BEA WebLogic Server™®

Beehive Integration in BEA WebLogic Server

Version 10.0
Document Revised: April 26, 2007

About This Document

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

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 Overrides"

Explains annotation overrides and provides code examples.

- 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 9.2.

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.

Table 0-1

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`*

Table 0-1

Convention	Item
`{ }`	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]`
`|`	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.

Beehive Applications

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

The following sectons provide an introduction to Beehive applications:

- "What Is a Beehive Application?" on page 1-1
- "Apache Beehive Documentation" on page 1-4
- "Running the WebLogic Server/Beehive Integration Samples" on page 1-4
- "Upgrade Paths" on page 1-7

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

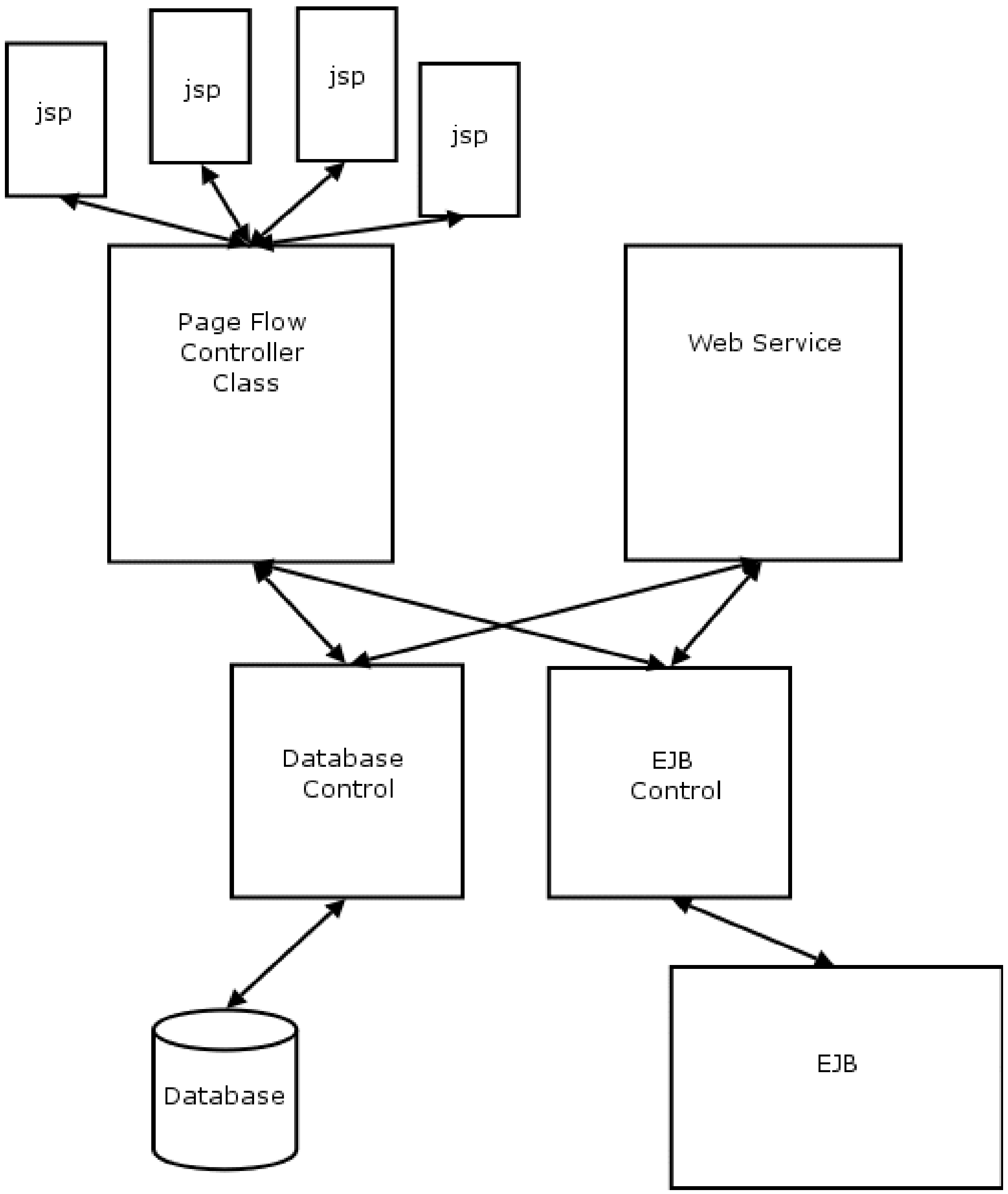
  A lightweight component framework that helps programmers encapsulate application logic and leverage metadata annotations into their programming model. Developers can create **custom controls** or use the pre-built **system controls**. The Beehive system controls provide access to these common J2EE components: 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 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-svn-snapshot\docs\docs`

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

Note: The version of Beehive shipped with WebLogic Server 9.2 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.

Upgrade Paths

Upgrading from 9.0

If you have Beehive applications that ran under WebLogic Server 9.0, you will have to make certain changes to them so that they will run under WebLogic Server 9.2. This section discusses those changes.

Beehive Directory Location

In the case of WebLogic Server 9.2, Beehive samples, ANT scripts, documentation, library JAR files, and other support files are installed on your system under the directory

```
<WEBLOGIC_HOME>\beehive\apache-beehive-svn-snapshot
```

This directory had a different name in 9.0 installations, usually

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

Check all your Ant scripts for references to this directory path and update any references to the old path.

Application Project Model

The project model for Beehive samples shipped with WebLogic Server 9.0, the source-in model, had this form:

```
module-name/
    WEB-INF/
        src/
```

The Beehive samples shipped with WebLogic Server 9.2 are structured according to a different project model, the source-out model. They have this form:

```
module-name/
    web/
        WEB-INF/
    src/
```

If you have a Beehive application that was based on one of the source-in samples, you should still be able to use your current build script, except for possible changes in path names.

However, if you change such an application to conform to the newer source-out model, you may encounter problems when running the Ant macros defined in the import files in

`<WEBLOGIC_HOME>\beehive\weblogic-beehive\ant\`

To avoid these problems, it will be necessary to copy some of the files in your application to new directories. This should be done as part of your `build.xml` build script. For guidance in writing the copy instructions appropriately, please see the build script in Chapter 5, "Beehive Tutorial," in particular, step 6 under the section titled "Create a New Page Flow Web Application" on page 5-6.

Upgrading from 9.1

If you have Beehive applications that ran under WebLogic Server 9.1, you will have to make certain changes to them so that they will run under WebLogic Server 9.2. This section discusses those changes.

Beehive Directory Location

In the case of WebLogic Server 9.2, Beehive samples, ANT scripts, documentation, library JAR files, and other support files are installed on your system under the directory

`<WEBLOGIC_HOME>\beehive\apache-beehive-svn-snapshot`

This directory had a different name in 9.1 installations, usually

`<WEBLOGIC_HOME>\apache-beehive-1.0-final`

Check all your Ant scripts for references to this directory path and update any references to the old path.

Ant Macros

The Ant macros defined in the import files in

`<WEBLOGIC_HOME>\beehive\weblogic-beehive\ant\`

make certain assumptions about the structure of your application. See "Application Project Model" on page 1-8 for an explanation of the two application models involved.

If your application conforms to the source-out model, it will be necessary to copy some of the files in your application to new directories. This should be done as part of your `build.xml` build script. For guidance in writing the copy instructions appropriately, please see the build script in Chapter 5, "Beehive Tutorial," in particular, step 6 under the section titled "Create a New Page Flow Web Application" on page 5-6.

Beehive System Controls

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

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.

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 3-9.

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 3-9.

EJB and JMS Controls

The EJB and JMS controls shipped with WebLogic Server 9.2 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:

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

and you must provide role/principal mappings using <security-role-assignment> in 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 2-3).

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

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

The following sections provide information on how to build Beehive applications:

- "Build Resources" on page 3-1
- "Ant Tasks" on page 3-3
- "Building Beehive Applications" on page 3-9
- "Template Build File" on page 3-12

Build Resources

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

```
   <WEBLOGIC_HOME>\beehive\weblogic-beehive\ant\weblogic-beehive-buildmodu
les.xml
```

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

`weblogic-beehive-buildmodules.xml` contains build macros (`<macrodef>` elements) for building a web app module and other module-leve artifacts. Use this file for easiest building.

`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. Use this file instead of `weblogic-beehive-buildmodules.xml` if you need more flexible control over project structure.

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

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

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

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

   <import file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-bui
ldmodules.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 3-12. 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*.

Note: These libraries allow you to use the Beehive JARs without copying them into your application or putting them on the systm CLASSPATH. For an overview and detailed information about shared libraries, see the WebLogic Server document Creating Shared Java EE Libraries and Optional Packages

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

Name	Definition
`tempdir`	Required. A temporary directory containing the generated sources and classes
`classpathref`	Required. The classpath used for compilation

Error during Build

Because of a known bug, you may encounter this error message when using the `build-control-module` task:

```
<drive>:\bea\wlserver_10.0\beehive\weblogic-beehive\ant\weblogic-beehive-b
uildmodules.xml:140: taskdef class
org.apache.beehive.controls.runtime.packaging.ControlJarTask cannot be
found
```

To correct this, you must modify your Ant build file.

After the line in the file where `weblogic-beehive-imports.xml` and `weblogic-beehive-tools.xml` are imported, and before the line where `weblogic-beehive-buildmodules.xml` is imported, insert the following line:

```
<property name="controls.jars.prop" refid="controls.dependency.path"/>
```

That section of your Ant build file should then look like this:

```
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-imports.xml"/>
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-tools.xml"/>
<property name="controls.jars.prop" refid="controls.dependency.path"/>
<import
file="${beehive.home}/weblogic-beehive/ant/weblogic-beehive-buildmodules.x
ml"/>
```

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

Note: As was mentioned above, you should only use this file if you want the added flexibility it provides. Otherwise, use `weblogic-beehive-buildmodules.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 3-11.

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

Name	Definition
`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 3-9.

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 3-12. 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 3-9 and "Building JMS and EJB Controls" on page 3-11.

For a detailed example of building a page flow web application, see "Create a New Page Flow Web Application" on page 5-6.

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="${src.dir}/services/MyService.wsdl"
     wsdlname="MyService.wsdl"
     wsdlservicename="MyService"
     tempdir="c:/temp/services"
     destdir="${dest.dir}/APP-INF/lib"
     typefamily="tylar"
   />
   ```

   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="${src.dir}/services/MyService.wsdl"
          wsdlservicename="MyService"
          servicecontrolname="MyServiceControl"
          destdir="${dest.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 2-3.

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, any annotations must be processed using apt.

```
<taskdef name="apt"

classname="org.apache.beehive.controls.runtime.generator.AptTask"
            classpathref="apt.task.classpath"
            onerror="fail"/>

        <apt srcdir="${src.dir}"
            destdir="${build.dir}/module"
            gendir="${build.dir}/aptgen"
            classpathref="classpath"
            srcExtensions="*.jws"
            />
```

5. After running apt, you build the initial version of the jws ejb jar. You do this by running jwsc against the code generated by apt, which is stored in the directory specified by the `gendir` attribute in the preceding step.

```
<taskdef name="jwsc"
            classpath="${weblogic.home}/server/lib/weblogic.jar"
            classname="weblogic.wsee.tools.anttasks.JwscTask"
            onerror="report" />

        <jwsc srcdir="${build.dir}/aptgen"
          destdir="${ent.app.dir}"
          classpathref="classpath"
          debug="on" keepGenerated="true">
```

```
              <jws file="hellotest/HelloTest.java"/>
            </jwsc>
```

6. Now the client must be assembled, which creates appropriate bindings between the client and the web service control. You do this with the `<assemble-controls>` task. 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 6 in a Java EE application context. Inside a Java EE 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.EJBAssemb
lyContext$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 4-6 in "Building Web Service Controls" on page 3-9: you must run assembly on the control and then compile the results of the assembly.

Detailed Example

For a detailed example of a build script that creates a web service application that uses a service control, see Appendix C, "Build Script Sample - Create Web Service That Uses a Service Control,".

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">
     <!-- Builds classpath for library modules referenced in
weblogic-applications.xml. -->
     <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-svn-snapshot/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 Overrides

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

Beehive's control architecture provides three techniques for setting the values of resource access attributes:

- Programmatic (setters)
- Externally configured property value or value override
- Metada annotations

This list shows the order of precedence, from highest to lowest. For example, an externally configured value can override the value set for that property by a metada annotation but not the value set by a setter method. This overriding of the value can happen at deployment time or run time.

WebLogic Server annotation overrides, the subject of this chapter, refers to the second of the above techniques. That is, annotation overrides is a technique for overriding, at run time, property values that were originally set by means of metadata annotations.

For background information on Beehive controls architecture and how it includes the above material, please see the documentation on the Apache Beehive site.

Process Flow

In order to use annotation overrides, you must perform specific actions at specific stages while creating a project. These stages and actions are summarized here and are then covered in more detail in the subsequent sections of this chapter.

- Code time

  In your code, you designate an annotated property value as externally configurable by annotating it with the following special meta attribute:

  ```
  @PropertySet(externalConfig=true)
  ```

- Build time

  You include various Ant macros in your `build.xml` file in order to create certain required artifacts — annotation manifests, annotation overrides files, and a deployment plan. Annotation manifests and annotation overrides files are explained below. For information about WebLogic Server deployment plans, see Understanding WebLogic Server Deployment Plans in *Deploying Applications to WebLogic Server*.

- Deployment time

  When the application is being deployed, WebLogic Server requires information that must be contained in a file named `weblogic-extension.xml`, stored in a specific directory.

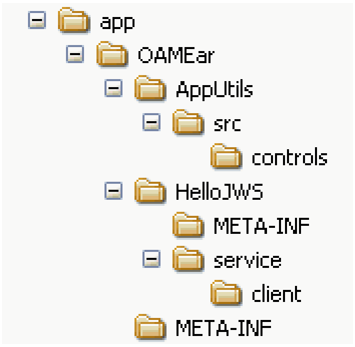
- Run time

  To change a property value dynamically while the application is running, you first use a Python script to assign the new value, and then you use a tool named `weblogic.Deployer` to make the application pick up and start using the new value.

Sample Code

For this chapter, we will use an application containing one module, OAMEar.

The application's directory structure looks like this:



In the following sections, we will be looking at these application files:

File	Location	Description
MyControl.java	AppUtils/src/controls	Public control interface
MyControlImpl.java	AppUtils/src/controls	Control implementation of MyControl.java
MyAnnotation.java	AppUtils/src/controls	Externally configurable annotation
MyJcx.java	AppUtils/src/controls	Control extension
HelloService.java	HelloJWS/service	Control client. Consumes control MyControlImpl.java.

In the following sections, for clarity, only part of the code of these files is displayed. The complete code for all the files mentioned in the table above can be found in Appendix A, "Code Samples for Annotation Overrides,".

Code Time

MyControl.java

```
...

@ControlInterface
public interface MyControl
```

```
{
  public String getAnnotationValue();
}
```

MyControlImp.java

```
...

@ControlImplementation
public class MyControlImpl implements MyControl, Extensible, Serializable
{
    @Context
    ControlBeanContext ctx;

    public String getAnnotationValue() {

        // This is a key part. The control framework will make sure
        // that the precedence is followed when looking up annotation
        // values for MyAnnotation
        MyAnnotation myAnnotation =
            ctx.getControlPropertySet(MyAnnotation.class);

        return myAnnotation.value();
    }

    public Object invoke(Method method, Object[] args) throws Throwable
    {
        // Control Extension not defined for simplicity
        return null;
    }
}
```

MyAnnotation.java

```
...

@PropertySet(externalConfig=true)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation
{
    String value() default "DEFAULT";
}
```

MyJcx.java

```
...

@PropertySet(externalConfig=true)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation
{
    String value() default "DEFAULT";
}
```

HelloService.java

```
...

@WebService(name="Hello")
@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)
@WLHttpTransport(contextPath="helloJWS", serviceUri="HelloService")
public class HelloService
{
    @Control()
    @MyAnnotation("Overriden@Field")
    protected MyJcxBean _jcxBean;

    @WebMethod
    public String sayHello()
    {
        return _jcxBean.getAnnotationValue();
    }
}
```

Build Time

Generate the Annotation Manifests

As part of your build process, you must generate artifacts called annotation manifests. These files, named `annotation-manifest.xml`, contain information required for annotation overrides.

There is one annotation manifest for each module in your application and one for the application root. In our example application, OAMEar, there is only module, the service, so our build script will generate two annotation manifests.

To generate the required set of `annotation-manifest.xml` files for your application, include a call to the `generate-manifests` Ant macro in your build script. Be sure that your build script performs all the compilation steps before it calls this macro.

The `generate-manifests` macro is defined in the file

`<weblogic.home>/beehive/weblogic-beehive/ant/weblogic-beehive-imports.xml`

which must be imported into your `build.xml` file so that it can use the macro.

The macro takes the following parameters:

- **destdir**

  Required.

  The absolute path where you want the task to generate the manifest file. Normally this would be the directory where the application or the module resides. The manifest file is created in the subdirectory `META-INF` under the destdir — that is, the macro creates `<destdir>/META-INF/annotation-manifest.xml`.

- **searchclasspathref**

  Required.

  The search path for the task to find overrideable annotations. Include only jars, zip files or class directories that need to be processed for annotation overrides. Examples of `searchClasspath` entries are: classes directory as result of apt, `WEB-INF/classes`, and `WEB-INF/lib/myExtensibleControl.jar`. `searchClasspath` can be a nested `<searchClasspath>` path structure, `searchClasspath` defined as a attribute, or a reference to a path that is defined elsewhere using `searchClasspathRef.`

- **classpathref**

  Required.

The classpath that allow the build-manifest task to load the necessary classes to generate and merge the manifest files. The classpath usually includes a few jars from `${wl.home}/server/lib`, such as `weblogic.jar` and `wlsbeehive.jar`. Again, the path can be defined as a nested `<classpath>` path structure, classpath defined as a attribute, or a reference to a path that is defined elsewhere using `classpathRef`.

- **tempdir**

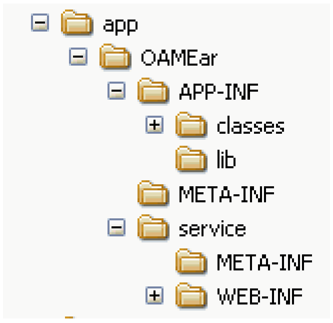
  Optional.

  Tells the build-manifest task which directory to use for temporary storage. Default value is `.staging` in the current directory.

For example, in our sample code, the call to the macro looks like this:

```
<generate-manifests destdir="${current.dir}"
                        searchclasspathref="app.inf.classes"
                        classpathref="app.build.classpath"
tempdir="${current.dir}/.staging"/>
```

After this macro call, our application includes the following directory structure:



The directory `OAMEar/META-INF` contains the following files:

```
application.xml
annotation-manifest.xml
weblogic-extension.xml
weblogic-application.xml
```

This `annotation-manifest.xml` file is the annotation manifest for the application root. It contains this XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<annotation-manifest
xmlns="http://www.bea.com/2004/03/wlw/external-config/">
  <annotated-class>
    <annotated-class-name>controls.MyJcx</annotated-class-name>
    <component-type>Control Extension (JCX)</component-type>
    <annotation>
      <annotation-class-name>controls.MyAnnotation</annotation-class-name>
      <member>
        <member-name>value</member-name>
        <member-value>OnClassDecl</member-value>
      </member>
    </annotation>
  </annotated-class>
  <annotation-definition>
    <annotation-class-name>controls.MyAnnotation</annotation-class-name>
    <allowed-on-declaration>false</allowed-on-declaration>
    <member-definition>
      <member-name>value</member-name>
      <is-array>false</is-array>
      <is-required>false</is-required>
      <simple-type-definition>
        <base-type>STRING</base-type>
        <default-value>DEFAULT</default-value>
      </simple-type-definition>
    </member-definition>
  </annotation-definition>
  <version>1.0.0</version>
</annotation-manifest>
```

The directory `service/META-INF` contains the following files:

```
annotation-manifest.xml
MANIFEST.MF
```

This `annotation-manifest.xml` file is the annotation manifest for the service module. It contains this XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<annotation-manifest
xmlns="http://www.bea.com/2004/03/wlw/external-config/">
  <annotated-class>
    <annotated-class-name>service.HelloService</annotated-class-name>
    <component-type>Web Service (JWS)</component-type>
    <field>
      <field-name>_jcxBean</field-name>
      <instance-type>controls.MyJcxBean</instance-type>
      <annotation>
      <annotation-class-name>controls.MyAnnotation</annotation-class-name>
        <member>
          <member-name>value</member-name>
          <member-value>Overriden@Field</member-value>
        </member>
      </annotation>
    </field>
  </annotated-class>
  <annotation-definition>
    <annotation-class-name>controls.MyAnnotation</annotation-class-name>
    <allowed-on-declaration>false</allowed-on-declaration>
    <member-definition>
      <member-name>value</member-name>
      <is-array>false</is-array>
      <is-required>false</is-required>
      <simple-type-definition>
        <base-type>STRING</base-type>
        <default-value>DEFAULT</default-value>
      </simple-type-definition>
    </member-definition>
  </annotation-definition>
  <version>1.0.0</version>
</annotation-manifest>
```

Note the bolded code above. This points to the code

```
@Control()
    @MyAnnotation("Overriden@Field")
    protected MyJcxBean _jcxBean;
```

in the service. (See `HelloService.java`, above.)

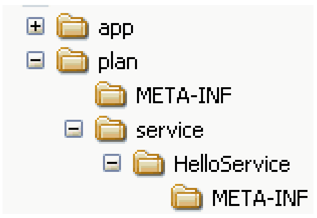
Create A Deployment Plan

Now you must create a deployment plan. You do this using the `weblogic.PlanGenerator` configuration tool. For documentation about the use of this tool, please see the BEA document weblogic.PlanGenerator Command Line Reference.

In our example, to use this tool, we open a command prompt in the parent directory where app resides and issue the following command:

```
java weblogic.PlanGenerator -plan plan/Plan.xml app/OAMEar
```

Our deployment directory now contains this directory structure:



The `directory plan/META-INF` contains a file named `annotation-overrides.xml`. This artifact corresponds to the annotation manifest file in the `app/OAMEar/META-INF` directory. This annotation overrides file contains the following XML:

```
<?xml version='1.0' encoding='UTF-8'?>
<annotation-overrides
xmlns="http://www.bea.com/2004/03/wlw/external-config/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <annotated-class>
    <annotated-class-name>controls.MyJcx</annotated-class-name>
    <component-type>Control Extension (JCX)</component-type>
    <annotation>
      <annotation-class-name>controls.MyAnnotation</annotation-class-name>
      <member>
        <member-name>value</member-name>
        <member-value>OnClassDecl</member-value>
        <cleartext-override-value>Application scope set at
-1868610374!!</cleartext-override-value>
```

```
      </member>
    </annotation>
  </annotated-class>
</annotation-overrides>
```

The directory `plan/HelloService/META-INF` contains a file named `annotation-overrides.xml`. This artifact corresponds to the annotation manifest file in the `app/OAMEar/service/META-INF` directory. This annotation overrides file contains the following XML:

```
<?xml version='1.0' encoding='UTF-8'?>
<annotation-overrides
xmlns="http://www.bea.com/2004/03/wlw/external-config/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <annotated-class>
    <annotated-class-name>service.HelloService</annotated-class-name>
    <component-type>Web Service (JWS)</component-type>
    <field>
      <field-name>_jcxBean</field-name>
      <instance-type>controls.MyJcxBean</instance-type>
      <annotation>
      <annotation-class-name>controls.MyAnnotation</annotation-class-name>
        <member>
          <member-name>value</member-name>
          <member-value>Overriden@Field</member-value>
          <cleartext-override-value>I hereby override this value to
-338576924</cleartext-override-value>
        </member>
      </annotation>
    </field>
  </annotated-class>
</annotation-overrides>
```

Note the bolded code, which corresponds to the bolded code in the corresponding annotation manifest file, above.

The new `plan` directory contains the newly generated deployment plan, `plan.xml`, which contains this segment of code:

```
<module-override>
    <module-name>service/HelloService.war</module-name>
    <module-type>war</module-type>
    <module-descriptor external="false">
      <root-element>weblogic-web-app</root-element>
      <uri>WEB-INF/weblogic.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>web-app</root-element>
      <uri>WEB-INF/web.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>weblogic-webservices</root-element>
      <uri>WEB-INF/weblogic-webservices.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>webservices</root-element>
      <uri>WEB-INF/webservices.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>webservice-policy-ref</root-element>
      <uri>WEB-INF/weblogic-webservices-policy.xml</uri>
    </module-descriptor>
    <module-descriptor external="true">
      <root-element>annotation-overrides</root-element>
      <uri>META-INF/annotation-overrides.xml</uri>
      <hash-code>1133371728402</hash-code>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>annotation-manifest</root-element>
      <uri>META-INF/annotation-manifest.xml</uri>
    </module-descriptor>
  </module-override>
```

Override the Property Values — Annotation Override

Overriding property values set by means of annotation can happen at application deployment time or while the application is running. In both cases, you change the property value set in the appropriate `annotation-overrides.xml` file.We recommend that you not edit those files directly. Rather, you should change the values by running a Python script.

Two Python script files are required, `OverrideUtils.py` and `driver.py`. Both scripts can be found in Appendix B, "Python Scripts for Annotation Overrides,".

In the `driver.py` script, you will find this section of code:

```
# Update a field's override value
moduleName='service/HelloService.war'
key=moduleName+'/service.HelloService/_jcxBean/value'
value='I hereby override this value to %d' % (r.nextInt())
util.applyOverride(moduleName, key, value)
```

(Note that this script assumes that the Hello service was packed as a WAR file.)

This is where you set the new property value. After you have edited the script to change the value, run this script and it will update the property value in the `annotation-overrides.xml` file. In our code examples, the script will reset the value of the property originally set in the code with the @MyAnnotation tag to the string "Application scope set at n!!" where n is a randomly generated integer.

To run this script, issue the following command-line command in the `plan` directory:

```
java weblogic.WLST driver.py
```

Deployment Time

When the application is deployed, the properties will have the values you last set by using the `driver.py` script, as opposed to the values originally set via annotation in the code.

Run Time

You can also change an application's property values dynamically while it is running.

First use the `driver.py` script, as described above, to change the values in the `annotation-overrides.xml` file.

Then use the `weblogic.Deployer` tool to cause the WebLogic Server to perform a hot update, causing the application to pick up the new overriden values while it is running. (For documentation about the use of this tool, please see the BEA document weblogic.Deployer Command Line Reference.)

Issue the following command-line command:

```
java weblogic.Deployer -adminurl t3://localhost:7001 -user username
-password password -name MyApp -update
```

That will trigger a dynamic update to the application and the application will pick up the annotation overrides.

CHAPTER 5

Beehive Tutorial

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

The following sections provide information on how to create and deploy a Beehive application on WebLogic Server

- "Create a Beehive-enabled Server Domain" on page 5-2
- "Create a New Java EE Application" on page 5-2
- "Create a New Page Flow Web Application" on page 5-6
- "Add a Java Control" on page 5-9
- "Add a Java Web Service" on page 5-13

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 Java EE Application

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

1. Underneath **base_domain**, create the following directory structure:

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

2. In the `beehive_tutorial\src` directory, create an Ant build file named `build.xml` and enter the following text in it. 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 5-2.

```
<?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-buildmodule
s.xml"/>
```

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

<!-- Builds classpath based on the libraries defined in
weblogic-application.xml. -->
<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-svn-snapshot/lib/netui">
         <include name="**/*.jar"/>
         <exclude name="**/beehive-netui-compiler.jar"/>
      </fileset>
   </path>
</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. In the `beehive_tutorial\src\META-INF` directory, create a configuration file named `application.xml` and enter the following text in it.

```
<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. In the `beehive_tutorial\src\META-INF` directory, create a configuration file named `weblogic-application.xml` and enter the following text in it. 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-controls-1.0</library-name>
   </library-ref>
   <library-ref>
      <library-name>beehive-controls-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>\wlserver_10.0\beehive\apache-beehive-svn-snapshot\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 structure exists:

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

3. Edit `beehive_tutorial\src\META-INF\application.xml` so it appears as follows. Code to add is shown 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\web\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. In the directory `beehive_tutorial\src\myWebApp\web\WEB-INF`, create a configuration named `weblogic.xml` and enter the following text in it.

```
<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=".">

   ...

   <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>

<copy todir="${src.dir}/myWebApp/">
        <fileset dir="${src.dir}/myWebApp/web/">
            <exclude name="**/*.java"/>
        </fileset>
    </copy>

    <copy todir="${dest.dir}/WEB-INF/classes">
        <fileset dir="${src.dir}/myWebApp/src/">
            <include name="sql/**"/>
            <include name="**/*.properties"/>
            <include name="**/*.xml"/>
        </fileset>
    </copy>

    <copy todir="${src.dir}/myWebApp/WEB-INF/src">
        <fileset dir="${src.dir}/myWebApp/src/">
            <exclude name="**/*.java"/>
        </fileset>
```

```
     </copy>

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

</project>
```

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\web\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\web\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**.

```
. . .

<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**.

   ```
   . . .
   ```

```
<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-svn-snapshot/lib/netui">
      <include name="**/*.jar"/>
      <exclude name="**/beehive-netui-compiler.jar"/>
    </fileset>
  </path>
  <!-- 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>
</target>

...

<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
   ```

Code Samples for Annotation Overrides

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

This appendix contains the complete code for the files used or referred to in Annotation Overrides.

- "MyControl.java" on page A-1
- "MyControlImp.java" on page A-2
- "MyAnnotation.java" on page A-3
- "MyJcx.java" on page A-3
- "HelloService.java" on page A-4

MyControl.java

```
package controls;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import org.apache.beehive.controls.api.bean.AnnotationMemberTypes;
import org.apache.beehive.controls.api.bean.ControlInterface;

import org.apache.beehive.controls.api.bean.AnnotationMemberTypes.Date;
import org.apache.beehive.controls.api.bean.AnnotationMemberTypes.Decimal;
import
org.apache.beehive.controls.api.bean.AnnotationMemberTypes.Optional;

import org.apache.beehive.controls.api.properties.PropertySet;
import weblogic.controls.annotations.RequiresEncryption;

@ControlInterface
public interface MyControl
{
  public String getAnnotationValue();
}
```

MyControlImp.java

```
package controls;

import java.lang.reflect.Method;
import java.io.Serializable;
import org.apache.beehive.controls.api.bean.ControlImplementation;

import org.apache.beehive.controls.api.bean.Extensible;
import org.apache.beehive.controls.api.bean.ControlBean;
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ControlBeanContext;

@ControlImplementation
public class MyControlImpl implements MyControl, Extensible, Serializable
{
    @Context
    ControlBeanContext ctx;

    public String getAnnotationValue() {
        // This is a key part. The control framework will make sure
        // that the precendence is followed when look up annotation
        // values for MyAnnotation
        MyAnnotation myAnnotation =
            ctx.getControlPropertySet(MyAnnotation.class);

        return myAnnotation.value();
    }

    public Object invoke(Method method, Object[] args) throws Throwable
    {
        // Control Extension not defined for simplicity
        return null;
    }
}
```

MyAnnotation.java

```
package controls;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import org.apache.beehive.controls.api.properties.PropertySet;

@PropertySet(externalConfig=true)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation
{
    String value() default "DEFAULT";
}
```

MyJcx.java

```
package controls;

import org.apache.beehive.controls.api.bean.ControlExtension;
```

```
@ControlExtension
@MyAnnotation("OnClassDecl")
public interface MyJcx extends MyControl
{
}
```

HelloService.java

```
/**
 * This class demonstrates the use of an override-able annotation
 * on a control instance. The field "_jcxBean" was declared with
 * MyAnnotation, which has an initial value of "Overriden@Field".
 * During runtime, the sayHello() method will actually return a
 * value that has been externally overriden via a deployment plan.
 *
 * There is no code change required for this class in order to
 * obtain a new externally-configured value.
 */
package service;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

import weblogic.jws.WLHttpTransport;
import org.apache.beehive.controls.api.bean.Control;
import controls.*;

@WebService(name="Hello")
@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)
@WLHttpTransport(contextPath="helloJWS", serviceUri="HelloService")
public class HelloService
{
    @Control()
    @MyAnnotation("Overriden@Field")
    protected MyJcxBean _jcxBean;

    @WebMethod
    public String sayHello()
    {
        return _jcxBean.getAnnotationValue();
    }
}
```

Python Scripts for Annotation Overrides

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

This appendix contains the complete code for the two Python scripts referred to in Annotation Overrides.

- "driver.py" on page B-1
- "OverrideUtils.py" on page B-3

Note: Customization is required before you can use these scripts. For example, you will have to set the username and password for the WebLogic Server domain you are using.

driver.py

```
import sys
from jarray import zeros, array
from java.lang import System
from java.lang import String
from java.util import Random

if(len(sys.argv) != 4):
    print "usage: scriptPath appPath planPath"
    sys.exit()

sys.path.append(sys.argv[1])
import OverrideUtils

connect('weblogic','weblogic')
print 'Connected'

appPath  = sys.argv[2]
planPath = sys.argv[3]

plan = loadApplication(appPath, planPath, 'false')
print 'Got plan'

util = OverrideUtils.PlanUpdater(plan, appPath, planPath)

r = Random()

# Update a field's override value
moduleName='service/HelloService.war'
key=moduleName+'/service.HelloService/_jcxBean/value'
value='I hereby override this value to %d' % (r.nextInt())
util.applyOverride(moduleName, key, value)

# The array case
key=moduleName+'/service.HelloService/_jcxBean/arrayValue'
val1=str(r.nextInt())
val2=str(r.nextInt())
val3=str(r.nextInt())
val4=str(r.nextInt())
arrArg=array([val1,val2,val3,val4],String)
util.applyOverride(moduleName, key, arrArg)

# Update a class' override value
moduleName='OAMEar'
key=moduleName+'/controls.MyJcx/controls.MyAnnotation/value'
value2='Application scope set at %d!!' % (r.nextInt())
```

```
util.applyOverride(moduleName, key, value2)

util.flush()
progress=updateApplication('OAMEar', planPath, stageMode='STAGE',
testMode='false')

f=open('override-value.txt', 'w')
f.writelines('override.value=' + value)
f.writelines('\n\n')
f.writelines('app.override.value=' + value2)
f.flush()
f.close()
```

OverrideUtils.py

```
import sys

class PlanUpdater:

    def __init__(self, plan, appPath, planPath):

        self.MANIFEST="META-INF/annotation-manifest.xml"
        self.plan = plan
        self.updateCount = 0



###########################################################################
##
    #
    # Call this method repeatedly to override different values.
    # In the end, call flush() to persist the changes
    #
    # There are 2 ways to construct the key here:
    #
    # 1. moduleName+"/"+className+"/"+annotationClassName+"/"+memberName
    # 2. moduleName+"/"+className+"/"+fieldName+"/"+memberName
    #
    #

############################################################################
##
    def applyOverride(self, moduleName, key, value):

        print
'\n---------------------------------------------------------------\n'
        if self.plan == None:
            print 'PlanUpdater was not initialized'
            return

        if moduleName == None or key == None or value == None:
         print 'Invalid input parameter to overrideValue() - null parameters
not allowed'
            return

        moduleDescriptor =  self.plan.getModuleDescriptor(self.MANIFEST,
moduleName)
        if moduleDescriptor == None:
            print 'unable to obtain moduleDescriptor for ' + moduleName
            return

        dConfigBean = self.plan.getDConfigBean(moduleDescriptor)
        classes = dConfigBean.getAnnotatedClasses()
        for clazz in classes:
            className = clazz.getAnnotatedClassName()
            annotations = clazz.getAnnotations()
            for annotation in annotations:
                annotationClassName = annotation.getAnnotationClassName()
                members = annotation.getMembers()
                for member in members:
                    memberName = member.getMemberName()
                    theKey =
moduleName+"/"+className+"/"+annotationClassName+"/"+memberName
                   ## FIXME: should probably only change if the values has
changed
                    if theKey == key:
```

```
                    originalValue = member.getMemberValue()
                    currentOverrideValue = member.getOverrideValue()
                    overrideValue = value

                    print '[    Updating Key]: '+ theKey
                    if originalValue != None:
                        print '[  Original Value]: '+ originalValue
                    if currentOverrideValue != None:
                     print '[Current Override]: '+ currentOverrideValue
                    if overrideValue != None:
                        print '[    New Override]: '+ overrideValue
                        print '\n'
                        member.setOverrideValue(overrideValue)
                        self.updateCount += 1
                    return

                arrayMembers = annotation.getArrayMembers()
                for arrayMember in arrayMembers:
                    memberName = arrayMember.getMemberName()
                    theKey =
moduleName+"/"+className+"/"+annotationClassName+"/"+memberName
                    if theKey == key:
                    print 'About to override array member %s' % (memberName)
                        original = arrayMember.getMemberValues()
                        currOvrdValue = arrayMember.getOverrideValues()
                        print 'There are %d items in the array' %
(len(currOvrdValue))

                        print '[    Updating Key]: '+ theKey
                        if original != None and len(original) > 0:
                            print '[  Original Value]:
'+self.arrayToString(original)
                      if currOvrdValue != None and len(currOvrdValue) > 0:
                            print '[Current Override]: '+
self.arrayToString(currOvrdValue)
                        if value != None and len(value) > 0:
                       print '[   New Override]: '+self.arrayToString(value)
                            print '\n'
```

```
                                arrayMember.setOverrideValues(value)
                                self.updateCount += 1
                            return

                fields = clazz.getFields()
                for field in fields:
                    fieldName = field.getFieldName()
                    annotations = field.getAnnotations()
                    for annotation in annotations:
                      annotationClassName = annotation.getAnnotationClassName()
                        members = annotation.getMembers()
                        for member in members:
                            memberName = member.getMemberName()
                            theKey =
moduleName+"/"+className+"/"+fieldName+"/"+memberName
                            if theKey == key:
                                originalValue = member.getMemberValue()
                              currentOverrideValue = member.getOverrideValue()
                                overrideValue = value
                                print '[    Updating Key]: '+ theKey

                                if originalValue != None:
                                    print '[  Original Value]: '+ originalValue
                                if currentOverrideValue != None:
                             print '[Current Override]: '+ currentOverrideValue
                                if overrideValue != None:
                                    print '[    New Override]: '+ overrideValue
                                    print '\n'
                                    member.setOverrideValue(overrideValue)
                                    self.updateCount += 1
                                return

                        arrayMembers = annotation.getArrayMembers()
                        if arrayMembers == None:
                            print '====> No arrayMembers found'
                        else:
                            print '====>>> got arrayMembers, size = %d' %
(len(arrayMembers))
```

```
                    for arrayMember in arrayMembers:
                        memberName = arrayMember.getMemberName()
                        theKey =
moduleName+"/"+className+"/"+fieldName+"/"+memberName
                        if theKey == key:
                            print 'About to override array member %s' %
(memberName)
                            original = arrayMember.getMemberValues()
                           currOvrdValue = arrayMember.getOverrideValues()

                            print '[    Updating Key]: '+ theKey
                            if original != None and len(original) > 0:
                                print '[  Original Value]:
'+self.arrayToString(original)
                       if currOvrdValue != None and len(currOvrdValue) > 0:
                                print '[Current Override]:
'+self.arrayToString(currOvrdValue)
                            if value != None and len(value) > 0:
                                print '[    New Override]:
'+self.arrayToString(value)
                                print '\n'
                                arrayMember.setOverrideValues(value)
                                self.updateCount += 1
                            return

    def arrayToString(self, values):
        if values == None or len(values) == 0:
            return ''
        else:
            result = '[ '
            for value in values:
                result = result + value + ' '
            result = result + ']'
            return result

    ########################################################
    #
    # flush() - Persist the changes to file
    #
    #######################################################

    def flush(self):

        if self.updateCount > 0:
            print 'Saving plan. There were %d changes in this update.' %
(self.updateCount)
            self.plan.save()
        else:
            print 'There has been no change to override values. plan not
saved.'
```

Build Script Sample - Create Web Service That Uses a Service Control

Warning: This document is deprecated as of version 10.0 of WebLogic Server. This deprecation warning applies only to this documentation, and not to the underlying functionality it describes nor to the open-source Beehive project. Users who wish to develop and deploy Beehive applications should do so using Workshop for WebLogic, which offers support for all aspects of Beehive technology. Current documentation for integrating Beehive and WebLogic Server can be found at Workshop for WebLogic Platform User's Guide.

```
<!-- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!! This build script takes you step by step through creating
  !!! a web service application that uses a service control.
  !!! It starts with a wsdl and a client jws.
  !!! There are three important directories invovled:
  !!!   src.dir   - which is where the source wsdl and jws are located
  !!!   build.dir - which is used to create build artifacts
  !!!   deploy.dir - which is the directory that you want to deploy
  !!!                the ear into.
  !!!
  !!! The build will take place in stages. Each step will be
  !!! contributing to an enterprise application directory
  !!! within the build.dir.
  !!! At the end of the build the entire application will be
  !!! built and put into an ear file in the ${build.dir}/deploy
  !!! directory.
  !!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
-->

<project name="serviceControlBuildExample" default="build">

    <property environment="env" />
    <property name="dev.home" value="${env.DEV_ROOT}" />
    <property name="weblogic.home" value="${env.WL_HOME}" />
    <property name="jdk.home" value="${env.JAVA_HOME}" />
    <property name="app.name" value="${ant.project.name}" />
    <import file="${dev.home}/wlw/controls/controls-imports.xml" />
    <import file="${dev.home}/wlw/netui/netui-imports.xml" />
    <import file="${dev.home}/common/beehive/beehive-imports.xml" />
    <import
file="${weblogic.beehive.home}/ant/weblogic-beehive-imports.xml" />
   <import file="${weblogic.beehive.home}/ant/weblogic-beehive-tools.xml"
/>
    <import
file="${weblogic.beehive.home}/ant/weblogic-beehive-buildmodules.xml" />

    <!-- this is the source directory -->
    <property name="src.dir" location="./src"/>

    <!-- this is where things are built into. the ear will wind up in the
deploy subdirectory -->
    <property name="build.dir" location="C:/tmp"/>

    <!-- this is the domain that you wish to deploy your application to -->
    <property name="deploy.dir"
location="${weblogic.home}/test/scexample/${app.name}/app"/>

    <!-- The ent.app.dir is where the enterprise application is built into.
      -->
    <property name="ent.app.dir" location="${build.dir}/${app.name}"/>

    <!-- this is a temporary directory used frequently for disposable work
-->
    <property name="temp.dir" location="${build.dir}/work"/>
    <delete dir="${temp.dir}"/>

    <!-- Setup classpath -->
    <property name="temp.lib.dir" value="${build.dir}/templibdir"/>
    <mkdir dir="${temp.lib.dir}"/>
    <taskdef name="libclasspath"
classname="weblogic.ant.taskdefs.build.LibClasspathTask" />
    <property name="common.dir"
location="${dev.home}/wlw/test/drt/domain/common" />
    <property name="deployable.lib.dir"
location="${weblogic.home}/common/deployable-libraries" />
    <libclasspath tmpdir="${temp.lib.dir}" basedir="${common.dir}"
classpathproperty="app.lib.classpath">
            <librarydir dir="${deployable.lib.dir}" />
    </libclasspath>
   <libclasspath tmpdir="${temp.lib.dir}" basedir="${common.dir}/testjpf"
classpathproperty="web.lib.classpath">
            <librarydir dir="${deployable.lib.dir}" />
    </libclasspath>
    <path id="classpath">
            <pathelement path="${app.lib.classpath}" />
            <pathelement path="${web.lib.classpath}" />
           <pathelement location="${dev.home}/external/junit/junit.jar" />
         <pathelement location="${weblogic.home}/server/lib/weblogic.jar"
/>
          <fileset dir="${ent.app.dir}/APP-INF/lib" includes="**/*.jar" />
            <pathelement path="${ent.app.dir}/APP-INF/classes" />
    </path>



    <!--
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
     !!! This is the primary build script which describes step by step
     !!! how to build a jws application that uses a service control.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
     -->
    <target name="build">
```

```
<!-- let's initialize the EAR directiry by creating it
  !! and seeding it with some initial descriptor files
  !! that provide support for controls
  -->
<mkdir dir="${ent.app.dir}"/>
<mkdir dir="${ent.app.dir}/APP-INF/lib"/>
<mkdir dir="${ent.app.dir}/APP-INF/classes"/>
<copy todir="${ent.app.dir}">
        <fileset dir="${src.dir}">
                <include name="META-INF/*.xml" />
        </fileset>
</copy>

<mkdir dir="${temp.dir}"/>

<!--
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!  BUILDING THE SERVICE CONTROL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 -->

<!-- first we need to build the types jar file. This will get
 !!! generated right into the ent.app.dir where our enterprise
 !!! application will be created. All you need is a wsdl!
 !!! The output from this step is a types jar file placed
 !!! in he /APP-INF/lib of the ear.
-->
<build-type-library
        wsdl="${src.dir}/wsdl/HelloTypesService.wsdl"
        tempdir="${temp.dir}"
        typefamily="tylar"
        destdir="${ent.app.dir}/APP-INF/lib"
        wsdlname="HelloTypesService"
        wsdlservicename="HelloTypesService"
   />
<delete dir="${temp.dir}"/>
```

```
<!-- next we need to build the actual service control jcx
 !!! which will be generated into a work directory
 !!! along with the wsdl.
 !!! The output from this step is the actual service control
 !!! extension (jcx) file placed in the "gensrc" directory.
 !!! In this case it is the hellotest.HelloTypesServiceControl.jcx
 !!! The WSDL is also placed in the same directory with the
 !!! service control extension.
-->
<property name="gen.src.dir" location="${build.dir}/gensrc"/>
<mkdir dir="${gen.src.dir}"/>
<generate-webservice-control
        wsdl="${src.dir}/wsdl/HelloTypesService.wsdl"
        wsdlservicename="HelloTypesService"
        destdir="${gen.src.dir}"
        appdir="${ent.app.dir}"
        servicecontrolname="HelloTypesServiceControl"
        package="hellotest"
        typefamily="tylar"
        classpathref="classpath"
 />


<!-- next we will build the control support classes
 !!! into the APP-INF/classes directory so
 !!! that all apps can take advantage of it.
 !!! This could be built into its own jar or
 !!! into the client too.
 !!! The following files are created and placed in APP-INF/classes
 !!!    HelloTypesServiceControl.class (interface)
 !!!    HelloTypesServiceControlBean.class (impl)
 !!!    HelloTypesServiceControlBean.class.manifest
 !!!    HelloTypesServiceControlBeanBeanInfo.class
 !!! These are all standard controls files
-->
<mkdir dir="${temp.dir}"/>
<mkdir dir="${ent.app.dir}/APP-INF/classes"/>
```

```
<build-controls srcdir="${gen.src.dir}"
                destdir="${ent.app.dir}/APP-INF/classes"
                tempdir="${temp.dir}"
                classpathref="classpath" />

<!--
     Important thing to remember is to move
     the wsdl into the same directory as
     the service control class.
-->
<copy todir="${ent.app.dir}/APP-INF/classes">
        <fileset dir="${gen.src.dir}" includes="**/*.wsdl" />
</copy>
<delete dir="${temp.dir}"/>

<mkdir dir="${build.dir}/module"/>
<mkdir dir="${build.dir}/aptgen"/>
<mkdir dir="${temp.dir}"/>

<!--
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!  BUILDING THE CLIENT THAT USES A SERVICE CONTROL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 -->

<!--  The first step is to run apt against the source code to
  !!! process the annotations.
  !!! The output from this step is source code placed in
  !!! the gendir (aptgen) directory which consists of
  !!! the HelloTest.jws converted to HelloTest.java
  !!! and a HelloTestClientInitializer.java to help initialize
  !!! the service control.
  !!! apt also compiles these classes into the destdir...
  !!! which in this case is the beginnings of our ejb jar
  !!! The following files are created:
  !!!   HelloTest.class (from source in gendir)
  !!!   HelloTestClientInitializer.class (from source in gendir)
  !!!   HelloTest.controls.properties file to help associate the
  !!!           service control with the client
  -->
<taskdef name="apt"
 classname="org.apache.beehive.controls.runtime.generator.AptTask"
    classpathref="apt.task.classpath"
    onerror="fail"/>

<apt srcdir="${src.dir}"
    destdir="${build.dir}/module"
    gendir="${build.dir}/aptgen"
    classpathref="classpath"
    srcExtensions="*.jws"
    />

<!--  The second step is to build the initial version of
  !!! the jws ejb jar. You do this by running jwsc against
  !!! the code generated by apt in the aptgen dir.
  !!! The output from this step is a jar file called
  !!! HelloTest.jar in the dest dir (ent.app.dir)
  !!! and the META-INF/application.xml gets updated
  !!! too with a reference to the ejb.
  !!! A lot of different things are generated here
  !!! to support the
 -->
<taskdef name="jwsc"
    classpath="${weblogic.home}/server/lib/weblogic.jar"
    classname="weblogic.wsee.tools.anttasks.JwscTask"
    onerror="report" />

<jwsc srcdir="${build.dir}/aptgen"
  destdir="${ent.app.dir}"
  classpathref="classpath"
  debug="on" keepGenerated="true">
  <jws file="hellotest/HelloTest.java"/>
</jwsc>


<!--  The third step is to run controls assembly
```

```
          !!! on the ejb application to provide the remaining
          !!! information needed by the service control.
          !!! To do this we unjar the HelloTest.jar into
          !!! the "module" directory we created above.
          !!! Then we run the assemble-controls task,
          !!! compile the new stuff that gets generated
          !!! into the module dir.
          !!! and then we re-jar the module directory back
          !!! into the ent.app.dir (ear).
          !!! The following artifcats are generated by assembly and
          !!! put in the module root:
          !!!    hellotest/HelloTypesServiceControlJaxRpcMap.xml - 109
bindings file
          !!!    hellotest/HelloTypesServiceControlServiceClassMemento.ser
-     runtime info used by the service control
          !!!
          !!! The ejb-jar.xml is updated to contain <service-ref tags that
          !!! provide a service reference to the jax-rpc stub with for the
          !!! service control.
          !!!
         !!! In the destdir (tempdir) the HelloTypesServiceControlSEI.java
file
          !!! is created which provides the Service Endpoint Interface for
JAX-RPC.
          !!! This will be compiled next.
          !!!
         -->
        <unjar src="${ent.app.dir}/hellotest/HelloTest.jar"
dest="${build.dir}/module"/>
        <assemble-controls
            moduledir="${build.dir}/module"
            destdir="${temp.dir}"
            classpathref="classpath"

assemblerclass="org.apache.beehive.controls.runtime.assembly.EJBAssemblyCo
ntext$Factory"/>

        <!-- we need to build the ServiceEndpointInterface -->
        <javac srcdir="${temp.dir}" classpathRef="classpath"
            destdir="${ent.app.dir}/APP-INF/classes"
            includes="**" />

       <!-- we can put the app module back together into a nice jar file -->
        <jar jarfile="${ent.app.dir}/hellotest/HelloTest.jar" >
            <fileset dir="${build.dir}/module" />
        </jar>
        <delete dir="${temp.dir}"/>


      <!-- The EAR is ready to be packed up...we put it in the deploy dir -->
        <mkdir dir="${build.dir}/deploy" />
        <jar jarfile="${build.dir}/deploy/${app.name}.ear" >
            <fileset dir="${ent.app.dir}" />
        </jar>

    </target>

    <target name="deploy">
        <echo message="Deploying application ${app.name}" />
        <copy file="${build.dir}/deploy/${app.name}.ear"
todir="${deploy.dir}" />
        <wldeploy action="deploy" source="${deploy.dir}/${app.name}.ear"
name="${app.name}" user="weblogic" password="weblogic" verbose="false"
adminurl="t3://localhost:7001" debug="false" targets="cgServer" />
    </target>

    <target name="undeploy">
      <fail message="The property app.name is not set." unless="app.name" />
        <echo message="Undeploying application ${app.name}" />
        <wldeploy action="undeploy" name="${app.name}" user="weblogic"
password="weblogic" verbose="false" adminurl="t3://localhost:7001"
debug="false" targets="cgServer" />
    </target>

    <target name="clean">
      <delete dir="${build.dir}"/>
    </target>

</project>
```