

Upgrading WebLogic Workshop 8.1 Applications

Using tools provided with Workshop for WebLogic 9.2, you can upgrade applications created with WebLogic Workshop 8.1 (SP4, SP5, or SP6). This section describes how to upgrade applications built with WebLogic Workshop. It describes the tools provided as well as tips and workarounds for issues you might encounter in your upgraded coded.

Current Release Information:

- [What's New in 9.2](#)
- [Upgrading from 8.1](#)

Useful Links:

- [Tutorials](#)
- [Tips and Tricks](#)

Other Resources:

- [Online Docs](#)
- [Dev2Dev](#)
- [Discussion Forums](#)
- [Development Blogs](#)

Topics Included in This Section

Overview: Upgrading from WebLogic Workshop 8.1

An overview of the upgrade process.

How To: Use the Import Wizard to Upgrade Version 8.1 Applications

Step-by-step guidance on using the wizard for upgrading.

Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2

A list of changes that will affect application during upgrade.

Upgrading Controls

Upgrade details and workarounds specific to controls.

Upgrading Web Services

Upgrade details and workarounds specific to web services.

Upgrading Page Flows

Upgrade details and workarounds specific to page flows.

Upgrading Enterprise JavaBeans

Upgrade details and workarounds specific to Enterprise JavaBeans.

Upgrading Annotations

Upgrade details and workarounds specific to annotations.

upgradeStarter Command

A command to perform the import wizard's upgrade work from the command line.

upgrade Ant Task

A task to perform the import wizard's upgrade work from Ant.

Related Topics

None

©2000-2006 BEA Systems, Inc. All Rights Reserved

Overview: Upgrading from WebLogic Workshop 8.1

Workshop for WebLogic provides tools to help ease the process of upgrading applications built with version 8.1. These tools are designed to read your version 8.1 code and generate a new corresponding version 9.2 workspace. The newly generated code is a migrated, upgraded version of the 8.1 code in which all of the clearly predictable changes have been made (note that your application logic is not altered by [upgrade tools](#)). The new code supports a new annotation model, a different project model, and other component-oriented changes.

Note: The Workshop for WebLogic upgrade documentation assumes that your application was developed using WebLogic Workshop version 8.1. If it wasn't, you must migrate your code so that it builds and runs in the WebLogic Workshop IDE version 8.1 SP4, SP5, or SP6 before using the tools described here to upgrade to Workshop for WebLogic version 9.2.

This topic provides an overview of the changes from version 8.1 to version 9.2, as well as suggested high-level steps for approaching upgrade.

[Differences Between the Version 8.1 and 9.2 IDEs, and the Applications Built with Them](#)

[Upgrade Process: High Level Steps](#)

[Preparing a Version 8.1 Application for Upgrade](#)

[Actions Performed By Upgrade Tools](#)

[Viewing the Upgrade Log](#)

[Notes About General Issues](#)

Differences Between the Version 8.1 and 9.2 IDEs and the Applications Built with Them

If you've been developing on WebLogic Workshop version 8.1, the most significant differences you'll notice in version 9.2 (and applications built with it) are likely to be:

- A [different IDE](#) codebase (it is now built on the Eclipse Platform).
- [Changes in the project model](#) (differences in the locations for project artifacts, file extensions, and so on).
- A [different annotation model](#) that uses the Java 5 standard (as opposed to the Javadoc-style annotations in previous releases).
- [Component-specific changes](#), such as updated, altered or omitted support for certain areas of functionality.

You might also be interested in the reading [Key Differences for WebLogic Workshop 8.1 Users](#).

Key IDE Differences

Unlike version 8.1, in version 9.2 the IDE is built on the Eclipse Platform. This is a change that offers many benefits, including: the transparency of an open-source architecture; commonly used features that are familiar because they're available with other widely used Eclipse-based IDEs; and a widely used extensibility model via the Eclipse plug-in framework. To this open framework, Workshop for WebLogic adds many features to support iterative development of the kinds of components you built with version 8.1.

Needless to say, moving from the version 8.1 IDE to the Eclipse-based version 9.2 means that there are significant user interface changes. The following list describes how familiar version 8.1 features are (or are not) rendered in version 9.2.

- Source types aren't rendered in a Design View. The version 8.1 IDE featured graphical Design View alternatives for most source artifacts, including web services, EJBs, controls, and JSPs. Only the version 8.1 Flow View has a corresponding graphical feature in version 9.2: the Page Flow Editor.
- Extensions to the version 9.2 IDE are based on the Eclipse plug-in framework. In other words, extensions to the version 8.1 IDE must be rewritten to work with version 9.2. Note that version 9.2 does not provide tooling for upgrading version 8.1 extensions.
- Some similar features are labeled differently.

Project Model Differences

Many of the changes from the version 8.1 to the version 9.2 project model are intended to align the model with broadly used Eclipse and Java conventions. If you've used other Eclipse-based IDEs, version 9.2 of Workshop for WebLogic should feel familiar. For a list of significant changes, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#).

Annotation Differences

Workshop for WebLogic version 9.2 supports the [Java 5 annotation model](#). Whereas in version 8.1 the annotations in your source code were embedded in Javadoc-style comments, their version 9.2 counterparts are outside the comment block. In the abstract, however, there are more similarities than differences between the annotation model in versions 8.1 and 9.2. In other words, for most version 8.1 annotations there are version 9.2 counterparts that use the new syntax. Note that as with version 8.1, the version 9.2 IDE provides an editor (the Annotations view) through which you can view and edit annotation attribute values. For more information on the changes, see [Upgrading Annotations](#).

Component-Specific Changes

These are changes that are specific to component types, such as web services, EJBs, controls, and so on. They primarily affect source code. You can read [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#) for a summary list, or see the individual technology-oriented topics listed at the bottom of this topic under Related Topics.

Upgrade Process: High Level Steps

These steps assume that your application was developed with WebLogic Workshop. If you wrote your code without using that IDE and you want to upgrade it using Workshop for WebLogic upgrade tools, you must first migrate your code so that it builds and runs in the WebLogic Workshop IDE.

1. Ensure that your WebLogic Workshop version 8.1 applications have been upgraded to SP4, SP5, or SP6. The [upgrade tools](#) included in this release are designed to upgrade from those versions only.
2. Undeploy WebLogic Workshop version 8.1 applications from your version 8.1 domain *before* you upgrade the server.

Upgrading domains and upgrading WebLogic Workshop applications are separate processes, but they're interrelated. For information about upgrading version 8.1 WebLogic Workshop domains, see [WebLogic Workshop Version 8.1 Domains Can Be Upgraded from Within Version 9.2](#).

3. Use the topics in this documentation to determine whether it would be useful for you to do preparatory work on your version 8.1 application before using the [upgrade tools](#) to upgrade.

For a list of suggested pre-upgraded changes, see [Preparing a Version 8.1 Application for Upgrade](#).

Also, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#). That topic lists upgrade-related issues, providing links to more information.

4. Use upgrade tools to upgrade the version 8.1 application.

[How To: Use the Import Wizard to Upgrade Version 8.1 Applications](#) describes how to use the tool.

You can also perform the wizard's work through the [upgradeStarter Command](#) or [upgrade Ant Task](#).

5. Use the upgrade log and these topics to determine how to start fixing any post-upgrade problems that might keep your upgraded application from running.

Preparing a Version 8.1 Application for Upgrade

If your applications are complex, upgrading them from version 8.1 to version 9.2 is likely to be a multi-step process. A key part of that process will be the upgrade support the IDE provides via [upgrade tools](#). But you'll also find that some preparatory work on your version 8.1 application makes the wizard's end result much easier to work with and get running.

Take a look through the upgrade documentation provided here. Many of the notes recommend ways to edit your version 8.1 application to make your upgraded code easier to get running.

Actions Performed By Upgrade Tools

Workshop for WebLogic includes three tools that automate most parts of the upgrade process. Each tool does essentially the same thing, allowing you to specify applications and parameters for upgrade: the [import wizard](#) is available as user interface in the IDE; [upgradeStarter Command](#) exposes options from the command line; and the [upgrade Ant task](#) exposes options for use from Ant.

Note: You can also upgrade individual files once they're in the version 9.2 IDE. To do this, right-click the file, then click Upgrade Source Files. For information about the error logging and message verbosity options, see [To Import and Upgrade a Version 8.1 Application](#).

The following briefly describes actions performed (and not performed) by the upgrader.


- The upgrader does not change version 8.1 code. Instead it writes upgraded versions of the code into a new version 9.2 workspace that you define.
- No source control actions are performed. In other words, your version 8.1 code is not checked out, nor is your upgraded code checked in to a source control repository. Before upgrade, you should check out your version 8.1 application sources. You should also choose a location for the new workspace that is a convenient place from which to check in the upgraded app.
- Wherever possible, version 8.1 annotations will be upgraded to their version 9.2 counterparts. For a list of version 8.1 annotations and corresponding 9.2 annotations, see [Upgrading Annotations](#).
- Version 8.1 annotations will be preserved in upgraded code. Version 9.2 of the IDE (including the runtime and compiler) does not see the version 8.1 annotation block as anything more than a Javadoc comment. In fact, you might find it helpful to have the older annotations present as you finish upgrading your application. You can delete the version 8.1 annotations at any time.
- The upgrader will migrate your version 8.1 source artifacts into a version 9.2 project model. This includes the following:
 - Converting version 8.1 project types to version 9.2 project types.
 - Optionally moving libraries from the version 8.1 application's Libraries folder to a new EAR project in the upgraded application. An EAR project is the preferred location for libraries used by multiple projects in version 9.2.
 - Moving JSP files into a WebContent directory.
 - Upgrading NetUI JSP tags to versions that are compatible with version 9.2 of the runtime.
 - Optionally migrating NetUI JSP tags to Apache Beehive JSP tags. See [Changes When Upgrading from Version 8.1 NetUI JSP Tags to Beehive NetUI JSP Tags](#) for more information.
 - Moving XSD files that are in a schema project into a version 9.2 utility project.
 - Moving Java packages and sources into a src directory if they are not in one already.


Viewing the Upgrade Log


Whether you [use the import wizard](#), [command-line](#), or [Ant task](#) to upgrade, Workshop for WebLogic will generate a log of the upgrade changes, errors, and warnings. If you use the import wizard, this log will also be displayed in a dialog you can review before completing the process.

Upgrade Preview Dialog

You can expand the node for each file to view the upgrade messages associated with that file. The following key describes the symbols displayed next to file names:

 Informational message.

 Warning message.

 Error message.

Text Log File

Upgrade tools generate a log file containing upgrade messages. This file is available at the following location after upgrade has completed:

```
UPGRADE_WORKSPACE_HOME\metadata\upgrade.log
```

A log message in the file will take the following form:

```
!SUBENTRY 1 com.bea.wlw.upgrade severity_level date time
!MESSAGE Upgrade-related message.
```

The *severity_level* will be two numbers, but they have the same meaning. The *date* and *time* entries refer to when the upgrade was attempted. The *upgrade-related message* describes what was done, warned about, or the error that occurred. The following is a snippet that shows two log entry examples:

```
!SUBENTRY 1 com.bea.wlw.upgrade 2 2 2006-02-27 17:17:53.687
!MESSAGE The 9.2 control context only supports a subset of the 8.1 control context APIs. Please see the
Workshop for WebLogic upgrade documentation for more information.
```

```
!SUBENTRY 1 com.bea.wlw.upgrade 1 1 2006-02-27 17:17:53.687
!MESSAGE The import "com.bea.control.JwsContext" needs to be updated.
```

Notes About General Issues

The following describes problems you might see after upgrade, but which aren't tied to particular component types or technologies. For upgrade-related issues associated with specific technologies, see the topics listed under Related Topics.

Version 8.1 Applications with Dependencies on Third-Party Library JARs Might Result in Version 9.2 Compile Errors

Generally speaking, the upgrade process will present a warning message when a precompiled library must be recompiled to be used in the version 9.2 environment. However, there might be cases when the library should be recompiled but no message is presented. In other words, consider recompiling these libraries when you see compile errors in a project that you can't otherwise account for.

Application Library JAR Files Whose Manifest Has Line Lengths Exceeding 72 Bytes Will Cause Build Errors

Unlike version 8.1, version 9.2 enforces the line length size limit in JAR file manifests when the JAR is used as an

application library. Upgraded applications containing JARs with lines exceeding the limit will generate errors at build time. To work around this change, either reduce the size of the lines or remove the manifest from the JAR.

Upgraded Code That Uses Internal APIs from weblogic.jar Will Break in Version 9.2

The use of internal APIs is not supported and the version 9.2 classpath does not (and should not) include weblogic.jar. Public APIs that used to be in weblogic.jar have been moved into wls-api.jar. The workaround for breakage due to the absence of weblogic.jar from the classpath is to rewrite code to use public APIs.

In addition, code that uses third-party APIs is not supported for upgrade to version 9.2.

Upgraded Code May Conflict with New Java 5 and JSP 2.0 Language Features

New language features in Java 5 and the JSP 2.0 expression language (which is used by the Beehive tag libraries) might make it necessary to rewrite portions of upgraded code.

For example, new features reserve words that were not reserved for code written in version 8.1. Workshop for WebLogic upgrade tools do not upgrade the use of these words, so code that uses them will not compile until you rewrite it so that it accounts for the new language features.

In particular, Java 5 adds the `enum` keyword. For more information, see [Enums](#) at the Sun web site.

JSP 2.0 reserved word are listed in [Reserved Words Can Not Be Used as Identifiers](#).

In addition, upgraded code might encounter many changes in the Java APIs. With the addition of generics to the Java language, method signatures that took Object parameters in Java 1.4 have become more strongly typed. This enables Workshop for WebLogic version 9.2 to detect many errors at compile time that would only appear at run time in version 8.1. For example, if you wrote:

```
String s;
Thing t;
s.compareTo(t);
```

you would see no compilation error in version 8.1, but a `ClassCastException` would occur at run time. In version 9.2, `compareTo(t)` will be flagged as a compile time error because `String.compareTo()` actually expects a `String` parameter in 9.2, but only requires an `Object` in 8.1.

As a general rule, debugged code will not contain these types of errors. But when they appear during post-upgrade compilation, the code must be fixed by the developer before the project will build successfully.

Upgraded Wildcard Import Statements Might Leave Some Types Stranded Without an Import

During the upgrade process, Workshop for WebLogic upgrade tools update package import statements to support the movement of key libraries. In some cases, this might break code that uses a class from the version 8.1 package that is still in that package. For example, [upgrade tools](#) will make the following changes:

```
import com.bea.wlw.netui.util.*;
```

... would be changed to...

```
import org.apache.beehive.netui.util.*
```

This leaves `com.bea.wlw.netui.util.TemplateHelper` stranded without an import because it is still in the older package. You can easily repair the code by using the Workshop for WebLogic quick fix feature to import required packages. Another, more thorough fix would be to remove wildcards and explicitly import classes in the version 8.1 code before upgrading it.

Version 8.1 Pointbase APIs are Not Supported in Upgraded Code

Version 8.1 includes APIs provided with the Pointbase database. These are not supported in version 9.2.

XML Schemas That Were Valid in Version 8.1 May Not be Valid in Version 9.2

Version 8.1 of the IDE allowed the use of schemas that were not standards-compliant; version 9.2 does not. As a result, invalid schemas will generate errors during upgrade. Note, however, that while errors will appear in the IDE, the errors should not effect runtime behavior.

If you want to continue using the schemas in their invalid state but don't want errors displayed in the IDE, you can turn off schema validation. If you turn off validation, it will be off for all schemas, even though you may want to have validated. To turn off schema validation:

1. Click **Window > Preferences**.
2. In the **Preferences** dialog, in the left pane, click **Validation**.
3. In the right pane, under **Validation**, clear the **XML Schema Validator** check box.
4. Click **OK**.

WebLogic Workshop Version 8.1 Domains Can Be Upgraded from Within Version 9.2

When you create a new server as described in [Creating a Server Definition for Use Within the IDE](#), you can upgrade a WebLogic Workshop version 8.1 domain.

1. In Workshop for WebLogic, click **File > New > Server**.
2. In the **New Server** dialog, under **Select the Server Type**, choose a new WebLogic Server type. For example, you can choose **BEA WebLogic v9.2 Server**.
3. Make other settings as needed, then click **Next**.
4. In the **Domain home** box, enter or browse for the WebLogic Workshop domain you want to upgrade.
5. With the path to a domain in the **Domain home** box, the dialog will display the following as a link:

An older version domain is detected. Click here to upgrade it with the Upgrade Wizard.
6. Click the "older version" link to launch the BEA WebLogic Upgrade Wizard.
7. Follow the steps in the domain upgrade wizard as described in [Procedure for Upgrading a WebLogic Domain](#).

Related Topics

[Upgrading Web Services](#)

[Upgrading Page Flows](#)

[Upgrading Controls](#)

[Upgrading Annotations](#)

[Upgrading Enterprise JavaBeans](#)

How To: Use the Import Wizard to Upgrade Version 8.1 Applications

Workshop for WebLogic version 9.2 provides functionality through which you can use the import wizard to upgrade version 8.1 applications. This wizard makes nearly all of the changes that upgrade a working version 8.1 application to a working version 9.2 application. Because the differences between version 8.1 and version 9.2 are many, there can be a lot of changes to make. The list of changes made, of course, omits those that would involve rewriting your code or making non-obvious assumptions about what you intend your application to do.

Note: Your version 8.1 application must have been upgraded to SP4, SP5, or SP6 before using the wizard.

Keep in mind that using the import wizard is usually only one part of the upgrade process. Depending on the technologies your application uses, you might find that some preparatory work makes an upgrade process that incorporates the wizard more efficient. Likewise, some work after using the wizard is likely to be needed to get your upgraded application compiling and running. For more at a high level on the upgrade process, see [Overview: Upgrading from WebLogic Workshop 8.1](#).

Note: Due to a [known issue](#) with the Sun JVM in which JAR files become locked, when upgrading applications on the Windows operating system there might be intermittent cases when files are left behind in the temp directory after upgrade. If this occurs you should be able to delete the files by first closing the IDE.

Setting Wizard Defaults for Upgrade

You can set defaults for some of the wizard's prompts. This is useful if you intend to upgrade multiple applications with the same settings. Use the following steps to set upgrade defaults:

1. In Workshop for WebLogic version 9.2, click **Window > Preferences**.
2. In the **Preferences** dialog, in the left pane, expand **WebLogic**, then click **Upgrade**.
3. Set defaults as needed, then click **OK**.

Note that the settings themselves are described in the [import and upgrade](#) procedure below.

Ensuring That the Wizard Has Enough Memory

Before using the import wizard to upgrade applications, consider temporarily increasing the maximum amount of memory that the Java Virtual Machine allows to Workshop for WebLogic. The upgrade process requires a compilation step that potentially includes a large number of files. The recommended maximum memory is 1 gigabyte.

You can increase maximum memory in the following way:

- Before starting the IDE, edit the workshop4WP.ini file to replace the `-Xmx` value with a sufficiently high memory maximum. By default, the workshop4WP.ini file is located here:

BEA_HOME\workshop92\workshop4WP\workshop4WP.ini

For example, you might change the setting from `-Xmx768m` (the default) to `-Xmx1G` (to set it to 1 gigabyte).

After you've finished using the upgrade tools, set the memory maximum back to a level that's appropriate for development.

To Import and Upgrade a Version 8.1 Application

1. If the files you're upgrading are in a source control system, check them out before running the wizard.
2. Start Workshop for WebLogic version 9.2.
3. Open or create the workspace that will be the destination for your upgraded files.
4. Click **File > Import**.
5. In the **Import** dialog, under **Select an import source**, select **Workshop 8.1 Application**, then click **Next**.
6. In the **Workshop 8.1 Application Upgrade** dialog, click the **Browse** button to locate the WORK file for the Workshop 8.1 application you want to upgrade.
7. After you have selected the WORK file, note that the application's projects are listed with check boxes.
8. In the project list, clear check boxes for projects you do not want to upgrade and import; ensure that check boxes are selected for desired projects.

While it is possible to deselect projects and so omit them from the upgrade, note that build and visibility dependencies between deselected projects and selected projects will be lost. You will need to manually reset these dependencies if you choose to upgrade the deselected projects at a later time.

9. Under **Source Upgrade**, expand **General**, **NetUI Project Upgrader Options**, **Properties file upgrade options**, and **JSP File Upgrader options** to consider the following options for upgrade:
 - Under **General**, select your preference for error handling and message verbosity. The following table lists the options:

Error handling options:

Setting	Description
Log errors but continue upgrade (default)	Upgrade will proceed regardless of whether there are errors; errors will be logged.
Log errors but abort upgrade	Upgrade will be aborted if there are errors; errors will be logged.
Display dialog on error	The IDE will display a dialog on each upgrade error. Note that errors will still be written to the upgrade log.

Message verbosity options:

Setting	Description
Include informational comments (default)	Upgrade messages will include all three levels of comment severity: information, warnings, and errors.
Include warning comments	Upgrade messages will include the top two levels of comment severity: warnings and errors.
Include error comments	Upgrade messages will include only the most severe comments: errors.

- Under **NetUI Project Upgrader Options**, select the **Use WebLogic J2EE Shared Libraries** check box to request that all upgraded projects in the application share a common set of runtime libraries.
- Under **Properties file upgrader options**, select the **Delete copied resource bundle files...** check box to have the upgrader remove unneeded resource bundle files from your upgraded project.
- Under **JSP File Migrator Options**, select the **Replace BEA NetUI tags...** check box to upgrade these version 8.1 JSP tags to Beehive versions.

For important information about upgrade and JSP tags, be sure to read about upgrading JSP tags in [Upgrading Page Flows](#) before upgrading JSP tags. Many of the NetUI custom JSP tags that were part of WebLogic Workshop 8.1 were donated to the Apache Beehive open source project. The latest versions of these tags are now in the Beehive project. Select this check box to have the version 8.1 tags in your upgraded projects replaced with Beehive tags.

10. Click **Next**. Workshop for WebLogic will import and compile the application in order to display file-by-file upgrade status messages.
11. If you are prompted with the Java license agreement, click **I Agree** or **I Disagree**. Note that clicking **I Disagree** aborts application upgrade.
12. Under **Upgrade Preview**, note the list of included files and related upgrade messages. As noted in [Viewing the Upgrade Log](#), you can expand the nodes in the dialog to view messages for each file. The messages here are also available in a log file after the wizard has done its work.
13. After reviewing the messages, click **Finish**.

Related Topics

[Overview: Upgrading from WebLogic Workshop 8.1](#)

Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2

This topic provides a list of the changes you're likely to see as you upgrade version 8.1 applications.

[Project Model Changes Include Folder Hierarchies, File Extensions, Project Types](#)

[Javadoc-Style Annotations Have Been Replaced with Java 5 Annotations](#)

[Version 8.1 Features Updated in Version 9.2](#)

[Version 8.1 Features Not Supported in Version 9.2](#)

[Scenarios Not Supported by Upgrade](#)

Project Model Changes Include Folder Hierarchies, File Extensions, Project Types

Many of the changes from the version 8.1 to the version 9.2 project model are intended to align the model with broadly used Eclipse and Java conventions. If you've used other Eclipse-based IDEs, version 9.2 of Workshop for WebLogic should feel familiar.

Note: You might also be interested in the reading [Key Differences for WebLogic Workshop 8.1 Users](#).

The following list summarizes the most significant differences between version 8.1 and version 9.2 project models.

- Semantically speaking, version 8.1 applications are replaced with version 9.2 workspaces; you build applications in version 9.2 workspaces as you did in version 8.1 applications. Note that version 9.2 workspaces are not represented by a single file you open in the IDE (such as the version 8.1 .work file). In version 9.2 a workspace is represented by a file system folder, and you select this folder when opening the workspace.
- Version 9.2 adds the EAR project type to contain application configuration information and libraries shared across projects.
- Version 9.2 adds the utility project type for developing shared code.
- There is no version 9.2 counterpart for the version 8.1 control project type.
- Version 9.2 does not feature a "schema project" type — a type of project specifically designed for compiling XML schemas (in XSD files) and WSDL files into XMLBeans. In version 9.2 a similar feature is provided by way of a project's XMLBeans Builder facet. For example, to generate XMLBeans types for use in multiple projects, create a Utility project and be sure to

select the XMLBeans check box when choosing project facets.

- Java source files all have .java file extensions. In version 8.1, while EJB, web service, and control sources were all either Java classes or interfaces, their file extensions were specific to file type: .ejb, .jws, .jcs, .jcx, and so on. In version 9.2 all such files use the .java extension.
- Version 9.2 provides support for library modules to enable sharing libraries across projects. Contrast this with version 8.1, in which library JAR files were required in the WEB-INF/lib directory of each project that used them regardless of whether the JAR files were used by multiple projects.
- Java source files can't be kept in a project's root directory (effectively putting them into the default package). Version 9.2 projects must specify a sub-directory for source files and Java sources must reside in package sub-directories of the source directory.
- In version 9.2 web applications, page flow controller files and JSP files reside in separate project folders. In version 8.1 they could be in the same folder (or *co-located*). JAVA files and JSP files are separated into version 9.2 web application src and web (or WebContent) folders, respectively.

Javadoc-Style Annotations Have Been Replaced with Java 5 Annotations

One of the largest aspects of this upgrade is the change from the Javadoc-comment style annotations used in version 8.1 to the Java 5 annotations supported for the current version. For more information, see [Upgrading Annotations](#). Most of the version 8.1 annotations have counterparts in version 9.2; for more information, see [Relationship Between Version 8.1 and Version 9.2 Annotations](#).

Version 8.1 Features Updated in Version 9.2

These features have updated counterparts in version 9.2 and may be deprecated.

WS-Security Should Be Upgraded to WS-Policy

Impacts: Web services

Upgrade strongly recommended. In version 8.1, web service message-level security is managed using WS-Security (WSSE) policy files. In version 9.2 you should use Web Services Policy Framework (WS-Policy).

For information on manually updating to WS-Policy security policies, see [Upgrading Security from from WS-Security to WS-Policy](#).

Security Settings for Service Control Specified in Multiple Locations

Due to changes in the web services security model, the means for specifying security characteristics through the version 9.2 service control differs from version 8.1. In version 8.1, you specified both security policies and values in a WebLogic Workshop WSSE policy file. In version

9.2, the means for specifying characteristics for these two aspects of security has been split into multiple locations.

For information on where security characteristics are now set, see [Configuring Run-Time Message-Level Security Via the Service Control](#).

Roles Defined in web.xml No Longer Assumed to be the Principal Name in Upgraded Projects

In the course of upgrading your version 8.1 applications, you might find that some of your application's security-related characteristics differ between its behavior on the domain shipped with Workshop for WebLogic and the upgraded domain to which you redeploy it. This is because the "new" 9.x domain included with Workshop for WebLogic is not backward compatible, whereas the upgraded domain to which you deploy your upgraded application is (because it has been upgraded).

For more information and workarounds, see [Resolving Issue of Unmapped Entries in web.xml](#).

Handling SOAP Faults with Wrapper Classes is Deprecated

Impacts: Web services

Supported but deprecated. Version 8.1 provided wrapper classes through which you could control the content of an outgoing SOAP fault, as well as APIs for retrieving content from an incoming SOAP fault. You can replace this functionality by using JAX-RPC to map SOAP faults to exceptions.

For more information, see [Alternative to Wrapper Classes for Handling SOAP Faults](#).

Automatic Transaction Rollback for a Checked Exception is Deprecated

Impacts: Web services

Supported but deprecated. In version 8.1, the runtime would roll back a container-managed transaction if a checked exception was thrown from the application or runtime. In version 9.2, this behavior is supported with an annotation added (by [upgrade tools](#) during upgrade) to the web service source code.

For more information about this change and the workaround, see [Upgrade Changes for Automatic Transaction Rollback](#).

Conversational Web Services Without START and FINISH Methods are Not Supported

Impacts: Web services

In version 8.1 it was possible to compile a "conversational" web service that did not have START

or FINISH operations. In other words, you could annotate the web service class as conversational but not annotate any of its operations with conversation phase attributes. In version 9.2 a conversational web service must have both a START and FINISH operation in order to compile.

For more information, see [Ensuring START and FINISH Methods for Conversations](#).

Combining Stateful and Stateless Operations in Web Services is Deprecated

Impacts: Web services

Version 9.2 supports web services that include both stateless and stateful operations, but the support is deprecated. In other words, in an upgraded conversational web service, stateless operations will be annotated with the deprecated annotation `@Conversation(value = Conversation.Phase.NONE)`.

For more information, see [Upgrade Changes for Web Services That Combine Stateful and Stateless Operations](#).

Same-Named Web Services Can Cause Deployment Error

Impacts: Web services

Due to a difference in the way versions 8.1 and 9.2 generate the default targetNamespace value for web services, you may encounter a deployment error if you have two or more web services with the same class name in an application. For web services in version 9.2, WebLogic Server uses the fully-qualified port name — which includes the web service's targetNamespace value — to bind resources it uses internally. As a result, the port name must be unique within an application.

For information on resolving the error, see [Resolving Deployment Error for Same-Named Web Services](#).

Mismatch Between Operation Parameter Names and Names in WSDL Can Cause Web Service Failure

Impacts: Web services

After upgrading a web service in which one or more method parameter names do not match their corresponding names in the WSDL from which the web service was created, you will need to add a `@WebParam` annotation to each parameter.

For details, see [Supporting Mismatch Between Operation Parameter Names and Names in WSDL](#).

Reliable Messaging Support Not Upgraded by Tools

Impacts: Web services

Workshop for WebLogic [upgrade tools](#) do not upgrade reliable messaging support (such as the `@jws:reliable` annotation) from version 8.1 to version 9.2. As noted in the version 8.1 documentation, that version's reliable messaging (RM) support was very limited and was not based on a specification that would be supported in future versions. You can manually upgrade reliable messaging support.

See [Upgrading Reliable Messaging Support — Basic Instructions](#) for high-level upgrade steps.

Version 9.0 and 9.1 WebLogic Server Web Services Might Need to Be Recompiled for Deployment in Version 9.2

Version 9.x doesn't support using the form-get and form-post message formats to receive messages sent from an HTML form. When upgrading web services that use these formats, you'll need to use another method for receiving data sent from a form in a web browser.

In other words, version 9.2 web services do not support message formats that do not include SOAP headers.

In version 8.1, the `@jws:protocol` annotation supported the following attributes and values:

- `@jws:protocol form-get="true"` — Indicated that the operation or web service supported receiving HTTP GET requests.
- `@jws:protocol form-post="true"` — Indicated that the operation or web service supported receiving HTTP POST requests.

These attributes have no counterparts in version 9.2 and there are no suggested workarounds. If you upgrade to version 9.2, [upgrade tools](#) will simply ignore a protocol setting that isn't supported.

Web Service from WSDL with `xs:anyType` Will Expect and Send Incorrect Message Payloads

Impacts: Web services

If you created a version 8.1 web service by generating it from a WSDL that specified `xs:anyType` instead of `xs:any`, the web service will expect and send incorrect XML payloads after upgrade to version 9.2.

You can ensure correct handling of `xs:anyType` by applying the `@WildcardBindings` annotation to the web service at the class level. For more information, see [Ensuring Correct Handling of `xs:anyType` in Messages](#).

Service Controls Must Be Associated with a WSDL

Impacts: Web services, control use

While version 8.1 allowed a Service control to not be associated with a WSDL, the control must be associated with a WSDL in order for it to upgrade successfully.

For more information on associating a WSDL with the service control, see [Associating a Service Control with a WSDL](#).

Service Controls Based on Abstract WSDLs are Not Fully Upgraded

Impacts: Web services

In version 8.1 it was possible to generate a service control from an abstract WSDL — that is, from a WSDL with no service definition. You could then configure the endpoint to call or listen on programmatically or by using annotations. However, WebLogic Server version 9.2 does not support abstract WSDLs. As a result, Workshop for WebLogic's [upgrade tools](#) are unable to upgrade a service control with the necessary annotations, leaving compiler errors in upgraded code. Likewise, if you try to generate a service control from an abstract WSDL in 9.x, you will receive an error stating that a service is required.

For more information and a suggested workaround, see [Upgrading Service Controls that are Based on an Abstract WSDL](#).

Service Control Methods `getEndPoint` and `setEndPoint` are Deprecated

The `ServiceControl.getEndPoint()` and `ServiceControl.setEndPoint(URL)` methods are deprecated in version 9.2 and may be removed in a future version. New code requiring this kind of API should use `ServiceControl.getEndpointAddress()` and `ServiceControl.setEndpointAddress(String)`, respectively.

For more detail, see [Replacing Service Control Methods `getEndPoint` and `setEndPoint`](#).

JMS Control Properties Not Automatically Bound to Method Parameters by Upgrade

Upgrade tools do not fully upgrade JMS control support for binding control method parameters to JMS properties. In particular, the method parameters themselves must be annotated but aren't. You can ensure support for these bindings by manually adding the required annotations after upgrading your version 8.1 code.

For more information and an example, see [Supporting Parameter Bindings for JMS Properties with the JMS Control](#).

Controls are Not Automatically Run Within the Scope of a Transaction

Impacts: Control use

In version 8.1, controls are automatically run within the scope of a transaction because they're run in the context of an Enterprise JavaBean. As Plain Old Java Objects (POJOs) in version 9.2, controls must be annotated with the `@TransactionAttribute` annotation for transaction support. During upgrade, [upgrade tools](#) will add this annotation.

For information on adding transaction support, see [Enabling Automatic Transaction Support in Controls](#).

Row Set Functionality is Supported Through a Backward Compatible JdbcControl

Impacts: Control use

The standard Beehive JdbcControl ([org.apache.beehive.controls.JdbcControl](#)) in version 9.2, which corresponds to the version 8.1 [Database control](#), does not support the version 8.1 "RowSet control" feature. To ensure that row set functionality is preserved during upgrade, the Database control is upgraded to the backward compatible JdbcControl ([com.bea.control.JdbcControl](#)) provided by BEA.

For more information, see [Changes to Support Database Control Row Set Functionality](#).

Control Event Handler Exception Handling More Restrictive

Impacts: Control use

In version 8.1, control event handlers (known in that version as "callback handlers") could throw exceptions caught from the control by throwing the caught exception or a superclass such as `java.lang.Exception`. In version 9.2, if the event handler throws the caught exception, it must instead throw a proper subset of the throws clause declared for the control `EventSet` method.

See [Upgrading Exception Handling in Control Event Handlers](#) for information on working around the new requirement.

Upgrading Message Buffering in Custom Controls

In version 8.1 you could apply the `common:message-buffer` tag to a custom control's interface or implementation code. In version 9.2 this annotation's counterpart, [com.bea.control.annotations.MessageBuffer](#), is supported only in the control interface code. To work around this change, you should remove the annotation from implementation code before upgrading the application.

Control and Web Services Context APIs Have Changed

Version 8.1 provided context APIs through which components such as web services (in version 8.1 JWS files) and custom controls could interact with their runtime environment. For web services, the location of `JwsContext` has changed, while some methods are no longer available. For

controls, changes brought by the Apache Beehive control model to version 9.2 have meant that several APIs exposed by the version 8.1 control context classes are either exposed in different ways or are no longer relevant and so not available.

For a list of the affected APIs, see [Handling Context API Changes](#).

Entity Beans are Not Automatically Run Within the Scope of a Transaction When the Transaction Isn't Specified

Impacts: Entity beans

In version 8.1, the EJB container would create a transaction for an entity bean if it ran in an unspecified transaction. In version 9.2, the default is *not* to create the transaction.

For information on ensuring the old default behavior, see [Enabling Automatic Transaction Support in Entity Beans](#).

Ambiguity May Occur When Annotation Type References are Added to Upgraded Code

Impacts: Potentially any code that uses annotations

Unlike version 8.1, in version 9.2 annotations are Java types that must either be imported or fully-qualified in code. Because code using these types is being added to your code in order to upgrade from annotations in your version 8.1 code, there can sometimes be ambiguity when the added annotations have the same names as types your code may already have been using.

For information on how to remedy this ambiguity, see [Resolving Ambiguity Related to Annotation Types](#).

Upgrading NetUI Tags to Beehive Tags Results in Changes to Some Tags and Attributes

Impacts: JSP files

Due to the differences between the version 8.1 NetUI tags and the Beehive NetUI tags, some changes might be made to JSP tags if you choose to upgrade them. For more information, see [Changes When Upgrading from Version 8.1 NetUI JSP Tags to Beehive NetUI JSP Tags](#).

Packages for Fully-Qualified Type Names are Not Upgraded in JPF File Method and JSP Code

When a type name is fully-qualified outside a type import (Java or JSP), the type's package is not upgraded to the package used in version 9.2. For more information, including a workaround, see [Fixing Package Names That are Not Upgraded in JPF File Method and JSP Code](#).

Upgrading NetUI Tags to Beehive Tags Sets the Default Expression Language to a Backward-Compatible Version

Impacts: JSP files

A backward-compatible version is set as the default for the upgraded application because it is more permissive than the current version, giving you an opportunity to migrate your version 8.1 code. For information on changing the default expression language, see [Changing the Default Expression Language Used by JSP Tags](#).

Impacts: JSP files

Upgrading NetUI Tags to Beehive Tags Does Not Fully Account for Expression Language Requirements

Impacts: JSP files

If you request it, [upgrade tools](#) will migrate your NetUI JSP tags to Beehive JSP tags. This includes migrating from the NetUI expression language syntax to the syntax of the Beehive JSP expression language. However, note that expressions in the Beehive JSP expression language are unable to bind to public fields, as was the case with NetUI expressions. For a full upgrade to Beehive JSP tags, in other words, you must manually add `get*` and `set*` accessors where public fields were used.

For more information on these differences, see [Changing Code to Support the Expression Language in Beehive NetUI JSP Tags](#).

Some PageFlowController and FlowController Methods Made Protected Instead of Public

To enhance application security, some public methods in [org.apache.beehive.netui.pageflow.PageFlowController](#) and [org.apache.beehive.netui.pageflow.FlowController](#) have been changed so that they're protected. This change means that these methods can no longer be invoked as bean properties from within JSP pages.

For more information, see [Details: Some PageFlowController and FlowController Methods Made Protected Instead of Public](#).

FlowController.getRequest Method Can No Longer Return a ScopedRequest Instance

In version 8.1, the return value of the `FlowController.getRequest` method could be cast to a `ScopedRequest` when running in the WebLogic Portal environment. In version 9.2, to improve performance the Beehive page flow APIs retrieve a `ScopedRequest` instance differently. If your portal code makes a call to this method, you will need to manually upgrade the code to avoid a

ClassCastException.

For more information and a suggested fix, see [Upgrading from getRequest Method Calls to Retrieve a ScopedRequest Instance](#).

XMLBeans Package is Changed; Schemas Must be Recompiled to Regenerate Types

Impacts: Code that uses XMLBeans, including web services, controls, JSP files, Enterprise JavaBeans

During upgrade, the package for types in the XMLBeans API will be changed from `com.bea.xml` to `org.apache.xmlbeans`. In addition, if your version 8.1 application included a schemas project (a project that supported automatic compilation of schemas into XMLBeans types), that project will be migrated to a version 9.2 project through which the schemas continue to be compiled into XMLBeans types.

XML Schemas are Validated More Strictly

Impacts: XML schemas

Version 9.2 uses a more strict schema validator than was used in version 8.1. The upgrade process does not repair invalid schemas that might have been considered valid in version 8.1. Because of this, you should ensure that schema validation is on when working with schemas whose validity is important.

XQuery Implementation Used by XMLBeans Updated from Working Draft 16 to Working Draft 23

Impacts: Code that uses XMLBeans and XQuery, including web services, controls, page flows, Enterprise JavaBeans

The older XQuery implementation is deprecated, but supported in this version for backward compatibility. Queries based on the older implementation will be kept, but a special `XmIOptions` parameter will be added to specify that the old implementation should be used.

One exception is XQuery use in JSP files, which is not updated by upgrade tools. You will need to make changes manually. For more information see [Updating XQuery Use to Support Upgraded XQuery Implementation](#).

Custom Ant Build Scripts are Not Upgraded by the Upgrade Tools

Impacts: Build processes

If you created custom Ant build scripts in version 8.1, you must manually upgrade them for use with version 9.2. You can migrate your modifications after upgrading your application by re-

exporting your Ant build file, then merging in your modifications.

Note: Unlike version 8.1, version 9.2 does not support building your application with Ant from within the IDE. You must export the build file and execute targets from the command line.

In version 9.2 use the following steps to export an Ant build file:

1. Right-click any file in your application, then click **Export**.
2. In the **Export** dialog, click **Workshop Ant Script**, then click **Next**.
3. In the **Ant Script Generation** dialog, under **Project**, select the workspace project for which you want to export an Ant script.
4. Under **Ant Script Generators**, select the WebLogic Server script you want to generate.

Note that to generate a script designed to build projects for use on WebLogic Server, you should choose a generator that has "(WebLogic Server)" in its name. Do not select "Java Project Build Script," which compiles JAVA files but is not designed to build projects for WebLogic Server.

5. Under **File Name**, enter (or browse for) the location for the Ant script file you are about to generate.
6. Click **Finish**.

You can edit the generated script file to support the customizations you want.

When running targets in the generated Ant script file, you will need to pass to the script a parameter value indicating the workspace's location. You can do this in one of two ways using the `-Dworkspace` command-line option. You can also specify the location of a workspace metadata file:

```
ant -buildfile build.xml -Dworkspace=c:\workspaces\MyWebServices\workspace.xml
```

To generate the metadata file, you export it as described above for the Ant script file. In the Export dialog, select Workspace Metadata for Workshop Ant Scripts. In subsequent dialogs, select the projects whose metadata you want the file to include, along with specific variable values you want. Note that as you change workspace settings — such as by adding a new project to an EAR — you will need to re-export the metadata file.

You can also give the workspace directory as the argument:

```
ant -buildfile build.xml -Dworkspace=c:\workspaces\MyWebServices
```

IDE Extensions Based on the Version 8.1 API are Not Supported

Impacts: IDE extensions

Version 9.2 of Workshop for WebLogic is based on Eclipse, which has its own model and API for developing extensions — known to Eclipse users as plug-ins. Version 8.1 extensions are not upgraded. If you wrote extensions based on the version 8.1 model, you need to rewrite them if you want to keep that functionality. If you used extensions based on the version 8.1 model, you need to find parallels in the Eclipse extension set.

For more information on Eclipse plug-ins, see Eclipse help in this documentation or [online](#).

Configuration Settings are Now Exposed Through WebLogic Administration Console or WLST

Impacts: Configuration

WebLogic Workshop version 8.1 provided several files you could use to configure the run-time environment for applications you built with WebLogic Workshop. These files are not included in version 9.2. Many of the properties set through them are now configurable using the WebLogic Administration Console, or are scriptable with the WebLogic Server Scripting Tool (WLST).

- [jws-config.properties](#) — Provided domain-wide configuration parameters for run-time components. This included information about conversations and JMS.
- [wlw-config.xml](#) — Primarily provided build-time directives to parameterize the code generation of EJBs that hosted 8.1 Web Services.
- [wlw-runtime-config.xml](#) — Provided run-time parameters of web resources.
- [wlw-manifest.xml](#) — Provided information about the server resources referenced in an EAR; useful to administrators determining the resources needed for successful deployment.

For more information about the administration console, see [Overview of the Administration Console](#) in the WebLogic Server documentation. For more about WLST, see [WebLogic Scripting Tool](#).

Some Version 8.1 Wildcard Import Statements Will Break Compilation on Version 9.2

Because the Workshop for WebLogic [upgrade tools](#) don't upgrade wildcard import statements, some of these statements will generate errors on version 9.2 because their libraries are not present there. In some cases, you can fix these by replacing them with their version 9.2 counterparts. For example, if

After upgrade, you might find that some import statements with wildcards have been changed to reflect their nearest parallel in version 9.2.

Version 8.1 Features Not Supported in Version 9.2

These features are no longer supported. The following describes functionality that version 8.1

supported, but which must be rewritten in order for upgraded code to work in version 9.2.

XQuery Maps Are Not Supported

Impacts: Web services, Service controls

Version 9.2 doesn't support XQuery maps, a version 8.1 feature through which you could use XQuery to reshape XML messages entering and leaving a web service operation. This change impacts not only the XML shapes supported by the web service itself, but creates a mismatch between the web service and service controls generated from it or its WSDL file. One workaround is to rewrite your code so that its WSDL matches the existing WSDL shape without the use of maps.

Note: The lack of support for XQuery maps does not mean that XQuery itself is not supported. You can still execute XQuery expressions using the XMLBeans API. For more information on upgrade changes impacting this API, see [Updating XQuery Use to Support Upgraded XQuery Implementation](#).

For more information, see [General Steps for Replacing XQuery Maps](#).

java.util.Map Can Not Be Returned from Web Service Operations

Impacts: Web services

Version 8.1 supported returning instances of java.util.Map from web service operations. The runtime provided a WebLogic Workshop-specific serialization of the Map to and from XML. The schema for that serialization was included in the WSDL for the Web Service. In version 9.2, java.util.Map instances can no longer be returned from web service operations.

For a suggested workaround, see [Replacing the Use of java.util.Map as a Web Service Operation Return Type](#).

Multiple SOAP Versions are Not Supported for Bindings Defined in a Web Service

Impacts: Web services

Unlike version 8.1, version 9.2 doesn't support using multiple SOAP versions for bindings defined in a web service. When upgrading, you'll need to manually edit any web services that use more than one SOAP version so that they use only one.

Version 8.1 Implementation of SOAP 1.2 is Not Supported

Impacts: Web services

Version 8.1 included a SOAP 1.2 implementation that was based on a working draft of the SOAP

1.2 specification. The version 9.2 implementation is based on the final version of the specification, and so differs from the older implementation. To ensure compatibility for clients of a SOAP 1.2 web service you created with version 8.1, you should rebuild the client using a WSDL generated from an upgraded (version 9.2) version of the web service.

For more information on generating a WSDL, see [Upgrading from Version 8.1 Implementation of SOAP 1.2](#).

Non-SOAP XML Message Format Over HTTP or JMS is Not Supported

Impacts: Web services

If you have version 8.1 web services that use the non-SOAP XML format over HTTP or JMS, you must change your web service on version 9.2 so that it either uses the SOAP protocol or some alternative.

For information on changing the message format in web services, see [Details: Non-SOAP XML Message Format Over HTTP or JMS is Not Supported](#).

Handlers Not Supported for Callbacks

Impacts: Web services

In version 8.1 the `@jc:handler` and `@jws:handler` annotations included a callback attribute that specified handlers to process SOAP messages associated with callbacks; version 9.2 does not include callback-specific handler support. For the counterparts of these annotations in version 9.2, see [Upgrading Annotations](#).

form-get and form-post Message Formats are Not Supported to Receive Data

Impacts: Web services

Version 9.2 doesn't support using the form-get and form-post message formats to receive messages sent from an HTML form. When upgrading web services that use these formats, you'll need to use another method for receiving data sent from a form in a web browser.

For more information, see [Details: form-get and form-post Message Formats are Not Supported to Receive Data](#).

Multiple Protocols for Web Service Operations is Not Supported

Impacts: Web services

While version 9.2 supports multiple protocols at the web service level, it does not continue the version 8.1 support for multiple protocols at the *operation* level. When upgrading, if a single web

service has operations that use different protocols, you'll need to create separate web services and divide the operations among them so that a single protocol is used in each service.

For information on making this change, see [Upgrade Changes for Multiple Protocols in a Web Service](#).

Mixed Operations with Document and RPC SOAP Bindings Will Result in Namespace Differences

Impacts: Web services

If a version 8.1 web service includes one or more operations that use the RPC SOAP binding and one or more operations that use the document SOAP binding, then after upgrade types generated for those operations will be placed into different namespaces. This will be different from the version 8.1 web service itself, in which the types were in the same namespace. A WSDL generated from the upgraded web service will differ from the version 8.1-generated WSDL.

For more information, including a workaround, see [Resolving Namespace Differences from Mixed Operations with Document and RPC SOAP Bindings](#).

Deprecated UseWLW81BindingTypes and WLWRollbackOnCheckedException Annotations are Added to Upgraded Code

Supported but deprecated. Upgraded web service and Service control code will include the `@UseWLW81BindingTypes` and/or `@WLWRollbackOnCheckedException` annotations applied at the class level. Even though these annotations are deprecated, they are required in order to support clients that used the version 8.1 code.

Web Services or Service Controls Whose WSDLs Define Multiple Services is Not Supported

Impacts: Web services

In version 8.1 it was possible to have a web service (JWS) or Service control whose WSDL defined multiple services. The web service or control would represent only one of these services. When upgrading such code to version 9.2 upgrade will fail. To ensure that upgrade succeeds for this code, you should edit the WSDL so that it defines only the service that is represented by the JWS or Service control.

Multiple <wsdl:import> Elements are Not Supported

Even though it was supported for WSDLs associated with version 8.1 service controls, in version 9.2 multiple occurrences of the `<wsdl:import>` element is not supported in the same WSDL. For example, you might have used one import to get WSDL portions of the WSDL and another import to get XSD portions for needed types.

More information and a workaround is available at [Upgrading WSDLs with Multiple <wsdl:import> Statements](#).

Types in 1999 Schema Namespace Not Supported

Version 8.1 supported using types in the 1999 schema namespace for service controls and web services generated from WSDLs that used the types. Because version 9.2 does not support types in this namespace, you will need to manually migrate the WSDL to the 2001 namespace.

For more information, see [Updating WSDLs from 1999 Namespace to 2001](#).

EJB Control Security Annotations Not Upgraded by Upgrade Tools

Due to the way the EJB control works (which differs from other kinds of controls), the upgrade tools do not automatically upgrade security annotations used in an EJB control. You can work around this difference by manually editing the EJB control extension file to include the needed methods and annotations.

For more information and an example, see [Upgrading Security in EJB Controls](#).

EJB Control getJNDIName and setJNDIName Methods Invoked Differently

Impacts: Control use

Differences from the getJNDIName and setJNDIName methods exposed from the version 8.1 EJB control will cause errors in upgraded code. In version 8.1, these methods were exposed as EJBControl.getJNDIName and EJBControl.setJNDIName; in version 9.2, they are exposed from a build-time-generated control bean class as getJndiName and setJndiName (note the case difference also).

For information on fixing method invocation code, see [Upgrading EJB Control getJNDIName and setJNDIName Method Invocations](#).

JDBC Control Does Not Support "All" array-max-length Value for Query Results

The version 8.1 database control supported getting all rows for a query by specifying "all" as a value for the @jc:sql array-max-length attribute. The version 9.2 JDBC control does not support this value; instead, specify a numerical value. This change is not automatically made by upgrade tools.

For more information, see [Replacing "All" Requests for Database Control Results](#).

Control Factories are Not Supported

Impacts: Control use

Version 9.2 does not support control factories, a version 8.1 feature in which multiple control instances could be created at run time from a single control. If your code uses control factories, you'll need to replace the functionality with an alternate solution.

For a suggested alternative, see [Replacing Control Factory Functionality](#).

JMS Controls Can't Be Used to Receive Messages

Impacts: JMS controls

In version 9.2, a JMS control can't be used to receive messages. In upgraded code, you can work around this by developing a message-driven bean (MDB) to receive the messages or by invoking a web service using asynchronous request-response.

JMS Control sendJMSMessage Method Takes a Different Message Parameter Type

Version 8.1 of the JMS control took a `JMSControl.Message` type as a parameter for its `sendJMSMessage` method; in version 9.2 the method takes an instance of `javax.jms.Message`. For a full upgrade, you must change your code accordingly.

For more information, see [Upgrading JMS Control sendJMSMessage Method Invocations](#).

Multiple Calls to TimerControl.start Method Have No Effect

Impacts: Control use

In version 8.1 it was possible to call the timer control's start method multiple times, without error, to start the timer; however, the `onTimeout` callbacks did not necessarily correspond to the separate calls to the start method. The version 9.2 timer control simplifies the API by disallowing multiple calls to the start method. Calls to the start method when the timer control is still running will have no effect.

Controls Used in a Page Flow Can't Receive External Callbacks

Impacts: Control use, page flows

In version 9.x custom controls used from a page flow can't receive external callbacks. In version 8.1, you could write a custom control that received callbacks from controls nested within it, such as service controls. By exposing a polling interface from your custom control, your page flow code could then retrieve responses received from the callbacks. This functionality is not supported in version 9.2.

In upgraded code, you can substitute for this functionality by replacing your custom control with a

web service. For more information on the version 8.1 feature and suggested version 9.2 workarounds, see [Providing Support for Callbacks from a Page Flow](#).

Some Version 8.1 Context and Control Context Events and APIs are Not Available in Version 9.2

Impacts: Web services, control creation

These are events and APIs that were either considered low usage or were not general enough in nature to be included in the Beehive controls runtime.

In situations such as with the the logging API, you can modify your applications to make direct use of standalone APIs or WebLogic Server-provided APIs (such as the logging API from Apache Commons). However, there is currently no workaround for situations where the APIs were closely associated with the runtime container.

ControlException.getNestedException Method is No Longer Available

Impacts: Control use.

The version 8.1 [com.bea.control.ControlException](#) featured a `getNestedException` method that is not included on its Beehive counterpart, [org.apache.beehive.controls.api.ControlException](#). Code that calls this method will represent a compilation error after upgrade. Because this method merely delegated to the `getCause` method of the `Throwable` class — which the Beehive `ControlException` class extends — working around this change is as simple as changing the `getNestedException` call to `getCause`.

Version 8.1 Control Annotation Definitions are Not Upgraded

Impacts: Control creation

Version 8.1 custom control annotation definitions are not upgraded to version 9.2. The means for defining annotations is based on the Java 5 annotations model. To upgrade controls written for version 8.1, you must rewrite the annotations definition in keeping with the new model.

See [Upgrading Custom Controls Featuring Custom Properties](#) for more information.

Co-Location of Controller File and JSP Files is Not Supported

Impacts: Page flows

Version 8.1 supported a project hierarchy in which the Controller file (`Controller.jspf`) and JSP files could be put into the same directory. This is not supported in version 9.2. During upgrade your project hierarchy will be changed so that the Controller file is no longer co-located with JSP files.

For more information, see [Upgrade Changes for Co-Location in Page Flows](#).

IDE Sync Features Not Included in this Version

The version 8.1 IDE included a set of features through which the IDE kept related files in sync. For example, after generating a ServiceControl from a WSDL, changes to the WSDL would cause the IDE to automatically re-generate the ServiceControl to match. This functionality is not supported in the version 9.2 IDE.

For suggested workarounds, see [Keeping Files in Sync in the Absence of IDE Support](#).

Scenarios Not Supported by Upgrade

Custom Tags That Extend NetUI Tags are Not Supported

If you created custom JSP tags in version 8.1 by extending NetUI tags, your tags will not be upgraded by Workshop for WebLogic tools. Extending NetUI tags was not supported. Note that if you elected not to migrate NetUI tags to Beehive tags, your tags may build within the application, but may not work as expected.

Likewise, extending Beehive JSP tags in version 9.2 is not supported.

IDE Erroneously Shows an Error When Variable Declarations are Made in an HTML Start Tag

The version 9.2 IDE incorrectly displays an error for code in which a variable declaration is made in an HTML start tag; the code is actually valid at run time. For example, in the following anchor tag the `<%=s%>` code using the String variable is flagged by the IDE as unresolved, but is actually valid.

[Fixing Erroneous IDE Error for Variable in an HTML Start Tag](#)

Related Topics

[Overview: Upgrading from WebLogic Workshop 8.1](#)

Upgrading Controls

This topic gives more detail about upgrade changes noted for controls.

For other upgrade issues related to controls, see the following topics:

[Replacing Callback-Enabled Controls Used in a Page Flow](#)

For a more complete list of changes affecting applications upgraded from version 8.1, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#).

Enabling Automatic Transaction Support in Controls

In version 8.1, controls are automatically run within the scope of a transaction because they're run in the context of an Enterprise JavaBean. In version 9.2 controls are Plain Old Java Objects (POJOs). As a result, if you want transaction support, your control interfaces must be annotated with the `@TransactionAttribute` annotation for transaction support.

During upgrade, [upgrade tools](#) will add an annotation for transaction support. The following version 8.1 and version 9.2 examples show a very simple control interface before and after upgrade.

In version 8.1, no annotation was required:

```
package localControls.nestedControls;

import com.bea.control.Control;

/**
 * This is the public interface for the VerifyFunds control.
 * The control is implemented in VerifyFundsImpl.jcs.
 */
public interface VerifyFunds extends Control
{
    interface Callback
    {
        void onTransactionComplete(String message, boolean isBalanceAvailable,
            boolean isInventoryAvailable);
    }

    /**
     * @common:operation
     */
    void submitPO(java.lang.String poNumberString,
        java.lang.String customerIDString, int itemNumber,
        int quantityRequested, double startingBalance);
}
```

In version 9.2, the `@TransactionAttribute` annotation signifies that transaction support is requested:

```
package localControls.nestedControls;

import com.bea.control.annotations.TransactionAttribute;
import com.bea.control.annotations.TransactionAttributeType;
import org.apache.beehive.controls.api.bean.ControlInterface;
import org.apache.beehive.controls.api.events.EventSet;

/**
 * Public interface for the VerifyFunds control.
 */
@ControlInterface()
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public interface VerifyFunds
{
    @EventSet(unicast = true)
    interface Callback
    {
        void onTransactionComplete(String message, boolean isBalanceAvailable,
            boolean isInventoryAvailable);
    }

    void submitPO(java.lang.String poNumberString,
```



```

        java.lang.String customerIDString, int itemNumber,
        int quantityRequested, double startingBalance);
}

```

Replacing Control Factory Functionality

Version 9.x does not support control factories, a version 8.1 feature in which multiple control instances could be created at run time from a single control. If your code uses control factories, you'll need to replace the functionality with an alternate solution.

You can support control factory-like functionality by using control-related APIs through which you explicitly instantiate multiple control instances. The following example illustrates by using a simplified version of the control factory example that was included with the version 8.1 samples application.

```

package controlfactory;

import java.util.HashMap;
import java.util.Map;
import java.io.Serializable;
import javax.jws.WebMethod;
import javax.jws.WebService;
import org.apache.beehive.controls.api.bean.ControlReferences;
import org.apache.beehive.controls.api.bean.Controls;
import weblogic.jws.Conversation;
import weblogic.jws.WLHttpTransport;
import javax.jws.soap.SOAPBinding;
import controlfactory.SlowServiceControlBean;
import controlfactory.SlowServiceControl;

/**
 * The following code demonstrates how you can achieve version 8.1 control
 * factory functionality by using APIs associated with controls.
 *
 * The code in this example is a somewhat simplified version of the control
 * factory example included in the version 8.1 SamplesApp application.
 */
@WebService(serviceUri = "controlfactory/ServiceFactoryClient.jws")
@WebService(serviceName = "ServiceFactoryClient",
    targetNamespace = "http://workshop.bea.com/ServiceFactoryClient")
@javax.jws.soap.SOAPBinding(style = javax.jws.soap.SOAPBinding.Style.DOCUMENT,
    use = javax.jws.soap.SOAPBinding.Use.LITERAL,
    parameterStyle = javax.jws.soap.SOAPBinding.ParameterStyle.WRAPPED)
@ControlReferences(SlowServiceControl.class)
public class ServiceFactoryClient implements java.io.Serializable {
    static final long serialVersionUID = 3553L;

    static Map<String, Long> serviceControlsMap = new HashMap<String, Long>();

    int m_numServices;

    /**
     * This method does the work of creating a quantity of control instances
     * based on a number received in the numServices param.
     */
    @Conversation(Conversation.Phase.START)
    @SOAPBinding(style = javax.jws.soap.SOAPBinding.Style.DOCUMENT,
        use = javax.jws.soap.SOAPBinding.Use.LITERAL,
        parameterStyle = javax.jws.soap.SOAPBinding.ParameterStyle.WRAPPED)
    @WebMethod()
    public void startServices(int numServices) {
        int i;

        if (numServices > 0) {
            try {
                for (i = 0; i < numServices; i++) {
                    m_numServices = numServices;

                    // Instead of using a control factory, instantiate separate
                    // instances of the control using the Controls.instantiate
                    // method.
                    SlowServiceControlBean bean;
                    bean = (SlowServiceControlBean) Controls.instantiate(Thread
                        .currentThread().getContextClassLoader(),
                        "controlfactory.SlowServiceControlBean", null);

                    // Also pair an instance of the callback handler

```

```

        // for each new control instance. This handler will listen
        // for the service's infoReady callback.
        SlowServiceCallbackHandler handler = new SlowServiceCallbackHandler(
            bean);
        bean.addCallbackListener(handler);

        // Call the Service control, passing a unique value to label
        // it for identifying later.
        String serviceName = "Service" + i;
        bean.requestInfo(serviceName);

        // Stash the time each control is launched, indexed by
        // instance name
        serviceControlsMap.put(serviceName, new Long(
            (new java.util.Date().getTime())));
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}

/**
 * Callback handler for the control's infoReady callback.
 */
public class SlowServiceCallbackHandler implements
    SlowServiceControl.Callback, Serializable {
    static final long serialVersionUID = 1L;

    private SlowServiceControlBean serviceControl;

    public SlowServiceCallbackHandler(SlowServiceControlBean control) {
        serviceControl = control;
    }

    public void infoReady(String name) {
        // Compute how many seconds since this control was launched.
        long timeTaken =
            (new java.util.Date().getTime() - ((Long) serviceControlsMap
                .get(name)).longValue()) / 1000;

        // A ControlBean instance provides many methods useful for
        // getting information about the instance.
        String controlId = serviceControl.getControlID();

        // Print the information discovered.
        printControlInfo(name, controlId, timeTaken);
    }

    public void onAsyncFailure(String arg0, Object[] arg1) {
        // TODO Auto-generated method stub
    }
}

@WebMethod
@Conversation(Conversation.Phase.FINISH)
public String finishServices() {
    return "Finished. " + m_numServices + " services invoked.";
}

public void printControlInfo(String serviceName, String controlId, long time) {
    System.out.println("Control callback received from " + serviceName
        + ":" + controlId + " after " + time + " seconds.");
}
}
}

```

This example includes the following APIs:

[org.apache.beehive.controls.api.bean.ControlReferences](#)

[org.apache.beehive.controls.api.bean.Controls](#)

[weblogic.jws.Conversation](#)

[weblogic.jws.WLHttpTransport](#)[javax.jws.soap.SOAPBinding](#)

Associating a Service Control with a WSDL

While version 8.1 allowed a service control to not be associated with a WSDL, the control must be associated with a WSDL in order for it to upgrade successfully.

You can associate the WSDL with the service control in WebLogic Workshop 8.1 before you upgrade your application. The easiest way to do this is by regenerating the Service control from the WSDL. In the Application tab, right-click the WSDL, then click Generate Service Control.

You can also associate the WSDL manually by pasting it into the Service control. Open the Service control in Source View, scroll to the end of the file and paste the WSDL's contents as the value of a `@common:define` annotation after all other code. Note in the following example that the value attribute encloses its value in double colons, and that the Javadoc comment continues after it.

```
/** @common:define name="MyServiceWsdL" value::
    ... WSDL contents ...
 * ::
 */
```

After adding the WSDL, add a `@jc:wsdl` annotation to the control declaration as follows:

```
/**
 * <other annotations>
 * @jc:wsdl file="#MyServiceWsdL"
 */
public interface MyServiceControl extends ControlExtension, ServiceControl
```

Note that the value of the file attribute is the same as the value of the `@common:define` name attribute, but with a `#` sign prepended.

Upgrading Service Controls that are Based on an Abstract WSDL

In version 8.1 it was possible to generate a service control from an abstract WSDL — that is, from a WSDL with no service definition. You could then configure the endpoint to call or listen on programmatically or by using annotations. However, WebLogic Server version 9.2 does not support abstract WSDLs. As a result, Workshop for WebLogic's [upgrade tools](#) are unable to upgrade a service control with the necessary annotations, leaving compiler errors in upgraded code. Likewise, if you try to generate a service control from an abstract WSDL in 9.x, you will receive an error stating that a service is required.

One workaround is to add a `<service>` definition into the WSDL before using [upgrade tools](#). That added entry will not be used in upgraded code because the endpoints will be taken from the annotations instead.

Replacing Service Control Methods `getEndPoint` and `setEndPoint`

The `ServiceControl.getEndPoint()` and `ServiceControl.setEndPoint(URL)` methods are deprecated in version 9.2 and may be removed in a future version. New code requiring this kind of API should use `ServiceControl.getEndpointAddress()` and `ServiceControl.setEndpointAddress(String)`, respectively.

Note that a `URL` instance (as used in the version 8.1 methods) isn't required in order to support the most common usage for these methods; a `String` instance suffices. The `get*` method is useful in debugging, providing a way to retrieve and log the endpoint location. The `set*` method is useful for dynamically configuring the endpoint location at run time, such as when the destination is on another server/cluster.

Configuring Run-Time Message-Level Security Via the Service Control

Due to changes in the web services security model, the means for specifying security characteristics through the version 9.2 service control differs from version 8.1. In version 8.1, you specified both security policies and values in a WebLogic Workshop WSSE policy file. In version 9.2, the means for specifying characteristics for these two aspects of security has been split into multiple locations.

Briefly, the model for specifying aspects of security in the version 9.2 service control is as follows:

- Security policies -- that is, the descriptions of what can be allowed -- are specified in either of the following ways:
 - By associating a WS-Policy file with the service control via the `@weblogic.jws.Policy` annotation.
 - By generating the service control from a WSDL that is aware of the policy.
- Security values -- that is, the descriptions of how security should be handled based on policies -- are specified in one of the following

ways:

- By using a credential mapper. (Note that when using usertoken in the credential mapper, each user must be explicitly mapped to a remote user; you can't specify group-level mappings.)
- By using the following methods exposed by the service control itself, as described in the [ServiceControl](#) reference:
 - setUsername(String username)
 - setPassword(String password)
 - setMessageUsername(String username)
 - setMessagePassword(String password)
 - setClientMessageCert(String alias, String password)
 - setServerMessageCert(X509Certificate cert)

Repairing Service Control JMS URL After Upgrade

When upgrading a service control, upgrade tools do not correctly upgrade a JMS URL specified in a `@jc:location` annotation. You will need to manually edit the URL. For example, compare the following URLs from version 8.1 and its upgraded counterpart:

Version 8.1

```
@jc:location jms-url="jms://localhost:7001/weblogic.jws.jms.QueueConnectionFactory/jws.queue?URI=/services/MyService.jws&java.naming.factory.initial=com.myco.jndi.factory"
```

Version 9.2

```
@ServiceControl.Location(urls = {
    "jms://localhost:7001/services/MyService.jws&java.naming.factory.initial=com.myco.jndi.factory?URI=jws.queue,FACTORY=weblogic.jws.jms.QueueConnectionFactory"
})
```

To fully upgrade the URL, make the following edits:

- If the JMS URL in your version 8.1 service control did not specify a connection factory, remove the FACTORY parameter from the upgraded URL. (In version 8.1, the factory was typically specified in the `jws-config.properties` file.) This will prompt the control to use the default factory or the factory specified for the domain. If your version 8.1 service control specified a connection factory, leave the parameter in the upgraded URL, but be make the change described below.
- If your upgraded URL will specify a connection factory, be sure to separate the FACTORY query string parameter from the rest of the URL with an ampersand (replacing the comma in the URL generated by upgrade tools).

After edits, your JMS URL might look like the following (if you elect to specify the connection factory):

```
@ServiceControl.Location(urls = {
    "jms://localhost:7001/services/MyService.jws&java.naming.factory.initial=com.myco.jndi.factory?URI=jws.queue&FACTORY=weblogic.jws.jms.QueueConnectionFactory"
})
```

Upgrading EJB Control getJNDIName and setJNDIName Method Invocations

Differences from the `getJNDIName` and `setJNDIName` methods exposed from the version 8.1 EJB control will cause errors in upgraded code. In version 8.1, these methods were exposed as `EJBControl.getJNDIName` and `EJBControl.setJNDIName`; in version 9.2, they are exposed from a build-time-generated control bean class as `getJndiName` and `setJndiName` (note the case difference also).

You should be able to correct upgraded code by replacing the method invocations. For example, in version 8.1 you would have call these methods from an instance of the control, as follows:

```
import mypackage.MyEJBControl;
...
private MyEJBControl ejbControl;

public String getEJBJNDIName()
{
    String jndiName = ejbControl.getJNDIName();
}
```

In version 9.2 you will call the method from an instance of a control bean class for your EJB control. (Note that the control bean isn't generated until you are able to successfully build the project.)

```
import mypackage.MyEJBControlBean;
...
private MyEJBControlBean ejbControlBean;

public String getEJBJNDIName()
{
    String jndiName = ejbControlBean.getJndiName()
}
```

Upgrading Security in EJB Controls

Due to the how the EJB control works (which differs from other kinds of controls), the upgrade tools do not automatically upgrade security annotations used in an EJB control. You can work around this difference by manually editing the EJB control extension file to include the needed methods and annotations.

Security annotations on controls are no longer supported at the class level. For controls that had security annotations at the class level, the upgrader will copy the security annotation to each method defined in the control. However, in the case of the EJB control, there are no methods defined directly in the control extension (the JCX file in version 8.1). Instead, the methods are inherited from the EJB home and interface definitions. Because omitting these annotations creates a security gap, upgrade tools will generate an error for any EJB control with security annotations.

To work around this difference, you can manually copy the EJB's methods into the control extension file and add the [com.bea.control.annotations.Security](#) annotation to each method. It is important to remember that any subsequent change to the EJB itself (such as adding new methods) may require updating the control extension to ensure that the security constraint formerly defined at the class level remains in force.

The following template-style example shows the kind of changes to make:

```
/**
 * @common:security [attributes here]
 */
public class MyEjb extends EJBControl, MyHome, MyLocal [other types] {
    // Nothing; methods were derived from the home and interface definitions.
}
```

After upgrade, you should edit the upgraded control extension so that the methods are declared along with the corresponding version 9.2 annotation on them:

```
public class MyEjb extends EJBControl, MyHome, MyLocal [other types] {

    @Security [attributes here]
    void methodFromHome()
    @Security [attributes here]
    void methodFromBean()
}
```

Replacing JMS Control Receive Functionality

In version 9.x, a JMS control can't be used to receive messages. During upgrade, if version 8.1 `@jc:jms` annotations include attributes that specify message receiving behavior, these attributes will be ignored. For example, the following version 8.1 annotation will be migrated to the version 9.2 annotation that follows it.

Version 8.1

```
@jc:jms receive-type="topic" receive-jndi-name="jms.AccountUpdate"
connection-factory-jndi-name="weblogic.jws.jms.QueueConnectionFactory"
```

Upgrade to version 9.2

```
@JMSControl.Destination(jndiConnectionFactory = "weblogic.jws.jms.QueueConnectionFactory")
```

Here are two suggestions for working around this in upgraded code:

If it's possible to update the receiving application, then replacing the receiving application with a JWS that has a callback would make it possible to replace the JMS control with a Service Control. If the API (between systems) included the use of JMS or user properties, then those properties would have to be added to the definition of the message(s) or added as custom SOAP headers.

If it's not possible to update the receiving application, then an alternative approach is to add a Timer Control to the calling application and use the Timer events to trigger polling for response messages. When a response message is present, the JMS message (and any relevant properties) would be manipulated (as needed) to match the signature of the event method. This functionality could be encapsulated in a custom control to minimize the impact on the calling application.

Upgrading JMS Control sendJMSMessage Method Invocations

Version 8.1 of the JMS control took a `JMSControl.Message` type as a parameter for its `sendJMSMessage` method; in version 9.2 the method takes an instance of `javax.jms.Message`. For a full upgrade, you must change your code accordingly.

Note that in upgrading this code, which uses the `@org.apache.beehive.controls.system.jms.JMSControl.Message` annotation, you may see the type ambiguity issue described in [Resolving Ambiguity Related to Annotation Types](#). See that section for information on resolving the issue with a fully-qualified type name.

Supporting Parameter Bindings for JMS Properties with the JMS Control

Upgrade tools do not fully upgrade JMS control support for binding control method parameters to JMS properties. In particular, the method parameters themselves must be annotated but aren't. You can ensure support for these bindings by manually adding the required annotations after upgrading your version 8.1 code.

In the following example, note that the `accountID` parameter is annotated with `@JMSControl.Property` to indicate which JMS property the parameter should be bound to.

```
@JMSControl.Properties({
    @JMSControl.PropertyValue(name = "transactionType",
        value = "DEPOSIT")
})
public void deposit(
    AccountTransaction transaction,
    @JMSControl.Property(name="accountIdentifier") String accountID);
```

Changes to Support Database Control Row Set Functionality

The standard Beehive `JdbcControl` ([org.apache.beehive.controls.JdbcControl](#)) in version 9.2, which corresponds to the version 8.1 [Database control](#), does not support the version 8.1 "RowSet control" feature. To ensure that row set functionality is preserved during upgrade, the Database control is upgraded to the backward compatible `JdbcControl` ([com.bea.control.JdbcControl](#)) provided by BEA.

If your upgraded application *does not* use the row set functionality, it is recommended that you update, after upgrading, from the backward compatible control to the Beehive control. If your upgraded application *does* use the row set functionality, note that this functionality is now represented by the `JbcControl.SQLRowSet` annotation's `rowsetSchema` attribute.

Note that when you insert a new `JdbcControl`, you'll get the Beehive control.

Replacing "All" Requests for Database Control Results

The version 8.1 database control supported getting all rows for a query by specifying "all" as a value for the `@jc:sql array-max-length` attribute. The version 9.2 JDBC control does not support this value; instead, specify a numerical value. This change is not automatically made by upgrade tools.

For example, you might make the following change:

Version 8.1: `@jc:sql array-max-length="all" statement="SELECT * FROM Customers"`

Version 9.2: `@JdbcControl.SQL(arrayMaxLength = 1024, statement = "SELECT * FROM Customers")`

Multiple Calls to TimerControl.start Method Have No Effect

In version 8.1 it was possible to call the timer control's `start` method multiple times, without error, to start the timer; however, the `onTimeout` callbacks did not necessarily correspond to the separate calls to the `start` method. The version 9.2 timer control simplifies the API by disallowing multiple calls to the `start` method. Calls to the `start` method when the timer control is still running will have no effect.

Upgrading Exception Handling in Control Event Handlers

In version 8.1, control event handlers (known in that version as "callback handlers") could throw exceptions of any type. In version 9.2, if the event handler throws the caught exception, it must now throw a proper subset of the throws clause declared for the control `EventSet` method.

Your event handler code can work around the version 9.2 requirement by modifying event handler exception handling after you have upgraded your application. Here are two suggestions:

- Add the throws clause your event handler is throwing to the EventSet event method. You will also need to add a corresponding try/catch block to code that invokes the event method.
- Remove the throws clause from the event handler code and implement another way to handle exceptions generated from the event.

Replacing Calls to Unsupported `ControlException.getNestedException` Method

The version 8.1 `com.bea.control.ControlException` featured a `getNestedException` method that is not included on its Beehive counterpart, `org.apache.beehive.controls.api.ControlException`. Code that calls this method will represent a compilation error after upgrade. Because this method merely delegated to the `getCause` method of the `Throwable` class — which the Beehive `ControlException` class extends — working around this change is as simple as changing the `getNestedException` call to `getCause`.

Upgrading Custom Controls Featuring Custom Properties

Version 8.1 custom control annotation definitions are not upgraded to version 9.2. The means for defining annotations is based on the Java 5 annotations model. To upgrade controls written for version 8.1, you must rewrite the annotations definition in keeping with the new model.

For more information on upgrading your custom annotations, take a look at the Apache Beehive source code for its system controls. These provide annotations that use the new model.

For information on how the control context APIs have changed from version 8.1, see [Handling Context API Changes](#).

Upgrading Message Buffering in Custom Controls

In version 8.1 you could apply the `common:message-buffer` tag to a custom control's interface or implementation code. In version 9.2 this annotation's counterpart, `com.bea.control.annotations.MessageBuffer`, is supported only in the control interface code. To work around this change, you should remove the annotation from implementation code before upgrading the application.

Handling Context API Changes

Version 8.1 provided context APIs through which components such as web services (in version 8.1 JWS files) and custom controls could interact with their runtime environment. The following provides an overview of how support for these APIs has (or hasn't) been migrated to version 9.2.

Web Service Context Changes

In general, the role played by the `com.bea.control.JwsContext` interface now belongs to `weblogic.wsee.jws.JwsContext`, used in conjunction with the `weblogic.jws.Context` annotation (both are described on the edocs web site).

However, some of the version 8.1 APIs were deprecated and are unavailable in version 9.2. Examples include callback-oriented methods on `com.bea.control.JwsContext` such as `getCallbackLocation`, `getCallbackPassword`, and others. For counterparts to these methods useful in version 9.2, see [weblogic.wsee.jws.CallbackInterface](#). For example, you can replace the `getCallbackLocation` method with the `CallbackInterface.getEndpointAddress` method.

Control Context Changes

Changes brought by the Apache Beehive control model to version 9.2 have meant that several APIs exposed by the version 8.1 control context classes are either exposed in different ways or are no longer relevant and so not available. Note that in many cases the lack of a workaround is due to the fact the Beehive model streamlines a control's role for interacting with aspects of its container's environment. In particular, conversation-related methods are no longer supported within a control.

The following table lists the version 8.1 control context-related classes and methods along with their version 9.2 counterparts, if any.

Version 8.1 Method	Version 9.2 Counterpart
Context	
finishConversation() : void	No workaround.
getCallerPrincipal() : Principal	See <code>com.bea.control.util.SecurityHelper.getCallerPrincipal()</code> .
getCurrentAge() : long	No workaround.
getCurrentIdleTime() : long	No workaround.
getLogger(String name) : Logger	Use logging services provided with WebLogic Server. For more information, see Understanding WebLogic Logging Services and Configuring WebLogic Logging Services .
getMaxAge() : long	No workaround.
getMaxIdleTime() : long	No workaround.
getService() : ServiceHandle	No workaround.
isCallerInRole(String role) : boolean	See <code>com.bea.control.util.SecurityHelper.isCallerInRole()</code> .
isFinished() : boolean	No workaround.
resetIdleTime() : void	No workaround.
setMaxAge(Date date) : void	No workaround.
setMaxAge(String duration) : void	No workaround.
setMaxIdleTime(long) : void	No workaround.
setMaxIdleTime(String duration) : void	No workaround.
Context.Callback	
onAgeTimeout(long age) : void	No workaround.
onAsyncFailure(String methodName, Object[] args) : void	No workaround.
onCreate() : void	See ControlBeanContext.Lifecycle.onCreate()
onException(Exception e, String methodName, Object[] args) : void	No workaround.
onFinish(boolean expired) : void	No workaround.
onIdleTimeout(long time) : void	No workaround.
ControlContext	
cancelEvents(String eventName) : void	No workaround.
getCallbackInterface() : Class	You can use the Java reflection API (java.lang.reflect) to retrieve the callback interface. Here's a brief example: <pre> Class controlInterface = context.getControlInterface(); Class callbackInterface = null; for(Class c : controlInterface.getDeclaredClasses()) { if("Callback".equals(c.getSimpleName()) && c.getAnnotation(EventSet.class) != null) { callbackInterface = c; System.out.println("Callback Interface: " + c.getName()); } } </pre>
getControlAttribute(String tagName, String attrName) : String	See ControlBeanContext.getControlPropertySet(Class<T> propSet)
getControlAttributes(String tagName) : List	No counterpart. The version 9.2 annotation model is based on JSR175, which doesn't support multiple attributes of the same name on an annotation. To work around this change, redesign your annotation so that it is a single annotation that can contain an array of annotations with the attribute that used to be repeated. For example, with the Beehive JMS control you can specify multiple property values by using multiple <code>@JMSControl.PropertyValue</code> annotations within a <code>@JMSControl.Properties</code> annotation: <pre> @JMSControl.Properties({ @JMSControl.PropertyValue(name="Property1", value="SomeValue") @JMSControl.PropertyValue(name="Property2", value="{arg2}") }) public void sendMessage(String arg1, String arg2) </pre> <p>Access to the property values would be through the <code>JMSControl.Properties</code> annotation rather than a context API.</p>
getControlInterface() : Class	See ControlBeanContext.getControlInterface()
getMethodArgument(String argName) : Object	Use Java reflection (see java.lang.reflect at the Sun web site).

getMethodArgumentNames() : String[]	See the Beehive method ControlBean.getParameterNames(Method method) .
getMethodAttribute(String tagName, String attrName) : String	See ControlBeanContext.getMethodPropertySet(Method m, Class<Annotation> propSet)
getMethodAttributes(String tagName) : List	<p>No counterpart. The version 9.2 annotation model is based on JSR175, which doesn't support multiple attributes of the same name on an annotation. To work around this change, redesign your annotation so that it is a single annotation that can contain an array of annotations with the attribute that used to be repeated. For example, with the Beehive JMS control you can specify multiple property values by using multiple <code>@JMSControl.PropertyValue</code> annotations within a <code>@JMSControl.Properties</code> annotation:</p> <pre>@JMSControl.Properties({ @JMSControl.PropertyValue(name="Property1", value="SomeValue") @JMSControl.PropertyValue(name="Property2", value="{arg2}") }) public void sendAMessage(String arg1, String arg2)</pre> <p>Access to the property values would be through the <code>JMSControl.Properties</code> annotation rather than a context API.</p>
raiseEvent() : Object	See the workaround for the <code>sendEvent</code> method below. A workaround for <code>raiseEvent</code> would involve executing similar code in the event handler for the event that should be raised (forwarded to the client). The code would find the current event name and arguments and forward it to the client by invoking the method through reflection.
scheduleEvent(String eventName, Object[] eventArgs, long time, boolean ignoreIfFinished) : void	See the workaround for the <code>sendEvent</code> method below. A workaround for <code>scheduleEvent</code> would involve executing similar code in the <code>onTimeout</code> event handler for a timer control.
sendEvent(String eventName, Object args[]) : Object	<p>You can work around the absence of <code>sendEvent</code> with the code below. Compare the following two code snippets.</p> <p>Version 8.1</p> <pre>context.sendEvent("mycallback", new String[]{"Hi from the callback (sendEvent)"});</pre> <p>Version 9.2</p> <pre>try { Method eventMethod = callback.getClass().getMethod("mycallback", new Class[] { String.class }); if (eventMethod != null) { eventMethod.invoke(callback, new Object[]{"Hi from the callback (invoke)"}); } } catch (Exception e) { System.err.println("Exception trying to 'sendEvent': " + e.getMessage()); }</pre>
ControlContext.Callback	
onAcquire() : void	See ResourceContext.ResourceEvents.onAcquire()
onRelease() : void	See ResourceContext.ResourceEvents.onRelease()
onReset() : void	No workaround.

Related Topics

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)

Upgrading Web Services

This topic gives more detail about upgrade changes noted for web services.

For a more complete list of changes affecting applications upgraded from version 8.1, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#).

Deprecated Use `WLW81BindingTypes` and `WLWRollbackOnCheckedException` Annotations are Added to Upgraded Code

Supported but deprecated. Upgraded web service and Service control code will include the `@UseWLW81BindingTypes` and/or `@WLWRollbackOnCheckedException` annotations applied at the class level. Even though these annotations are deprecated, they are required in order to support clients that used the version 8.1 code.

Upgrade Changes for Web Services That Combine Stateful and Stateless Operations

Supported but deprecated. In version 9.x stateless operations must be marked with the `NONE` phase constant. During upgrade, [upgrade tools](#) will ensure that all operations are legal by adding an annotation to the version 8.1 stateless operations so that they are marked with `@Conversation(value = Conversation.Phase.NONE)`.

Note: The `NONE` phase constant is deprecated, included for backward compatibility only. To ensure that your web services functionality is supported in future releases, you should plan to rewrite the service so that stateless and stateful functionality is placed into separate web services.

Version 8.1. For example, the following version 8.1 snippet shows a stateful operation preceding a stateless operation.

```
/**
 * @common:operation
 * @jws:conversation phase="start"
 */
public void startShopping(int customerId)
{
    ... stateful ...
}

// Other stateful operations, including a FINISH operation.

/**
 * @common:operation
 */
public String buyWithOneClick(int customerId)
{
    ... stateless ...
}
```

Version 9.2. When upgraded, this would result in something like the following:

```
@Conversation(value = Conversation.Phase.START)
@WebMethod()
@WebResult(name = "startShoppingResult")
public void startShopping(int customerId)
{
    ...
}

// Other stateful operations, including a FINISH operation.
```

```

@Conversation(value = Conversation.Phase.NONE)
@WebMethod()
@WebResult(name = "noPhaseResult")
public String butWithOneClick(int customerId)
{
    ...
}

```

Ensuring START and FINISH Methods for Conversations

In version 8.1 it was possible to compile a "conversational" web service that did not have START or FINISH operations. In other words, you could annotate the web service class as conversational (setting, for example, the annotation's `maxAge` attribute) but not annotate any of the web service's operations with conversation phase attributes. In version 9.2 a conversational web service must have both a START and FINISH operation in order to compile.

Setting Conversation Phase

In version 8.1, you can set conversation phase by opening the web service in Design view, then selecting operations and setting their phase attribute each in turn. In version 9.2, open the web service source code and place a cursor at the operation whose phase attribute you want to set, then locate and set the Conversation value in the Annotations view. You can also apply the annotation in source, as shown here for a START method:

```

@WebMethod()
@Conversation(Conversation.Phase.START)
public void startConversation()
{
    ....
}

```

Multiple SOAP Versions are Not Supported for Bindings Defined in a Web Service

Unlike version 8.1, version 9.x doesn't support using multiple SOAP versions for bindings defined in a web service. When upgrading, you'll need to manually edit any web services that use more than one SOAP version so that they use only one.

Upgrading from Version 8.1 Implementation of SOAP 1.2

Version 8.1 included a SOAP 1.2 implementation that was based on a working draft of the SOAP 1.2 specification. The version 9.2 implementation is based on the final version of the specification, and so differs from the older implementation. After you upgrade a web service that uses the version 8.1 SOAP 1.2 implementation, the service's clients will no longer be able to use it.

To ensure compatibility for clients of a SOAP 1.2 web service you created with version 8.1, you should rebuild the client using a WSDL generated from an upgraded (version 9.2) version of the web service. In version 9.2, you can generate a WSDL by right-clicking the web service file in Package Explorer view, then clicking Generate WSDL.

Details: Non-SOAP XML Message Format Over HTTP or JMS is Not Supported

If you have version 8.1 web services that use a non-SOAP XML format over HTTP or JMS, you must change your web service on version 9.x so that it either uses the SOAP protocol or some alternative. In other words, version 9.2 web services do not support message formats that do not include SOAP headers.

In version 8.1, the `@jws:protocol` annotation supported the following attributes and values:

- `@jws:protocol http-xml="true"` — Indicated that the operation or web service supported receiving XML messages over HTTP, without SOAP headers.
- `@jws:protocol jms-xml="true"` — Indicated that the operation or web service supported receiving XML messages over the Java Message Service (JMS), without SOAP headers.

These attributes and values have no counterparts in version 9.2. If you upgrade to version 9.2, [upgrade tools](#) will simply ignore a protocol setting that isn't supported.

Details: form-get and form-post Message Formats are Not Supported to Receive Data

Version 9.x doesn't support using the form-get and form-post message formats to receive messages sent from an HTML form. When upgrading web services that use these formats, you'll need to use another method for receiving data sent from a form in a web browser.

In other words, version 9.2 web services do not support message formats that do not include SOAP headers.

In version 8.1, the `@jws:protocol` annotation supported the following attributes and values:

- `@jws:protocol form-get="true"` — Indicated that the operation or web service supported receiving HTTP GET requests.
- `@jws:protocol form-post="true"` — Indicated that the operation or web service supported receiving HTTP POST requests.

These attributes have no counterparts in version 9.2 and there are no suggested workarounds. If you upgrade to version 9.2, [upgrade tools](#) will simply ignore a protocol setting that isn't supported.

Upgrade Changes for Multiple Protocols in a Web Service

While version 9.x supports multiple protocols at the web service level, it does not continue the version 8.1 support for protocols set at the operation level. For example, in version 8.1 the following annotation was supported on a web service operation:

```
@jws:protocol jms-soap="true"
public void myOperation()
```

During upgrade, this annotation will be interpreted as a desire to support web service invocations over JMS. Your upgraded code will feature an annotation at the web service level, but not at the operation level:

```
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT,
              use = SOAPBinding.Use.LITERAL,
              parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
@WljmsTransport(queue = "jws.queue",
               serviceUri = "myservices/MyWebService.jws")
@WebService(serviceName = "MyWebService",
           targetNamespace = "http://www.openuri.org/")
public class MyWebService
{
    ...
}
```

If supporting multiple protocols for different operations is a requirement, consider splitting your web services code into multiple web services, each with its own protocol support. For example, operations requiring JMS support would go into a web service designed for that purpose, while operations to be invoked over HTTP would go into another web service.

Resolving Namespace Differences from Mixed Operations with Document and RPC SOAP Bindings

If a version 8.1 web service includes one or more operations that use the RPC SOAP binding and one or more operations that use the document SOAP binding, then after upgrade types generated for those operations will be placed into different namespaces. This will be different from the version 8.1 web service itself, in which the types were in the same namespace. A WSDL generated from the upgraded web service will differ from the version 8.1-generated WSDL.

Note that this issue will probably only arise for web services that were written from scratch in Java, rather than generated from an existing WSDL file. Also, this may not be an issue at all if it is not important that your web service interface honor a WSDL.

If you need to honor a WSDL contract, you can work around this issue by generating a WSDL in version 8.1, then creating a new web service in version 9.2 from that WSDL. You can then copy and paste the implementation code from the version 8.1 service to the version 9.2 service.

Upgrading Web Services or Service Controls Whose WSDLs Define Multiple Services

In version 8.1 it was possible to have a web service (JWS) or Service control whose WSDL defined multiple services. The web service or control would represent only one of these services. When upgrading such code to version 9.2 upgrade will fail. To ensure that upgrade succeeds for this code, you should edit the WSDL so that it defines only the service that is represented by the JWS or Service control.

Upgrading WSDLs with Multiple <wsdl:import> Statements

Even though it was supported for WSDLs associated with version 8.1 service controls, in version 9.2 multiple occurrences of the <wsdl:import> element is not supported in the same WSDL. For example, you might have used one import to get WSDL portions of the WSDL and another import to get XSD portions for needed types.

You can work around this change by, before upgrading, including only one <wsdl:import> statement whose namespace attribute value is the namespace of the imported WSDL. The WSDL and XSD portions will both be imported with the single statement.

Ensuring Correct Handling of xs:anyType in Messages

If you created a version 8.1 web service by generating it from a WSDL that specified xs:anyType instead of xs:any, the web service will expect and send incorrect XML payloads after upgrade to version 9.2. You can ensure correct handling of xs:anyType by applying the following annotation to the web service at the class level:

```
@WildcardBindings(
    @WildcardBinding(className="org.apache.xmlbeans.XmlObject",
        binding=WildcardParticle.ANYTYPE)
)
```

Updating WSDLs from 1999 Namespace to 2001

Version 8.1 supported using types in the 1999 schema namespace for Service controls and web services generated from WSDLs that used the types. Because version 9.2 does not support types in this namespace, you will need to manually migrate the WSDL to the 2001 namespace. In simple cases, this will mean changing the following URIs:

```
http://www.w3.org/1999/XMLSchema
http://www.w3.org/1999/XMLSchema-instance
```

... to these:

```
http://www.w3.org/2001/XMLSchema
http://www.w3.org/2001/XMLSchema-instance
```

In more complex WSDLs, you will need to change parts of the WSDL to migrate the types themselves. Keep in mind that these changes may break clients that communicate with your web service.

Supporting Mismatch Between Operation Parameter Names and Names in WSDL

After upgrading a web service in which one or more method parameter names do not match their corresponding names in the WSDL from which the web service was created, you will need to add a `@WebParam` annotation to each parameter.

You might have this situation if a parameter name was modified after generating the web service, or if the WSDL name is not valid as a Java identifier. To work around this issue, add a `@WebParam` annotation with the matching WSDL name to each parameter in the JWS operation.

Resolving Deployment Error for Same-Named Web Services

Due to a difference in the way versions 8.1 and 9.2 generate the default `targetNamespace` value for web services, you may encounter a deployment error if you have two or more web services with the same class name in an application. For web services in version 9.2, WebLogic Server uses the fully-qualified port name — which includes the web service's `targetNamespace` value — to bind resources it uses internally. As a result, the port name must be unique within an application.

In version 8.1, the `targetNamespace` value defaulted to `"http://www.openuri.org/"`. Because it is the default, this qualifying value could potentially be the same for multiple web services in the application. As a result, after upgrading to version 9.2, if there are two or more web services with the same simple class name (name without package) and this default value, a conflict can occur that results in a deployment error. The error will take the form `"<web_service_name> is already bound."`

If you encounter this error you can work around it by setting the value of the `@java.jws.WebService` annotation's `name` and `serviceName` attributes. This changes the name values within the WSDL portType and port elements. Even so, it should not affect the web service's interaction with its version 8.1 clients.

Note that for web services created with version 9.2, the default `targetNamespace` value is based on the web service package name rather than the same value for all. You can specify another value with the `@WebService` annotation's `targetNamespace` attribute.

Upgrading Security from from WS-Security to WS-Policy

Upgrade required. In version 8.1, web service message-level security is managed using WS-Security (WSSE) policy files. In version 9.2 you should use Web Services Policy Framework (WS-Policy). Workshop for WebLogic upgrade tools do not perform this aspect of upgrade. This section describes the key differences between the version 8.1 and 9.2 models.

- The security information maintained in WSSE policy files has been separated into two places in the WS-Policy framework. Whereas WSSE stored both the "what" (the kinds security policies in effect) and the "how" (how those security policies are evaluated), these have been split for both the service control and the web service it communicates with.
- This split of responsibilities is handled differently in the web service itself than it is in a service control that communicates with the web service from client code.
 - In the web service, use `@weblogic.jws.Policy` annotations to specify the policies that should be used. For specific security configuration needs, use the `weblogic.jws.security.WssConfiguration` annotation to specify a named configuration managed through the WebLogic Server console. For example, you might need to create a named configuration that includes information needed to handle encryption and digital signatures. See [Create a Web Service security configuration](#) for information on creating the named configuration.
 - In the service control, there are two options.
 - A service control generated from a WSDL that contains the security configuration information should have the information required to communicate successfully with the web service. For example, for an upgraded web service whose security is fully configured on WebLogic Server, retrieving the WSDL with `<servicepath>?WSDL` should produce a WSDL that has the required information.
 - A service control that has not been generated from a fully-configured web service's WSDL can be annotated to support security configuration settings. This is a case in which you generate a service control in order to test and debug your application's functionality (say, with a test version of the web service) before taking the step of adding security constraints. To add security constraints, you annotate the service control with `@ServiceControl.Policy` annotations that specify XML fragments containing security configuration information. You specify run-time values in one of two ways, as described in [Configuring Run-Time Message-Level Security](#)

Via the [Service Control](#) for more information.

- o Values from a WSSE policy file map approximately into the WS-Policy framework.

8.1 WSSE File Element	9.2 Annotation	Direction
wsSecurityIn/token	@Policy(uri="policy:Auth.xml, direction=Policy.Direction.inbound)	Inbound
wsSecurityIn/encryptionRequired	@Policy(uri="policy:Encrypt.xml")	Both
wsSecurityOut/encryption	@Policy(uri="policy:Encrypt.xml")	
wsSecurityIn/signatureRequired	@Policy(uri="policy:Sign.xml")	Both
wsSecurityOut/signatureKey	@Policy(uri="policy:Sign.xml")	
wsSecurityOut/additionalSignedElements	Define a custom policy file and link to it with the @Policy annotation.	Both
wsSecurityOut/additionalEncryptedElements	Define a custom policy file and link to it with the @Policy annotation.	Both
keystore	Define a named configuration and link to it with the @WssConfiguration annotation.	N/A

Resolving Issue of Unmapped Entries in web.xml

In the course of upgrading your version 8.1 applications, you might find that some of your application's security-related characteristics differ between its behavior on the domain shipped with Workshop for WebLogic and the upgraded domain to which you redeploy it. This is because the "new" 9.x domain included with Workshop for WebLogic is not backward compatible, whereas the upgraded domain to which you deploy your upgraded application is (because it has been upgraded).

Applications on new domains default to having Combined Role Mapping enabled. This causes roles with no matching entry in weblogic.xml to have no (an empty) mapping.

If you had unmapped entries in web.xml, you will have to do one of the following:

- Add an explicit mapping(s) in weblogic.xml
- Disable the Combined Role Mapping feature (see the documentation referenced below).

You can disable combined role mapping by setting the <combined-role-mapping> property of the realm:

```
security-configuration>
  <name>workshop</name>
  <realm>
    ...
    <sec:combined-role-mapping-enabled>>false</sec:combined-role-mapping-enabled>
    <sec:name>myrealm</sec:name>
  </realm>
  <default-realm>myrealm</default-realm>
</security-configuration>
```

For more information on the combined role mapping setting and how it impacts security, see [Understanding the Combined Role Mapping Enabled Setting](#).

Upgrading Reliable Messaging Support — Basic Instructions

Workshop for WebLogic [upgrade tools](#) do not upgrade reliable messaging support (such as the @jws:reliable annotation) from version 8.1 to version 9.2. As noted in the version 8.1 documentation, that version's reliable messaging (RM) support was very limited and was not based on a specification that would be supported in future versions. This section provides high level suggestions for adding reliable messaging support to a web service in version 9.2.

In version 8.1, you added support for reliable messaging in part by using annotations such as `@jws:reliable` and `@jc:reliable`. For example, at the web service class level you indicated the message time to live, while at the operation level you indicated that the operation could be invoked reliably. Here's an example:

```
/**
 * @jws:reliable message-time-to-live="600 seconds"
 */
public class ReliableService implements com.bea.jws.WebService
{
    static final long serialVersionUID = 1L;

    /**
     * @common:operation
     * @jws:reliable enable="true"
     * @jws:protocol form-get="false" form-post="false"
     */
    public void doSomething()
    {
        ...method body...
    }
    ...
}
```

Version 9.2 annotations related to reliable messaging are not automatically added to upgraded code. Instead, you can manually add version 9.2 reliable messaging support using the following high-level steps. Note that you should consider these a minimum list of changes; for a more complete picture, see [Using Web Service Reliable Messaging](#) on the edocs web site.

1. Use [Workshop for WebLogic tools](#) to upgrade your projects from WebLogic Workshop version 8.1.
2. Create a WS-Policy XML file that describes the web services reliable messaging support. There are also default policy files provided with WebLogic Server. For more information, see [Use of WS-Policy Files for Web Service Reliable Messaging Configuration](#) on the edocs web site. Note that you can use the policy file to specify an "expiration" value that corresponds to the "message-time-to-live" attribute of the `@jws:reliable` annotation.
3. Annotate the web service class with a `@weblogic.jws.Policy` annotation that references the policy file. For information on this annotation, see [Using the @Policy Annotation](#) on the edocs web site.
4. Annotate the reliable method with the `@javax.jws.Oneway` annotation. All "reliable" operations must be "Oneway" when not using the WebLogic Server asynchronous request-response feature. For more information, see [Using the @Oneway Annotation](#) and [Invoking a Web Service Using Asynchronous Request-Response](#) on the edocs web site.

Note: For an overview of options related to asynchrony in web services, see [Using Events and Callbacks to Enable Long-Running Operations](#).

Here's a very simple example of how the upgraded web service might look:

```
@Transactional(true)
@UseWLW81BindingTypes()
@WLHttpTransport(serviceUri = "reliable/ReliableService.jws")
@WLWRollbackOnCheckedException()
@WebService(serviceName = "ReliableService",
            targetNamespace = "http://www.openuri.org/")
@javax.jws.soap.SOAPBinding(style = javax.jws.soap.SOAPBinding.Style.DOCUMENT,
                            use = javax.jws.soap.SOAPBinding.Use.LITERAL,
                            parameterStyle = javax.jws.soap.SOAPBinding.ParameterStyle.WRAPPED)
@Policy(uri="ReliableServicePolicy.xml", direction=Policy.Direction.both, attachToWsdl=true)
public class ReliableService implements java.io.Serializable
{
    static final long serialVersionUID = 1L;
```



```

@SOAPBinding(style = javax.jws.soap.SOAPBinding.Style.DOCUMENT,
             use = javax.jws.soap.SOAPBinding.Use.LITERAL,
             parameterStyle = javax.jws.soap.SOAPBinding.ParameterStyle.WRAPPED)
@WebMethod()
@Oneway()
@WebResult(name = "doSomethingResult")
public void doSomething()
{
    ...
}
...
}

```

5. Account for the fact that the version 9.2 service control does not provide the version 8.1 `onDeliveryFailure` callback method. The `onDeliveryFailure` method was designed to provide notification that a reliable message could not be delivered. To support this functionality in a web service upgraded to version 9.2, you can use the [@webllogic.jws.AsyncFailure](#) annotation in conjunction with WebLogic Server's asynchronous request-response feature.

Note: To support failure notification, you must migrate your code to the WebLogic Server asynchronous request-response technology. For more information, see [Invoking a Web Service Using Asynchronous Request-Response](#) on the edocs web site.

The basic outline of the client code would look something like the following.

```

@WebService
public class ClientService
{
    @Control()
    private ReliableServiceControl reliableServiceControl;

    @WebMethod()
    public void someMethod(String parm)
    {
        // Invoke the service control method.
        reliableServiceControl.doSomething(parm);
    }

    @AsyncFailure(target="reliableServiceControl", operation="doSomething")
    public void onDoSomethingFailure(AsyncPostCallContext apc, Throwable e)
    {
        // Error handling here
    }
}

```

6. Finally, you'll want to upgrade client service controls. To do so, you'll need a WSDL file representing the fully upgraded web service. The WSDL must contain the policy information needed for reliable messaging. To get a WSDL useful for this purpose, you can use a web browser to display the web service's WSDL using a URL of the following form: `<URL_of_deployed_web_service>?WSDL`

With the WSDL in hand, you can do one of the following:

- o Regenerate the service control from the new WSDL file.
- o Add the [@com.bea.control.ServiceControl.Policy](#) annotation to the existing service control (at the class level) in order to specify the policy file you created for the web service. Here's an example of policy information you might get from the WSDL:

```

<s0:Policy s1:Id="ReliableServicePolicy.xml">
  <wsrm:RMAssertion xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm">
    <wsrm:InactivityTimeout Milliseconds="600000"/>
    <wsrm:AcknowledgementInterval Milliseconds="200"/>
    <wsrm:BaseRetransmissionInterval Milliseconds="3000"/>
  </wsrm:RMAssertion>
</s0:Policy>

```

```

        <wsrm:ExponentialBackoff/>
        <beapolicy:Expires Expires="PlD" xmlns:beapolicy="http://www.bea.com/wsrp/policy"/>
    </wsrm:RMAssertion>
</s0:Policy>
<wsp:UsingPolicy n1:Required="true" xmlns:n1="http://schemas.xmlsoap.org/wsdl/" />

```

For the annotation's uri attribute, specify the value in the <Policy> element's Id attribute. For the annotation's direction attribute value, specify [ServiceControl.Direction.both](#). Here's an example:

```

@ServiceControl.Policy(uri="ReliableServicePolicy.xml", direction=ServiceControl.Direction.both)
public interface ReliableServiceControl extends com.bea.control.ServiceControl

```

Alternative to Wrapper Classes for Handling SOAP Faults

Supported but deprecated. Version 8.1 provided wrapper classes through which you could control the content of an outgoing SOAP fault, as well as APIs for retrieving content from an incoming SOAP fault. Code using this feature is supported in code upgraded to version 9.2, but the feature is deprecated.

Note that since this feature is deprecated in this release, you should plan to find another solution for future releases, and should not use this feature for new code. For example, consider using the `javax.xml.rpc.soap.SOAPFaultException` class provided by JAX-RPC. For more information, see [Throwing Exceptions](#) in the WebLogic Server documentation on developing web services.

Handlers Not Supported for Callbacks

In version 8.1 the `@jc:handler` and `@jws:handler` annotations included a callback attribute that specified handlers to process SOAP messages associated with callbacks; version 9.2 does not include callback-specific handler support. For the counterparts of these annotations in version 9.2, see [Upgrading Annotations](#).

Upgrade Changes for Automatic Transaction Rollback

Supported but deprecated. In version 8.1, the runtime would roll back a container-managed transaction if a checked exception was thrown from the application or runtime. In version 9.2, this behavior is supported with the `@Transactional` and `@WLWRollbackOnCheckedException()` annotations added (by upgrade tools) to the web service source code.

General Steps for Replacing XQuery Maps

Version 9.x doesn't support XQuery maps, a version 8.1 feature through which you could use XQuery to reshape XML messages entering and leaving a web service operation. As you might imagine, the shape of the WSDL on which the web service's clients were built is defined in part by XQuery maps because the maps specify the types that map-annotated operations expect to receive or send. With the map removed, expected types are almost certainly different, changing the web service's interface, and causing client calls to fail. Note that this would also change the shape of a WSDL generated from the web service; any other files, such as service controls, generated from a version 8.1 copy of that WSDL will no longer match the web service itself.

Note: The lack of support for XQuery maps does not mean that XQuery itself is not supported. You can still execute XQuery expressions using the XMLBeans API. For more information on upgrade changes impacting this API, see [Updating XQuery Use to Support Upgraded XQuery Implementation](#).

One workaround is to rewrite your web service's operations so that its post-upgrade WSDL matches the version 8.1 WSDL shape, but without XQuery maps. That way, clients calling your web service can rely on the interface contract already established by the version 8.1 web service.

Here are the high-level steps to accomplish this.

1. In version 8.1, generate a WSDL file from the version 8.1 web service.
 - o In WebLogic Workshop version 8.1, in the Application tab, right-click the JWS file, then click Generate WSDL File.

2. In version 8.1, use the generated WSDL file to generate source code for a new web service that you will import into your version 9.2 application. This should give you a web service whose operations send and receive XML messages in the same shapes as the mapped operations, but without the maps.
 1. In version 8.1, right-click the WSDL file you generated, then click Duplicate; this will prevent you from overwriting the existing JWS file.
 2. Right-click the duplicate WSDL file, then click Generate Web Service.

You'll receive a prompt asking whether you'd like to use XMLBeans types for methods of the web service. If you select Yes, the IDE will create methods whose parameters are XMLBeans types to which parts of the incoming message can be bound (this is essentially what the XQuery map was doing). If you select No, the IDE will generate code for inner classes that can be used as parameter types.

You might find that keeping the XMLBeans types is a more reliable way to ensure that the XML shape required by the operation remains valid after you upgrade the web service.

3. Using code from the original mapped-operation web service, implement the original service's logic in the newly-generated mapless web service. As part of this process, test the new mapless web service with existing clients to ensure that the old functionality is still present despite the absence of maps.
4. Use the [upgrade tools](#) to upgrade the version 8.1 application (including the mapless web service) to version 9.2. This will do nearly all of the work to update the application (update types, annotations, project structure, and so on).
5. Test the upgraded web service with existing clients.

Replacing the Use of `java.util.Map` as a Web Service Operation Return Type

Version 8.1 supported returning instances of `java.util.Map` from web service operations. The runtime provided a WebLogic Workshop-specific serialization of the Map to and from XML. The schema for that serialization was included in the WSDL for the Web Service. In version 9.2, `java.util.Map` instances can no longer be returned from web service operations.

Provide an application-defined type that supports the key/value features provided by `java.util.Map`. That type must conform to JAX/RPC Java <-> XML serialization rules. If the application-defined type will contain subclasses of the type's key or value type, then you must use the `weblogic.jws.Types` annotation to specify the types that could be contained at run time. Web services (and their clients) that previously returned a `java.util.Map` will have to be manually updated to use this new application-defined type.

Updating XQuery Use to Support Upgraded XQuery Implementation

Impacts: Code that uses XMLBeans and XQuery, including web services, controls, page flows, Enterprise JavaBeans

The older XQuery implementation is deprecated, but supported in this version for backward compatibility. Queries based on the older implementation will be kept, but a special `xmlOptions` parameter will be added to specify that the old implementation should be used.

Note: The older implementation is not automatically updated in JSP files. You must manually add the `Path._forceXqrl2002ForXpathXQuery` option to XQuery code in these files.

The following example shows how upgraded code would look with the `xmlOptions` parameter specifying the older XQuery engine.

```
import org.apache.xmlbeans.impl.store.Path;
import org.apache.xmlbeans.XmlOptions;

String queryExpression =
    "for $e in $this/xq:employees/xq:employee " +
    "let $s := $e/xq:address/xq:state " +
    "where $s = 'WA' " +
    "return $e";
```

```

try{
    XmlCursor resultCursor = empCursor.executeQuery(m_namespaceDeclaration + queryExpression,
        (new XmlOptions()).put(Path._forceXqrl2002ForXPathXQuery));
    resultXML = resultCursor.getObject();
    resultCursor.dispose();
}catch(Exception e){
    System.out.println(e.getLocalizedMessage());
}

```

Also, you should begin to upgrade XQuery strings so that they conform with the standard supported in version 9.2. For more information on both the old and the new standards, see the links in the following table:

Specification Used by Version 8.1	Specifications Used by Version 9.2
XQuery 1.0 and XPath 2.0 Functions and Operators	XQuery 1.0 and XPath 2.0 Data Model
	XQuery 1.0: An XML Query Language

Version 9.0 and 9.1 WebLogic Server Web Services Might Need to Be Recompiled for Deployment in Version 9.2

If you intend to deploy WebLogic Server web services compiled on version 9.0 or 9.1, you might need to recompile them before deploying on version 9.2. Recompilation is necessary if the web service code contains any of the following annotations.

```

weblogic.jws.Conversation
weblogic.jws.Context
weblogic.jws.Callback
weblogic.jws.ServiceClient
org.apache.beehive.controls.api.bean.Control

```

Keeping Files in Sync in the Absence of IDE Support

The version 8.1 IDE included a set of features through which the IDE kept related files in sync. For example, after generating a ServiceControl from a WSDL, changes to the WSDL would cause the IDE to automatically re-generate the ServiceControl to match. This functionality is not supported in the version 9.2 IDE.

For the following cases, after changes to the "source" WSDL file, the generated file must be manually re-generated using the context menu and the appropriate "Generate..." action:

- A ServiceControl is generated from a WSDL
- A Web Service is generated from a WSDL

Version 8.1 Clients are Generally Supported for Interop with Version 9.2 Web Services

However, there will be cases in which version 8.1 clients are only supported for communication with version 9.2 that were upgraded from version 8.1. For example, a version 9.2 web service that has operations annotated with @Oneway results in a WSDL that is not supported in version 8.1.

Related Topics

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)

Upgrading Page Flows

This topic gives more detail about upgrade changes noted for page flows.

For a more complete list of changes affecting applications upgraded from version 8.1, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#).

Changes When Upgrading from Version 8.1 NetUI JSP Tags to Beehive NetUI JSP Tags

When upgrading using [upgrade tools](#), you'll have the option to choose to replace your version 8.1 NetUI JSP tags with the Beehive NetUI JSP tags, which provide enhanced functionality over the version 8.1 tags. (The Beehive tags are the default for new web projects in Workshop for WebLogic version 9.2.) The default behavior is not to replace version 8.1 tags with Beehive tags. Instead, by default version 8.1 tags are upgraded to a version of the 8.1 tags that is compatible with a version 9.x server.

Note: A JSP page can use a combination of compatible 8.1 tags and Beehive tags. However, this is only supported when the combination is due to an upgrade outcome in which some version 8.1 tags have no Beehive counterparts.

As you might imagine, the decision you make about the tags will have a broad impact on the JSP portion of your application. Before choosing, consider the following:

- In general, if you want to use the Beehive tags, you're likely to find that migrating to Beehive is best done as a two-step process: upgrading to the compatible 8.1 tags, then migrating to the Beehive tags. This gives you an opportunity to ensure that your upgraded application runs on a version 9.x server before migrating to Beehive technology. Note, too, that you can later migrate portions of your application to Beehive as you revise your application, creating new JSP pages or upgrading existing ones.
- If your application is in maintenance mode, consider *not replacing* version 8.1 tag with Beehive tags. Upgrading to the compatible 8.1 tags (the default behavior) is likely to get your version 9.2 application up and running more quickly.
- If your application is still in development, consider replacing version 8.1 tags with Beehive tags. You're likely to find that the advantages of the enhanced Beehive tags are worth the effort of migrating fully to Beehive.

Tag Changes

If you choose to replace version 8.1 tags with Beehive tags during application upgrade, many of the version 8.1 tags will be replaced with easily identifiable Beehive counterparts. There will be exceptions to this predictability, as the following table describes.

Version 8.1 Tag	Version 9.2 Tag
tree	None; deprecated
grid	None; deprecated
label	span
choice	None; deprecated
choiceMethod	None; deprecated
repeater	repeater
content	None; deprecated
callControl	None; deprecated
declareControl	None; deprecated
visible	None; deprecated
getNetuiTagName	None; deprecated
anchor	anchor (Not migrated if the forward attribute is set.)

imageAnchor

imageAnchor
(Not migrated if the forward attribute is set.)

Attribute Changes

During upgrade some JSP tag attributes may be migrated to new attributes, or simply removed. The following table describes possible changes.

Version 8.1 Tag	Attribute	Action
anchor	id	Removed if a tagId attribute is present; otherwise migrated to tagId.
anchor	forward	Tag is not migrated if this attribute is present.
anchor	page	Removed if an href attribute is present; otherwise migrated to href by removing the initial '/'
bindingUpdateErrors	id	Removed
checkBoxGroup	tagId	Removed
form	id	Removed if a tagId attribute is present; otherwise migrated to tagId
form	tabindex	Removed
form	name	Change to beanName
form	type	Change to beanType
form	scope	Change to beanScope
image	lowsrc	Removed; netui:attribute tag added as a child of the image tag, for example: <pre></netui:image> <netui:attribute name="lowsrc" value="blurryFoo.gif"/> </netui:image></pre>
image	page	Removed if a src attribute is present, otherwise migrate to src by removing the initial '/'
image	id	Removed if a tagId attribute is present, otherwise migrate to tagId
imageAnchor	lowsrc	lowsrc attribute and netui:attribute tag added as a child of the image tag, for example: <pre></netui:imageAnchor> <netui:attribute name="lowsrc" value="blurryFoo.gif"/> </netui:imageAnchor></pre>
imageAnchor	scope	Removed
imageAnchor	forward	Tag not migrated if this attribute is present.
imageAnchor	page	Removed if an href attribute is present; otherwise migrated to href by removing the initial '/'
imageButton	align	Removed
imageButton	height	Removed
imageButton	width	Removed
imageButton	hspace	Removed
imageButton	vspace	Removed
imageButton	id	Removed if a tagId attribute is present; otherwise migrated to tagId
imageButton	page	Removed if a src attribute is present; otherwise migrated to src
label	dataFormatas	Removed
label	id	Removed if a tagId attribute is present; otherwise migrated to tagId
label	tabIndex	Removed
radioButtonGroup	tagId	
select	onSelect	Removed
select	nullableTop	Removed
selectOption	id	Removed if a tagId attribute is present; otherwise migrated to tagId
selectOption	tabIndex	Removed
scriptContainer	scopeld	Changed to idScope

html	scopeId	Changed to idScope
error	value	Changed to key
error	bundle	Name migrated to "bundleName"
errors	bundle	Name migrated to "bundleName"
template:section	visibility	If the visibility attribute is present and it has a value, the value is migrated to the visible attribute; otherwise if the visibility attribute is not present, the visible attribute is migrated.

Upgrade Sets the Default Expression Language to a Backward-Compatible Version

When upgrading an application, you'll be prompted to upgrade version 8.1 NetUI JSP tags to Beehive NetUI JSP tags. Whether or not you choose to upgrade to Beehive, the default expression language in your upgraded application will be a backward-compatible version, rather than the version used for new applications. This is because the backward-compatible version is more permissive, allowing you to continue with your existing code until you are ready to upgrade fully. If and when you choose to upgrade fully to the Beehive JSP tag technologies, you should change the default expression language to the version that supports Beehive tags.

If you add Beehive JSP tags to your pages, they will use the Beehive version of the expression language. Until that time, you should continue to use the backward-compatible expression language as the default. See [Changing the Default Expression Language Used by JSP Tags](#) for more information.

Note: In general, you should regard the version 8.1 NetUI tags as deprecated; new code should use the Beehive NetUI JSP tags.

Changing the Default Expression Language Used by JSP Tags

When you have fully upgraded your JSP pages so that they use the Beehive JSP tags, you should change the default expression language from the backward-compatible version to the current version.

Note: Before making this change, you should be aware of [code changes that might be required](#).

You can change the default expression language by editing the `beehive-netui-config.xml` file in your project. You'll find that file at a path such as the following:

```
WORKSPACE_HOME\PROJECT_HOME\web\WEB-INF\beehive-netui-config.xml
```

Possible settings for default language include `netuiel` (for the JSP 2.0 expression language) and `compat-netuiel` (for the version 8.1 NetUI expression language). In other words, to change to the default to the JSP 2.0 expression language, change the file so that it includes the following:

```
<expression-languages>
  <default-language>netuiel</default-language>
  . . .
</expression-languages>
```

Changing Code to Support the Expression Language in Beehive NetUI JSP Tags

If you choose to upgrade version 8.1 NetUI JSP tags to Beehive NetUI JSP tags, changes to your JSP code will include expression changes to support the expression language used by the Beehive tags. The following describes changes made by the tools during upgrade, as well as changes you might need to make to your own non-JSP code.

Expression Changes

The following changes are made to expressions during upgrade.

- Use of `dataSource` attributes is changed from `dataSource="{actionForm.name}"` to `dataSource="actionForm.name"`.
- Expressions of the form `{expression}` are changed to `${expression}` (except for the `dataSource` attribute).
- Version 8.1 binding contexts are changed according to the following table:

8.1 Binding Context	9.2 Implicit Object	Comment
<code>url</code>	<code>param</code>	Changed to the JSP 2.0 implicit object name
<code>request</code>	<code>requestScope</code>	Changed to the JSP 2.0 implicit object name
<code>application</code>	<code>applicationScope</code>	Changed to the JSP 2.0 implicit object name
<code>session</code>	<code>sessionScope</code>	Changed to the JSP 2.0 implicit object name
<code>pageContext</code>	<code>pageScope</code>	Changed to the JSP 2.0 implicit object name
<code>actionForm</code>	<code>actionForm</code>	
<code>pageInput</code>	<code>pageInput</code>	
<code>container</code>	<code>container</code>	
<code>pageFlow</code>	<code>pageFlow</code>	
<code>globalApp</code>	<code>globalApp</code>	
<code>bundle</code>	<code>bundle</code>	
	<code>sharedFlow</code>	This is a new binding context.
	<code>backing</code>	This is a new binding context used with JavaServer Faces.

Changes Needed to Bind to Public Data Members

If you upgrade version 8.1 NetUI JSP tags to Beehive NetUI JSP tags, you may need to change how data held by public members is exposed to tags for binding. Whereas version 8.1 supported binding expression code to public fields, in Beehive expressions can only bind to public accessors.

In other words, although [upgrade tools](#) will upgrade the expression language syntax used in JSP tags, they will of course not add accessors to classes where needed. For a full upgrade to Beehive JSP tags, you must manually add `get*` and `set*` accessors where public fields were used.

Reserved Words Can Not Be Used as Identifiers

The JSP 2.0 expression language reserves the following words; your code should not use them as identifiers.

`and` `eq` `gt` `true` `instanceof`

`or` `ne` `le` `false` `empty`

`not` `lt` `ge` `null` `div` `mod`

Note that you can inadvertently use a reserved word by including that word in the name of an accessor designed to expose a property to expression language code. For example, the following code would be designed to expose an `empty` property:

```
public String empty;
public String getEmpty() {return empty;}
```

In the JSP page, of course, this would result in using "empty," which is a reserved word, as a property name:

```
<netui:content value=?${pageFlow.empty}? defaultValue=?NO VALUE?/>
```

Instead, trying using a naming scheme that results in a method name such as "getEmptyVal", then bind to them with `pageFlow.emptyVal`.

For more information about the expression language, see [Expression Language](#) in the J2EE 1.4 tutorial.

Form Classes Must Include a Default Constructor

Unlike version 8.1, version 9.2 enforces a requirement that forms used in page flow (such as those which extended the now-deprecated `FormData` class) must have a default constructor.

You can work around this change by adding a default constructor after upgrading your code.

Updating XQuery Use to Support Upgraded XQuery Implementation

In version 8.1, XMLBeans supported XQuery Working Draft 16; Working Draft 23 is used in version 9.2. Upgrade tools will generally update XMLBeans code that executes XQuery expressions, but not code in JSP files. You will need to make the required change manually for code based on the older implementation to work.

For more information, see [Updating XQuery Use to Support Upgraded XQuery Implementation](#).

Fixing Package Names That are Not Upgraded in JPF File Method and JSP Code

When a type name is fully-qualified outside a type import (Java or JSP), the type's package is not upgraded to the package used in version 9.2. For example, in the following after-upgrade code, note that the package for `Forward` has not been upgraded from `com.bea.wlw.netui.pageflow` to `org.apache.beehive.netui.pageflow`.

```
/**
 * @jpf:action
 * @jpf:forward name="begin" path="Begin.jsp"
 */
@Jpf.Action(forwards = {
    @Jpf.Forward(name = "begin",
        path = "Begin.jsp")
})
public Forward begin()
{
    return new com.bea.wlw.netui.pageflow.Forward("begin");
}
```

The errors will be highlighted in source code. You can make fixes by using the Workshop for WebLogic quick fix feature to replace the type with the correct type of the same name. You can also search and replace to fix.

Custom Tags That Extend NetUI Tags are Not Supported

If you created custom JSP tags in version 8.1 by extending NetUI tags, your tags will not be upgraded by Workshop for WebLogic tools. Extending NetUI tags was not supported. Note that if you elected not to migrate NetUI tags to Beehive tags, your tags may build within the application, but may not work as expected.

Likewise, extending Beehive JSP tags in version 9.2 is not supported.

Details: Some PageFlowController and FlowController Methods Made Protected Instead of Public

To enhance application security, some public methods in [org.apache.beehive.netui.pageflow.PageFlowController](#) and [org.apache.beehive.netui.pageflow.FlowController](#) have been changed so that they're protected. This change means that these methods can no longer be invoked as bean properties from within JSP pages. In addition, new public versions of the methods were created for access from outside page flow code, but given names that begin "the" instead of "get" so that they're not accessible as bean properties.

In summary, this change includes the following:

- The following PageFlowController methods were changed from public to protected: `getSharedFlows`, `getCurrentPageInfo`, `getPreviousPageInfo`, and `getPreviousActionInfo`.
- The following public PageFlowController methods were added: `theSharedFlows`, `theCurrentPageInfo`, `thePreviousPageInfo`, and `thePreviousActionInfo`.
- The following FlowController methods were changed from public to protected: `getModuleConfig`, `getActions`
- The following public FlowController method was added: `theModuleConfig`.

In general, fixing broken code after upgrade involves searching for the method name that begins with "get" and replacing it with the corresponding method that begins with "the". For example, the following code would need to be revised:

```
PageFlowController pageFlowController =
    PageFlowUtils.getCurrentPageFlow(this.Request());
if(pageFlowController != null){
    if(pageFlowController.getCurrentPageInfo().getForward().getRedirect() == true) {
        return new Forward("main_page");
    }
}
```

The following reflects changes that would be made (note that the `getCurrentPageFlow` method above, from version 8.1, is deprecated; use the two-parameter version below instead):

```
PageFlowController pageFlowController =
    PageFlowUtils.getCurrentPageFlow(this.getRequest(), this.getServletContext());
// Note "theCurrentPageInfo" here where "getCurrentPageInfo" is used above.
if(pageFlowController != null){
    if(pageFlowController.theCurrentPageInfo().getForward().getRedirect() == true) {
        return new Forward("main_page");
    }
}
```

Note that code written on versions of the Beehive APIs included in WebLogic Platform prior to version 9.2 might also need to be changed. Workshop for WebLogic [upgrade tools](#) do not upgrade code written on 9.x versions of the platform.

Upgrading from getRequest Method Calls to Retrieve a ScopedRequest Instance

In version 8.1, the return value of the `FlowController.getRequest` method could be cast to a `ScopedRequest` when running in the WebLogic Portal environment. In version 9.2, to improve performance the Beehive page flow APIs retrieve a `ScopedRequest` instance differently. This change is not upgraded by Workshop for WebLogic upgrade tools. If your portal code makes a call to this method, you will need to manually upgrade the code to avoid a `ClassCastException`. Your code should instead use the `ScopedServletUtils.unwrapRequest` method to retrieve the `ScopedRequest` instance.

For example, if your page flow code had the following:

```
ScopedRequest s = (ScopedRequest)getRequest();
```

you would change it to:

```
ScopedRequest s = ScopedServletUtils.unwrapRequest(getRequest());
```

Updating Code to Reflect Changes in PageFlowStack Class

Differences between the version 8.1 `com.bea.wlw.netui.pageflow.PageFlowStack` class and its Beehive counterpart in version 9.2, `org.apache.beehive.netui.pageflow.PageFlowStack`, may make it necessary for you to modify some of your code after upgrade. In particular, the version 8.1 class extended [java.util.Stack](#), but the 9.2 class extends `Object` directly.

One scenario in which this could create a problem is if your code casts the return value of the static method `PageFlowUtils.getPageFlowStack` (which is a `Stack` object) to `PageFlowStack`. This cast is of course not valid with the Beehive `PageFlowStack` class. If you were casting to `PageFlowStack`, then using `Stack` methods, keep in mind that several of these methods are implemented in the Beehive [PageFlowStack](#) class; however, you now get the `PageFlowStack` instance using static `PageFlowStack.get` methods.

If you were directly calling the `Stack` instance returned from `PageFlowUtils.getPageFlowStack`, but you aren't casting to `PageFlowStack`, you can still use the `Stack`. Keep in mind, however, that the `getPageFlowStack` method is deprecated.

Updating Code to Reflect Changes in State Tracking API

In version 9.2, several methods in the NetUI API have been changed so that, under certain circumstances, they return `IllegalStateException` rather than `null`. This is an improvement over the version 8.1 API, in which `null` was returned regardless of whether the requested return value was actually null or simply unknown because it wasn't being tracked.

You can avoid handling `IllegalStateException` by overriding the `alwaysTrackPreviousAction` method in your page flow controller so that that method returns `true`. Note that this might negatively impact the performance of your application. Here's how it would look:

```
protected boolean alwaysTrackPreviousAction()
{
    return true;
}
```

The following is a list of affected methods:

- `getPreviousActionURI`
- `getPreviousActionInfo`
- `getCurrentPageInfo`
- `getPreviousPageInfo`
- `getPreviousFormBean`
- `getCurrentForwardPath`
- `getPreviousForwardPath`

The following lists detailed circumstances surrounding the change:

- This change is only relevant if the current page flow does not use a `@jpf:forward` with a `return-to="previousAction"` attribute.
- If `alwaysTrackPreviousAction` is not overridden to return `true`, the version 9.2 methods will return `java.lang.IllegalStateException`.
- If `alwaysTrackPreviousAction` is overridden to return `true`, then the version 9.2 methods will either return the requested value or `null` if the value is in fact null.

Replacing Callback-Enabled Controls Used in a Page Flow

In version 9.x custom controls used from a page flow can't receive external callbacks. In version 8.1, you could write a custom control that received callbacks from controls nested within it, such as service controls. By exposing a polling interface from your custom control, your page flow code could then retrieve responses received from the callbacks. This functionality is not supported in version 9.2.

In upgraded code, you can substitute for this functionality by replacing your custom control with a web service. For more information on the version 8.1 feature and suggested version 9.2 workarounds, see [Providing Support for Callbacks from a Page Flow](#).

Ensuring Assembly of Controls Used from a JSP Page

In version 9.2, some controls require a compile-time assembly process that generates artifacts needed by the runtime. These controls include ServiceControl, EJBCControl, and TimerControl.

This assembly process is only triggered for these controls when they're referenced within a Java file — effectively excluding controls that are referenced only from JSP pages. In upgraded code, you can work around this change from version 8.1 by referencing the control from a Java file.

The best place for this reference is the page flow controller file; if there's no controller in your application, you can add the reference to another Java file that's compiled with the application. You do this by adding an [org.apache.beehive.controls.api.bean.ControlReferences](#) annotation that names the control class used in the JSP page. Here's an example that references a single control:

```
@ControlReferences(value={mycontrols.MyServiceControl.class})
public class Controller extends PageFlowController {
    ...
}
```

Resolving Name Conflicts Between Jpf.Forward and validationErrorForward

Version 8.1 allowed the `jpf:validation-error-forward` and `jpf:forward` annotations to have the same value for their name attributes; this is not allowed in version 9.2. For example, you might have had version 8.1 code such as the following, in which "failure" is used twice as a name attribute value:

```
/**
 * @jpf:action
 * @jpf:forward name="success" return-action="addProfileDone"
 * @jpf:forward name="failure" path="addClient.jsp"
 * @jpf:validation-error-forward name="failure" return-to="currentPage"
 */
protected Forward addClientProfile(MaintainIndividualProfileForm form)
{
    ...
}
```

The upgraded counterparts to this code (in the `Jpf.Forward` annotation and the `Jpf.Action validationErrorForward` attribute) will generate a build error in version 9.2. You can work around this change by changing one or both of the attribute values so they're not the same.

Upgrade Changes for Co-Location in Page Flows

Version 8.1 supported a project hierarchy in which the Controller file (`Controller.jpf`) and JSP files could be put into the same directory. This is not supported in version 9.2.

For example, during upgrade the following hierarchy change will occur for upgraded page flows:

Version 8.1

```
webapp/
  Controller.jspf
  index.jsp
```

Version 9.2 (after upgrade)

```
webapp
  src
    Controller.java
  web
    index.jsp
```

The version 8.1 style is referred to as *co-location*. In version 9.2 (including for upgraded applications), JAVA source files and JSPs are kept in different directories. Note that if you create a new page flow called `myPageFlow`, it would feature two parallel directories called `myPageFlow` — one for JAVA source code and one for the JSPs. Here's a simplified example:

```
webapp
  src
    myPageFlow
      myPageFlowController.java
      Controller.java
  WebContent
    index.jsp
    myPageFlow
      index.jsp
      page1.jsp
      other JSPs
```

Fixing Erroneous IDE Error for Variable in an HTML Start Tag

The version 9.2 IDE incorrectly displays an error for code in which a variable declaration is made in an HTML start tag; the code is actually valid at run time. For example, in the following anchor tag the `<%=s%>` code using the String variable is flagged by the IDE as unresolved, but is actually valid:

```
<a href="<%=s%>">Test</a>
```

This also applies to WebLogic Portal tags that set Java variables, such as `<render:getJspUri>` and `<render:getSkinPath>`.

You can work around this in code upgraded from version 8.1 by moving the variable declaration to a line outside a start tag, as follows:

```
<%=s%>
<a href="<%=s%>">Test</a>
```

Import and Use of `com.bea.wlw.netui.util.XMLString` are Not Supported and Will Not be Upgraded

This `XMLString` class (not to be confused with `org.apache.xmlbeans.XmlString`) is not included in version 9.2. Upgraded code that imports the type will generate compilation errors. In version 8.1 this class was used in conjunction with the XScript expression language, which has been removed from version 9.2.

Related Topics

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)

Upgrading Enterprise JavaBeans

This topic gives more detail about upgrade changes noted for Enterprise JavaBeans.

For a more complete list of changes affecting applications upgraded from version 8.1, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#).

Enabling Automatic Transaction Support in Entity Beans

In version 8.1, the EJB container would create a transaction for an entity bean if it ran in an unspecified transaction. In version 9.2, the default is *not* to create the transaction for the transaction attribute values indicating "not supported," "supports," and "never."

To support the old behavior, ensure that the `TransactionAttribute.REQUIRED` constant is used in the `@RemoteMethod` annotation applied to entity bean methods. Here's an example of the annotation's syntax:

```
@RemoteMethod(transactionAttribute=Constants.TransactionAttribute.REQUIRED)
public void myMethod()
{
    ...
}
```

Related Topics

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)

Upgrading Annotations

This release incorporates support for the standard for Java annotations released with Java 5; that standard is defined in [JSR 175](#). This topic describes how version 8.1 annotations are treated during upgrade to version 9.2. An obvious change is the difference in syntax between the two versions; version 8.1 annotations were embedded in a Javadoc-style comment, for example. When you use [upgrade tools](#) to upgrade a version 8.1 application, the tools upgrade all of the annotations that have counterparts in version 9.2. Upgrade tools generally leave the old-style annotations, which are ignored by the version 9.2 runtime.

Note: For general information about Java 5 annotations, see the [Annotations](#) topic at the Sun web site.

For a more complete list of changes affecting applications upgraded from version 8.1, see [Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#).

Relationship Between Version 8.1 and Version 9.2 Annotations

To take advantage of the Java 5 annotations feature, version 9.2 of Workshop for WebLogic uses migrated versions of the annotations that version 8.1 supported. So, for example, the version 8.1 `@common:operation` annotation is replaced in version 9.2 with the `@WebMethod` annotation to signify that a method should be used as a web service operation.

For the most part, version 8.1 annotations have counterparts in version 9.2. However, there are a few exceptions. It's also worth noting that most of the migrated annotations are now part of the [Apache Beehive](#) project.

Note: [Upgrade tools](#) will upgrade all annotations for which there is a version 9.2 counterpart.

Version 8.1 Annotation	Version 9.x Annotation	Containing Component Type
common:context	weblogic.jws.Context	Web service
common:context	org.apache.beehive.controls.api.context.Context	Control
common:control	org.apache.beehive.controls.api.bean.Control	Multiple
common:define	No version 9.x counterpart.	
common:message-buffer	weblogic.jws.MessageBuffer	Web service
common:message-buffer	com.bea.control.annotations.MessageBuffer . Must be used in the control interface; no longer supported on the control implementation.	Control
common:operation	javax.jws.WebMethod	Web service
common:operation	No version 9.x counterpart	Control

common:schema	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	
common:security	com.bea.control.annotations.Security	Control
common:security (callback-roles-allowed attribute)	weblogic.jws.security.CallbackRolesAllowed	Web service
common:security (run-as attribute)	weblogic.jws.security.RunAs	Web service
common:security (roles-referenced attribute)	weblogic.jws.security.RolesReferenced	Web service
common:target-namespace	targetNamespace attribute in javax.jws.WebService annotation.	Web service
common:xmlns	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Web service
editor-info:code-gen	No version 9.x counterpart.	Custom control source
editor-info:link	No version 9.x counterpart. See Keeping Files in Sync in the Absence of IDE Support for a suggested workaround.	Web service
ejbgen:automatic-key-generation	weblogic.ejbgen.AutomaticKeyGeneration	EJBs
ejbgen:cmp-field	weblogic.ejbgen.CmpField	EJBs
ejbgen:cmr-field	weblogic.ejbgen.CmrField	EJBs
ejbgen:compatibility	weblogic.ejbgen.Compatibility	EJBs
ejbgen:create-default-dbms-tables	weblogic.ejbgen.CreateDefaultDbmsTables	EJBs
ejbgen:ejb-client-jar	weblogic.ejbgen.EjbClientJar	EJBs
ejbgen:ejb-interface	weblogic.ejbgen.EjbInterface	EJBs
ejbgen:ejb-local-ref	weblogic.ejbgen.EjbLocalRef	EJBs
ejbgen:ejb-ref	weblogic.ejbgen.EjbRef	EJBs
ejbgen:entity-cache-ref	weblogic.ejbgen.EntityCacheRef	EJBs
ejbgen:entity	weblogic.ejbgen.Entity	EJBs
ejbgen:env-entry	weblogic.ejbgen.EnvEntry	EJBs
ejbgen:file-generation	weblogic.ejbgen.FileGeneration	EJBs
ejbgen:finder	weblogic.ejbgen.Finder	EJBs
ejbgen:foreign-jms-provider	weblogic.ejbgen.ForeignJmsProvider	EJBs
ejbgen:jar-settings	weblogic.ejbgen.JarSettings	EJBs
ejbgen:jndi-name	weblogic.ejbgen.JndiName	EJBs
ejbgen:local-home-method	weblogic.ejbgen.LocalHomeMethod	EJBs
ejbgen:local-method	weblogic.ejbgen.LocalMethod	EJBs
ejbgen:message-driven	weblogic.ejbgen.MessageDriven	EJBs
ejbgen:method-isolation-level-pattern	weblogic.ejbgen.MethodIsolationLevelPattern	EJBs

ejbgen:method-permission-pattern	weblogic.ejbgen.MethodPermissionPattern	EJBs
ejbgen:relation	weblogic.ejbgen.Relation	EJBs
ejbgen:relationship-caching-element	weblogic.ejbgen.RelationshipCachingElement	EJBs
ejbgen:remote-home-method	weblogic.ejbgen.RemoteHomeMethod	EJBs
ejbgen:remote-method	weblogic.ejbgen.RemoteMethod	EJBs
ejbgen:resource-env-ref	weblogic.ejbgen.ResourceEnvRef	EJBs
ejbgen:resource-ref	weblogic.ejbgen.ResourceRef	EJBs
ejbgen:role-mapping	weblogic.ejbgen.RoleMapping	EJBs
ejbgen:security-role-ref	weblogic.ejbgen.SecurityRoleRef	EJBs
ejbgen:select	weblogic.ejbgen.Select	EJBs
ejbgen:session	weblogic.ejbgen.Session	EJBs
ejbgen:value-object	weblogic.ejbgen.ValueObject	EJBs
jc:connection	com.bea.control.JdbcControl.ConnectionDataSource	Database control
jc:conversation	com.bea.control.ServiceControl.Conversation	Service control and others.
jc:ejb	org.apache.beehive.controls.system.ejb.EJBControl.EJBHome	EJB control
jc:handler	com.bea.control.ServiceControl.Handler	Service control
jc:jms	org.apache.beehive.controls.system.jms.JMSControl.Destination	JMS control
jc:jms-headers	org.apache.beehive.controls.system.jms.JMSControl.HeaderType , org.apache.beehive.controls.system.jms.JMSControl.Priority , org.apache.beehive.controls.system.jms.JMSControl.Expiration , org.apache.beehive.controls.system.jms.JMSControl.Type	JMS control
jc:jms-property	org.apache.beehive.controls.system.jms.JMSControl.PropertyValue	JMS control
jc:location	com.bea.control.ServiceControl.Location	Service control
jc:log	No version 9.x counterpart. The Beehive implementation of the Database control does not currently support logging.	Database control
jc:parameter-xml	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Service control
jc:protocol	com.bea.control.ServiceControl.SOAPBinding and com.bea.control.ServiceControl.JmsSoapProtocol .	Service control
jc:reliable	No version 9.x counterpart.	Service control
jc:return-xml	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Service control

jc:selector	No version 9.x counterpart. The JMS control in version 9.2 does not support receiving messages.	Service Broker control
jc:sql	com.bea.control.JdbcControl.SQL	Database control
jc:timer	com.bea.control.TimerControl.TimerSettings	Timer control
jc:ws-security-callback	No version 9.x counterpart. This was used to support WS-Security in web services. Upgraded code should use the Web Services Policy Framework (WS-Policy) instead.	Service control
jc:ws-security-service	No version 9.x counterpart. This was used to support WS-Security in web services. Upgraded code should use the Web Services Policy Framework (WS-Policy) instead.	Service control
jcs:control-tags	No version 9.x counterpart.	Custom control source
jcs:ide	No version 9.x counterpart.	Custom control source
jcs:jc-jar	No version 9.x counterpart.	Custom control source
jcs:suppress-common-tags	No version 9.x counterpart.	Custom control source
jws:context	weblogic.jws.Context	Web service
jws:control	org.apache.beehive.controls.api.bean.Control	Web service
jws:conversation	weblogic.jws.Conversation	Web service
jws:conversation-lifetime	weblogic.jws.Conversational	Web service
jws:define	No version 9.x counterpart.	Web service
jws:handler	javax.jws.HandlerChain (for files) and javax.jws.soap.SOAPMessageHandlers (for individual classes)	Web service
jws:location	com.bea.control.ServiceControl.Location	Web service
jws:message-buffer	weblogic.jws.MessageBuffer	Web service
jws:operation	javax.jws.WebMethod	Web service
jws:parameter-xml	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Web service
jws:protocol	javax.jws.soap.SOAPBinding	Web service
jws:reliable	Use WS-Policy files and the weblogic.jws.Policies annotation.	Web service
jws:return-xml	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Web service
jws:schema	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Web service
jws:target-namespace	targetNamespace attribute in javax.jws.WebService annotation.	Web service
jws:wSDL	wSDLLocation attribute in javax.jws.WebService annotation.	Web service

jws:ws-security-callback	No version 9.x counterpart. This was used to support WS-Security in web services. Upgraded code should use the Web Services Policy Framework (WS-Policy) instead.	Web service
jws:ws-security-service	Use the weblogic.jws.Policy and weblogic.jws.Policies annotations.	Web service
jws:xmlns	No version 9.x counterpart. This was used in conjunction with XQuery maps, which are no longer supported.	Web service
jpf:controller	org.apache.beehive.netui.pageflow.annotations.Jpf.Controller	Page flow
jpf:action	org.apache.beehive.netui.pageflow.annotations.Jpf.Action	Page flow
jpf:forward	org.apache.beehive.netui.pageflow.annotations.Jpf.Forward	Page flow
jpf:catch	org.apache.beehive.netui.pageflow.annotations.Jpf.Catch	Page flow
jpf:validation-error-forward	org.apache.beehive.netui.pageflow.annotations.Jpf.Forward	Page flow
jpf:exception-handler	org.apache.beehive.netui.pageflow.annotations.Jpf.ExceptionHandler	Page flow
jpf:message-resources	org.apache.beehive.netui.pageflow.annotations.Jpf.MessageBundle	Page flow

Resolving Ambiguity Related to Annotation Types

Unlike version 8.1, in version 9.2 annotations are Java types that must either be imported or fully-qualified in code. Because code using these types is being added to your code in order to upgrade from annotations in your version 8.1 code, there can sometimes be ambiguity when the added annotations have the same names as types your code may already have been using.

For example, imagine that your version 8.1 session bean happens to use the JMS `Session` interface and imports `javax.jms.*`. Your upgraded session bean will be marked with the `@Session` annotation (`weblogic.ejbgen.Session`). Your imports will include `javax.jms.*` and `weblogic.ejbgen.Session`, but the compiler doesn't know which is intended with a simple `@Session` annotation.

In general, be aware that compilation errors in your upgraded code may simply be due to this ambiguity. The remedy for this issue is to fully qualify annotation type names where they're used.

Related Topics

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)

upgradeStarter Command

Upgrades a WebLogic Workshop application version 8.1 (SP4, SP5, or SP6) to a Workshop for WebLogic, version 9.2 workspace.

Note: This command is also available as an Ant task. For more information, see [upgrade Ant Task](#).

Note: Your version 8.1 application must have been upgraded to SP4, SP5, or SP6 before using this command.

This command exposes from the command line essentially the same functionality exposed by the import wizard when upgrading an application. (See [How To: Use the Import Wizard to Upgrade Version 8.1 Applications](#) for information on the wizard.) One exception to this support is that it is not possible with this command to specify a subset of an application to upgrade. For example, this command does not support specifying which projects to upgrade; all projects are upgraded.

As with the import wizard, this command does not delete or change the version 8.1 application.

Environment

You must use a version 1.5 implementation of the JRE. Also, the classpath must include startup.jar (see the `ECLIPSE_HOME` argument description below).

Syntax

```
java -cp
  %ECLIPSE_HOME%/startup.jar
  -Dwlw.application=%WORK_FILE%
  -Dweblogic.home=%WL_HOME%
  org.eclipse.core.launcher.Main
  -application com.bea.wlw.upgrade.upgradeStarter
  -data %WORKSPACE%
  [-pluginCustomization %PREFS_FILE%]
```

Arguments

ECLIPSE_HOME

Required. The path to the directory that contains startup.jar. By default for Workshop for WebLogic, this is:

BEA_HOME/workshop92/eclipse

-Dweblogic.home=WL_HOME

Required. The location of WebLogic Server root folder. By default, this is:

BEA_HOME/weblogic92

-Dwlv.application=WORK_FILE

Required. Specifies the application to upgrade. Replace *WORK_FILE* with the WORK file name corresponding to the WebLogic Workshop 8.1 you want to upgrade.

-application com.bea.wlv.upgrade.upgradeStarter

Required. The Eclipse plugin extension point used to execute this command.

-data WORKSPACE

Required. The name of the target workspace where you want the upgraded application to go. This can be any directory to which you want the version 9.2 application files generated.

[-pluginCustomization PREFS_FILE]

Optional. Specifies a properties file to set options for upgrade. Replace *PREFS_FILE* with the name of a properties file containing a number of key-value pairs. See the remarks below for a list of possible properties.

Remarks

The following lists the properties supported in a *PREFS_FILE* specified by the `-pluginCustomization` argument.

- `com.bea.wlv.upgrade/upgradeHarnessAbortOnError = true | false`

Specify `true` to have the upgrade process fail on any error. By default upgrade will attempt to continue after an error. Errors will always appear in the log file. The default is `false`.

- `com.bea.wlv.upgrade/upgradeHarnessMessageLevel = INFO | WARNING | ERROR`

Optional. Takes a message level setting. When not specified, upgrade will log all messages. The following values can be specified:

`INFO` — Display all messages. This is the default.

`WARNING` — Display warning, error and fatal messages. Suppress informational messages.

`ERROR` — Display only error and fatal messages

- `com.bea.wlv.upgrade/migrateJSPPreference = true | false`

Specify `true` to have version 8.1 NetUI JSP tags replaced with their Beehive counterparts (where counterparts exist) as part of the upgrade process. The default is `false`, in which case the tags are instead upgraded to versions of the 8.1 tags that are compatible with the version 9.x server environment.

- `com.bea.wlv.upgrade/useJ2EESharedLibraries = true | false`

Specify `false` to have the web application libraries copied to `WEB-INF/lib`. The default is `true`

(use shared J2EE libraries).

- `com.bea.wlw.upgrade/upgradeProjectImportOverwrite = true | false`

Specify `true` to have an existing project overwritten in the event of a project name conflict. The default is `false`.

- `com.bea.wlw.upgrade/upgradeProjectImportPrefix = "PREFIX"`

Specify an optional prefix prepended to all imported projects.

- `com.bea.wlw.upgrade/upgraderPrefMoveResourceBundle = "true | false"`

Specify `true` to have files with the `.properties` extension moved from the web content folder to the source file folder. The default is `false` (make copies instead of moving the files).

- `com.bea.wlw.upgrade/upgradeHarnessReportOnly = true | false`

Specify `true` to generate an upgrade report, but not upgrade the application. The default is `false`, meaning both report and upgrade will be performed.

- `com.bea.wlw.upgrade/upgradeHarnessLogFile = LOG_FILE_LOCATION`

Specify the location of the log file this command generates. The default value is `WORKSPACE/.metadata/upgrade.log`

Related Topics

[upgrade Ant Task](#)

[How To: Use the Import Wizard to Upgrade Version 8.1 Applications](#)

[Overview: Upgrading from WebLogic Workshop 8.1](#)

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)

upgrade Ant Task

Upgrades a WebLogic Workshop application version 8.1 (SP4, SP5, or SP6) to a Workshop for WebLogic, version 9.2 workspace.

Note: This Ant task is also available as a command-line command. For more information, see [upgradeStarter command](#).

Note: Your version 8.1 application must have been upgraded to SP4, SP5, or SP6 before using this task.

This task exposes through [Ant](#) essentially the same functionality exposed by the import wizard when upgrading an application. (See [How To: Use the Import Wizard to Upgrade Version 8.1 Applications](#) for information on the wizard.) One exception to this support is that it is not possible with this task to specify a subset of an application to upgrade. For example, this task does not support specifying which projects to upgrade; all projects are upgraded.

This task requires `com.bea.wlw.upgrade.cmdline.UpgradeTask`, which is available in the `wlw-upgrade.jar`. This jar is installed by default to `WORKSHOP_HOME/eclipse/plugins/com.bea.wlw.upgrade_9.2.0`.

Environment

The task classpath (as specified by the `classpathref` attribute in the example below) must include `startup.jar` (see the `eclipseHome` attribute description below).

Attribute	Description	Required
<code>data</code>	The Eclipse workspace to which the specified 8.1 application will be imported and upgraded.	Yes.
<code>eclipseHome</code>	The Eclipse directory that contains <code>startup.jar</code> . By default this is <code>BEA_HOME/workshop92/eclipse</code> .	Yes.
<code>weblogicHome</code>	The location of the WebLogic Server root folder.	
<code>pluginCustomization</code>	The location of an optional preference file to use during import and upgrade. For more on this file, see the Remarks for the <code>upgradeStarter</code> command.	No.
<code>wlwApplication</code>	The location of the <code>WORK</code> file for the version 8.1 application that you are upgrading.	Yes.

Example

The following illustrates how to invoke this task.

```
<target name="workshopUpgrade">
```

```
<echo message="\${workshop.home}/eclipse" />
<path id="eclipse.classpath">
  <fileset dir="\${workshop.home}/eclipse/plugins"
    includes="com.bea.wlw.**/wlw-upgrade.jar" />
</path>

<taskdef name="upgradeTask"
  classname="com.bea.wlw.upgrade.cmdline.UpgradeTask"
  classpathref="eclipse.classpath" />

<upgradeTask
  data=%WORKSPACE%
  eclipseHome=%ECLIPSE_HOME%
  weblogicHome=%WL_HOME%
  pluginCustomization=%PREFS_FILE%
  wlwApplication=%WORK_FILE% />

</target>
```

Related Topics

[upgradeStarter Command](#)

[How To: Use the Import Wizard to Upgrade Version 8.1 Applications](#)

[Overview: Upgrading from WebLogic Workshop 8.1](#)

[Changes During Upgrade from WebLogic Workshop 8.1 to Version 9.2](#)