. The types are packaged as a JAR file. If the WSDL contains complex Java types, it is necessary to run this target as part of the process of generating a service control from the WSDL. For details on generating a service control from a WSDL, see "Building Web Service Controls" on page 3-7.

The task takes the following parameters:

Name	Definition
`wsdl`	Required. The WSDL file from which the type library is to be generated
`wsdlname`	Required. The relative path of the WSDL to the project in which it is placed (without the .wsdl extension)
`wsdlservicename`	Required. The service name in the WSDL containing the complex-types from which the type-library will be generated
`tempdir`	Required. A temporary directory
`destdir`	Required. The directory where the type-library JAR will be placed
`typefamily`	Optional. The type family; default value is **tylar**

generate-webservice-client

Generates a web service client from a WSDL file.

The task takes the following parameters:

Name	Definition
`wsdl`	Required. The WSDL file from which the client will be generated
`tempdir`	Required. A temporary directory containing the generated sources
`destjar`	Required. The directory where the client will be placed
`package`	Required. The package of the generated source

generate-webservice-control

Generates a web service control from a WSDL file.

The task takes the following parameters:

Name	Definition
`wsdl`	Required. The WSDL file from which the service control is to be generated
`wsdlservicename`	Required. The WSDL service name
`servicecontrolname`	Required. The service control class name
`destdir`	Required. The directory where the web service control class and WSDL will be placed
`appdir`	Required. The application directory
`package`	Required.
`typefamily`	Optional. The type family; the default value is **no_complex_types**
`classpathref`	Required. The classpath used for compilation

Building Beehive Applications

As a starting point for building Beehive applications, use the build template file provided in "Template Build File" on page 3-9. For each component in your Beehive application, add build elements to the template build file.

For each web application (any component containing page flows and Java Controls) call `<build-webapp>`.

Web services should be compiled using the `<jwsc>` (`weblogic.wsee.tools.anttasks.JwscTask`) target.

To build applications that contain the system controls, see "Building Web Service Controls" on page 3-7 and "Building JMS and EJB Controls" on page 3-9.

Building Web Service Controls

The following instructions explain how to create a web service control, integrate it into a Beehive application, and compile the resulting application. The instructions assume that you have access to the target web service's WSDL file.

1. Acquire the WSDL for the target web service.
2. Call `<build-type-library>` (defined in `weblogic-beehive-tools.xml`) if there are any complex Java types passed by the target web service.

   This will generate a type library JAR file, which should be saved in the application's `APP-INF\lib` directory.

   ```
   <build-type-library
     wsdl="${src.dir}/services/MyService.wsdl"
     wsdlname="MyService.wsdl"
     wsdlservicename="MyService"
     tempdir="c:/temp/services"
     destdir="${src.dir}/APP-INF/lib"
   />
   ```

   Calling `<build-type-library>` is only necessary if the web service contains complex Java types; if the web service contains only standard Java types, this step can be skipped.
3. Call `<generate-webservice-control>` (defined in `weblogic-beehive-tools.xml`).

   This will generate a Java source file for the web service control.

   If the web service does not contain complex types, then the typefamily attribute should have the value **no_complex_types**.

   ```
   <generate-webservice-control
             wsdl="${src.dir}/services/MyService.wsdl"
             wsdlservicename="MyService"
             servicecontrolname="MyServiceControl"
             destdir="${src.dir}/services"
             appdir="${src.dir}"
             package="pkg"
             typefamily="no_complex_types"
             classpathref="app.classpath"/>
   ```

   The Java source file that is created should not be modified, except for the addition of security annotations. For details on the available security annotations, see "Control Security" on page 2-2.
4. Place the Java web service control source file (and any type library JAR file) into the client application. Before the client can use the web service control, the client needs to be assembled, which creates appropriate bindings between the client and the web service control.

   Call the `<assemble-controls>` task.
5. Calling <assemble-controls> creates Java source code (and modifies some deployment descriptors), which needs to be compiled in turn.

The following Ant targets illustrate how to implement steps 4 and 5 in a J2EE application context. Inside a J2EE application, the `assemblerclass` attribute should have the value **org.apache.beehive.controls.runtime.assembly.EJBAssemblyContext$Factory**. Inside of a web application context, `assemblerclass` should have the value **org.apache.beehive.controls.runtime.assembly.WebAppAssemblyContext$Factory.**

```
<!-- Assemble the controls to a tmp dir, then compile the assembly output
-->

<assemble-controls
    moduledir="${dest.dir}/myApp"
    destdir="${assembly.build.tmp.dir}"
    classpathref="myWeb.assembly.classpath"
    assemblerclass="org.apache.beehive.controls.runtime.assembly.EJBAss
emblyContext$Factory"
    />

<javac srcdir="${assembly.build.tmp.dir}"
    destdir="${dest.dir}/myWeb/WEB-INF/classes"
    classpath="customerLoanWeb.assembly.classpath"
    />
```

Building JMS and EJB Controls

The process for generating JMS and EJB controls is identical to steps 4-5 in "Building Web Service Controls" on page 3-7: you must run assembly on the control and then compile the results of the assembly.

Template Build File

Use the following Ant build file as a template to build and deploy your Beehive applications.

Note that this template file uses the task `<wlcompile>`, which ensures that split source compilation and deployment will succeed. `<wlcompile>` also automatically searches for and compiles all Java files in your application. Be aware that `<wlcompile>` will fail if it encounters any Beehive-specific classes in your application. For this reason, you should exclude your Beehive projects from the `<wlcompile>` search. For example:

```
<wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
excludes="myBeehiveProj1,myBeehiveProj2"/>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<project default="build" basedir=".">
```

```
<property environment="env"/>

<!-- provide overrides -->
<property file="build.properties"/>

<property name="src.dir" value="${basedir}"/>
<property name="dest.dir" value="${basedir}/../build"/>
<property name="dist.dir" value="${basedir}/../dist"/>

<property name="app.name" value="beehive_tutorial"/>
<property name="ear.path" value="${dist.dir}/${app.name}.ear"/>
<property name="tmp.dir" value="${java.io.tmpdir}"/>
<property name="weblogic.home" value="${env.WL_HOME}"/>

<property name="user" value="weblogic"/>
<property name="password" value="weblogic"/>

<fail unless="weblogic.home" message="WL_HOME not set in environment"/>

<property name="beehive.home" value="${weblogic.home}/beehive"/>

<!-- beehive-imports defines dependency paths that are required by
beehive-tools.xml -->
  <import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-imports.xml"/>
  <!-- defines macros for build-schemas, build-controls, build-pageflows -->
  <import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-tools.xml"/>
  <!-- defines macros for build-webapp -->
  <import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-buildmodules.x
ml"/>

<taskdef name="libclasspath"
classname="weblogic.ant.taskdefs.build.LibClasspathTask"/>

<target name="init.app.libs">
     <libclasspath basedir="${src.dir}" tmpdir="c:/tmp/wls_lib_dir"
property="app.lib.classpath">
        <librarydir dir="${weblogic.home}/common/deployable-libraries/" />
     </libclasspath>
     <echo message="app.lib.claspath is ${app.lib.classpath}"
```

```
level="info"/>
</target>

<target name="init.dirs">
     <mkdir dir="${dest.dir}/APP-INF/classes"/>
     <mkdir dir="${dest.dir}/APP-INF/lib"/>
     <mkdir dir="${dist.dir}"/>
</target>

<target name="init" depends="init.app.libs,init.dirs">
    <path id="app.classpath">
      <pathelement location="${src.dir}/APP-INF/classes"/>
      <pathelement location="${dest.dir}/APP-INF/classes"/>
      <pathelement path="${app.lib.classpath}"/>
      <fileset dir="${src.dir}/APP-INF/lib">
        <include name="**/*.jar"/>
      </fileset>
      <fileset dir="${dest.dir}/APP-INF/lib">
        <include name="**/*.jar"/>
       </fileset>
       <fileset
dir="${beehive.home}/apache-beehive-incubating-1.0m1/lib/netui">
          <include name="**/*.jar"/>
          <exclude name="**/beehive-netui-compiler.jar"/>
       </fileset>
    </path>
    <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"/>
</target>

<target name="build" depends="compile,appc"/>

<target name="compile" depends="init"/>

<target name="clean" depends="init.dirs,clean.dest,clean.dist"/>

<target name="clean.dest">
    <echo message="deleting dest.dir:${dest.dir}"/>
    <delete includeemptydirs="true" >
     <fileset dir="${dest.dir}" excludes=".beabuild.txt" includes="**/*" />
    </delete>
</target>
```

```
<target name="clean.dist">
    <echo message="deleting dest.dir:${dest.dir}"/>
    <delete includeemptydirs="true" >
      <fileset dir="${dist.dir}" includes="**/*" />
    </delete>
</target>

<target name="appc" depends="init" >
    <wlappc source="${dest.dir}"
librarydir="${weblogic.home}/common/deployable-libraries/"/>
</target>

<target name="pkg.exploded">
    <antcall target="clean.dist"></antcall>
    <wlpackage toDir="${dist.dir}" srcdir="${src.dir}"
destdir="${dest.dir}" />
</target>

<target name="deploy.exploded" >
    <wldeploy user="${user}" password="${password}" action="deploy"
name="${app.name}" source="${dist.dir}"/>
</target>

<target name="deploy" >
    <wldeploy user="${user}" password="${password}" action="deploy"
name="${app.name}" source="${dest.dir}"/>
</target>

<target name="redeploy">
    <wldeploy user="${user}" password="${password}" action="redeploy"
name="${app.name}"/>
</target>

</project>
```

Annotation for Rowsets

SQLRowSet

The SQLRowSet metadata annotation has been added in WebLogic Server 9.0. SQLRowSet is a method-level annotation that supports RowSet functionality. It has the following members.

SQLRowSet Members

Member Name	Type	Required	Use	Values
commandType	CommandType	No	RowSet command to execute	CommandType.NONE (default) CommandType.GRID CommandType.DETAIL CommandType.UPDATE CommandType.INSERT CommandType.DELETE CommandType.TEMPLATE_ROW CommandType.INSERTED_ROW CommandType.NEW_KEY

SQLRowSet Members

Member Name	Type	Required	Use	Values
rowsetName	String	No	Name of RowSet object	Default: empty string
rowsetSchema	String	No	XML schema name	Default: empty string

Example

The sample code described in this section is intended to be included in a JDBC control file. For information about such files, see http://beehive.apache.org.

Include the following code in your control file:

```
@SQLRowSet(commandType = com.bea.control.JdbcControl.CommandType.NONE,
rowsetName="aRowSet")
```

```
@JdbcControl.SQL(statement="SELECT * FROM WEBLOGIC.VISITORS_CONTROL")
public RowSet getRowSet() throws SQLException;
```

Suppose your control file is named `JdbcControlSample.jcx`. An instantiable Java class named `JdbcControlSample.class` will be generated, containing a method named `getRowSet()` that returns a RowSet object populated with the results of executing the SQL statement shown in the sample code.

You could use this functionality as follows:

```
@Control() protected JdbcControlSample myCtl;
try {
    RowSet rs = myCtl.getRowSet();
    . . .
} catch (Exception e) {
    . . .
}
```

Usage Restrictions

The following rules govern the use of this annotation:

- If the annotation for a method specifies CommandType.INSERT, CommandType.UPDATE, or CommandType.DELETE, then the method must have a RowSet as its first parameter.
- If the annotation for a method specifies CommandType.INSERT, then the RowSet must contain at least one inserted row. Otherwise, the method call will throw a ControlException.
- If the annotation for a method specifies CommandType.INSERT, then the RowSet must not contain any deleted or updated rows. Otherwise, the method call will throw a ControlException.
- If the annotation for a method specifies CommandType.DELETE, then the RowSet must contain at least one deleted row. Otherwise, the method call will throw a ControlException.
- If the annotation for a method specifies CommandType.DELETE, then the RowSet must not contain any inserted or updated rows. Otherwise, the method call will throw a ControlException.
- If the annotation for a method specifies CommandType.UPDATE, then the RowSet must contain at least one updated row. Otherwise, the method call will throw a ControlException.
- If the annotation for a method specifies CommandType.UPDATE, then the RowSet must not contain any inserted or deleted rows. Otherwise, the method call will throw a ControlException.
- If the annotation for a method specifies CommandType.TEMPLATE_ROW, then the RowSet must not have been assigned a name by a previous call to the JDBC control. Otherwise, the method call will throw an exception. In addition, the method must return a RowSet object.

CHAPTER 5

Beehive Tutorial

This tutorial will show you how to create and deploy a Beehive application on WebLogic Server. The tutorial shows you how to build the three basic Beehive components:

- Apage flow web application
- AJava control
- AJava web service

Note: If this is your second time going through the tutorial, be aware that if you change the location of the application source files you will not be able to deploy to the same server domain. To avoid this problem either (1) create a new server domain from scratch (= the first step of the tutorial) or, if you want to skip this step, (2) undeploy the application from the server first. To undeploy, run the `undeploy` Ant target.

Create a Beehive-enabled Server Domain

In this step you will create a new server domain that can support Beehive applications.

1. Start the domain configuration wizard: Start > All Programs > BEA Products > Tools > Configuration Wizard.
2. Click **Next** on each page without changing any values, except for the second page (named **Select Domain Source**) and the third page (named **Configure Administrator Username and Password**).

   On the second page, place a check in the **Apache Beehive** checkbox before clicking the **Next** button.

On the third page, in the **User password** and **Confirm user password** fields, enter `weblogic`.

At the conclusion of the wizard, a domain named **base_domain** is created at `<BEA_HOME>\user_projects\domains\base_domain`.

3. Start a server instance in the domain by running

   ```
   <BEA_HOME>\user_projects\domains\base_domain\startWebLogic.cmd
   ```

   on Windows or

   ```
   <BEA_HOME>/user_projects/domains/base_domain/startWebLogic.sh
   ```

   on UNIX.

4. Open a new command window. Set the development environment in the command window by running

   ```
   <BEA_HOME>\user_projects\domains\base_domain\bin\setDomainEnv.cmd
   ```

   on Windows or

   ```
   <BEA_HOME>/user_projects/domains/base_domain/bin/setDomainEnv.sh
   ```

   on UNIX.

Create a New J2EE Application

In this step you will create the basic directory structure, and build configuration files for your application.

1. Create the following directory structure:

   ```
   beehive_tutorial
     src
       APP-INF
         lib
         classes
       META-INF
   ```

2. Create the following Ant build file at `beehive_tutorial\src\build.xml`. Note that the properties user/password have the values weblogic/weblogic; these values are designed to match the Beehive-enabled server domain created in "Create a Beehive-enabled Server Domain" on page 5-1.

   ```
   <?xml version="1.0" encoding="UTF-8" ?>
   <project default="build" basedir=".">

   <property environment="env"/>
   ```

```
<!-- provide overrides -->
<property file="build.properties"/>

<property name="src.dir" value="${basedir}"/>
<property name="dest.dir" value="${basedir}/../build"/>
<property name="dist.dir" value="${basedir}/../dist"/>

<property name="app.name" value="beehive_tutorial"/>
<property name="ear.path" value="${dist.dir}/${app.name}.ear"/>
<property name="tmp.dir" value="${java.io.tmpdir}"/>
<property name="weblogic.home" value="${env.WL_HOME}"/>

<property name="user" value="weblogic"/>
<property name="password" value="weblogic"/>

<fail unless="weblogic.home" message="WL_HOME not set in environment"/>

<property name="beehive.home" value="${weblogic.home}/beehive"/>

<!-- beehive-imports defines dependency paths that are required by
beehive-tools.xml -->
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-imports.xml"/>
<!-- defines macros for build-schemas, build-controls, build-pageflows -->
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-tools.xml"/>
<!-- defines macros for build-webapp -->
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-buildmodules.xml"
/>

<taskdef name="libclasspath"
classname="weblogic.ant.taskdefs.build.LibClasspathTask"/>

<target name="init.app.libs">
   <libclasspath basedir="${src.dir}" tmpdir="${tmp.dir}"
property="app.lib.classpath">
      <librarydir dir="${weblogic.home}/common/deployable-libraries/" />
   </libclasspath>
   <echo message="app.lib.claspath is ${app.lib.classpath}" level="info"/>
</target>

<target name="init.dirs">
   <mkdir dir="${dest.dir}/APP-INF/classes"/>
   <mkdir dir="${dest.dir}/APP-INF/lib"/>
   <mkdir dir="${dist.dir}"/>
</target>

<target name="init" depends="init.app.libs,init.dirs">
   <path id="app.classpath">
      <pathelement location="${src.dir}/APP-INF/classes"/>
      <pathelement location="${dest.dir}/APP-INF/classes"/>
      <pathelement path="${app.lib.classpath}"/>
      <fileset dir="${src.dir}/APP-INF/lib">
         <include name="**/*.jar"/>
      </fileset>
      <fileset dir="${dest.dir}/APP-INF/lib">
         <include name="**/*.jar"/>
      </fileset>
      <fileset
dir="${beehive.home}/apache-beehive-incubating-1.0m1/lib/netui">
          <include name="**/*.jar"/>
          <exclude name="**/beehive-netui-compiler.jar"/>
      </fileset>
   </path>

   <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"/>
</target>

<target name="build" depends="compile,appc"/>

<target name="compile" depends="init"/>

<target name="clean" depends="init.dirs,clean.dest,clean.dist"/>

<target name="clean.dest">
   <echo message="deleting dest.dir:${dest.dir}"/>
   <delete includeemptydirs="true" >
      <fileset dir="${dest.dir}" excludes=".beabuild.txt" includes="**/*" />
   </delete>
</target>

<target name="clean.dist">
   <echo message="deleting dist.dir:${dist.dir}"/>
   <delete includeemptydirs="true" >
      <fileset dir="${dist.dir}" includes="**/*" />
   </delete>
</target>

<target name="appc" depends="init" >
   <wlappc source="${dest.dir}"
librarydir="${weblogic.home}/common/deployable-libraries/"/>
</target>

<target name="pkg.exploded">
   <antcall target="clean.dist"></antcall>
   <wlpackage toDir="${dist.dir}" srcdir="${src.dir}" destdir="${dest.dir}"
/>
</target>

<target name="deploy.exploded" >
   <wldeploy user="${user}" password="${password}" action="deploy"
name="${app.name}" source="${dist.dir}"/>
</target>

<target name="deploy" >
   <wldeploy user="${user}" password="${password}" action="deploy"
name="${app.name}" source="${dest.dir}"/>
</target>

<target name="redeploy">
   <wldeploy user="${user}" password="${password}" action="redeploy"
name="${app.name}"/>
</target>

<!-- This target is useful if you want to move the location of the
tutorial source files. Undeploy the app from the server, move the
source files, and then deploy the app again to the server.-->
<target name="undeploy">
```

```
   <wldeploy user="${user}" password="${password}" action="undeploy"
name="${app.name}"/>
</target>

</project>
```

3. Create the following configuration file at `beehive_tutorial\src\META-INF\application.xml`:

```
<application xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.4">
   <display-name>beehive_tutorial</display-name>
</application>
```

4. Create the following configuration file at `beehive_tutorial\src\META-INF\weblogic-application.xml`. (This configuration file informs WebLogic Server that your application uses Beehive functionality, so that the appropriate libraries, in the form of JAR files, are made available at build-time and run-time.)

```
<weblogic-application xmlns="http://www.bea.com/ns/weblogic/90">
   <library-ref>
      <library-name>weblogic-beehive-1.0</library-name>
   </library-ref>
</weblogic-application>
```

Create a New Page Flow Web Application

In this step you will create a page flow web application. Page flows form the user interface for Beehive applications, allowing users to interact with your application through JSPs.

1. Copy the folder

   <BEA_HOME>`\weblogic90\beehive\apache-beehive-incubating-1.0m1\samples\netui-blank`

   into `beehive_tutorial\src` (netui-blank is a template web application).

2. Rename the folder `netui-blank` to `myWebApp`. Before proceeding, ensure that the following directory exists:

```
beehive_tutorial
   src
      myWebApp
         resources
         WEB-INF
         Controller.java
         index.jsp
```

3. Edit `beehive_tutorial\src\META-INF\application.xml` so it appears as follows. Code to add appears in **bold**.

```
<application xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.4">
   <display-name>beehive_tutorial</display-name>
   <module>
      <web>
         <web-uri>myWebApp</web-uri>
         <context-root>/myWebApp</context-root>
      </web>
   </module>
</application>
```

4. Edit `beehive_tutorial\src\myWebApp\index.jsp` so it appears as follows. Code to edit appears in **bold**.

```
<%@ page language="java" contentType="text/html;charset=UTF-8"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-databinding-1.0"
prefix="netui-data"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-html-1.0"
prefix="netui"%>
<%@ taglib uri="http://beehive.apache.org/netui/tags-template-1.0"
prefix="netui-template"%>
<netui:html>
   <head>
      <title>Beehive Tutorial Test Page</title>
      <netui:base/>
   </head>
   <netui:body>
      <h3>
         Beehive Tutorial Test Page
      </h3>
      <p>
         Welcome to the Beehive Tutorial!
      </p>
   </netui:body>
</netui:html>
```

5. Create the following configuration file at `beehive_tutorial\src\myWebApp\WEB-INF\weblogic.xml`.

```
<weblogic-web-app
  xmlns="http://www.bea.com/ns/weblogic/90"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.bea.com/ns/weblogic/90
  http://www.bea.com/ns/weblogic/90/weblogic-web-app.xsd">
    <library-ref>
```

```
        <library-name>beehive-netui-1.0</library-name>
    </library-ref>
    <library-ref>
        <library-name>struts-1.1</library-name>
    </library-ref>
    <library-ref>
        <library-name>jstl-1.1</library-name>
    </library-ref>
</weblogic-web-app>
```

6. Edit `beehive_tutorial\src\build.xml` so it appears as follows. Code to add appears in **bold**.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project default="build" basedir=".">

    ...
        <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
excludes="myWebApp"/>
    </target>

. . .

    <target name="compile" depends="init,compile.myWebApp"/>
. . .
    <target name="redeploy">
        <wldeploy user="${user}" password="${password}" action="redeploy"
name="${app.name}"/>
    </target>

    <target name="compile.myWebApp" depends="init" >
        <libclasspath basedir="${src.dir}/myWebApp" tmpdir="${tmp.dir}"
property="myWebApp.lib.classpath">
            <librarydir
dir="${weblogic.home}/common/deployable-libraries/" />
        </libclasspath>
        <path id="myWebApp.build.classpath">
            <path refid="app.classpath"/>
            <path path="${myWebApp.lib.classpath}"/>
        </path>

        <build-webapp
          webapp.src.dir="${src.dir}/myWebApp"
          webapp.build.dir="${dest.dir}/myWebApp"
          app.build.classpath="myWebApp.build.classpath"/>
    </target>

</project>
```

Note: You may be wondering, "Why are we excluding myWebApp from the `<wlcompile>` step?" The answer is: for our Beehive app, `<wlcompile>` is intended to facilitate split-source deployment, *not* compilation of Java source files. For our purposes, the `<wlcompile>` element is present in order to generate the `.beabuild` file, which enables split source build/deployment of the application. (Split source build/deployment is an especially clean way to build and deploy an application. It allows the strict separation of the original source files and the generated build-artifacts.) Of course, `<wlcompile>` also searches for all Java files in the application and compiles them. But `<wlcompile>` will fail if it encounters any Beehive-specific classes. By excluding the `myWebApp` directory from the search, we prevent a compile error that would have been caused if `<wlcompile>` had encountered the Beehive-specific classes in `myWebApp`.

7. In the command window, navigate to `beehive_tutorial\src\`. (Make sure you use the same command window where you ran `setDomainEnv.cmd` or `setDomainEnv.sh`.)
8. To compile the application, run the following Ant command:

   ```
   ant clean build pkg.exploded
   ```

9. To deploy the application, run the following Ant command:

   ```
   ant deploy.exploded
   ```

10. To test the application, visit the following URL in a browser:

    http://localhost:7001/myWebApp/begin.do

You should see the `index.jsp` page:

```
Welcome to the Beehive Tutorial!
```

Add a Java Control

A Beehive Java Control allow you to encapsulate functionality in your applications. Typically Java Controls are used to perform the following functions:

- Business logic
- Accessing backend resources, for example, databases
- Accessing external resources, for example, web services

Beehive ships with a number of "system controls," that is, controls that are designed with some particular functionality in mind: these system controls include database, JMS, EJB, and Web Service controls.

In the following step you will add a custom control with a very simple function: it returns the message “Hello, World!”

Inside the directory `beehive_tutorial\src\`, create a directory named `controls`.

1. Inside the directory `beehive_tutorial\src\controls`, create a directory named `pkg`.

   Before proceeding, confirm that the following directory structure exists:

   ```
      beehive_tutorial
         src
            controls
               pkg
   ```

2. In the folder `beehive_tutorial\src\controls\pkg\`, create a file named `HelloWorld.java`. Edit `HelloWorld.java` so it appears as follows:

   ```
   package pkg;

   import org.apache.beehive.controls.api.bean.*;

   @ControlInterface
   public interface HelloWorld
   {
         String hello();
   }
   ```

3. In the folder `beehive_tutorial\src\controls\pkg\`, create a file named `HelloWorldImpl.java`. Edit `HelloWorldImpl.java` so it appears as follows:

   ```
   package pkg;

   import org.apache.beehive.controls.api.bean.*;

   @ControlImplementation(isTransient=true)
   public class HelloWorldImpl implements HelloWorld
   {
         public String hello()
         {
               return "hello!";
         }
   }
   ```

4. Edit `beehive_tutorial\src\myWebApp\Controller.java` so it appears as follows. Code to add and edit appears in **bold**.

   ```
   import javax.servlet.http.HttpSession;

   import org.apache.beehive.netui.pageflow.Forward;
   import org.apache.beehive.netui.pageflow.PageFlowController;
   import org.apache.beehive.netui.pageflow.annotations.Jpf;
   ```

```
import org.apache.beehive.controls.api.bean.Control;
import pkg.HelloWorld;

@Jpf.Controller(
    simpleActions={
        @Jpf.SimpleAction(name="begin_old", path="index.jsp")
    },
    sharedFlowRefs={
        @Jpf.SharedFlowRef(name="shared", type=shared.SharedFlow.class)
    }
)
public class Controller
    extends PageFlowController
{
    @Jpf.SharedFlowField(name="shared")
    private shared.SharedFlow sharedFlow;

    @Control
    private HelloWorld _helloControl;

    @Jpf.Action(
      forwards={
        @Jpf.Forward(name="success", path="index.jsp")
      }
    )
    protected Forward begin() throws Exception
    {
        Forward f = new Forward("success");
        f.addActionOutput("helloMessage", _helloControl.hello());
        return f;
    }

    /**
     * Callback that is invoked when this controller instance is created.
     */
    protected void onCreate()
    {
    }

    /**
     * Callback that is invoked when this controller instance is
destroyed.
     */
    protected void onDestroy(HttpSession session)
    {
    }
}
```

5. Edit the file `beehive_tutorial/src/myWebApp/index.jsp` so it appears as follows.

```
<netui:html>
   <head>
      <title>Beehive Tutorial Test Page</title>
   <netui:base/>
   </head>
   <netui:body>
      <h3>
         Beehive Tutorial Test Page
      </h3>
      <p>
         Welcome to the Beehive Tutorial!
      </p>
      <p>
         Response from the hello() method on the HelloWorld Control:
         <netui:span style="color:#FF0000"
value="${pageInput.helloMessage}"/>
      </p>
   </netui:body>
</netui:html>
```

6. Edit **beehive_tutorial/src/build.xml** so it appears as follows. Code to add appears in **bold**.

```
<?xml version="1.0" encoding="UTF-8" ?>

<project default="build" basedir=".">

...

   <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
excludes="myWebApp,controls"/>

. . .

<target name="compile"
depends="init,compile.helloWorldControl,compile.myWebApp"/>

. . .

<target name="compile.helloWorldControl" depends="init">
   <build-controls
     srcdir="${src.dir}"
     destDir="${dest.dir}/APP-INF/classes"
     tempdir="${tmp.dir}/${app.name}/controls/build-controls"
     classpathRef="app.classpath"
   />
   <!-- clean up duplicate myWebApp <apt> output -->
   <delete>
      <fileset dir="${dest.dir}/APP-INF/classes"
excludes="pkg/**/*.class"/>
   </delete>
</target>
```

```
</project>
```

7. To compile the application, run the following Ant command. Make sure that you use the same command window in which you ran `setDomainEnv.cmd` or `setDomainEnv.sh`.

   Note: In the following build process, you will see some build warnings. These warnings are harmless and do not affect the final product of compilation.

   ```
   ant clean build pkg.exploded
   ```

8. To deploy the application, run the following Ant command:

   ```
   ant deploy.exploded
   ```

9. To test the application, visit the following URL in a browser:

   http://localhost:7001/myWebApp/begin.do

   You should see the following JSP page.

   ```
   Beehive Tutorial Test Page

   Welcome to the Beehive Tutorial!

   Response from the hello() method on the HelloWorld Control: hello!
   ```

Add a Java Web Service

Web services provide an XML-based interface for your application. In this step you will create an XML-based way to communicate with your application.

1. In the directory `beehive_tutorial\src\` create a folder named `services`.
2. In the directory `beehive_tutorial\src\services\` create a folder named `pkg`.
3. In the directory `beehive_tutorial\src\services\pkg\` create a file named `HelloWorldService.java`.
4. Edit `HelloWorldService.java` so it appears as follows:

   ```
   package pkg;

   import javax.jws.*;
   import org.apache.beehive.controls.api.bean.Control;
   import pkg.HelloWorld;

   @WebService
   public class HelloWorldService
   {
       @Control
       private HelloWorld _helloControl;
   ```

```
    @WebMethod()
    public String hello()
    {
        String message = _helloControl.hello();
        return message;
    }
}
```

5. Edit `beehive_tutorial\src\build.xml` so it appears as follows. Code to add appears in **bold**.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project default="build" basedir=".">

...
<!-- Merge the application.xml file with the auto-generated
       application.xml file created by the web service compilation
       process. -->
<copy todir="${dest.dir}/META-INF">
   <fileset dir="${src.dir}/META-INF"/>
</copy>

<wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
excludes="myWebApp,controls,services"/>

...

<target name="compile"
depends="init,compile.helloWorldControl,compile.helloWorldService,compi
le.myWebApp"/>

...

<target name="compile.helloWorldService" depends="init" >

   <taskdef name="jwsc"
classname="weblogic.wsee.tools.anttasks.JwscTask"/>

   <jwsc
     verbose="true"
     tempdir="${tmp.dir}"
     destdir="${dest.dir}"
     keepgenerated="true"
     srcdir="${basedir}/services">
       <jws file="pkg/HelloWorldService.java" name="HelloWorldService"
explode="true"/>
       <classpath>
          <path refid="app.classpath"/>
       </classpath>
   </jwsc>
</target>

</project>
```

6. To build the application, run the following Ant command. Make sure that you use the same command window in which you ran `setDomainEnv.cmd` or `setDomainEnv.sh`:

   ```
   ant clean build pkg.exploded
   ```

7. To deploy the application, run the following Ant command:

   ```
   ant deploy.exploded
   ```

8. To test the web service, visit the following URL in a browser:

   http://localhost:7001/HelloWorldService/HelloWorldService

   To see the web service's WSDL, visit the following URL:

   http://localhost:7001/HelloWorldService/HelloWorldService?WSDL