BEA WebLogic Server®

Programming WebLogic Deployment

Version 9.1
Revised: December 19, 2005

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Introduction and Roadmap

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

- "Document Scope and Audience" on page 1-1
- "Guide to This Document" on page 1-2
- "Related Documentation" on page 1-3
- "Samples for the Deployment API Developer" on page 1-3
- "Release-Specific WebLogic Deployment API Information" on page 1-4
- "Summary of WebLogic Deployment API Features" on page 1-4
- "Unsupported WebLogic Deployment API Features" on page 1-4

Document Scope and Audience

This document is a resource for software developers who want to understand the packages of the WebLogic Deployment API. This API adheres to the API specifications described in the J2EE Deployment API standard (JSR-88) and extends the interfaces provided by that standard. To use the information here, it is essential that you understand the J2EE Deployment API standard (JSR-88) and how its interfaces are extended.

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

`Deployer`. `Deployer` is the recommended deployment tool for the WebLogic Server Environment (see Deploying Applications to the WebLogic Server for information on how to use `Deployer` and the WebLogic Server Administration Console). Therefore, implementations of the Deployment API only apply to a few use cases, which include:

- The developer wishes to use their own model implementation (see the definition of a model in the J2EE Deployment API standard) and want to interface with the WebLogic Service Provider Interface (SPI, see the J2EE Deployment API standard). In this case, the WebLogic Deployment API deployment factory (see "Application Evaluation" on page 3-8) is used to obtain a `WebLogicDeploymentManager`, which extends `javax.enterprise.deploy.spi.DeploymentManager` for use with the WebLogic SPI.
- The developer wishes to use an interface of their own design instead of the WebLogic Server Administration Console and/or `Deployer`. In this case, you may implement some or all "Phases of Deployment" on page 2-2 using the WebLogic Deployment API classes and interfaces.

This document also contains useful information for system architects who are evaluating WebLogic Server or considering the use of the WebLogic Deployment API for a particular deployment strategy.

The document is relevant to the 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. For links to the WebLogic Server® documentation and resources for other production topics, see "Related Documentation" on page 1-3.

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. It emphasizes the value-added features and how to manage application deployment with the WebLogic Deployment API.

Guide to This Document

This document is organized as follows:

- This chapter, Introduction and Roadmap, describes the audience and scope of this guide and summarizes the features of the WebLogic Deployment API.
- Chapter 2, Understanding the WebLogic Deployment API, describes the packages, interfaces, and classes of the API, including 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, contains instructions on how to configure every aspect of deployment programmatically.
- Chapter 4, Performing Deployment Operations, completes the discussion of the deployment API with the deployment life cycle and controls for a deployed application.

Related Documentation

This document contains WebLogic Deployment API-specific design and development information. For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- Deploying Applications to WebLogic Server is a guide to using the deployment tool packaged with weblogic server. Developers who are new to deployment to the WebLogic Server should read this document first, before reading the Deployment API developers guide, to understand what is involved in deployment - configuration, deployment, and the deployment life cycle.
- Developing WebLogic Server Applications is a guide to developing WebLogic Server components (such as Web applications and EJBs) and applications.
- Developing Web Applications for WebLogic Server is a guide to developing Web applications, including servlets and JSPs, that are deployed and run on WebLogic Server.
- Programming WebLogic Enterprise Java Beans is a guide to developing EJBs that are deployed and run on WebLogic Server.
- Programming WebLogic XML is a guide to designing and developing applications that include XML processing.
- Overview of WebLogic Server System Administration provides an overview of administering WebLogic Server and its deployed applications.

Samples for the Deployment API Developer

API examples are available for download at http://dev2dev.bea.com. These examples are distributed as zipped files that you can unzip into an existing WebLogic Server samples directory structure.

You build and run the downloadable examples in the same manner as you would an installed WebLogic Server example. See the download pages of individual examples for more information.

Release-Specific WebLogic Deployment API Information

For release-specific information, see the following documents:

- The What's New in WebLogic Server section in the *WebLogic Server Release Notes* lists new, changed, and deprecated features. If you are not familiar with the new features provided in version 9.0 of WebLogic Server, see the What's New in WebLogic Server 9.0 section of the *WebLogic Server Release Notes*.
- WebLogic Server Known and Resolved Issues lists known problems by general release and service pack, for all WebLogic Server APIs, including the WebLogic Deployment API.

Summary of WebLogic Deployment API Features

The WebLogic Deployment API provides extensions the J2EE Deployment API standard (JSR-88). See the J2EE Deployment API standard (JSR-88) on the Sun Microsystems Java support site.

Unsupported WebLogic Deployment API Features

The 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 because fallbacks are not trivial operations.

Understanding the WebLogic Deployment API

The following sections describe the structure and functionality of the WebLogic Deployment API:

Overview of the 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. This document and the WebLogic Deployment API JavaDocs are intended to be used by developers and Independent Software Vendors (ISVs) who want to perform deployment operations programmatically for the WebLogic Server.

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

Phases of Deployment

The functionality described here is in support of a deployer tool's ability to effectively configure and deploy applications. The deployment process has the following phases, which a tool may implement and all of which are optional:

1) Application evaluation - this phases inspects and evaluates the application files to determine the structure of the application and content of the standard descriptors embedded in it.

2) Front-end configuration - this phase establishes configuration information based on information embedded within the application. This information may be in the form of WebLogic Server descriptors, defaults, and user provided deployment plans.

3) Deployment configuration - this phase involves a conversation with the user to establish desired configuration and tuning for the specific deployment. This phase resolves previously unresolved elements and allows for overriding existing configuration and/or establishment of environment specific information.

4) Deployment preparation - this phase generates the final deployment plan and performs some level of client-side validation of the application.

5) Application deployment - this phases handles the distribution of the application and plan to the admin server for server-side processing and application startup.

The implementations of these phases are described in the Configuring Applications for Deployment and Performing Deployment Operations chapters that follow this chapter. This chapter introduces the packages necessary to perform these operations and general information about each of them.

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 (model, shared, and spi). In addition, the WebLogic Deployment API provides a Tools package with helper classes to perform common deployment tasks easily.

WebLogic supports the “Product Provider” role described in the J2EE Deployment API standard (JSR-88) and thus 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 (enables you to add optional information which only makes sense to WebLogic server. For example, starting in Admin mode, redeploying with versioning, etcetera)
- Finer grain control (module level targeting – not part of J2EE Deployment API)
- Support of WebLogic module extensions (JMS, JDBC, Interception, Application Specific Configuration[Custom/Config modules]) these are not required by J2EE Deployment API, it is extended for additional configuration.
- Remote Operation
- Additional Operations
  - The 'Deploy' verb: combines 'distribute' and 'start'
  - Partial Redeployment: redeployment/removal of parts of an application
  - Configuration update: redeployment of a new deployment with only dynamic configuration changes

The SPI Package

As a J2EE product provider, BEA extends the javax SPI package to control how configuration and deployment is done to the WebLogic Server specifically. The core interface for this package is the `DeploymentManager`, from which all other deployment activities can be 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. This and other ways to obtain a `WebLogicDeploymentManager` are described in the discussion on "Application Evaluation" on page 3-8.

weblogic.deploy.api.spi

The `weblogic.deploy.api.spi` package provides the interfaces required to configure and deploy applications for deployment to WebLogic Server targets. This package enables a deployment tool to represent the WebLogic Server-specific deployment configuration for an Enterprise Application or standalone module.

`weblogic.deploy.api.spi` includes the `WebLogicDeploymentManager` interface, as noted above. Deployment tools use this deployment manager to perform all deployment-related operations such as distributing, starting, and stopping applications in WebLogic Server. The `WebLogicDeploymentManager` provides important extensions to the J2EE `DeploymentManager` interface to support 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 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 - `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 only deploying all modules to the same set of targets as specified by jsr88). The tools provided in support for module targeting are the `WebLogicDeploymentManager.createTargetModuleID` methods. Using `TargetModuleID's` is described in the discussion on "Target Objects" on page 4-12.

The `WebLogicTargetModuleID` class contains the WebLogic Server extensions to the `javax.enterprise.deploy.spi.TargetModuleID` interface. The `WebLogicTargetModuleIDs` in this class have a close relationship to the configured `TargetInfoMBeans` (`AppDeploymentMBean` and `SubDeploymentMBean`). The `TargetModuleID`'s provide more detailed descriptions of the application modules and their relationship to targets than those in `MBeans`.

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.
`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(string)` and the following values yielding the following results:

- STAGE - Force copying of files to target servers.
- NO_STAGE - Inhibit copying of files to target servers.
- EXTERNAL_STAGE - files are to be staged manually.

DConfigBean Validation

The property setters in a `DConfigBean` will 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 will also do more semantic validation, 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

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 `WebLogicDeployableObjects` and `WebLogicDDBeans` for the application's deployment descriptors. Basic configuration tasks are accomplished by associating these `WebLogicDDBeans` with `WebLogicDConfigBeans`, which represent the server configuration properties required for deploying the application on a WebLogic Server. These are discussed in the "Overview of the Configuration Process" on page 3-3.

Unlike `DConfigbeans`, which contain configuration information specifically for a server environment (in this case WebLogic server), the `DDBean` objects take in the general deployment descriptor elements for the application. For example, if you were deploying a Web application, the deployment descriptors that would end up 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 `DDBeans` describe the application while the `DConfigBeans` configure 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. The possible operations that can occur during a Configuration Session are described in "Configuring an Application" on page 3-1.

The Shared Package

The WebLogic Deployment API extends the `weblogic.deploy.api.shared` interfaces and provides the types listed below for complete server operability.

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 model and SPI package members.

The definitions of the standard `javax.enterprise.deploy.shared` classes `ModuleType` and `CommandType` are extended in this package to include more specific information that the WebLogic Server can use. The additions provided by the extensions are as follows:

- Module types (JMS, JDBC, Interception, Submodules, Diagnostics, WSEE, Custom)
- Commands (deploy, update)

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. It defines the following target types (server, cluster, virtual hosts, JMS servers). See “Target Objects” on page 4-12 for more information on targets.

Command Types for Deploy and Update

The deploy and update command types were added to the required command types defined in the `javax.enterprise.spi.shared package`. These commands are therefore 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 (Software Agent Framework) Agent
- Server
- Virtual Host

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

The Tools Package

Use the tools in this package to perform common deployment tool tasks with a minimum of number of controls and explicit object manipulations. This includes controlling, `WebLogicDeploymentManagers`, `WebLogicDeployableObjects` and generating deployment plans.

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` has simplified methods for this. These methods include:

- `getDeplymentManager, getDisconnectedDeploymentManager, getRemoteDeploymentManager`: Accessing a `WebLogicDeploymentManager`
- `setApplication`, `setApplicationRoot`, `setPlan` - identify the application and deployment plan to use in the session
- `initializeConfiguration`, `inspect`: Initializing a configuration session
- `savePlan`: Saving a deployment
- `getdefaultJMSTargetModuleIDs, getJMSDescriptor`: JMS Submodule targeting
- `getApplication`, `getDescriptorURIs`, `getModuleInfo`, etcetera: Application inspection - this provides information about the application components.

If your tools code directly to WebLogic Server's J2EE Deployment API implementation, you should always use `SessionHelper`.

As noted in the discussion of The SPI Package, `SessionHelper` can obtain a `WebLogicDeploymentManager` automatically 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 that the above structure be used for applications, although it is a preferred approach as it serves to keep the application and its configuration artifacts under a common root, thus providing `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, trees of typed Java Bean objects that represent the individual descriptor elements. These bean tress 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 via 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 standard and WebLogic Server descriptors included in an application. The resulting plan will describe the application structure, identify all deployment descriptors and will export a subset of the application's configurable properties. Exporting a property exposes it to tools like the WebLogic Server console which can use the plan to assist the administrator in providing appropriate values for those properties. By default, the `PlanGenerator` tool only exports application dependencies; those properties required for a successful deployment. This behavior can be overridden with one of the following options:

- Dependencies: Export resources referenced by the application (this is the 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

Configuring Applications for Deployment

The term *configuration* refers to the process of preparing an application or deployable resource for deployment to WebLogic Server. The J2EE Deployment API standard (JSR-88) differentiates between a configuration session and deployment. They are distinguished as follows:

- Application Configuration involves: Generation of WebLogic descriptors, which go into a Deployment Plan
- Deployment tasks are: Distributing, Starting, Stopping, Redeploying, Undeploying

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

- "Configuring an Application" on page 3-1
- "Overview of the Configuration Process" on page 3-3
- "Application Evaluation" on page 3-8
- "Performing Front-End Configuration" on page 3-13
- "Customizing Deployment Configuration" on page 3-19
- "Deployment Preparation" on page 3-25

Configuring an Application

Taking a look once again at the deployment phases, those phases that are covered by configuration are the first four:

1. Application evaluation - this phase inspects and evaluates the application files to determine the structure of the application and content of the standard descriptors embedded in it.
   - Initialize a deployment session by obtaining a `WebLogicDeploymentManager`. See "Application Evaluation" on page 3-8.
   - Create a `WebLogicJ2eeApplicationObject` or `WebLogicDeployableObject` to represent the J2EE Configuration of an Enterprise Application (EAR) or standalone module (WAR, EAR, RAR, or CAR). If it is an EAR, there will be child objects generated. This is the deployable object as described in the J2EE Deployment API standard (JSR-88). See "Creating a Deployable Object" on page 3-12.
2. Front-end configuration - this phase establishes configuration information based on information embedded within the application. This information 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 begins a process that result in a Deployment Plan for this deployment of this object. See "Deployment Configuration" on page 3-14.
   - Restore existing WebLogic Server configuration values from an existing deployment plan, if available. See "Performing Front-End Configuration" on page 3-13.
3. Customizing deployment configuration - this phase involves a conversation with the user to establish desired configuration and tuning for the specific deployment. This phase resolves previously unresolved elements and allows for overriding existing configuration and/or establishment of environment specific information.

   Modify individual WebLogic Server configuration values based on user inputs and the selected WebLogic Server targets. See "Customizing Deployment Configuration" on page 3-19.
4. Deployment preparation - this phase generates the final deployment plan and performs some level of client-side validation of the application.

   Save the modified WebLogic Server configuration information to a new deployment plan or to variable definitions in an existing Deployment Plan.
5. Application deployment - this phase handles the distribution of the application and plan to the administration server for server-side processing and application startup.

Within the first four phases, a deployment configuration tool performs several steps. Each step of each phase is described in detail below. But before embarking upon coding these tasks, it is necessary to take note of the tools WebLogic provides to handle configuration information.

The WebLogic Server SessionHelper Class

SessionHelper simplifies the most common operational patterns such as accessing a `DeploymentManager`, Initializing a configuration session, saving deployment plans, JMS Submodule targeting, and Inspecting applications.

Tools that code directly to WebLogic J2EE Deployment API implementations are encouraged to use `SessionHelper` and each of the sections that follow, a subsection devoted to `SessionHelper` will appear so that you can use it in your implementation. Also see the Javadocs for more information about `SessionHelper`.

Session Cleanup

Temporary files are created during a configuration session. Archives get exploded into the temp area. They can be removed only after session is complete. There is no standard api defined to close out a session. There are `close()` methods to `WebLogicDeployableObject` and `WebLogicDeploymentConfiguration`. `SessionHelper.close()` cleans everything up after that. It is up to the tool programmer to use these methods or the temp directories may fill up over time.

Overview of the Configuration Process

One of the tasks handled by deployer tools is configuring an application for a successful deployment. Most configuration information for an application is provided in its deployment descriptors. The supported descriptors are listed below. Certain elements in these descriptors refer to external objects and/or must be handled in server-specific ways. Different server vendors manage this in different ways. WebLogic Server uses descriptor extensions for this purpose. These are the WebLogic Server specific deployment descriptors. The mapping between standard descriptors and WebLogic Server descriptors is managed via `DDBeans` and `DConfigBeans`, which are described in this document.

Types of Configuration Information

The primary configuration information for an application falls into two distinct but related categories:

- J2EE Configuration
- WebLogic Server Configuration

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 "Performing Front-End Configuration" on page 3-13 for more information.

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 deployer tool acquires an instance of `WebLogicDeployableObject` object for an application via 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. These would be WARs, EJBs, RARs and CARs.

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.

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.

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.

DConfigBeans

`DConfigBeans` (config beans) are the objects used to convey server configuration requirements to a deployer 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 style deployer 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 (e.g. resource references) will 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 via variable assignments. If the application does not come with the relevant WebLogic Server descriptors, they will be 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

The Application Evaluation phase consists of the following possible deployment operations:

- Initialize a deployment session by obtaining a `WebLogicDeploymentManager`.
- Create a `WebLogicJ2eeApplicationObject` or `WebLogicDeployableObject` to represent the J2EE Configuration of an Enterprise Application (EAR) or standalone module (WAR, EAR, RAR, or CAR). If it is an EAR, there will be child objects generated. This is the deployable object as described in the J2EE Deployment API standard (JSR-88)

A deployment manager provides an interface to the WebLogic Server deployment infrastructure. To initialize a deployment session for a deployment tool, you create a new deployment manager.

The deployment manager is implemented using the factory pattern. You first obtain a deployment factory class by specifying its name, `weblogic.deployer.spi.factories.internal.DeploymentFactoryImpl`, and then register the factory class with a `javax.enterprise.deploy.spi.DeploymentFactoryManager` instance. For instance:

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

```
DeploymentFactoryManager.getInstance().registerDeploymentFactory(myDeploymentF
actory);
```

After you have registered the deployment factory, you can use it with a specific *URI* to obtain a deployment manager that has the functionality you require. This requires you to choose a type of 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* you specify 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. You can use a disconnected deployment manager to configure an application on a remote client machine, but you cannot use it to perform deployment operations. (For example, you 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 can be used both to configure and deploy applications.

A connected deployment manager can be 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 each basic type of deployment manager.

Table 3-3 WebLogic Server Deployment Manager Usage

Deployment Manager Connectivity	Type	Usage	Notes
Disconnected	n/a	Configuration tools only	Cannot perform deployment operations

Deployment Manager Connectivity	Type	Usage	Notes
Connected	Local	Configuration and deployment tools local to the Administration Server	All files are assumed to be 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

All `DeploymentManager`'s obtained from `WebLogicDeploymentFactory` support the WebLogic Server extensions. You 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 you want to obtain. Connected deployment managers require that you pass 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 manager.

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(myDeploymentF
actory);
WebLogicDeploymentManager myDisconnectedManager =
(WebLogicDeploymentManager)myDeploymentFactory.getDisconnectedDeploymentManage
r(WebLogicDeploymentFactory.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(myDeployme
ntFactory.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 in Listing 3-1. The `SessionHelper` code required to obtain a disconnected deployment manager consists of a single line:

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

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

```
DeploymentManager myConnectedManager =
SessionHelper.getDeploymentManager("adminhost", "7001", "weblogic",
"weblogic"));
```

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

Creating a Deployable Object

The second part of Application Evaluation is creating the Deployable Object, a container for the application you are intending to deploy. Once you have initialized a configuration session by obtaining a WebLogicDeploymentManager, you can create the deployable object in one of two ways. The direct approach uses the WebLogicDeployableObject class of the model package. You do this 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 convenience method to help you easily obtain a deployable object. The `SessionHelper` code required to obtain a deployable object is pretty simple:

```
SessionHelper.setApplicationRoot(root);

WebLogicDeployableObject myDeployableObject =
SessionHelper.getDeployableObject();
```

As you can see, there is no application specified in the `getDeployableObject()` call. `SessionHelper` assumes that the application in the root directory set by `setApplicationRoot()` is the one being used. This is the structure described in the section of the previous chapter. Once you have set the application root directory, `SessionHelper` can perform other operations automatically without further complications, such as explicitly naming the dispatch file location, the deployment plan location, etcetera.

If this is not the directory structure you are using or if there are several applications in the same directory, you may set the application file name using the `setApplication` method. This is done as follows:

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

Thus you can continue to use `SessionHelper` even if you have another directory structure in mind. The `getDeployableObject` method will automatically assume the application you have set is being deployed.

Summary

Application Evaluation consists of obtaining a deployment manager and a deployable object container for your application. These set the stage for further deployment operations.

Performing Front-End Configuration

The Front-End configuration phase is comprised of two possible logical operations:

- Create a `WebLogicDeploymentConfiguration` object to represent the WebLogic Server configuration of an Application. This begins a process that result in a Deployment Plan for this deployment of this object.
- Restore existing WebLogic Server configuration values from an existing deployment plan, if available

A deployment plan is an XML document that contains the environmental configuration for an application, referred to as its front-end configuration. It separates the environment specific details of an application from the logic of the application. A deployment plan is not required for every application. It is required if the application expects specific attributes to be provided by the descriptors, as opposed to the generic values generated automatically. You may have a deployment plan for each environment in which the application will be deployed.

The deployment plan has several purposes:

- It describes the application structure. It will tell you what modules are in the application.
- It allows developers and administrators to update the configuration of an application without modifying the application archive.
- It contains environment specific descriptor override information. By modifying this file, you can override environment variables for the application so that it can be used in a different environment.

The information in a deployment plan can be loaded into and extracted 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.

Deployment Configuration

The server configuration for an application is encapsulated in the `javax.enterprise.deploy.spi.DeploymentConfiguration` interface. A `DeploymentConfiguration` can also be viewed as the object representation of a deployment plan. A `DeploymentConfiguration` is associated with a `DeployableObject` via the `DeploymentManager.createConfiguration` method. Following the creation of the `DeploymentConfiguration`, 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 in the application, then a `DConfigBean` tree is created using available default values. Binding properties that have no defaults will be left unset.

It is the responsibility of the deployer tool to ensure the `DConfigBean` tree is fully populated before using it to distribute an application. The `DConfigBeans` can be populated from scratch as shown in the configuration code below:

```
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 via `DeploymentConfiguration.save()`. A deployer 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 via 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()` provides 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` will process 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 may read in a deployment plan with the `DeploymentConfiguration.restore()` method, in accordance with the J2EE Deployment API standard (JSR-88). 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`. At this point, you begin constructing the `WebLogicDConfigBean` tree as usual.

Validating a Configuration

Validation of the configuration occurs mostly during the parsing of the descriptors, which occurs when the app's descriptors are processed. Validation consists of ensuring the descriptors are valid xml documents and that they abide by their respective schemas.

Summary

Performing Front-End configuration involves creating a `WebLogicDeploymentPlan` and populating it and its associated bean trees with configuration information from a deployment plan. Although a deployment plan is optional, a valid `WebLogicDeploymentConfiguration` is required in order to change configuration values in the deployment configuration (described below).

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. The configuration at this point 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 the 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. e.g. there is no standard descriptor element they are directly related to.

Editing a configuration is accomplished by invoking the property setters.

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

Introspection is, of course, required in both approaches in order to present or modify descriptor properties. The difference is in how the tool presents the information: either driven by standard descriptor content or WebLogic Server descriptor content.

A system of modifying configuration information would include a user interface to ask for changes. A system for obtaining and setting such changes is illustrated in the code sample below:

```
  // 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.getDDBean().getRoot().getFilename());

    }

    // get property descriptor for the bean

    BeanInfo info =
Introspector.getBeanInfo(dcb.getClass(),Introspector.USE_ALL_BEANINFO);

    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, String
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 would enable changes to the configuration by an administrator or user would use the property setters as shown above.

Targets

Targets are associated with WebLogic servers, clusters, web servers, virtual hosts and JMS servers. This is captured in the `weblogic.deploy.api.spi.WebLogicTarget` Javadocs.

Application Naming

In WebLogic Server, application names are provided by the 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 deployer tool prior to distribution. This representation should be consistent with the names assigned by the server when the application is finally distributed.

The tool plug-in constructs its view of an application via the `DeployableObject` and `J2eeApplicationObject` classes. These represent standalone modules and EARs, respectively. Each of these has a direct relationship to a `DDBeanRoot` object. When presented with a distribution where the name is not configured, one will be derived for it. If the distribution is a `File` object, the file's name will be used. If the archive is offered as an input stream, a random name will be used for the root module.

Deployment Preparation

The deployment preparation phase involves saving the resulting plan from a configuration session. It is handled by the `DeploymentConfiguration.save()` method (another standard J2EE Deployment API method). You can use the `SessionHelper.savePlan()` method to do this; it saves new copy of deployment plan along with any external documents in the plan directory.

The `DeploymentConfiguration.save` methods will create an XML file based on the deployment plan schema, consisting of a serialization of the current collection of `DConfigBeans`, along with any variable assignments and definitions. The `DConfigBean` trees are always saved as external descriptors. These descriptors will only be saved if they do not already exist in the application archive or the external configuration area. i.e., the save operation will not overwrite existing descriptors. The `DeploymentConfiguration.saveDConfigBean` method will, however, overwrite files. This is not to say that any changes made to the configuration are lost. Rather they are handled using variable assignments.

As noted before, the `DeploymentConfiguration.restore` methods are used to create config beans based on a previously saved deployment plan (see "Performing Front-End Configuration" on page 3-13). An entire collection of config beans may be restored or the tool can restore a subset of the config beans. e.g., it is possible to save/restore just the config beans for a specific module in an application. This allows for a degree of flexibility in configuring an application.

CHAPTER 4

Performing Deployment Operations

The following sections describe how to implement deployment operations for a WebLogic deployment API constructed deployment tool.

Application Deployment

Up to this point, we have described how to create elaborate configurations but have not touched only actually putting those configurations to a practical use. Application Deployment, the fifth phase of deployment, uses any information setup by configuration operations to handle the distribution of the application and plan to the administration server for server-side processing and application startup. This chapter describes all of the necessary components to deploy and control an application to the WebLogic Server environment.

Deployment Factories

A deployer tool must allocate a `DeploymentManager` from a `DeploymentFactory,` which is registered with the `DeploymentFactoryManager` class, in order to perform deployment operations. In addition to configuration (described in "Overview of the Configuration Process" on page 3-3), the `DeploymentManager` is responsible for establishing a connection to a J2EE server. The `DeploymentManager` implementation is accessed via a `DeploymentFactory`.

A deployer tool is responsible for instantiating and registering any `DeploymentFactory` objects it uses. The WebLogic factory is advertised via the manifest file in the `wldeploy.jar` archive. The manifest contains entries defining the fully-qualified class names of the factories, separated by whitespace. For example:

```
MANIFEST.MF:

    Manifest-version: 1.0

    Implementation-Vendor: BEA Systems

   Implementation-Title: WebLogic Server 9.0 Mon Nov 11 08:16:47 PST 2002 221755

    Implementation-Version: 9.0.0.0

    J2EE-DeploymentFactory-Implementation-Class:

    weblogic.deploy.spi.factories.DeploymentFactoryImpl
```

A deployer tool can define any mechanism for managing the SPI plug-ins it recognizes. For simplicity's sake, assume the tool requires that all SPI plug-ins reside in some fixed location, and that they are also in its classpath. Upon startup, the tool would register the plug-ins as shown in the example.

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.

DeploymentManager Behaviors

Only one `DeploymentManager` implementation is provided. Depending on the URI specified when allocating the `DeploymentManager`, it will take on the following characteristics:

- `deployer:WebLogic`: The `DeploymentManager` assumes it is running locally on an administration server. Thus any files referenced during the deployment session are assumed to be local to the administration server.
- `deployer:WebLogic.remote`: The `DeploymentManager` will assume it is running remotely to the WebLogic administration server, thus any files referenced during the deployment session are not assumed to be local to the administration server. i.e. a distribute operation will include 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` will assume it is running locally on a WebLogic administration server. Thus any files referenced during the deployment session are assumed to be local to the administration server.

Server Connectivity

A deployment tool allocates a `DeploymentManager` from a registered `DeplomentFactory` (see above). `DeploymentManagers` can be 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` will revert to a disconnected state.

Explicitly disconnecting a `DeploymentManager` is accomplished via the `DeploymentManager.release` method. There is no corresponding method for reconnecting the `DeploymentManager`. Instead the deployer tool must allocate a new `DeploymentManager`. This should not affect any configuration information being maintained within the tool through a `DeploymentConfiguration` object.

`DeploymentManagers` are identified by an opaque URI. All `DeploymentManagers` for WebLogic Server have the URI scheme, `deployer:WebLogic<.`*`type`*`>`. The `DeploymentFactory.getDeploymentManager` method takes a URI, userid and password as arguments. In this case the URI must also include the host and port for the admin server, e.g. `deployer:WebLogic:`*`localhost:7001`*. When obtaining a disconnected

`DeploymentManager`, only the URI is necessary because no actual connection to a server is made. In this case, the URI can simply be `deployer:WebLogic`, although extending the URI with the host and port is allowed.

The userid and password arguments are ignored if the deployer tool uses a pre-authenticated `DeploymentManager`; The factory presumes that the user has already been authenticated.

The URI of any `DeploymentManager` implementation should be accessed using the `DeploymentFactory.getUris()` method. `getUris` is an extension of `DeploymenFactory`.

The behavior of the `DeploymentManager` operations is guided by the URI specified when it is allocated from the factory. These behaviors apply to relevant operations supported by the `DeploymentManager`. They include:

- File Upload: determines whether files are uploaded to the admin server. This functionality is automatically enabled for the remote `DeploymentManagers`
- Authentication Required:

These behaviors also can be explicitly enabled using the `WebLogicDeploymentManager` method `enableFileUploads()`.

Deployment Processing

Most of the functional components of a `DeploymentManager` are defined in the J2EE Deployment API specification. The specification does not describe a `DeploymentManager` that is sufficient to support the needs of WebLogic Server customers, hence a number of extensions were introduced. These are documented in the Javadocs for `weblogic.deploy.api.spi.WebLogicDeploymentManager`. The relationship between the SPI operations and WebLogic Server is provided here.

The intent is not to change the programming model defined by the J2EE deployment API specification, but rather to extend the `DeploymentManager` interface (`DeploymentManager`) with the capabilities required and expected by existing WebLogic Server-based deployment tools.

The JSR-88 programming model revolves around employing `TargetModuleID` objects (`TargetModuleIDs`) and `ProgressObject` objects. In general, the 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 its deployment operations (start, stop, and so on) and tracks their progress. The deployer tool queries progress via a `ProgressObject` returned for each operation. When the `ProgressObject` indicates the operation is completed or failed, the operation is done.

The following sections describe the different `DeploymentManager` operations and their extensions. Full descriptions are provided in the Javadocs. This section serves as a general overview. The extensions cover the following features:

- General usability/convenience
- Side-by-side version support
- module level targeting
- partial redeployment
- administration (test) mode applications

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 for standard behaviors. Some of the options supported in this release are:

- administration (test) mode
- Retirement Policy
- Staging

See the `DeploymentOptions` Javadoc for a full list of options.

Distribution

The distribution of new applications results in the application archive and plan being staged on all targets, and the application being configured into the domain. Redistribution will honor the staging mode already configured for the application.

The standard distribute operations do not allow for version naming, leaving this to be generated by the system. `WebLogicDeploymentManager` extends the standard with a distribute operation that allows the user to specify a version name to associate with the new 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` will represent the application's module availability on each target.

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

When using the `distribute(Target[],InputStream,InputStream)` method to distribute, the archive and plan represented by the input streams will be copied from the streams into a temp area prior to deployment. It is not recommended to use this distribute method for performance reasons.

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)` would look like this:

```
myapp on serverx (implied)

 webapp1 on serverx

 webapp2 on serverx
```

So `start(tmid)` would start *webapp1* and *webapp2* on `serverx`.

To just start *webapp1*, module level control is required. This is achievable as follows by manually creating a `TargetModuleID` hierarchy.

```
WebLogicTargetModuleID root =
dm.createTargetModuleID("myapp",ModuleType.EAR,getTarget(serverx));

WebLogicTargetModuleID web =
dm.createTargetModuleID(root,"webapp1",ModuleType.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, and is provided as a convenience. Web Applications may be deployed in normal or administration (test) mode. Application staging can also be specified via the `DeploymentOptions` argument on deploy. The deploy operations use `TargetModuleIDs` instead of `Targets` for targeting, thus 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 discussion for more details on module level control.

The `stop` operation does not modify the application configuration.

Version support is built into the `TargetModuleIDs`, allowing the user to stop a specific version of an application.

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 for removal of static pages and the like.

Version support is built into the `TargetModuleIDs`, allowing the user to undeploy a specific version of an application.

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 as follows:

- redeploying individual modules in an application. This implies an in-place redeploy (no new version). The version information in the new application must be the same as the old application, but with a different value (for example 1.0, 1.1, and so on.). Module targeting must be specified.
- Specifying a retirement policy for an old application.
- Partial redeployment - this involves adding or replacing specific files in an existing deployment. This is an in-place redeployment.
- Redeploying applications in administration (test) mode.
- updating configuration - the redeployment of a deployment plan.

Version support is provided via arguments in the above operations. The version, if relevant, is specified in the `TargetModuleID`.

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 that the old version will not be retired if the new version is in administration (test) mode.

Module Targeting

`DeploymentManager's` will honor the JSR-88 specification and restrict operations to root modules. Module level control is provided by manually constructing a module specific `TargetModuleID` hierarchy via `WebLogicDeploymentManager.createTargetModuleID`

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 with no interruption in service to its current clients. Details on deploying side-by-side versions can be found in the Redeployment Strategies documentation. Briefly, an archive manifest can specify the WebLogic Server application version. The deployment plan also supports a version identifier. The combination of these 2 version identifiers is used internally to distinguish between versions of applications, and may also be used by developers and administrators to assist in version control.

This provides the basis for production redeployment as it is implemented for the WebLogic Server Deployment Service. Please refer to the Redeployment Strategies documentation for more information.

Administration (Test) Mode

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

Progress Reporting

`ProgressObjects` are the interface for the deployment state in the SPI. 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 live connection with the deployment framework, allowing it to provide the tools with up-to-date deployment status. Once this status goes to completed, failed or released, the tools should not expect any further changes. 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` were 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:

```
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;
}
}
}
```

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.

Target Objects

The J2EE Deployment API specification's definition of a target does not include any notion of its type. WebLogic Server recognizes types - servers, clusters, JMS servers and virtual hosts - as valid targets for deployment. These concepts are introduced into the API via the `weblogic.deploy.api.spi.WebLogicTarget` and `weblogic.deploy.api.spi.WebLogicTargetType` classes. `WebLogicTargetType` follows the form of the classes in the `javax.enterprise.deploy.shared`. It enumerates the known target types.

`WebLogicTarget` and `WebLogicTargetType` are defined in more detail in the Javadocs. `WebLogicTarget` extends `javax.enterprise.deploy.spi.Target`.

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,TWebLogicT
argetType=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 have a notion of sub-module, hence any JMS module may have child `TargetModuleIDs` as dictated by the JMS deployment descriptor. These may be children of an embedded module or the root module. Therefore, there can be 3 levels of `TargetModuleIDs` for an application.

`TargetModuleIDs` are generally acquired via some deployment operation, or one of the `DeploymentManager.get*Modules()` methods. These will only provide `TargetModuleIDs` based on existing configuration. In certain scenarios where more specific targeting is desired than is currently defined in the configuration, the `createTargetModuleID` method is provided. This method will create a root `TargetModuleID` that is specific to a module or submodule within the application. This `TargetModuleID` can then be used in any deployment operation. For operations that include the application archive (e.g. `deploy()`), using one of these `TargetModuleIDs` may result in the application being reconfigured. For example, given the following configuration

```
<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>
```

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-- 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 App may include a `webservers.xml` descriptor. If present the module will be automatically configured with the WebLogic Server equivalent descriptor for configuring the web services. These are treated as secondary descriptors in JSR-88 terminology. The deployment plan includes these descriptors as part of the module, not as a separate module.

CMP

CMP support in EJBs is configured via RDBMS descriptors that are identified for CMP beans in the `weblogic-ejb-jar.xml` descriptor. The RDBMS descriptors support CMP11 and CMP20 currently. Any number of RDBMS descriptors may be included with an EJB module. These descriptors must be provided in the application archive or configuration area (approot/plan). They are not created by the configuration process, but may be modified as with any other descriptor. RDBMS descriptors are treated as secondary descriptors in the deployment plan.

JDBC

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

If the JDBC module is part of an EAR its configuration overrides are treated as with secondary descriptors; they 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. There is no archive involved. If part of an EAR the JMS descriptors are specified in `weblogic-application.xml`. These are configurable properties. JMS modules can be 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 treated as with secondary descriptors; they 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. There is no archive involved. If part of an EAR the Intercept descriptors are specified in `weblogic-application.xml`. These are configurable properties. Intercept modules can be 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 treated as with secondary descriptors; they are incorporated in the deployment plan as part of the EAR, not as separate modules.

Examples

Consider the deployment of a standalone JMS module, one that employs sub-modules. The module is defined by the file, `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,pword);
```

```
// get list of all configured targets
// The filter method is where the tool might 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);
```