Oracle® WebLogic Server

Programming WebLogic Deployment

10*g* Release 3 (10.3)

July 2008

ORACLE

Oracle WebLogic Server Programming WebLogic Deployment, 10*g* Release 3 (10.3)

Contents

1. Introduction and Roadmap

2. Understanding the WebLogic Deployment API

3. Configuring Applications for Deployment

4. Performing Deployment Operations

CHAPTER 1

Introduction and Roadmap

The following sections describe the contents and organization of this guide—*Programming WebLogic Deployment*:

Document Scope and Audience

This document is a resource for:

- Software developers who want to understand the WebLogic Deployment API. This API adheres to the specifications described in the J2EE Deployment API standard (JSR-88) and extends the interfaces provided by that standard.
- Developers and Independent Software Vendors (ISVs) who want to perform deployment operations programmatically for WebLogic Server applications.
- System architects who are evaluating WebLogic Server or considering the use of the WebLogic Deployment API.
- Design, development, test, and pre-production phases of a software project. It does not directly address production phase administration, monitoring, or tuning application performance with the WebLogic Deployment API. The deployment API includes utilities

to make software updates during production but it mirrors the functionality of the deployment tools already available.

This guide emphasizes:

- Value-added features of the WebLogic Deployment API
- How to manage application deployment using the WebLogic Deployment API.

Note: It is assumed that the reader is familiar with J2EE concepts, the J2EE Deployment API standard (JSR-88), the Java programming language, Enterprise Java Beans (EJBs), and Web technologies.

Guide to This Document

- This chapter, Introduction and Roadmap, describes the organization and scope of this guide.
- Chapter 2, Understanding the WebLogic Deployment API, describes the packages, interfaces, and classes of the API. This section also includes information on extensions to the J2EE Deployment API standard (JSR-88), utilities, helper classes, and new concepts for WebLogic Server deployment.
- Chapter 3, Configuring Applications for Deployment, describes the process of preparing an application or deployable resource for deployment to WebLogic Server.
- Chapter 4, Performing Deployment Operations, provides information on the deployment life cycle and controls for a deployed application.

Related Documentation

For additional information about deploying applications and modules to WebLogic Server, see these documents:

- *Developing Applications for WebLogic Server* describes how to deploy applications during development using the `wldeploy` Ant task, and provides information about the WebLogic Server deployment descriptor for Enterprise Applications.
- The WebLogic Server J2EE programming guides describe the J2EE and WebLogic Server deployment descriptors used with each J2EE application and module:
  - *Developing Web Applications, Servlets, and JSPs for WebLogic Server*
  - *Programming WebLogic Enterprise JavaBeans (EJB)*

  - *Programming WebLogic Server Resource Adapters*
  - *Getting Started With WebLogic Web Services Using Jax-WS*
  - *Programming WebLogic Deployment*

- *Programming WebLogic JDBC* describes the XML deployment descriptors for JDBC application modules.
- *Programming WebLogic JMS* describes the XML deployment descriptors for JMS application modules.

New and Changed Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see "What's New in WebLogic Server" in *Release Notes*.

CHAPTER 2

Understanding the WebLogic Deployment API

The WebLogic Deployment API implements and extends the J2EE Deployment API standard (JSR-88) interfaces to provide specific deployment functionality for WebLogic Server applications. The following sections describe the structure and functionality of the WebLogic Deployment API:

The WebLogic Deployment API

Note: WebLogic Server 9.0 deprecates the use of the `weblogic.management.deploy` API used in earlier releases.

The following sections provide an overview of the WebLogic Server Deployment API:

- "When to Use the WebLogic Deployment API" on page 2-3

WebLogic Deployment API Deployment Phases

The J2EE Deployment API standard (JSR-88) differentiates between a configuration session and deployment. They are distinguished as follows:

- Application Configuration which involves the generation of descriptors for a deployment plan
- Deployment tasks such as Distributing, Starting, Stopping, Redeploying, Undeploying

In order to effectively manage the deployment process in your environment, you must use the WebLogic Deployment API to:

- "Configure an Application for Deployment" on page 2-2
- "Deploy an Application" on page 2-3

Configure an Application for Deployment

In this document, the term *configuration* refers to the process of preparing an application or deployable resource for deployment to a WebLogic Server instance. Configuring an application consists of the following phases:

- **Application Evaluation**—Inspection and evaluation of application files to determine the structure of the application and content of the embedded descriptors. See "Application Evaluation" on page 3-7.
- **Front-end Configuration**—Creation of configuration information based on content embedded within the application. This content may be in the form of WebLogic Server descriptors, defaults, and user provided deployment plans. See "Perform Front-end Configuration" on page 3-12.
- **Deployment Configuration**—Modification of individual WebLogic Server configuration values based on user inputs and the selected WebLogic Server targets. See "Customizing Deployment Configuration" on page 3-17.
- **Deployment preparation**—Generation of the final deployment plan and preliminary client-side validation of the application. See "Deployment Preparation" on page 3-23.

Deploy an Application

Application deployment is the process of distributing an application and plan to the administration server for server-side processing and application startup. See "Performing Deployment Operations" on page 4-1.

weblogic.Deployer Implementation of the WebLogic Deployment API

WebLogic Server provides a packaged deployment tool, `webLogic.Deployer`, to provide deployment services for WebLogic Server. Any deployment operation that can be implemented using the WebLogic Deployment API is implemented, either in part or in full, by `webLogic.Deployer`.

When to Use the WebLogic Deployment API

Note: `webLogic.Deployer` is the recommended deployment tool for the WebLogic Server Environment. See Deploying Applications to the WebLogic Server for information on how to use `webLogic.Deployer` and the WebLogic Server Administration Console.

You may need to implement the WebLogic Deployment API in the following cases:

- You need to model your own implementation and interface with the WebLogic Service Provider Interface (SPI). In this case, the WebLogic Deployment API deployment factory is used to obtain a `WebLogicDeploymentManager`, which extends `javax.enterprise.deploy.spi.DeploymentManager` for use with the `weblogic.deploy.api.spi`. See "Application Evaluation" on page 3-7 and the J2EE Deployment API standard.
- You need to create your own deployment interface instead using the WebLogic Server Administration Console and/or `webLogic.Deployer`. In this case, you may implement some or all "WebLogic Deployment API Deployment Phases" on page 2-2 using the WebLogic Deployment API classes and interfaces.

J2EE Deployment API Compliance

The WebLogic Deployment API classes and interfaces extend and implement the J2EE Deployment API standard (JSR-88) interfaces, which are described in the `javax.enterprise.deploy` sub-packages. The WebLogic Deployment API provides the following packages:

WebLogic Server Value-Added Deployment Features

WebLogic supports the "Product Provider" role described in the J2EE Deployment API standard (JSR-88) and provides utilities specific to the WebLogic Server environment in addition to extensible components for any J2EE network client. These extended features include:

- Support for WebLogic features, such as starting in `admin` mode or redeploying with versioning.
- Fine grain control, such as:
  - module level targeting
  - Partial Redeployment, the redeployment or removal of parts of an application.
  - Dynamic configuration changes.
- Support of WebLogic module extensions such as JMS, JDBC, Interception, and Application Specific Configuration (Custom/Configuration) modules.
- Additional operations, such as the `Deploy` verb which combines `distribute` and `start`.

Note: The WebLogic Deployment API does not support an automated fallback procedure for a failed application update. The policy and procedures for this behavior must be defined and configured by the developers and administrators for each deployment environment.

The Service Provider Interface Package

As a J2EE product provider, Oracle extends the Sun Microsystems `javax` Service Provider Interface (SPI) package to provide specific configuration and deployment control for WebLogic Server. The core interface for this package is the `DeploymentManager`, from which all other deployment activities are initiated, monitored, and controlled.

The `WebLogicDeploymentManager` interface provides WebLogic Server extensions to the `javax.enterprise.deploy.spi.DeploymentManager` interface. A

`WebLogicDeploymentManager` object is a stateless interface for the Weblogic Server deployment framework. It provides basic deployment features as well as extended WebLogic Server deployment features such as production redeployment and partial deployment for modules in an Enterprise Application. You generally acquire a `WebLogicDeploymentManager` object using `SessionHelper.getDeploymentManager` method from the `SessionHelper` helper class from the Tools package. See "Application Evaluation" on page 3-7.

The following sections provide basic information on the functionality of the WebLogic Server SPI:

weblogic.deploy.api.spi

The `weblogic.deploy.api.spi` package provides the interfaces required to configure and deploy applications to a target (see "Support for Querying WebLogic Target Types" on page 2-6 for valid target types). This package enables you to create deployment tools that can implement a WebLogic Server-specific deployment configuration for an Enterprise Application or stand-alone module.

`weblogic.deploy.api.spi` includes the `WebLogicDeploymentManager` interface. Use this deployment manager to perform all deployment-related operations such as distributing, starting, and stopping applications in WebLogic Server. The `WebLogicDeploymentManager` also provides important extensions to the J2EE `DeploymentManager` interface for features such as module-level targeting for Enterprise Application modules, production redeployment, application versioning, application staging modes, and constraints on Administrative access to deployed applications.

The `WebLogicDeploymentConfiguration` and `WebLogicDConfigBean` classes in the `weblogic.deploy.api.spi` package represent the deployment and configuration descriptors (WebLogic Server deployment descriptors) for an application.

- A `WebLogicDeploymentConfiguration` object is a wrapper for a deployment plan.

- A `WebLogicDConfigBean` encapsulates the properties in Weblogic deployment descriptors.

weblogic.deploy.api.spi.factories

This package contains only one interface, the `WebLogicDeploymentFactory`. This is a WebLogic extension to `javax.enterprise.deploy.spi.factories.DeploymentFactory`. Use this factory interface to select and allocate `DeploymentManager` objects that have different characteristics. The `WebLogicDeploymentManager` characteristics are defined by public fields in the `WebLogicDeploymentFactory`.

Module Targeting

Module targeting is deploying specific modules in an application to different targets as opposed to deploying all modules to the same set of targets as specified by JSR-88. Module targeting is supported by the `WebLogicDeploymentManager.createTargetModuleID` methods.

The `WebLogicTargetModuleID` class contains the WebLogic Server extensions to the `javax.enterprise.deploy.spi.TargetModuleID` interface. This class is closely related to the configured `TargetInfoMBeans` (`AppDeploymentMBean` and `SubDeploymentMBean`). The `WebLogicTargetModuleID` class provides more detailed descriptions of the application modules and their relationship to targets than those in `TargetInfoMBeans`. See "Target Objects" on page 4-12.

Support for Querying WebLogic Target Types

For WebLogic Server, the `WebLogicTarget` class provides a direct interface for maintaining the target types available to WebLogic Server. Target accessor methods are described in Table 2-1.

Table 2-1 Target Accessor Methods

Method	Description
`boolean isCluster()`	Indicates whether this target represents a cluster target.
`boolean isJMSServer()`	Indicates whether this target represents a JMS server target.
`boolean isSAFAgent()`	Indicates whether this target represents a SAF agent target.

Table 2-1 Target Accessor Methods

`boolean isServer()`	Indicates whether this target represents a server target.
`boolean isVirtualHost()`	Indicates whether this target represents a virtual host target.

Server Staging Modes

The staging mode of an application affects its deployment behavior. The application's staging behavior is set using `DeploymentOptions.setStageMode(`*`stage mode`*`)` where the value of *`stage mode`* is one of the following:

- `STAGE`—Force copying of files to target servers.
- `NO_STAGE`—Files are not copied to target servers.
- `EXTERNAL_STAGE`—Files are staged manually.

DConfigBean Validation

The property setters in a `DConfigBean` reject attempts to set invalid values. This includes property type validation such as attempting to set an integer property to a non-numeric value. Some properties perform semantic validations, such as ensuring a maximum value is not smaller than its associated minimum value.

The Model Package

These classes are the WebLogic Server extensions to and implementations of the `javax.enterprise.deploy.model` interfaces. The model interfaces describes the standard elements, such as deployment descriptors, of a J2EE application.

- "weblogic.deploy.api.model" on page 2-7
- "Accessing Deployment Descriptors" on page 2-8

weblogic.deploy.api.model

This package contains the interfaces used to represent the J2EE configuration of a deployable object. A deployable object is a deployment container for an Enterprise Application or standalone module.

The WebLogic Server implementation of the `javax.enterprise.deploy.model` interfaces enable you to work with applications that are stored in a WebLogic Server application installation directory, a formal directory structure used for managing application deployment files, deployments, and external WebLogic Deployment descriptors generated during the configuration process. See Preparing Applications and Modules for Deployment for more information about the layout of an application installation directory. It supports any J2EE application, with extensions to support applications residing in an application installation directory.

Note: `weblogic.deploy.api.model` does not support dynamic changes to J2EE deployment descriptor elements during configuration and therefore does not support registration and removal of Xpath listeners. `DDBean.addXPathListener` and `removeXPathListener` are not supported.

The `WebLogicDeployableObject` class and `WebLogicDDBean` interface in the `weblogic.deploy.api.model` package represent the standard deployment descriptors in an application.

Accessing Deployment Descriptors

J2EE Deployment API dictates that J2EE deployment descriptors be accessed through a `DeployableObject`. A `DeployableObject` represents a module in an application. Elements in the descriptors are represented by `DDBeans`, one for each element in a deployment descriptor. The root element of a descriptor is represented by a `DDBeanRoot` object. All of these interfaces are implemented in corresponding interfaces and classes in this package.

The `WebLogicDeployableObject` class, which is the Weblogic Server implementation of `DeployableObject`, provides the `createDeployableObject` methods, which create the `WebLogicDeployableObject` and `WebLogicDDBean` for the application's deployment descriptors. Basic configuration tasks are accomplished by associating the `WebLogicDDBean` with a `WebLogicDConfigBean`, which represent the server configuration properties required for deploying the application on a WebLogic Server. See “Application Evaluation” on page 3-7.

Unlike a `DConfigbean`, which contain configuration information specifically for a server environment (in this case WebLogic server instance), a `DDBean` object takes in the general deployment descriptor elements for the application. For example, if you were deploying a Web application, the deployment descriptors in `WebLogicDDBeans` come from `WEB-INF/web.xml` file in the `.war` archive. The information for the `WebLogicDConfigBeans` would come from `WEB-INF/weblogic.xml` in the `.war` archive based on the `WebLogicDDBeans`. Though they serve the same fundamental purpose of holding configuration information, they are logically separate as a `DDBean` describes the application while a `DConfigBeans` configures the application for a specific environment.

Both of these objects are generated during the initiation of a configuration session. The `WebLogicDeployableObject`, `WebLogicDDBeans`, and `WebLogicDConfigBeans` are all instantiated and manipulated in a configuration session. See "Overview of the Configuration Process" on page 3-1.

The Shared Package

The following sections provide information on classes that represent WebLogic Server-specific deployment commands, module types, and target types as classes:

weblogic.deploy.api.shared

The `weblogic.deploy.api.shared` package provides classes that represent the WebLogic Server-specific deployment commands, module types, and target types as classes. These objects can be shared by "weblogic.deploy.api.model" on page 2-7 and "weblogic.deploy.api.spi" on page 2-5 packages.

The definitions of the standard `javax.enterprise.deploy.shared` classes `ModuleType` and `CommandType` are extended to provide support for:

- The module type, see "Support for Module Types" on page 2-10
- Commands, see "Command Types for Deploy and Update" on page 2-9

The `WebLogicTargetType` class, which is not required by the J2EE Deployment API standard (JSR-88), enumerates the different types of deployment targets supported by WebLogic Server. This class does not extend a javax deployment class. See "Support for all WebLogic Server Target Types" on page 2-10.

Command Types for Deploy and Update

The `deploy` and `update` command types are added to the required command types defined in the `javax.enterprise.spi.shared` package and are available to a `WebLogicDeploymentManager`.

Support for Module Types

Supported module types include JMS, JDBC, Interception, WSEE, Config, and WLDF. These are defined in the `weblogic.deploy.api.shared.WebLogicModuleType` class as fields.

Support for all WebLogic Server Target Types

Targets, which were not implemented in the J2EE Deployment API specification, are implemented in the WebLogic Deployment API. The valid target values are:

- Cluster
- JMS Server
- SAF (Store-and-Forward) Agent
- Server
- Virtual Host

These are enumerated field values in the `weblogic.deploy.api.shared.WebLogicTargetType` class.

The Tools Package

The following sections provide information on API tools you can use to perform common deployment tool tasks with a minimum of number of controls and explicit object manipulations:

- "weblogic.deploy.api.tools" on page 2-10
- "SessionHelper" on page 2-11
- "Deployment Plan Creation" on page 2-12

weblogic.deploy.api.tools

The `weblogic.deploy.api.tools` package provides convenience classes that can help you:

- Obtain a `WebLogicDeploymentManager`
- Populate a configuration for an application
- Create a new or updated deployment plan

The classes in the tools package are not extensions of the J2EE Deployment API standard (JSR-88) interfaces. They provide easy access to deployment operations provided by the WebLogic Deployment API.

SessionHelper

Although configuration sessions can be controlled from a `WebLogicDeploymentManager` directly, `SessionHelper` provides simplified methods. If your tools code directly to WebLogic Server's J2EE Deployment API implementation, you should always use `SessionHelper`.

Use `SessionHelper` to obtain a `WebLogicDeploymentManager` with one method call. To do this effectively, it must be able to locate the application. The `SessionHelper` views an application and deployment plan artifacts using an "install root" abstraction, which ideally is the actual organization of the application. The install root appears as follows:

```
install-root (eg myapp)
-- app
----- archive (eg myapp.ear)
-- plan
----- deployment plan (eg plan.xml)
----- external descriptors (eg META-INF/weblogic-application.xml...)
```

There is no requirement to mandate that this structure be used for applications. It is a preferred approach because it serves to keep the application and its configuration artifacts under a common root and provides `SessionHelper` with a format it can interpret.

`SessionHelper.getModuleInfo()` returns an object that is useful for understanding the structure of an application without having to work directly with `DDBeans` and `DeployableObjects`. It provides such information as

- Names and types of modules and submodules in the application
- Names of web services provided by the application
- Context roots for web applications
- Names of enterprise beans in an EJB

Internally, the deployment descriptors are represented as descriptor bean trees and trees of typed Java Bean objects that represent the individual descriptor elements. These bean trees are easier to work with than the more generic `DDBean` and `DConfigBean` objects. The descriptor bean trees for

each module are directly accessible from the associated `WebLogicDDBeanRoot` and `WebLogicDConfigBeanRoot` objects for each module using their `getDescriptorBean` methods. Modifying the bean trees obtained from a `WebLogicDConfigBean` has the same effect as modifying the associated `DConfigBean`, and therefore the application's deployment plan.

Deployment Plan Creation

`weblogic.PlanGenerator` creates a deployment plan template based on the J2EE and WebLogic Server descriptors included in an application. The resulting plan describes the application structure, identifies all deployment descriptors, and exports a subset of the application's configurable properties. Export properties to expose them to tools like the WebLogic Server console which then uses the plan to assist the administrator in providing appropriate values for those properties. By default, the `weblogic.PlanGenerator` tool only exports application dependencies; those properties required for a successful deployment. This behavior can be overridden using of the following options:

- `Dependencies`: Export resources referenced by the application (default)
- `Declarations`: Export resources defined by the application
- `Configurables`: Export non-resource oriented configurable properties
- `Dynamics`: Export properties that may be changed in a running application
- `All`: Export all changeable properties
- `None`: Export no properties

CHAPTER 3

Configuring Applications for Deployment

In the context of this document, configuration is the process of preparing an application or deployable resource for deployment to a WebLogic Server instance. Most configuration information for an application is provided in its deployment descriptors. Certain elements in these descriptors refer to external objects and may require special handling depending on the server vendor. WebLogic Server uses descriptor extensions—WebLogic Server specific deployment descriptors. The mapping between standard descriptors and WebLogic Server descriptors is managed using `DDBeans` and `DConfigBeans`.

The following sections describe how to configure an application for deployment using the WebLogic Deployment API:

Overview of the Configuration Process

This section provides information on the basic steps a deployment tool must implement to configure an application for deployment:

1. **Application Evaluation**—Inspection and evaluation of application files to determine the structure of the application and content of the embedded descriptors.
   - Initialize a deployment session by obtaining a `WebLogicDeploymentManager`. See "Application Evaluation" on page 3-7.
   - Create a `WebLogicJ2eeApplicationObject` or `WebLogicDeployableObject` to represent the J2EE Configuration of an Enterprise Application (EAR) or standalone module (WAR, EAR, RAR, or CAR). If the object is an EAR, child objects are generated. See J2EE Deployment API standard (JSR-88) and "Create a Deployable Object" on page 3-11.
2. **Front-end Configuration**—Creation of configuration information based on content embedded within the application. This content may be in the form of WebLogic Server descriptors, defaults, and user provided deployment plans.
   - Create a `WebLogicDeploymentConfiguration object` to represent the WebLogic Server configuration of an application. This is the first step in creating a deployment plan for this object. See "Deployment Configuration" on page 3-13.
   - Restore existing WebLogic Server configuration values from an existing deployment plan, if available. See "Perform Front-end Configuration" on page 3-12.
3. **Deployment Configuration**—Modification of individual WebLogic Server configuration values based on user inputs and the selected WebLogic Server targets.

   A deployment tool must provide the ability to modify individual WebLogic Server configuration values based on user inputs and selected WebLogic Server targets. See "Customizing Deployment Configuration" on page 3-17.
4. **Deployment preparation**—Generation of the final deployment plan and preliminary client-side validation of the application.

   A deployment tool must have the ability to save the modified WebLogic Server configuration information to a new deployment plan or to variable definitions in an existing Deployment Plan.

Types of Configuration Information

The following sections provide background information on the types of configuration information, how it is represented, and the relationship between J2EE and WebLogic Server descriptors:

J2EE Configuration

The J2EE configuration for an application defines the basic semantics and runtime behavior of the application, as well as the external resources that are required for the application to function. This configuration information is stored in the standard J2EE deployment descriptor files associated with the application, as listed in Table 3-1.

Table 3-1 Standard J2EE Deployment Descriptors

Application or Standalone Module	J2EE Descriptor
Enterprise Application	`META-INF/application.xml`
Web Application	`WEB-INF/web.xml`
Enterprise JavaBean	`META-INF/ejb.xml`
Resource Adapter	`META-INF/ra.xml`
Client Application Archive	`META-INF/application-client.xml`

Complete and valid J2EE deployment descriptors are a required input to any application configuration session.

Because the J2EE configuration controls the fundamental behavior of an application, the J2EE descriptors are typically defined only during the application development phase, and are not modified when the application is later deployed to a different environment. For example, when you deploy an application to a testing or production domain, the application's behavior (and therefore its J2EE configuration) should remain the same as when application was deployed in the development domain. See "Perform Front-end Configuration" on page 3-12 for more information.

WebLogic Server Configuration

The WebLogic Server descriptors provide for enhanced features, resolution of external resources, and tuning associated with application semantics. Applications may or may not have these descriptors embedded in the application. The WebLogic Server configuration for an application:

- Binds external resource names to resource definitions in the J2EE deployment descriptor so that the application can function in a given WebLogic Server domain
- Defines tuning parameters for the application containers
- Provides enhanced features for J2EE applications and standalone modules

The attributes and values of a WebLogic Server configuration are stored in the WebLogic Server deployment descriptor files, as shown in Table 3-2.

Table 3-2 WebLogic Server Deployment Descriptors

Application or Standalone Module	WebLogic Server Descriptor
Enterprise Application	`META-INF/weblogic-application.xml`
Web Application	`WEB-INF/weblogic.xml`
Enterprise JavaBean	`META-INF/weblogic-ejb-jar.xml`
Resource Adapter	`META-INF/weblogic-ra.xml`
Client Archive	`META-INF/weblogic-appclient.xml`

Because different WebLogic Server domains provide different types of external resources and different levels of service for the application, the WebLogic Server configuration for an application typically changes when the application is deployed to a new environment. For example, a production staging domain might use a different database vendor and provide more usable memory than a development domain. Therefore, when moving the application from development to the staging domain, the application's WebLogic Server descriptor values need to be updated in order to make use of the new database connection and available memory.

The primary job of a deployment configuration tool is to ensure that an application's WebLogic Server configuration is valid for the selected WebLogic targets.

Representing J2EE and WebLogic Server Configuration Information

Both the J2EE deployment descriptors and any available WebLogic Server descriptors are used as inputs to the application configuration process. You use the deployment API to represent both the J2EE configuration and WebLogic Server configuration as Java objects.

The J2EE configuration for an application is obtained by creating either a `WebLogicJ2eeApplicationObject` for an EAR, or a `WeblogicDeployableObject` for a standalone module. (A `WebLogicJ2eeApplicationObject` contains multiple `DeployableObject` instances to represent individual modules included in the EAR.)

Each `WebLogicJ2eeApplicationObject` or `WeblogicDeployableObject` contains a `DDBeanRoot` to represent a corresponding J2EE deployment descriptor file. J2EE descriptor properties for EARs and modules are represented by one or more `DDBean` objects that reside beneath the `DDBeanRoot`. `DDBean` components provide standard getter methods to access individual deployment descriptor properties, values, and nested descriptor elements.

DDBeans

`DDBeans` are described by the `javax.enterprise.deploy.model` package. These objects provide a generic interface to elements in standard deployment descriptors, but can also be used as an XPath based mechanism to access arbitrary XML files that follow the basic form of the standard descriptors. Examples of such files would be WebLogic Server descriptors and Web Services descriptors.

The `DDBean` representation of a descriptor is a tree of `DDBeans`, with a specialized `DDBean`, a `DDBeanRoot`, at the root of the tree. `DDBeans` provide accessors for the element name, id attribute, root, and text of the descriptor element they represent.

The `DDBeans` for an application are populated by the model plug-in, the tool provider implementation of `javax.enterprise.deploy.model`. An application is represented by the `DeployableObject` interface. The WebLogic Server implementation of this interface is a public class, weblogic.deploy.api.model.`WebLogicDeployableObject`. A WebLogic Server based deployment tool acquires an instance of `WebLogicDeployableObject` object for an application using the `createDeployableObject` factory methods. This results in the `DDBean` tree for the application being created and populated by the elements in the J2EE descriptors embedded in the application. If the application is an EAR, multiple `WebLogicDeployableObject` objects are

created. The root `WebLogicDeployableObject`, extended as `WebLogicJ2eeApplicationObject`, would represent the EAR module, with its child `WebLogicDeployableObject` instances being the modules contained within the application, such as WARs, EJBs, RARs and CARs.

The Relationship Between J2EE and WebLogic Server Descriptors

J2EE descriptors and WebLogic Server descriptors are directly related in the configuration of external resources. A J2EE descriptor defines the types of resources that the application requires to function, but it does not identify the actual resource names to use. The WebLogic Server descriptor binds the resource definition in the J2EE descriptor name to the name of an actual resource in the target domain.

The process of binding external resources is a required part of the configuration process. Binding resources to the target domain ensures that the application can locate resources and successfully deploy.

J2EE descriptors and WebLogic Server descriptors are also indirectly related in the configuration of tuning parameters for WebLogic Server. Although no elements in the standard J2EE descriptors *require* tuning parameters to be set in WebLogic Server, the presence of individual descriptor files indicates which tuning parameters are of interest during the configuration of an application. For example, although the `ejb.xml` descriptor does not contain elements related to tuning the WebLogic Server EJB container, the presence of an `ejb.xml` file in the J2EE configuration indicates that tuning properties can be configured before deployment.

DConfigBeans

`DConfigBeans` (config beans) are the objects used to convey server configuration requirements to a deployment tool, and are also the primary source of information used to create deployment plans. Config beans are Java Beans and can be introspected for their properties. They also provide basic property editing capabilities.

`DConfigBeans` are created from information in embedded WebLogic Server descriptors, deployment plans, and input from an IDE deployment tool.

A `DConfigBean` is potentially created for every weblogic Descriptor element that is associated with a dependency of the application. Descriptors are entities that describe resources that are available to the application, represented by a JNDI name provided by the server.

Descriptors are parsed into memory as a typed bean tree while setting up a configuration session. The `DConfigBean` implementation classes delegate to the WebLogic Server descriptor beans. Only beans with dependency properties, such as resource references, have a `DConfigBean`. The root of descriptor always has a `DConfigBeanRoot`.

Bean Property accessors return a child `DConfigBean` for elements that require configuration or a descriptor bean for those that do not. Property accessors return data from the descriptor beans.

Modifications to bean properties result in plan overrides. Plan overrides for existing descriptors are handled using variable assignments. If the application does not come with the relevant WebLogic Server descriptors, they are automatically created and placed in an external plan directory. For external deployment descriptors, the change is made directly to the descriptor. Embedded descriptors are never modified on disk.

Application Evaluation

Application Evaluation consists of obtaining a deployment manager and a deployable object container for your application. Use the following steps:

1. Obtain a deployment factory class by specifying its name, `weblogic.deployer.spi.factories.internal.DeploymentFactoryImpl`.
2. Register the factory class with a `javax.enterprise.deploy.spi.DeploymentFactoryManager` instance.

   For instance:

```
Class WlsFactoryClass =
   Class.forname("weblogic.deployer.spi.factories.internal.DeploymentFacto
ryImpl");
DeploymentFactory myDeploymentFactory =
   (DeploymentFactory) WlsFactoryClass.newInstance();
DeploymentFactoryManager.getInstance().registerDeploymentFactory(myDeploym
entFactory);
```

3. "Obtain a Deployment Manager" on page 3-8
4. "Create a Deployable Object" on page 3-11.

Obtain a Deployment Manager

The following sections provide information on how to obtain a deployment manager:

Types of Deployment Managers

WebLogic Server provides a single implementation for `javax.enterprise.deploy.spi.DeploymentManager` that behaves differently depending on the `URI` specified when instantiating the class from a factory. WebLogic Server provides two basic types of deployment manager:

- A *disconnected deployment manager* has no connection to a WebLogic Server instance. Use a disconnected deployment manager to configure an application on a remote client machine. It cannot be used it to perform deployment operations. (For example, a deployment tool cannot use a disconnected deployment manager to distribute an application.)
- A *connected deployment manager* has a connection to the Administration Server for the WebLogic Server domain, and by a deployment tool to both to configure and deploy applications.

A connected deployment manager is further classified as being either local to the Administration Server, or running on a remote machine that is connected to the Administration Server. The local or remote classification determines whether file references are treated as being local or remote to the Administration Server.

Table 3-3 summarizes deployment manager types.

Table 3-3 WebLogic Server Deployment Manager Usage

DeploymentManager Connectivity	Type	Usage	Notes
Disconnected	n/a	Configuration tools only	Cannot perform deployment operations
Connected	Local	Configuration and deployment tools local to the Administration Server	All files are local to the Administration Server machine
	Remote	Configuration and Deployment for Tools on a remote machine (not on the Administration Server)	Distribution and Deployment operations cause local files to be uploaded to the Administration Server

Connected and Disconnected Deployment Manager URIs

Each `DeploymentManager` obtained from the `WebLogicDeploymentFactory` supports WebLogic Server extensions. When creating deployment tools, obtain a specific type of deployment manager by calling the correct method on the deployment factory instance and supplying a string constant defined in `weblogic.deployer.spi.factories.WebLogicDeploymentFactory` that describes the type of deployment manager required. Connected deployment managers require a valid server `URI` and credentials to the method in order to obtain a connection to the Administration Server.

Table 3-4 summarizes the method signatures and constants used to obtain the different types of deployment managers.

Table 3-4 URIs for Obtaining a WebLogic Server Deployment Manager

Type of Deployment Manager	Method	Argument
disconnected	`getDisconnectedDeploymentManager()`	String value of `WebLogicDeploymentFactory.LOCAL_DM_URI`
connected, local	`getDeploymentManager()`	`URI` consisting of: • `WebLogicDeploymentFactory.LOCAL_DM_URI` • Administration Server host name • Administration Server port • Administrator username • Administrator password
connected, remote	`getDeploymentManager()`	`URI` consisting of: • `WebLogicDeploymentFactory.REMOTE_DM_URI` • Administration Server host name • Administration Server port • Administrator username • Administrator password

The sample code in Listing 3-1 shows how to obtain a disconnected deployment manager.

Listing 3-1 Obtaining a Disconnected Deployment Manager

```
Class WlsFactoryClass = Class.forname("weblogic.deployer.spi.factories.internal.DeploymentFactoryImpl");
DeploymentFactory myDeploymentFactory = (DeploymentFactory) WlsFactoryClass.newInstance();
DeploymentFactoryManager.getInstance().registerDeploymentFactory(myDeploymentFactory);
```

```
WebLogicDeploymentManager myDisconnectedManager = (WebLogicDeploymentManag
er)myDeploymentFactory.getDisconnectedDeploymentManager(WebLogicDeployment
Factory.LOCAL_DM_URI);
```

The deployment factory contains a helper method, `createUri()` to help you form the `URI` argument for creating connected deployment managers. For example, to create a disconnected remote deployment manager, replace the final line of code with:

```
(WebLogicDeploymentManager)myDeploymentFactory.getDeploymentManager(myDepl
oymentFactory.createUri(WebLogicDeploymentFactory.REMOTE_DM_URI,
“localhost”, “7001”, “weblogic”, “weblogic”));
```

Using SessionHelper to Obtain a Deployment Manager

The `SessionHelper` helper class provides several convenience methods to help you easily obtain a deployment manager without manually creating and registering the deployment factories as shown in Listing 3-1. The `SessionHelper` code required to obtain a disconnected deployment manager consists of a single line:

```
   DeploymentManager myDisconnectedManager = SessionHelper.getDisconnected
DeploymentManager();
```

You can use the `SessionHelper` to obtain a connected deployment manager, as shown below:

```
   DeploymentManager myConnectedManager = SessionHelper.getDeploymentManag
er(“adminhost”, “7001”, “weblogic”, “weblogic”));
```

This method assumes a remote connection to an administration server (`adminhost`). See the Javadocs for more information about `SessionHelper`.

Create a Deployable Object

The following sections provide information on how to create a deployable object, which is the container your deployment tool uses to deploy applications. Once you have initialized a configuration session by “Obtain a Deployment Manager” on page 3-8, create a deployable object for your deployment tool in one of the following ways:

- “Using the WebLogicDeployableObject class” on page 3-12
- “Using SessionHelper to obtain a Deployable Object” on page 3-12

Using the WebLogicDeployableObject class

The direct approach uses the WebLogicDeployableObject class of the model package as shown below:

```
WebLogicDeployableObject myDeployableObject = WebLogicDeployableObject.
createWebLogicDeployableObject("myAppFileName");
```

Once the deployable object is created, a configuration can be created for the applications deployment.

Using SessionHelper to obtain a Deployable Object

The `SessionHelper` helper class provides a convenient method to obtain a deployable object. The `SessionHelper` code required to obtain a deployable object is shown below:

```
SessionHelper.setApplicationRoot(root);
WebLogicDeployableObject myDeployableObject = SessionHelper.getDeployab
leObject();
```

There is no application specified in the `getDeployableObject()` call. `SessionHelper` uses the application in the root directory set by `setApplicationRoot()`. Once the application root directory is set, `SessionHelper` can be used to perform other operations, such as explicitly naming the dispatch file location or the deployment plan location.

You can also set the application file name using the `setApplication` method as shown below:

```
SessionHelper.setApplication(AppFileName);
```

This method allows you to continue using `SessionHelper` independent of the directory structure. The `getDeployableObject` method returns the application specified.

Perform Front-end Configuration

Front-end configuration involves creating a `WebLogicDeploymentPlan` and populating it and its associated bean trees with configuration information:

- "What is Front-end Configuration" on page 3-13
- "Deployment Configuration" on page 3-13
- "Validating a Configuration" on page 3-17

What is Front-end Configuration

Front-end configuration phase is consists of two logical operations:

- Loading information from a deployment plan to a deployment configuration. If a deployment configuration does not yet exist, this includes creating a `WebLogicDeploymentConfiguration` object to represent the WebLogic Server configuration of an application. This is the first step in the process of process of creating a deployment plan for this object.
- Restoring any existing WebLogic Server configuration values from an existing deployment plan.

A deployment tool must be able to:

Extract information from a deployment configuration. The deployment configuration is the active java object that is used by the Deployment Manager to obtain configuration information. The deployment plan exists outside of the application so that it can be changed without manipulating the application.

A deployment plan is an XML document that contains the environmental configuration for an application and is sometimes referred to as an application's front-end configuration. A deployment plan:

- Separates the environment specific details of an application from the logic of the application.
- Is not required for every application. However, a deployment plan typically exists for each environment an application is deployed to.
- Describes the application structure, such as what modules are in the application.
- Allows developers and administrators to update the configuration of an application without modifying the application archive.
- Contains environment specific descriptor override information (tunables). By modifying a deployment plan, you can provide environment specific values for tunable variables in an application.

Deployment Configuration

The server configuration for an application is encapsulated in the `javax.enterprise.deploy.spi.DeploymentConfiguration` interface. A `DeploymentConfiguration` provides an object representation of a deployment plan. A

DeploymentConfiguration is associated with a DeployableObject using the DeploymentManager.createConfiguration method. Once a DeploymentConfiguration object is created, a DConfigBean tree representing the configurable and tunable elements contained in any and all WebLogic Server descriptors is available. If there are no WebLogic Server descriptors for an application, then a DConfigBean tree is created using available default values. Binding properties that have no defaults are left unset.

When creating a deployment tool, you must ensure that the DConfigBean tree is fully populated before the tool distributes an application.

Example Code

The following code provides an example on how to populate DConfigBeans:

Listing 3-2 Example Code to Populate DConfigBeans

```
public class DeploymentSession {
  DeploymentManager dm;
  DeployableObject dObject = null;
  DeploymentConfiguration dConfig = null;
  Map beanMap = new HashMap();
.
.
.
  // Assumes app is a web app.
  public void initializeConfig(File app) throws Throwable {
    /**
    * Init the wrapper for the DDBeans for this module. This example assumes
    * it is using the WLS implementation of the model api.
     */
    dObject= WebLogicDeployableObject.createDeployableObject(app);
    //Get basic configuration for the module
    dConfig = dm.createConfiguration(dObject);
    /**
     * At this point the DeployableObject is populated. Populate the
     * DeploymentConfigurationbased on its content.
     * We first ask the DeployableObject for its root.
     */
```

```
    DDBeanRoot root = dObject.getDDBeanRoot();
    /**
     * The root DDBean is used to start the process of identifying the
     * necessary DConfigBeans for configuring this module.
     */
    System.out.println("Looking up DCB for "+root.getXpath());
    DConfigBeanRoot rootConfig = dConfig.getDConfigBeanRoot(root);
    collectConfigBeans(root, rootConfig);
    /**
     * The DeploymentConfiguration is now initialized, although not
necessarily
     * completely setup.
     */
    FileOutputStream fos = new FileOutputStream("test.xml");
    dConfig.save(fos);

  }

  // bean and dcb are a related DDBean and DConfigBean.
 private void collectConfigBeans(DDBean bean, DConfigBean dcb) throws
Throwable{
    DConfigBean configBean;
    DDBean[] beans;
    if (dcb == null) return;
    /**
     * Maintain some sort of mapping between DDBeans and DConfigBeans
     * for later processing.
     */
    beanMap.put(bean,dcb);
    /**
     * The config bean advertises xpaths into the web.xml descriptor it
     * needs to know about.
     */
    String[] xpaths = dcb.getXpaths();
    if (xpaths == null) return;
    /**
     * For each xpath get the associated DDBean and collect its associated
    * DConfigBeans. Continue this recursively until we have all DDBeans and
```

```
     * DConfigBeans collected.
     */
    for (int i=0; i<xpaths.length; i++) {
      beans = bean.getChildBean(xpaths[i]);
      for (int j=0; j<beans.length; j++) {
        /**
         * Init the DConfigBean associated with each DDBean
         */
        System.out.println("Looking up DCB for "+beans[j].getXpath());
        configBean = dcb.getDConfigBean(beans[j]);
        collectConfigBeans(beans[j], configBean);
      }
    }
```

This example merely iterates through the `DDBean` tree, requesting the `DConfigBean` for each `DDBean` to be instantiated.

`DeploymentConfiguration` objects may be persisted as deployment plans using `DeploymentConfiguration.save()`. A deployment tool may allow the user to import a saved deployment plan into the `DeploymentConfiguration` object instead of populating it from scratch. `DeploymentConfiguration.restore()` provides this capability. This supports the idea of having a repository of deployment plans for an application, with different plans being applicable to different environments.

Similarly the `DeploymentConfiguration` may be pieced together using partial plans, which were presumably saved in a repository from a previous configuration session. A partial plan maps to a module-root of a `DConfigBean` tree. `DeploymentConfiguration.saveDConfigBean()` and `DeploymentConfiguration.restoreDConfigBean()` provide this capability.

Parsing of the WebLogic Server descriptors in an application occurs automatically when a `DeploymentConfiguration` is created. The descriptors ideally conform to the most current schema. For older applications that include descriptors based on WebLogic Server 8.1 and earlier DTDs, a transformation is performed. Old descriptors are supported but they cannot be modified using a deployment plan. Therefore, any DOCTYPE declarations must be converted to name space references and element specific transformations must be performed.

Reading In Information with SessionHelper

`SessionHelper.initializeConfiguration` processes all standard and WebLogic Server descriptors in the application.

Prior to invoking `initializeConfiguration`, you can specify an existing deployment plan to associate with the application using the `SessionHelper.setPlan()` method. With a plan set, you can read in a deployment plan using the `DeploymentConfiguration.restore()` method. In addition, the `DeploymentConfiguration.initializeConfiguration()` method automatically restores configuration information once a plan is set.

When initiating a configuration session with the `SessionHelper` class, you can easily initiate and fill a `deploymentConfiguration` object with deployment plan information as illustrated below:

```
   DeploymentManager dm = SessionHelper.getDisconnectedDeploymentManager()
;
 SessionHelper helper = SessionHelper.getInstance(dm);
 // specify location of archive
 helper.setApplication(app);
 // specify location of existing deployment plan
 helper.setPlan(plan);
 // initialize the configuration session
 helper.initializeConfiguration();
 DeploymentConfiguration dc = helper.getConfiguration();
```

The above code produces the deployment configuration and its associated `WebLogicDDBeanTree`.

Validating a Configuration

Validation of the configuration occurs mostly during the parsing of the descriptors which occurs when an application's descriptors are processed. Validation consists of ensuring the descriptors are valid XML documents and that the descriptors conform to their respective schemas.

Customizing Deployment Configuration

The Customizing Deployment Configuration phase involves modifying individual WebLogic Server configuration values based on user inputs and the selected WebLogic Server targets.

- "Modifying Configuration Values" on page 3-18

- "Targets" on page 3-23
- "Application Naming" on page 3-23

Modifying Configuration Values

In this phase, a configuration is only as good as the descriptors or pre-existing plan associated with the application. The `DConfigBeans` are designed as Java Beans and can be introspected, allowing a tool to present their content in some meaningful way. The properties of a `DConfigBean` are, for the most part, those that are configurable. Key properties (those that provide uniqueness) are also exposed. Setters are only exposed on those properties that can be safely modified. In general, properties that describe application behavior are not modifiable. All properties are typed as defined by the descriptor schemas.

The property getters return subordinate `DConfigBeans`, arrays of `DConfigBeans`, descriptor beans, arrays of descriptor beans, simple values (primitives and `java.lang` objects), or arrays of simple values. Descriptor beans represent descriptor elements that, while modifiable, do not require `DConfigBean` features, meaning there are no standard descriptor elements they are directly related to. Editing a configuration is accomplished by invoking the property setters.

The Java JSR-88 `DConfigBean` class allows a tool to access beans using the `getDConfigBean(DDBean)` method or introspection. The former approach is convenient for a tool that presents the standard descriptor based on the `DDBeans` in the application's `DeployableObject` and provides direct access to each `DDBean`'s configuration (its `DConfigBean`). This provides configuration of the essential resource requirements an application may have. Introspection allows a tool to present the application's entire configuration, while highlighting the required resource requirements.

Introspection is required in both approaches in order to present or modify descriptor properties. The difference is in how a tool presents the information:

- Driven by standard descriptor content or
- WebLogic Server descriptor content.

A system of modifying configuration information must include a user interface to ask for configuration changes. See Listing 3-3.

Listing 3-3 Code Example to Modify Configuration Information

```
.
.
.
// Introspect the DConfigBean tree and ask for input on properties with
setters
  private  void processBean(DConfigBean dcb) throws Exception {
    if (dcb instanceof DConfigBeanRoot) {

System.out.println("Processing configuration for descriptor: "+dcb.getDDBe
an().getRoot().getFilename());
    }
    // get property descriptor for the bean
    BeanInfo info =
       Introspector.getBeanInfo(dcb.getClass(),Introspector.USE_ALL_BEANIN
FO);
    PropertyDescriptor[] props = info.getPropertyDescriptors();
    String bean = info.getBeanDescriptor().getDisplayName();
    PropertyDescriptor prop;
    for (int i=0;i<props.length;i++) {
      prop = props[i];
      // only allow primitives to be updated
      Method getter = prop.getReadMethod();
      if (isPrimitive(getter.getReturnType())) // see isPrimitive method
below
      {
        writeProperty(dcb,prop,bean); //see writeProperty method below
      }
      // recurse on child properties
      Object child = getter.invoke(dcb,new Object[]{});
      if (child == null) continue;
      // traversable if child is a DConfigBean.
      Class cc = child.getClass();
      if (!isPrimitive(cc)) {
        if (cc.isArray()) {
          Object[] cl = (Object[])child;
          for (int j=0;j<cl.length;j++) {
```

```
        if (cl[j] instanceof DConfigBean) processBean((DConfigBean) cl[j]);
           }
         } else {
         if (child instanceof DConfigBean) processBean((DConfigBean) child);
         }
       }
     }
   }

    // if the property has a setter then invoke it with user input

    private void writeProperty(DConfigBean dcb, PropertyDescriptor prop, St
ring bean)
     throws Exception {
     Method getter = prop.getReadMethod();
     Method setter = prop.getWriteMethod();
     if (setter != null) {
       PropertyEditor pe =
           PropertyEditorManager.findEditor(prop.getPropertyType());
       if (pe == null &&
String[].class.isAssignableFrom(getter.getReturnType())) pe =
new StringArrayEditor();  // see StringArrayEditor class below
       if (pe != null) {
         Object oldValue = getter.invoke(dcb,new Object[0]);
         pe.setValue(oldValue);
         String val =
 getUserInput(bean,prop.getDisplayName(),pe.getAsText());
         // see getUserInput method below
         if (val == null || val.length() == 0) return;
         pe.setAsText(val);
         Object newValue = pe.getValue();
         prop.getWriteMethod().invoke(dcb,new Object[]{newValue});
       }
     }
   }
```

```
  private String getUserInput(String element, String property, String curr)
{
    try {
     System.out.println("Enter value for "+element+"."+property+". Current
value is: "+curr);
      return br.readLine();
    } catch (IOException ioe) {
      return null;
    }
  }
  // Primitive means a java primitive or String object here
  private  boolean isPrimitive(Class cc) {
    boolean prim = false;
    if (cc.isPrimitive() || String.class.isAssignableFrom(cc)) prim = true;
    if (!prim) {
      // array of primitives?
      if (cc.isArray()) {
        Class ccc = cc.getComponentType();
       if (ccc.isPrimitive() || String.class.isAssignableFrom(ccc)) prim =
true;
      }
    }
    return prim;
  }

  /**
   * Custom editor for string arrays. Input text is converted into tokens
using
   * commas as delimiters
   */
  private class StringArrayEditor extends PropertyEditorSupport {
    String[] curr = null;

    public StringArrayEditor() {super();}

    // comma separated string
    public String getAsText() {
      if (curr == null) return null;
```

```
      StringBuffer sb = new StringBuffer();
      for (int i=0;i<curr.length;i++) {
        sb.append(curr[i]);
        sb.append(',');
      }
      if (curr.length > 0) sb.deleteCharAt(sb.length()-1);
      return sb.toString();
    }

    public Object getValue() { return curr; }

    public boolean isPaintable() { return false; }

    public void setAsText(String text) {
      if (text == null) curr = null;
      StringTokenizer st = new StringTokenizer(text,",");
      curr = new String[st.countTokens()];
     for (int i=0;i<curr.length;i++) curr[i] = new String(st.nextToken());
    }

    public void setValue(Object value) {
      if (value == null) {
        curr = null;
      } else {
       String[] v = (String[])value; // let caller handle class cast issues
        curr = new String[v.length];
        for (int i=0;i<v.length;i++) curr[i] = new String(v[i]);
      }
    }
  }
.
.
.
```

Beyond the mechanics of the rudimentary user interface, any interface that enables changes to the configuration by an administrator or user can use the property setters shown in Listing 3-3.

Targets

Targets are associated with WebLogic servers, clusters, web servers, virtual hosts and JMS servers. See `weblogic.deploy.api.spi.WebLogicTarget` and "Support for Querying WebLogic Target Types" on page 2-6.

Application Naming

In WebLogic Server, application names are provided by a deployment tool. Names of modules contained within an application are based on the associated archive or root directory name of the modules. These names are persisted in the configuration `MBeans` constructed for the application.

In J2EE deployment there is no mention of the configured name of an application or its constituent modules, other than in the `TargetModuleID` object. Yet `TargetModuleIDs` exist only for applications that have been distributed to a WebLogic Server domain. Hence there is a need to represent application and module names in a deployment tool prior to distribution. This representation should be consistent with the names assigned by the server when the application is finally distributed.

Your deployment tool plug-in must construct a view of an application using the `DeployableObject` and `J2eeApplicationObject` classes. These classes represent standalone modules and EARs, respectively. Each of these classes is directly related to a `DDBeanRoot` object. When presented with a distribution where the name is not configured, the deployment tool must create a name for the distribution. If the distribution is a `File` object, use the filename of the distribution. If an archive is offered as an input stream, a random name is used for the root module.

Deployment Preparation

The deployment preparation phase involves saving the resulting plan from a configuration session. Use the `DeploymentConfiguration.save()` method (a standard J2EE Deployment API method). You can also use the `SessionHelper.savePlan()` method to save a new copy of deployment plan along with any external documents in the plan directory.

The `DeploymentConfiguration.save` methods creates an XML file based on the deployment plan schema that consists of a serialization of the current collection of `DConfigBeans`, along with any variable assignments and definitions. `DConfigBean` trees are always saved as external descriptors. These descriptors are only be saved if they do not already exist in the application archive or the external configuration area, meaning a save operation does not overwrite existing descriptors. The `DeploymentConfiguration.saveDConfigBean` method does overwrite files.

This is does not mean that any changes made to a configuration are lost, it means that they are handled using variable assignments.

As noted before, the `DeploymentConfiguration.restore` methods are used to create configuration beans based on a previously saved deployment plan (see "Perform Front-end Configuration" on page 3-12). You can restore an entire collection of configuration beans or you can restore a subset of the configuration beans. It is also possible to save or restore the configuration beans for a specific module in an application.

Session Cleanup

Temporary files are created during a configuration session. Archives are exploded into the temp area and can only be removed after session is complete. There is no standard API defined to close out a session. Use the `close()` methods to `WebLogicDeployableObject` and `WebLogicDeploymentConfiguration`. `SessionHelper.close()` to clean up after a session. If you do not clean up after closing sessions, the disk containing your temp directories may fill up over time.

CHAPTER 4

Performing Deployment Operations

Application deployment distributes the information created in "Configuring Applications for Deployment" on page 3-1 to the administration server for server-side processing and application startup. Your deployment tool must be able to successfully complete the following deployment operations:

Register Deployment Factory Objects

Your deployment tool must instantiate and register the `DeploymentFactory` objects it uses. You can implement your own mechanism for managing `DeploymentFactory` objects. WebLogic Server `DeploymentFactory` objects are advertised in a manifest file stored in the `wldeploy.jar` file. The manifest contains entries of the fully qualified class names of the factories, separated by whitespace. For example, if you assume that the `DeploymentFactory`-objects reside in a fixed location and are included in the deployment tool classpath, the deployment tool registers any `DeploymentFactory` objects it recognizes at startup. See Listing 4-1.

Listing 4-1 Registered Deployment Factory in the Manifest File

```
   MANIFEST.MF:
      Manifest-version: 1.0
      Implementation-Vendor: BEA Systems
      Implementation-Title: WebLogic Server 9.0 Mon May 29 08:16:47 PST 20
06 221755
      Implementation-Version: 9.0.0.0
      J2EE-DeploymentFactory-Implementation-Class:
      weblogic.deploy.spi.factories.DeploymentFactoryImpl
   .
   .
   .
```

The standard `DeploymentFactory` interface is extended by `weblogic.deploy.api.WebLogicDeploymentFactory` . The additional methods provided in the extension are:

- `String[] getUris()`: Returns an array of URI's that are recognized by `getDeploymentManager`. The first URI in the array is guaranteed to be the default `DeploymentManager` URI, `deployer:WebLogic`. Only published URI's are returned in this array.
- `String createUri(String protocol, String host, String port)`: Returns a usable URI based on the arguments.

Allocate a DeploymentManager

Your deployment tool must allocate a `DeploymentManager` from a `DeploymentFactory`, which is registered with the `DeploymentFactoryManager` class, in order to perform deployment operations. In addition to configuring an application for deployment, the `DeploymentManager` is responsible for establishing a connection to a J2EE server. The `DeploymentManager` implementation is accessed using a `DeploymentFactory`.

The following sections provide information on how a `DeploymentManager` connects to a server instance:

- "Getting a DeploymentManager Object" on page 4-3

Getting a DeploymentManager Object

Use the `DeploymentFactory.getDeploymentManager` method to get a `DeploymentManager` object. This method takes a URI, user ID and password as arguments. The URI has the following patterns:

- `deployer:WebLogic<:host:port>`
- `deployer:WebLogic.remote<:host:port>`
- `deployer:WebLogic.authenticated<:host:port>`

When connecting to an administration server, the URI must also include the host and port, such as `deployer:WebLogic:localhost:7001`. See "Understanding DeploymentManager URI Implementations" on page 4-3.

The following provides additional information on `DeploymentManager` arguments:

- When obtaining a disconnected `DeploymentManager`, you do not need to include the *`host:port`* because there is no connection to an administration server. For example, the URI can be `deployer:WebLogic`.
- The user ID and password arguments are ignored if the deployment tool uses a pre-authenticated `DeploymentManager`.
- You can access the URI of any `DeploymentManager` implementation using the `DeploymentFactory.getUris()` method. `getUris` is an extension of `DeploymenFactory`.

Understanding DeploymentManager URI Implementations

Depending on the URI specified during allocation, the `DeploymentManager` object will have one of the following characteristics:

- `deployer:WebLogic`: The `DeploymentManager` is running locally on an administration server and any files referenced during the deployment session are treated as if they are local to the administration server.
- `deployer:WebLogic.remote`: The `DeploymentManager` is running remotely to the WebLogic administration server and any files referenced during the deployment session are treated as being remote to the administration server and may require uploading. For

example, a distribute operation includes uploading the application files to the administration server.

- `deployer:WebLogic.authenticated`: This is an internal, unpublished URI, usable by applications such as a console servlet that is already authenticated and has access to the domain management information. The `DeploymentManager` is running locally on a WebLogic administration server and any files referenced during the deployment session are treated as if they are local to the administration server.

You can explicitly force the uploading of application files by using the `WebLogicDeploymentManager method enableFileUploads()` method.

Server Connectivity

`DeploymentManagers` are either connected or disconnected. Connected `DeploymentManagers` imply a connection to a WebLogic administration server. This connection is maintained until it is explicitly disconnected or the connection is lost. If the connection is lost, the `DeploymentManager` reverts to a disconnected state.

Explicitly disconnecting a `DeploymentManager` is accomplished using the `DeploymentManager.release` method. There is no corresponding method for reconnecting the `DeploymentManager`. Instead the deployment tool must allocate a new `DeploymentManager`.

Note: Allocating a new `DeploymentManager` does not affect any configuration information being maintained within the tool through a `DeploymentConfiguration` object.

Deployment Processing

Most of the functional components of a `DeploymentManage`r are defined in the J2EE Deployment API specification. However, Oracle has extended the `DeploymentManager` interface with the capabilities required by existing WebLogic Server-based deployment tools. Oracle WebLogic Server deployment extensions are documented at `weblogic.deploy.api.spi.WebLogicDeploymentManager`.

The JSR-88 programming model revolves around employing `TargetModuleID` objects (`TargetModuleIDs`) and `ProgressObject` objects. In general, target modules are specified by a list of `TargetModuleIDs` which are roughly equivalent to deployable root modules and sub-module level mbeans. The `DeploymentManager` applies the `TargetModuleIDs` to deployment operations and tracks their progress. A deployment tool needs to query progress using a `ProgressObject` returned for each operation. When the `ProgressObject` indicates the operation is completed or failed, the operation is done.

The following sections provide an overview of WebLogic `DeploymentManager` features:

DeploymentOptions

The WebLogic Server allows for a `DeploymentOptions` argument (`weblogic.deploy.api.spi.DeploymentOptions`) which supports the overriding of certain deployment behaviors. The argument may be `null`, which provides standard behavior. Some of the options supported in this release are:

- `admin` (test) mode
- Retirement Policy
- Staging

See `DeploymentOptions` Javadoc.

Distribution

Distribution of new applications results in:

- the application archive and plan is staged on all targets
- the application being configured into the domain.

Note: Redistribution honors the staging mode already configured for an application.

The standard distribute operations does not support version naming. WebLogic Server provides `WebLogicDeploymentManager` to extend the standard with a distribute operation that allows you to associate a version name with an application.

The `ProgressObject` returned from a distribute provides a list of `TargetModuleIDs` representing the application as it exists on the target servers. The targets used in the distribute are

any of the supported targets. The TargetModuleID represents the application's module availability on each target.

For new applications, TargetModuleIDs represent the top level AppDeploymentMBean objects. TargetModuleIDs do not have child TargetModuleIDs based on the modules and sub-modules in the application since the underlying MBeans would only represent the root module. For pre-existing applications, the TargetModuleIDs are based on DeployableMBeans and any AppDeploymentMBean and SubAppDeploymentMBean in the configuration.

If you use the distribute(Target[],InputStream,InputStream) method to distribute an application, the archive and plan represented by the input streams are copied from the streams into a temporary area prior to deployment which impacts performance.

Application Start

The standard start operation only supports root modules; implying only entire applications can be started. Consider the following configuration.

```
<AppDeployment Name="myapp">
   <SubDeployment Name="webapp1", Targets="serverx"/>
   <SubDeployment Name="webapp2", Targets="serverx"/>
</AppDeployment>
```

The TargetModuleID returned from getAvailableModules(ModuleType.EAR) looks like:

```
myapp on serverx (implied)
   webapp1 on serverx
   webapp2 on serverx
```

and start(tmid) would start webapp1 and webapp2 on serverx.

To start webapp1, module level control is required. Configure module level control by manually creating a TargetModuleID hierarchy.

```
   WebLogicTargetModuleID root = dm.createTargetModuleID("myapp",ModuleTyp
e.EAR,getTarget(serverx));
   WebLogicTargetModuleID web = dm.createTargetModuleID(root,"webapp1",Mod
uleType.WAR);
   dm.start(new TargetModuleID[]{web});
```

This approach uses the TargetModuleID creation extension to manually create an explicit TargetModuleID hierarchy. In this case the created TargetModuleID would look like

```
myapp on serverx (implied)
   webapp1 on serverx
```

The `start` operation does not modify the application configuration. Version support is built into the `TargetModuleIDs`, allowing the user to start a specific version of an application. Applications may be started in normal or administration (test) mode.

Application Deploy

The `deploy` operation combines a `distribute` and `start` operation. Web Applications may be deployed in normal or administration (test) mode. You can specify application staging using the `DeploymentOptions` argument. `deploy` operations use `TargetModuleIDs` instead of `Targets` for targeting, allowing for module level configuration.

The `deploy` operation may change the application configuration based on the `TargetModuleIDs` provided.

Application Stop

The standard `stop` operation only supports root modules; implying only entire applications can be stopped. See the "Application Start" on page 4-6.

Oracle provides versioning support, allowing you to stop a specific version of an application. The `stop` operation does not modify the application configuration. See "Version Support" on page 4-9.

Undeployment

The standard undeploy operation removes an application from the configuration, as specified by the `TargetModuleIDs`. Individual modules can be undeployed. The result is that the application remains on the target, but certain modules are not actually configured to run on it. See the "Application Start" on page 4-6 section for more detail on module level control.

The `WebLogicDeploymentManager` extends undeploy in support of removing files from a distribution. This is a form of in-place redeployment that is only supported in web applications, and is intended to allow you to remove static pages. See "Version Support" on page 4-9.

Production Redeployment

Standard redeployment support only applies to entire applications and employs side-by-side versioning to ensure uninterrupted session management. The `WebLogicDeploymentManager` extends the `redeploy()` method and provides the following additional support:

- “In-Place Redeployment” on page 4-8:
- “Module level Targeting” on page 4-8
- “Retirement Policy” on page 4-8
- “Version Support” on page 4-9
- .“Administration (Test) Mode” on page 4-9

In-Place Redeployment

The In-Place redeployment strategy works by immediately replacing a running application's deployment files with updated deployment files, such as:

- Partial redeployment which involves adding or replacing specific files in an existing deployment.
- Updating a configuration using a redeployment of a deployment plan

Module level Targeting

A `DeploymentManager` implements the JSR-88 specification and restricts operations to root modules. Module level control is provided by manually constructing a module specific `TargetModuleID` hierarchy using `WebLogicDeploymentManager.createTargetModuleID`

Retirement Policy

When a new version of an application is redeployed, the old version should eventually be retired and undeployed. There are 2 policies for retiring old versions of applications:

1. (Default) old version is retired when new version is active and old version finishes its in-flight operations.
2. The old version is retired when new version is active, retiring the old after some specified time limit of the new version being active.

Note: The old version is not retired if the new version is in administration (test) mode.

Version Support

Side-by-side versioning is used to provide retirement extensions, as suggested in the JSR-88 redeployment specification. This ensures that an application can be redeployed without interruption in service to its current clients. Details on deploying side-by-side versions can be found in Updating Applications in a Production Environment in *Deploying Applications to WebLogic Server*.

Administration (Test) Mode

A web application may be started in normal or administration (test) mode. Normal mode indicates the web application is fully accessible to clients. Administration (test) mode indicates the application only listens for requests using the `admin` channel. Administration (test) mode is specified by the `DeploymentOptions` argument on the WebLogic Server extensions for `start`, `deploy` and `redeploy`. See `DeploymentOptions` Javadoc.

Progress Reporting

Use `ProgressObjects` to determine deployment state of your applications. These objects are associated with `DeploymentTaskRuntimeMBeans`. `ProgressObjects` support the cancel operation but not the stop operation.

`ProgessObjects` are associated with one or more `TargetModuleIDs`, each of which represents an application and its association with a particular target. For any `ProgressObject`, its associated `TargetModuleIDs` represent the application that is being monitored.

The `ProgressObject` maintains a connection with the deployment framework, allowing it to provide a deployment tool with up-to-date deployment status. The deployment state transitions from running to completed or failed only after all `TargetModuleIDs` involved have completed their individual deployments. The resulting state is `completed` only if all `TargetModuleIDs` are successfully deployed.

The `released` state means that the `DeploymentManager` was disconnected during the deployment. This may be due to a manual release, a network outage, or similar communication failures.

The following code sample shows how a `ProgressObject` can be used to wait for a deployment to complete:

Listing 4-2 Example Code to Wait for Completion of a Deployment

```
package weblogic.deployer.tools;

import javax.enterprise.deploy.shared.*;
import javax.enterprise.deploy.spi.*;
import javax.enterprise.deploy.spi.status.*;

/**
 * Example of class that waits for the completion of a deployment
 * using ProgressEvent's.
 */
public class ProgressExample implements ProgressListener {

 private boolean failed = false;
 private DeploymentManager dm;
 private TargetModuleID[] tmids;

 public void main(String[] args) {
    // set up DeploymentManager, TargetModuleIDs, etc
    try {
      wait(dm.start(tmids));
} catch (IllegalStateException ise) {
      //... dm not connected
}
    if (failed) System.out.println("oh no!");
}

 void wait(ProgressObject po) {
    ProgressHandler ph = new ProgressHandler();
    if (!po.getDeploymentStatus().isRunning()) {
      failed = po.getDeploymentStatus().isFailed();
      return;
}
    po.addProgressListener(ph);
    ph.start();
    while (ph.getCompletionState() == null) {
      try {
```

```
        ph.join();
} catch (InterruptedException ie) {
        if (!ph.isAlive()) break;
}
}
    StateType s = ph.getCompletionState();
    failed = (s == null ||
              s.getValue() == StateType.FAILED.getValue());
    po.removeProgressListener(ph);
}

 class ProgressHandler extends Thread implements ProgressListener {
    boolean progressDone = false;
    StateType finalState = null;
    public void run(){
      while(!progressDone){
        Thread.currentThread().yield();
}
}
    public void handleProgressEvent(ProgressEvent event){
      DeploymentStatus ds = event.getDeploymentStatus();
      if (ds.getState().getValue() != StateType.RUNNING.getValue()) {
        progressDone = true;
        finalState = ds.getState();
}
}
    public StateType getCompletionState(){
      return finalState;
}
}
}
```

Target Objects

The following section provides information on how to target objects:

Module Types

The standard modules types are defined by `javax.enterprise.deploy.shared.ModuleType`. This is extended to support WebLogic Server-specific module types: JMS, JDBC, INTERCEPT and CONFIG.

Extended Module Support

JSR-88 defines a secondary descriptor as additional descriptors that a module can refer to or make use of. These descriptors are linked to the root `DConfigBean` of a module such that they are visible to a Java Beans based tool as they are child properties of a `DConfigBeanRoot` object. Secondary descriptors are automatically included in the configuration process for a module.

Web Services

An EJB or web application may include a `webservers.xml` descriptor. If present, the module is automatically configured with the WebLogic Server equivalent descriptor for configuring web services as secondary descriptors. The deployment plan includes these descriptors as part of the module, not as a separate module.

CMP

CMP support in EJBs is configured using RDBMS descriptors that are identified for CMP beans in the `weblogic-ejb-jar.xml` descriptor. The RDBMS descriptors support CMP11 and CMP20. Any number of RDBMS descriptors may be included with an EJB module. Provide these descriptors in the application archive or configuration area (`approot/plan`). Although they are

not created by the configuration process, they may be modified like any other descriptor. RDBMS descriptors are treated as secondary descriptors in the deployment plan.

JDBC

JDBC modules are described by a single deployment descriptor with no archive. If the module is part of an EAR, the JDBC descriptors are specified in `weblogic-application.xml` as configurable properties. You can deploy JDBC modules to WebLogic servers and clusters. Configuration changes to JDBC descriptors are handled as overrides to the descriptor.

If a JDBC module is part of an EAR, its configuration overrides are incorporated in the deployment plan as part of the EAR, not as separate modules.

JMS

JMS modules are described by a single deployment descriptor with no archive. If the module is part of an EAR, the JMS descriptors are specified in `weblogic-application.xml` as configurable properties. JMS modules are deployed to JMS servers. Configuration changes to JMS descriptors are handled as overrides to the descriptor. JMS descriptors may identify "targetable groups". These groups are treated as sub-modules during deployment.

If the JMS module is part of an EAR, its configuration overrides are incorporated in the deployment plan as part of the EAR, not as separate modules.

INTERCEPT

Intercept modules are described by a single deployment descriptor with no archive. If the module is part of an EAR, the Intercept descriptors are specified in `weblogic-application.xml` as configurable properties. Intercept modules are deployed to WebLogic Server servers and clusters. Configuration changes to Intercept descriptors are handled as overrides to the descriptor.

If the INTERCEPT module is part of an EAR, its configuration overrides are incorporated in the deployment plan as part of the EAR, not as separate modules.

Recognition of Target Types

The J2EE Deployment API specification's definition of a target does not include any notion of its type. WebLogic Server supports standard modules and Oracle-specific module types as valid deployment targets. Target support is provided by the `weblogic.deploy.api.spi.WebLogicTarget` and `weblogic.deploy.api.spi.WebLogicTargetType` classes. See "Module Types" on page 4-12.

TargetModuleID Objects

The `TargetModuleID` objects uniquely identify a module and a target it is associated with. `TargetModuleIDs` are the objects that specify where modules are to be started and stopped. The object name used to identify the `TargetModuleID` is of the form:

`Application=`*`parent-name`*`,Name=`*`configured-name`*`,Target=`*`target-name`*`,TWebLogicTargetType=`*`target-type`*

where

- *`parent-name`* is the name of the ear this module is part of.
- *`configured-name`* is the name used in the WebLogic Server configuration for this application or module
- *`target-name`* is the server, cluster or virtual host where there module is targeted
- *`target-type`* is the description of the target derived from Target.getDescription.

`TargetModuleID.toString()` will return this object name.

WebLogic Server TargetModuleID Extensions

`TargetModuleID` is extended by `weblogic.deploy.api.spi.WebLogicTargetModuleID`. This class provides the following additional functionality:

- `getServers` - servers associated with the `TargetModuleID`'s target.
- `isOnCluster` - whether target is a cluster
- `isOnServer` - whether target is a server
- `isOnHost` - whether target is a virtual host
- `isOnJMSServer` - whether target is a JMS server
- `getVersion` - the version name
- `createTargetModuleID` - factory for creating module specific targeting

`WebLogicTargetModuleID` is defined in more detail in the Javadocs.

The `WebLogicDeploymentManager` is also extended with convenience methods that simplify working with `TargetModuleIDs`. They are:

- `filter` - returns a list of `TargetModuleIDs` that match on application, module, and version
- `getModules` - creates `TargetModuleIDs` based on an `AppDeploymentMBean`

`TargetModuleIDs` have a hierarchical relationship based on the application upon which they are based. The root `TargetModuleID` of an application represents an EAR module or a standalone module. Child `TargetModuleIDs` are modules that are defined by the root module's descriptor. For EARs, these are the modules identified in the `application.xml` descriptor for the EAR. JMS modules may have child `TargetModuleIDs` (sub-modules) as dictated by the JMS deployment descriptor. These may be children of an embedded module or the root module. Ttherefore, JMS modules can have three levels of `TargetModuleIDs` for an application.

Typically, you get `TargetModuleIDs` in a deployment operation or one of the `DeploymentManager.get*Modules()` methods. These operations provide `TargetModuleIDs` based on the existing configuration. In certain scenarios where more specific targeting is desired than is currently defined in the configuration, you may use the `createTargetModuleID` method. This method creates a root `TargetModuleID` that is specific to a module or sub-module within the application. This `TargetModuleID` can then be used in any deployment operation. For operations that include the application archive, such as `deploy()`, using one of these `TargetModuleIDs` may result in the application being reconfigured. For example:

```
<AppDeployment Name="myapp", Targets="s1,s2"/>
```

the application is currently configured for all modules to run on `s1` and `s2`. To provide more specific targeting, a deployment tool can do the following:

```
Target s1 = find("s1",dm.getTargets());
// find() is not part of this api
WebLogicTargetModuleID root =
   dm.createTargetModuleID("myapp",ModuleType.EAR,s1);
WebLogicTargetModuleID web =
   dm.createTargetModuleID(root,"webapp1",ModuleType.WAR);
dm.deploy(new TargetModuleID[]{web},myapp,myplan,null);
```

`myapp` is reconfigured and `webapp` is specifically targeted to only run on `s1`. The new configuration is:

```
<AppDeployment Name="myapp", Targets="s1,s2">
   <SubDeployment Name="webapp", Targets="s1"/>
</AppDeployment>
```

Example Module Deployment

Consider the deployment of a standalone JMS module, one that employs sub-modules. The module is defined by the file, simple-`jms.xml`, which defines sub-modules, `sub1` and `sub2`. The descriptor is fully configured for the environment hence no deployment plan is required, although the scenario described here would be the same if there was a deployment plan.

The tool to deploy this module performs the following steps:

```
// init the jsr88 session. This uses a WLS specific helper class,
// which does not employ any WLS extensions
DeploymentManager dm = SessionHelper.getDeploymentManager(host,port,user,p
word);

// get list of all configured targets
// The filter method is a location where you could ask the user
// to select from the list of all configured targets

Target[] targets = filter(dm.getTargets());

// the module is distributed to the selected targets
ProgressObject po = dm.distribute(targets,new File("jms.xml"),plan);

// when the wait comes back the task is done
waitForCompletion(po);

// It is assumed here that it worked (there is no exception handling)
// the TargetModuleIDs (tmids) returned from the PO correspond to all the
// configured app/module mbeans for each target the app was distributed to.
// This should include 3 tmids per target: the root module tmid and the
// submodules' tmids.
TargetModuleID[] tmids = po.getResultTargetModuleIDs();

// then to deploy the whole thing everywhere you would do this
po = dm.start(tmids);
// the result is that all sub-modules would be deployed on all the selected
// targets, since they are implicitly targeted wherever the their parent is
// targeted
```

```
// To get sub-module level deployment you need to use WebLogic Server
// extensions to create TargetModuleIDs that support module level targeting.
// The following deploys the topic “xyz” on a JMS server
WebLogicTargetModuleID root =
   dm.createTargetModuleID(tmids[i].getModuleID(),tmids[i],jmsServer);
WebLogicTargetModuleID topic =
   dm.createTargetModuleID(root,"xyz",WebLogicModuleType.JMS);

// now we can take the original list of tmids and let the user select
// specific tmids to deploy
po = dm.start(topic);
```