BEA WebLogic Server®

Configuring WebLogic Server Environments

Version 10.0
Revised: March 30, 2007

Contents

3. Avoiding and Managing Overload

4. Configuring Network Resources

5. Configuring Web Server Functionality

6. Using the WebLogic Persistent Store

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Configuring WebLogic Server Environments*.

Document Scope and Audience

This document describes how you design, configure, and manage WebLogic Server® environments. It is a resource for system administrators and operators responsible for implementing a WebLogic Server installation. This document is relevant to all phases of a software project, from development through test and production phases.

It is assumed that the reader is familiar with Java EE and Web technologies, object-oriented programming techniques, and the Java programming language.

Guide to This Document

The document is organized as follows:

- This chapter, "Introduction and Roadmap," describes the scope of this guide and lists related documentation.

- Chapter 2, "Using Work Managers to Optimize Scheduled Work," describes the WebLogic Server execution model and the process of configuring application access to the execute queue.
- Chapter 3, "Avoiding and Managing Overload," describes detecting, avoiding, and recovering from overload conditions.
- Chapter 4, "Configuring Network Resources," describes optimizing your WebLogic Server domain for your network.
- Chapter 5, "Configuring Web Server Functionality," describes using WebLogic Server as a Web server.
- Chapter 6, "Using the WebLogic Persistent Store," describes configuring and monitoring the persistent store, a built-in, high-performance storage solution for WebLogic Server subsystems and services that require persistence.

Related Documentation

- *Understanding Domain Configuration*
- *Administration Console Online Help*

New and Changed Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see "What's New in WebLogic Server 10" in *Release Notes*.

CHAPTER 2

Using Work Managers to Optimize Scheduled Work

WebLogic Server allows you to configure how your application prioitizes the execution of its work. Based on rules you define and by monitoring actual runtime performance, WebLogic Server can optimize the performance of your application and maintain service level agreements. You define the rules and constraints for your application by defining a Work Manger and applying it either globally to WebLogic Server domain or to a specific application component.

Understanding How WebLogic Server Uses Thread Pools

In previous versions of WebLogic Server, processing was performed in multiple execute queues. Different classes of work were executed in different queues, based on priority and ordering requirements, and to avoid deadlocks. In addition to the default execute queue,

`weblogic.kernel.default`, there were pre-configured queues dedicated to internal administrative traffic, such as `weblogic.admin.HTTP` and `weblogic.admin.RMI`.

Users could control thread usage by altering the number of threads in the default queue, or configure custom execute queues to ensure that particular applications had access to a fixed number of execute threads, regardless of overall system load.

WebLogic Server uses a single thread pool, in which all types of work are executed. WebLogic Server prioritizes work based on rules you define, and run-time metrics, including the actual time it takes to execute a request and the rate at which requests are entering and leaving the pool.

The common thread pool changes its size automatically to maximize throughput. The queue monitors throughput over time and based on history, determines whether to adjust the thread count. For example, if historical throughput statistics indicate that a higher thread count increased throughput, WebLogic increases the thread count. Similarly, if statistics indicate that fewer threads did not reduce throughput, WebLogic decreases the thread count. This new strategy makes it easier for administrators to allocate processing resources and manage performance, avoiding the effort and complexity involved in configuring, monitoring, and tuning custom executes queues.

Understanding Work Managers

WebLogic Server prioritizes work and allocates threads based on an execution model that takes into account administrator-defined parameters and actual run-time performance and throughput.

Administrators can configure a set of scheduling guidelines and associate them with one or more applications, or with particular application components. For example, you can associate one set of scheduling guidelines for one application, and another set of guidelines for other application. At run-time, WebLogic Server uses these guidelines to assign pending work and enqueued requests to execution threads.

To manage work in your applications, you define one or more of the following Work Manager components:

- Fair Share Request Class:
- Response Time Request Class:
- Min Threads Constraint:
- Max Threads Constraint:
- Capacity Constraint

- Context Request Class:

For more information on these components, see "Request Classes" on page 2-4 or "Constraints" on page 2-6

You can use any of these components to control the performance of your application by referencing the name of the component in the application's deployment descriptor. In addition, you may define a *Work Manager* that encapsulates all of the above components (except Context Request Class. See "Context Request Class" on page 2-5) and reference the name of the Work Manager in your application's deployment descriptor. You can define multiple Work Managers—the appropriate number depends on how many distinct demand profiles exist across the applications you host on WebLogic Server.

Work Managers can be configured at the domain level, application level, and module level in one of the following configuration files:

- `config.xml`—Work Managers specified in `config.xml` can be assigned to any application, or application component, in the domain. You can use the Administration Console to define a Work Manager.
- `weblogic-application.xml`—Work Managers specified at the application level can be assigned to that application, or any component of that application.
- `weblogic-ejb-jar.xml` or `weblogic.xml`—Work Managers specified at the component-level can be assigned to that component.
- `weblogic.xml`—Work Managers specified for a Web Application.

Listing 2-1 is an example of a Work Manager definition.

Listing 2-1 Work Manager Stanza

```
<work-manager>
<name>highpriority_workmanager</name>
   <fair-share-request-class>
      <name>high_priority</name>
      <fair-share>100</fair-share>
   </fair-share-request-class>
   <min-threads-constraint>
      <name>MinThreadsCountFive</name>
      <count>5</count>
```

```
   </min-threads-constraint>
</work-manager>
```

To reference the Work Manager used in the example in Listing 2-1 in the dispatch policy of a Web Application, add the code in Listing 2-2 to the Web Application's `web.xml` file:

Listing 2-2 Referencing the Work Manager in a Web Application

```
<init-param>
   <param-name>wl-dispatch-policy</param-name>
   <param-value>highpriority_workmanager</param-value>
</init-param>
```

The components you can define and use in a Work Manager are described in following sections.

Request Classes

A request class expresses a scheduling guideline that WebLogic Server uses to allocate threads to requests. Request classes help ensure that high priority work is scheduled before less important work, even if the high priority work is submitted after the lower priority work. WebLogic Server takes into account how long it takes for requests to each module to complete

There are multiple types of request class, each of which expresses a scheduling guideline in different terms. A Work Manager may specify only one request class.

- `fair-share-request-class`—Specifies the average thread-use time required to process requests.

  For example, assume that WebLogic Server is running two modules. The Work Manager for `ModuleA` specifies a `fair-share-request-class` of 80 and the Work Manager for `ModuleB` specifies a `fair-share-request-class` of 20.

  During a period of sufficient demand, with a steady stream of requests for each module such that the number requests exceed the number of threads, WebLogic Server will allocate 80% and 20% of the thread-usage time to `ModuleA` and `ModuleB`, respectively.

Note: The value of a fair share request class is specified as a relative value, not a percentage. Therefore, in the above example, if the request classes were defined as 400 and 100, they would still have the same relative values.

- `response-time-request-class`—This type of request class specifies a response time goal in milliseconds. Response time goals are not applied to individual requests. Instead,

WebLogic Server computes a tolerable waiting time for requests with that class by subtracting the observed average thread use time from the response time goal, and schedules requests so that the average wait for requests with the class is proportional to its tolerable waiting time.

For example, given that ModuleA and ModuleB in the previous example, have response time goals of 2000 ms and 5000 ms, respectively, and the actual thread use time for an individual request is less than its response time goal. During a period of sufficient demand, with a steady stream of requests for each module such that the number requests exceed the number of threads, and no "think time" delays between response and request, WebLogic Server will schedule requests for `ModuleA` and `ModuleB` to keep the average response time in the ratio 2:5. The actual average response times for `ModuleA` and `ModuleB` might be higher or lower than the response time goals, but will be a common fraction or multiple of the stated goal. For example, if the average response time for `ModuleA` requests is 1,000 ms., the average response time for `ModuleB` requests is 2,500 ms.

- `context-request-class`—This type of request class assigns request classes to requests based on context information, such as the current user or the current user's group.

  For example, the `context-request-class` in "Context Request Class" on page 2-5 assigns a request class to requests based on the value of the request's `subject` and `role` properties.

Context Request Class

A context request class allows you to define request classes in an application's deployment descriptor based on a user's context. For example:

Listing 2-3 Context Request Class

```
<work-manager>
  <name>responsetime_workmanager</name>
    <response-time-request-class>
      <name>my_response_time</name>
      <goal-ms>2000</goal-ms>
   </response-time-request-class>
</work-manager>

<work-manager>
  <name>context_workmanager</name>
  <context-request-class>
    <name>test_context</name>
    <context-case>
```

```
      <user-name>system</user-name>
      <request-class-name>high_fairshare</request-class-name>
   </context-case>
   <context-case>
      <group-name>everyone</group-name>
      <request-class-name>low_fairshare</request-class-name>
    </context-case>
  </context-request-class>
</work-manager>
```

Above, we explained the request classes based on fair share and response time by relating the scheduling to other work using the same request class. A mix of fair share and response time request classes is scheduled with a marked bias in favor of response time scheduling.

Constraints

A constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests.

You can define the following types of constraints:

- `max-threads-constraint`—This constraint limits the number of concurrent threads executing requests from the constrained work set. The default is unlimited. For example, consider a constraint defined with maximum threads of 10 and shared by 3 entry points. The scheduling logic ensures that not more than 10 threads are executing requests from the three entry points combined.

  A `max-threads-constraint` can be defined in terms of a the availability of resource that requests depend upon, such as a connection pool.

  A `max-threads-constraint` might, but does not necessarily, prevent a request class from taking its fair share of threads or meeting its response time goal. Once the constraint is reached the server does not schedule requests of this type until the number of concurrent executions falls below the limit. The server then schedules work based on the fair share or response time goal.

- `min-threads-constraint`—This constraint guarantees a number of threads the server will allocate to affected requests to avoid deadlocks. The default is zero. A `min-threads-constraint` value of one is useful, for example, for a replication update request, which is called synchronously from a peer.

  A `min-threads-constraint` might not necessarily increase a fair share. This type of constraint has an effect primarily when the server instance is close to a deadlock condition.

In that case, the constraint will cause WebLogic Server to schedule a request even if requests in the service class have more requests than its fair share recently.

- `capacity`—This constraint causes the server to reject requests only when it has reached its capacity. The default is -1. Note that the capacity includes all requests, queued or executing, from the constrained work set. Work is rejected either when an individual capacity threshold is exceeded or if the global capacity is exceeded. This constraint is independent of the global queue threshold.

Stuck Thread Handling

In response to stuck threads, you can define a Stuck Thread Work Manager component that can shut down the Work Manager, move the application into admin mode, or mark the server instance as failed.

For instance, the Work Manager defined in Listing 2-4 shuts down the Work Manager when two threads are stuck for longer than 30 seconds.

Listing 2-4 Stuck-Thread Work Manager

```
<work-manager>
   <name>stuckthread_workmanager</name>
   <work-manager-shutdown-trigger>
      <max-stuck-thread-time>30</max-stuck-thread-time>
      <stuck-thread-count>2</stuck-thread-count>
   </work-manager-shutdown-trigger>
</work-manager>
```

Work Manager Scope

Essentially, there are three types of Work Managers, each one characterized by its scope and how it is defined and used. The three types are:

- The default Work Manager
- Global Work Managers
- Application-scoped Work Managers

These three types of Work Managers are described in the following sections:

The Default Work Manager

To handle thread management and perform self-tuning, WebLogic Server implements a default Work Manager. This Work Manager is used by an application when no other Work Managers are specified in the application's deployment descriptors.

In many situations, the default Work Manager may be sufficient for most application requirements. WebLogic Server's thread-handling algorithms assign each application its own fair share by default. Applications are given equal priority for threads and are prevented from monopolizing them.

Overriding the Default Work Manager

You can override the behavior of the default Work Manager by creating and configuring a global Work Manager called `default`. This allows you to control the default thread-handling behavior of WebLogic Server.

When to Use Work Managers

Following are guildelines to help you determine when you might want to use Work Managers to customize thread management:

- The default fair share is not sufficient.

  This usually ocurrs in situations where one application needs to be given higher priority over another.

- A response time goal is required.
- A minimum thread constraint needs to be specified to avoid server deadlock

Global Work Managers

You can create Work Managers that are available to all applications and modules deployed on a server. Global Work Managers are created in the WebLogic Administration Console and are defined in `config.xml`.

An application uses a globally defined Work Manager as a template. Each application creates its own instance which handles the work associated with that application and separates that work from other applications. This separation is used to handle traffic directed to two applications which are using the same dispatch policy. Handling each application's work separately, allows an application to be shut down without affecting the thread management of another application.

Although each application implments its own Work Manager instance, the underlying components are shared.

Application-scoped Work Managers

In addition to globally-scoped Work Managers, you can also create Work Managers that are available only to a specific application or module. Work Managers can be specified in the following descriptors:

- `weblogic-application.xml`
- `weblogic-ejb-jar.xml`
- `weblogic.xml`

If you do not explicitly assign a Work Manager to an application, it uses the default Work Manager.

A method is assigned to a Work Manager, using the `<dispatch-policy>` element in the deployment descriptor. The `<dispatch-policy>` can also identify a custom execute queue, for backward compatibility. For an example, see Listing 2-2, “Referencing the Work Manager in a Web Application,” on page 2-4.

Using Work Managers, Request Classes, and Constraints

Work Managers, Request Classes, and Constraints require the following:

- A definition. You may define a Work Managers, Request Classes, or Constraints globally in the domain’s configuration using the Administration Console, (see Environments > Work Managers in the Administration Console) or you may define them in one of the deployment descriptors listed above. In either case, you assign a name to each.
- A mapping. In your deployment descriptors you reference one of the Work Managers, Request Classes, or Constraints by its name.

Dispatch Policy for EJB

`weblogic-ejb-jar.xml`—the value of the existing `dispatch-policy` tag under `weblogic-enterprise-bean` can be a named `dispatch-policy`. For backwards compatibility, it can also name an ExecuteQueue. In addition, we allow `dispatch-policy`, `max-threads`, and `min-threads`, to specify named (or unnamed with numeric value for constraints) policy and constraints for a list of methods, analogously to the present `isolation-level` tag.

Dispatch Policy for Web Applications

`weblogic.xml`—also supports mappings analogous to the `filter-mapping` of the `web.xml`, where named dispatch-policy, max-threads, or min-threads are mapped for url-patterns or servlet names.

Deployment Descriptor Examples

This section contains examples for defining Work Managers in various types of deployment descriptors.

For additional reference, see also the schema for these deployment descriptors:

- weblogic-ejb-jar.xml schema
- weblogic-application.xml schema
- weblogic.xml schema

Listing 2-5 weblogic-ejb-jar.xml With Work Manager Entries

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

<weblogic-enterprise-bean>
   <ejb-name>WorkEJB</ejb-name>
   <jndi-name>core_work_ejb_workbean_WorkEJB</jndi-name>
   <dispatch-policy>weblogic.kernel.System</dispatch-policy>
</weblogic-enterprise-bean>

<weblogic-enterprise-bean>
   <ejb-name>NonSystemWorkEJB</ejb-name>
   <jndi-name>core_work_ejb_workbean_NonSystemWorkEJB</jndi-name>
   <dispatch-policy>workbean_workmanager</dispatch-policy>
</weblogic-enterprise-bean>
```

```
<weblogic-enterprise-bean>
   <ejb-name>MinThreadsWorkEJB</ejb-name>
   <jndi-name>core_work_ejb_workbean_MinThreadsWorkEJB</jndi-name>
   <dispatch-policy>MinThreadsCountFive</dispatch-policy>
</weblogic-enterprise-bean>

<work-manager>
   <name>workbean_workmanager</name>
</work-manager>

<work-manager>
   <name>stuckthread_workmanager</name>
   <work-manager-shutdown-trigger>
      <max-stuck-thread-time>30</max-stuck-thread-time>
      <stuck-thread-count>2</stuck-thread-count>
   </work-manager-shutdown-trigger>
</work-manager>

<work-manager>
   <name>minthreads_workmanager</name>
   <min-threads-constraint>
      <name>MinThreadsCountFive</name>
      <count>5</count>
   </min-threads-constraint>
</work-manager>

<work-manager>
   <name>lowpriority_workmanager</name>
   <fair-share-request-class>
      <name>low_priority</name>
      <fair-share>10</fair-share>
   </fair-share-request-class>
</work-manager>

<work-manager>
<name>highpriority_workmanager</name>
   <fair-share-request-class>
      <name>high_priority</name>
```

```
      <fair-share>100</fair-share>
   </fair-share-request-class>
</work-manager>

<work-manager>
<name>veryhighpriority_workmanager</name>
   <fair-share-request-class>
      <name>veryhigh_priority</name>
      <fair-share>1000</fair-share>
   </fair-share-request-class>
</work-manager>
```

Listing 2-6 weblogic-ejb-jar.xml with Connection Pool Based Max Thread Constraint

These EJBs are configured to get as many threads as there are instances of a resource they depend upon—a connection pool, and an application scoped connection pool.

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

   <weblogic-enterprise-bean>
     <ejb-name>ResourceConstraintEJB</ejb-name>
      <jndi-name>core_work_ejb_resource_ResourceConstraintEJB</jndi-name>
      <dispatch-policy>test_resource</dispatch-policy>
   </weblogic-enterprise-bean>

   <weblogic-enterprise-bean>
      <ejb-name>AppScopedResourceConstraintEJB</ejb-name>
      <jndi-name>core_work_ejb_resource_AppScopedResourceConstraintEJB
      </jndi-name>
      <dispatch-policy>test_appscoped_resource</dispatch-policy>
   </weblogic-enterprise-bean>
```

```
<work-manager>
   <name>test_resource</name>
    <max-threads-constraint>
      <name>pool_constraint</name>
      <pool-name>testPool</pool-name>
   </max-threads-constraint>
</work-manager>

<work-manager>
   <name>test_appscoped_resource</name>
   <max-threads-constraint>
      <name>appscoped_pool_constraint</name>
      <pool-name>AppScopedDataSource</pool-name>
   </max-threads-constraint>
</work-manager>
</weblogic-ejb-jar>
```

Listing 2-7 weblogic-ejb-jar.xml with commonJ Work Managers

For information using commonJ, see "Using CommonJ With WebLogic Server"and the commonJ Javadocs.

Listing 2-8 weblogic-application.xml

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

   <max-threads-constraint>
      <name>j2ee_maxthreads</name>
      <count>1</count>
```

```
   </max-threads-constraint>

   <min-threads-constraint>
      <name>j2ee_minthreads</name>
      count>1</count>
   </min-threads-constraint>

   <work-manager>
      <name>J2EEScopedWorkManager</name>
      </work-manager>
</weblogic-application>
```

Listing 2-9 web application descriptor

This Web Application is deployed as part of the Enterprise Application defined in Listing 2-8, "weblogic-application.xml," on page 2-13. This Web Application's descriptor defines two Work Managers. Both Work Managers point to the same max threads constraint, `j2ee_maxthreads` which is defined in the application's `weblogic-application.xml` file. Each Work Manager specifies a different response time request class.

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

   <work-manager>
      <name>fast_response_time</name>
      <response-time-request-class>
         <name>fast_response_time</name>
         <goal-ms>2000</goal-ms>
      </response-time-request-class>
      <max-threads-constraint-name>j2ee_maxthreads
      </max-threads-constraint-name>
   </work-manager>

   <work-manager>
```

```
      <name>slow_response_time</name>
      <max-threads-constraint-name>j2ee_maxthreads
      </max-threads-constraint-name
      <response-time-request-class>
         <name>slow_response_time</name>
         <goal-ms>5000</goal-ms>
      </response-time-request-class>
   </work-manager>

</weblogic>
```

Listing 2-10 web application descriptor

This descriptor defines a Work Manager using the context-request-class.

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

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

 <work-manager>
  <name>foo-servlet-1</name>
  <request-class-name>test-fairshare2</request-class-name>
  <max-threads-constraint>
    <name>foo-mtc</name>
    <pool-name>oraclePool</pool-name>
  </max-threads-constraint>
 </work-manager>

 <work-manager>
  <name>foo-servlet</name>
  <context-request-class>
  <name>test-context</name>
  <context-case>
    <user-name>anonymous</user-name>
```

```
    <request-class-name>test-fairshare1</request-class-name>
  </context-case>
   <context-case>
    <group-name>everyone</group-name>
    <request-class-name>test-fairshare2</request-class-name>
   </context-case>
  </context-request-class>
 </work-manager>
</weblogic-web-app>
```

Work Managers and Execute Queues

This section discusses how to enable backward compatibility with Execute Queues and how to migrate applications from using Execute Queues to Work Managers.

Enabling Execute Queues

WebLogic Server, Version 8.1 implemented Execute Queues to handle thread management which allowed you to create thread-pools to determine how workload was handled. WebLogic Server still provides Execute Queues for backward compaitibility, primarily to facilitate application migration. However, new application development should utilize Work Managers to peform thread management more efficiently.

You can enable Execute Queues in the following ways:

- Using the command line option

  `-Dweblogic.Use81StyleExecuteQueues=true`

- Setting the `Use81StyleExecuteQueues` property via the Kernel MBean in `config.xml`.

Enabling Execute Queues disables all Work Manager configuration and thread self tuning. Execute Queues behave exactly as they did in WebLogic Server 8.1. See Using User-Defined Execute Queues in *WebLogic Server Performance and Tuning*.

When enabled, Work Managers are converted to Execute Queues based on the following rules:

- If the Work Manager implements a minimum or maximum threads constraint, then an Execute Queue is created with the same name as the Work Manager. The thread count of the Execute Queue is based on the value defined in the constraint.

- If the Work Manager does not implement any constraints, the the global default Execute Queue is used.

Migrating from Execute Queues to Work Managers

When an application is migrated from WebLogic Server 8.1, any Execute Queues defined in the server configuration before migration will still be present. WebLogic Server does not automatically convert the Execute Queues to Work Managers.

When an 8.1 application implementing Execute Queues is deployed on WebLogic Server 9.x, the Execute Queues are created to handle thread management for requests. However, only those requests whose dispatch-policy maps to an Execute Queue will take advantage of this feature.

Accessing Work Managers Using MBeans

Work Managers can be accessed using the WorkManagerMBean configuration MBean. For more information, see WorkManagerMBean.

WorkManagerMBean is accessed in the runtime tree or configuration tree depending on how the Work Manager is accessed by an application.

- If the Work Manager is defined at the module level, the WorkManagerRuntime MBean is available through the corresponding ComponentRuntimeMBean.
- If a Work Manager is defined at the application level, then WorkManagerRuntime is available through ApplicationRuntime.
- If a Work Manager is defined globally in config.xml, each application creates its own instance of the Work Manager. Each application has its own corresponding WorkManagerRuntime available at the application level.

Using CommonJ With WebLogic Server

WebLogic Server Work Managers provide server-level configuration that allows administrators a way to set dispatch-policies to their servlets and EJBs.

WebLogic Server also provides a programmatic way of handling work from within an application. This is provided via the CommonJ API. Weblogic Server implements the `commonj.work` and `commonj.timers` packages of the CommonJ specificaion.

For general information on the CommonJ specification, see http://dev2dev.bea.com/wlplatform/commonj/twm.html. For specific information WebLogic Server's implementation of CommonJ, see the CommonJ Javadocs.

The WebLogic Server implementation of CommonJ enables an application to break a single request task into multiple work items, and assign those work items to execute concurrently using multiple Work Managers configured in WebLogic Server. Applications that do not need to execute concurrent work items can also use configured Work Managers by referencing or creating Work Managers in their deployment descriptors or, for Java EE Connectors, using the JCA API.

Following are some differences between the WebLogic Server implementation and the CommonJ specification:

- The RemoteWorkItem interface is an optional interface provided by the CommonJ specification and is not supported in WebLogic Server. WebLogic Server implements its own cluster load balancing and failover policies. Workload management is based on these policies.
- WebLogic CommonJ timers behave differently than `java.util.Timer`. When the execution is greater that twice the period, the WebLogic CommonJ timer will skip some periods to avoid falling further behind. The `java.util.Timer` does not do this.

Accessing CommonJ Work Managers

Unlike WebLogic Server Work Managers, which can only be accessed from an application via dispatch policies, you can access CommonJ Work Managers directly from an application. The following code example demonstrates how to lookup a CommonJ work manager using JNDI:

```
InitialContext ic = new InitialContext();
commonj.work.WorkManager wm =
(commonj.work.WorkManager)ic.lookup("java:comp/env/wm/myWM");
```

For more information on using CommonJ Work Managers, see the CommonJ Javadocs.

Mapping CommonJ to WebLogic Server Work Managers

You can map an externally defined CommonJ Work Manager to a WebLogic Server Work Manager. For example, if you have a CommonJ Work Manager defined in a descriptor, `ejb-jar.xml` for example, as:

```
<resource-ref>
<res-ref-name>minthreads_workmanager</res-ref-name>
```

```
<res-type>commonj.work.WorkManager</res-type>
<res-auth>Container</res-auth>
<res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

You can link this to a WebLogic Server Work Manager by ensuring that the `name` element is identical in the WebLogic Server descriptor such as `weblogic-ejb-jar.xml`:

```
<work-manager>
<name>minthreads_workmanager</name>
<min-threads-constraint>
<count>5</count>
</min-threads-constraint>
</work-manager>
```

This procedure is similar for a resource-ref defined in web.xml. The WebLogic Server Work Manager can be defined in either a module descriptor (`weblogic-ejb-jar.xml` or `weblogic.xml`, for example) or in the application descriptor (`weblogic-application.xml`).

CHAPTER 3

Avoiding and Managing Overload

WebLogic Server has features for detecting, avoiding, and recovering from overload conditions. WebLogic Server's overload protection features help prevent the negative consequences—degraded application performance and stability—that can result from continuing to accept requests when the system capacity is reached.

Configuring WebLogic Server to Avoid Overload Conditions

When system capacitiy is reached, if an application server continues to accept requests, application performance and stability can deteriorate. The following sections demonstrate how you can configure WebLogic Server to minimize the negative results of system overload.

Limiting Requests in the Thread Pool

In WebLogic Server, all requests, whether related to system administration or application activity—are processed by a single thread pool. An administrator can throttle the thread pool by defining a maximum queue length. Beyond the configured value, WebLogic Server will refuse requests, except for requests on administration channels.

Note: Administration channels allow access only to administrators. The limit you set on the execute length does not effect administration channel requests, to ensure that reaching the maximum thread pool length does not prevent administrator access to the system. To limit the number of administration requests allowed in the thread pool, you can configure an administration channel, and set the MaxConnectedClients attribute for the channel.

When the maximum number of enqueued requests is reached, WebLogic Server immediately starts rejecting:

- Web application requests.
- Non-transactional RMI requests with a low fair share, beginning with those with the lowest fair share.

  If the overload condition continues to persist, higher priority requests will start getting rejected, with the exception of JMS and transaction-related requests, for which overload management is provided by the JMS and the transaction manager.

Throttle the thread pool by setting the `Shared Capacity For Work Managers` field in the Administration Console (see Environments > Servers > Threads and select an execute queue). The default value of this field is 65536.

Work Managers and Thread Pool Throttling

An administrator can configure Work Managers to manage the thread pool at a more granular level, for sets of requests that have similar performance, availability, or reliability requirements. A Work Manager can specify the maximum requests of a particular request class that can be queued. The maximum requests defined in a Work Manager works with global thread pool value. The limit that is reached first is honored.

See "Using Work Managers to Optimize Scheduled Work" on page 2-1.

Limiting HTTP Sessions

An administrator can limit the number of active HTTP sessions based on detection of a low-memory condition. This is useful in avoiding out of memory exceptions.

WebLogic Server refuses requests that create new HTTP sessions after the configured threshold has been reached. In a WebLogic Server cluster, the proxy plug-in redirects a refused request to another Managed Server in the cluster. A non-clustered server instance can re-direct requests to alternative server instance.

The servlet container takes one of the following actions when maximum number of sessions is reached:

- If the server instance is in a cluster, a the servlet container throws a SessionCreationException. Your application code should handle this runtime exception and send a relevant response.

  To implement overload protection, you should handle this exception and send a 503 response explicitly. This response can then be handled by the proxy or load balancer.

You set a limit for the number of simultaneous HTTP sessions in the deployment descriptor for the Web Application. For example, the following element sets a limit of 12 sessions:

```
<session-descriptor>
  <max-in-memory-sessions>12</max-in-memory-sessions>
</session-descriptor>
```

Exit on Out of Memory Exceptions

Administrators can configure WebLogic Server to exit upon an out of memory exception. This feature allows you to minimize the impact of the out of memory condition—automatic shutdown helps avoid application instability, and you can configure Node Manager or another HA tool to automatically restart WebLogic Server, minimizing down-time.

You can configure this using the Administration Console, or by editing the following elements in config.xml:

```
<overload-protection>
   <panic-action>system-exit</panic-action>
</overload-protection>
```

For more information, see the attributes of the OverloadProtectionMBean.

Stuck Thread Handling

WebLogic Server checks for stuck threads periodically. If all application threads are stuck, a server instance marks itself failed, if configured to do so, exits. You can configure Node Manager or a third-part high-availability solution to restart the server instance for automatic failure recovery.

You can configure these actions to occur when not all threads are stuck, but the number of stuck threads have exceeded a configured threshold,

- Shut down the Work Manager if it has stuck threads. A Work Manager that is shut down will refuse new work and reject existing work in the queue by sending a rejection message. In a cluster, clustered clients will fail over to another cluster member.

- Shut down the application if there are stuck threads in the application. The application is shutdown by bringing it into admin mode. All Work Managers belonging to the application are shut down, and behave as described above.
- Mark the server instance as failed and shut it down it down if there are stuck threads in the server. In a cluster, clustered clients that are connected or attempting to connect will fail over to another cluster member.

For more information, see the attributes of the OverloadProtectionMBean.

WebLogic Server Self-Monitoring

The following sections describe WebLogic Server features that aid in determining and reporting overload conditions.

Overloaded Health State

WebLogic Server has a new health state—`OVERLOADED`—which is returned by the `ServerRuntimeMBean.getHealthState()` when a server instance whose life cycle state is `RUNNING` becomes overloaded. This condition occurs as a result of low memory.

The server instances health state returns to `OK` after the overload condition passes. An administrator can suspend or shut down an `OVERLOADED` server instance.

WebLogic Server Exit Codes

When WebLogic Server exits it returns an exit code. The exit codes can be used by shell scripts or HA agents to decide whether a server restart is necessary. See "WebLogic Server Exit Codes and Restarting After Failure" in *Managing Server Startup and Shutdown*.

CHAPTER 4

Configuring Network Resources

Configurable WebLogic Server resources, including network channels and domain-wide administration ports, help you effectively utilize the network features of the machines that host your applications and manage quality of service.

The following sections describe configurable WebLogic Server network resources, examples of their use, and the configuration process:

Overview of Network Configuration

For many development environments, configuring WebLogic Server network resources is simply a matter of identifying a Managed Server's listen address and listen port. However, in most production environments, administrators must balance finite network resources against the demands placed upon the network. The task of keeping applications available and responsive can be complicated by specific application requirements, security considerations, and maintenance tasks, both planned and unplanned.

WebLogic Server allows you to control the network traffic associated with your applications in a variety of ways, and configure your environment to meet the varied requirements of your applications and end users. You can:

- Designate the Network Interface Cards (NICs) and ports used by Managed Servers for different types of network traffic.
- Support multiple protocols and security requirements.
- Specify connection and message timeout periods.
- Impose message size limits.

You specify these and other connection characteristics by defining a network channel—the primary configurable WebLogic Server resource for managing network connections. You configure a network channel with the Servers-->Protocols-->Channels tab of the Administration Console or by using `NetworkAccessPointMBean`.

New Network Configuration Features in WebLogic Server

In this version of WebLogic Server, the functionality of network channels is enhanced to simplify the configuration process. Network channels now encompass the features that, in WebLogic Server 7.x, required both network channels and network access points. In this version of WebLogic Server, network access points are deprecated. The use of `NetworkChannelMbean` is deprecated in favor of `NetworkAccessPointMBean`.

Understanding Network Channels

The sections that follow describe network channels and the standard channels that WebLogic Server pre-configures, and discusses common applications for channels.

What Is a Channel?

A network channel is a configurable resource that defines the attributes of a network connection to WebLogic Server. For instance, a network channel can define:

- The protocol the connection supports.
- The listen address.
- The listen ports for secure and non-secure communication.
- Connection properties such as the login timeout value and maximum message sizes.
- Whether or not the connection supports tunneling.

- Whether the connection can be used to communicate with other WebLogic Server instances in the domain, or used only for communication with clients.

Rules for Configuring Channels

Follow these guidelines when configuring a channel.

- You can assign a particular channel to only one server instance.
- You can assign multiple channels to a server instance.
- Each channel assigned to a particular server instance must have a unique combination of listen address, listen port, and protocol.
- If you assign non-SSL and SSL channels to the same server instance, make sure that they do not use the same port number.

Custom Channels Can Inherit Default Channel Attributes

If you do not assign a channel to a server instance, it uses WebLogic Server's default channel, which is automatically configured by WebLogic Server, based on the attributes in `ServerMBean` or `SSLMBean`. The default channel is described in "The Default Network Channel" on page 4-5.

`ServerMBean` and `SSLMBean` represent a server instance and its SSL configuration. When you configure a server instance's Listen Address, Listen Port, and SSL Listen port, using the Server-->Configuration-->General tab, those values are stored in the `ServerMBean` and `SSLMBean` for the server instance.

If you do not specify a particular connection attribute in a custom channel definition, the channel inherits the value specified for the attribute in `ServerMBean`. For example, if you create a channel, and do not define its Listen Address, the channel uses the Listen Address defined in `ServerMBean`. Similarly, if a Managed Server cannot bind to the Listen Address or Listen Port configured in a channel, the Managed Server uses the defaults from `ServerMBean` or `SSLMBean`.

Why Use Network Channels?

You can use network channels to manage quality of service, meet varying connection requirements, and improve utilization of your systems and network resources. For example, network channels allow you to:

- **Segregate different types of network traffic**—You can configure whether or not a channel supports outgoing connections. By assigning two channels to a server instance—one that supports outgoing connections and one that does not—you can independently

configure network traffic for client connections and server connections, and physically separate client and server network traffic onto different listen addresses or listen ports.

You can also segregate instance administration and application traffic by configuring a domain-wide administration port or administration channel. For more information, see "Administration Port and Administrative Channel" on page 4-5.

- **Support varied application or user requirements on the same Managed Server—**You can configure multiple channels on a Managed Server to support different protocols, or to tailor properties for secure vs. non-secure traffic.
- **Segregate internal application network traffic**—You can assign a specific channel to a an EJB.

If you use a network channel with a server instance on a multi-homed machine, you must enter a valid Listen Address either in `ServerMBean` or in the channel. If the channel and `ServerMBean` Listen Address are blank or specify the localhost address (IP address 0.0.0.0 or 127.*.*.*), the server binds the network channel listen port and SSL listen ports to all available IP addresses on the multi-homed machine. See "The Default Network Channel" on page 4-5 for information on setting the listen address in `ServerMBean`.

Handling Channel Failures

When initiating a connection to a remote server, and multiple channels with the same required destination, protocol and quality of service exist, WebLogic Server will try each in turn until it successfully establishes a connection or runs out of channels to try.

Upgrading Quality of Service Levels for RMI

For RMI lookups only, WebLogic Server may upgrade the service level of an outgoing connection. For example, if a T3 connection is required to perform an RMI lookup, but an existing channel supports only T3S, the lookup is performed using the T3S channel.

This upgrade behavior does not apply to server requests that use URLs, since URLs embed the protocol itself. For example, the server cannot send a URL request beginning with `http://` over a channel that supports only `https://`.

Standard WebLogic Server Channels

WebLogic Server provides pre-configured channels that you do not have to explicitly define.

- Default channel—Every Managed Server has a default channel.

- Administrative channel—If you configure a domain-wide Administration Port, WebLogic Server configures an Administrative Channel for each Managed Server in the domain.

The Default Network Channel

Every WebLogic Server domain has a default channel that is generated automatically by WebLogic Server. The default channel is based on the Listen Address and Listen Port defined in the `ServerMBean` and `SSLMBean`. It provides a single Listen Address, one port for HTTP communication (7001 by default), and one port for HTTPS communication (7002 by default). You can configure the Listen Address and Listen Port using the Configuration-->General tab in the Administration Console; the values you assign are stored in attributes of the `ServerMBean` and `SSLMBean`.

The default configuration may meet your needs if:

- You are installing in a test environment that has simple network requirements.
- Your server uses a single NIC, and the default port numbers provide enough flexibility for segmenting network traffic in your domain.

Using the default configuration ensures that third-party administration tools remain compatible with the new installation, because network configuration attributes remain stored in `ServerMBean` and `SSLMBean`.

Even if you define and use custom network channels for your domain, the default channel settings remain stored in `ServerMBean` and `SSLMBean`, and are used if necessary to provide connections to a server instance.

Administration Port and Administrative Channel

You can define an optional administration port for your domain. When configured, the administration port is used by each Managed Server in the domain exclusively for communication with the domain's Administration Server.

Administration Port Capabilities

An administration port enables you to:

- Start a server in standby state. This allows you to administer a Managed Server, while its other network connections are unavailable to accept client connections. For more information on the standby state, see Standby State in *Managing Server Startup and Shutdown*.

- Separate administration traffic from application traffic in your domain. In production environments, separating the two forms of traffic ensures that critical administration operations (starting and stopping servers, changing a server's configuration, and deploying applications) do not compete with high-volume application traffic on the same network connection.
- Administer a deadlocked server instance using the `weblogic.Admin` command line utility. If you do not configure an administration port, administrative commands such as `THREAD_DUMP` and `SHUTDOWN` will not work on deadlocked server instances.

If a administration port is enabled, WebLogic Server automatically generates an administration channel based on the port settings upon server instance startup.

Administration Port Restrictions

The administration port accepts only secure, SSL traffic, and all connections via the port require authentication. Enabling the administration port imposes the following restrictions on your domain:

- The Administration Server and all Managed Servers in your domain must be configured with support for the SSL protocol. Managed Servers that do not support SSL cannot connect with the Administration Server during startup—you will have to disable the administration port in order to configure them.
- Because all server instances in the domain must enable or disable the administration port at the same time, you configure the administration port at the domain level. You can change an individual Managed Server's administration port number, but you cannot enable or disable the administration port for an individual Managed Server. The ability to change the port number is useful if you have multiple server instances with the same Listen Address.
- After you enable the administration port, you must establish an SSL connection to the Administration Server in order to start any Managed Server in the domain. This applies whether you start Managed Servers manually, at the command line, or using Node Manager. For instructions to establish the SSL connection, see "Administration Port Requires SSL" on page 4-7.
- After enabling the administration port, all Administration Console traffic *must* connect via the administration port.
- If multiple server instances run on the same computer in a domain that uses a domain-wide administration port, you must either:
  - Host the server instances on a multi-homed machine and assign each server instance a unique listen address, or

– Override the domain-wide port on all but one of one of the servers instances on the machine. Override the port using the Local Administration Port Override option on the Advanced Attributes portion of the Server->Connections->SSL Ports page in the Administration Console.

Administration Port Requires SSL

The administration port requires SSL, which is enabled by default when you install WebLogic Server. If SSL has been disabled for any server instance in your domain, including the Administration Server and all Managed Servers, re-enable it using the Server--> Configuration-->General tab in the Administration Console.

Ensure that each server instance in the domain has a configured default listen port or default SSL listen port. The default ports are those you assign on the Server-->Configuration-->General tab in the Administration Console. A default port is required in the event that the server cannot bind to its configured administration port. If an additional default port is available, the server will continue to boot and you can change the administration port to an acceptable value.

By default WebLogic Server is configured to use demonstration certificate files. To configure production security components, follow the steps in Configuring SSL in *Managing WebLogic Security*.

Configure Administration Port

Enable the administration port as described in Enabling the Domain-Wide Administration Port in *Administration Console Online Help*.

After configuring the administration port, you must restart the Administration Server and all Managed Servers to use the new administration port.

Booting Managed Servers to Use Administration Port

If you reboot Managed Servers at the command line or using a start script, specify the Administration Port in the port portion of the URL. The URL must specify the `https://` prefix, rather than `http://`, as shown below.

```
-Dweblogic.management.server=https://host:admin_port
```

Note: If you use Node Manager for restarting the Managed Servers, it is not necessary to modify startup settings or arguments for the Managed Servers. Node Manager automatically obtains and uses the correct URL to start a Managed Server.

If the hostname in the URL is not identical to the hostname in the Administration Server's certificate, disable hostname verification in the command line or start script, as shown below:

```
-Dweblogic.security.SSL.ignoreHostnameVerification=true
```

Booting Managed Servers to Use Administrative Channels

To allow a managed server to bind to an administrative channel during reboot, use the following command line option:

```
-Dweblogic.admin.ListenAddress=<addr>
```

This allows the managed server to startup using an administrative channel. After the initial bootstap connection, a standard administrative channel is used.

Note: This option is useful to ensure that the appropriate NIC semantics are used before config.xml is downloaded.

Custom Administrative Channels

If the standard WebLogic Server administrative channel does not satisfy your requirements, you can configure a custom channel for administrative traffic. For example, a custom administrative channel allows you to segregate administrative traffic on a separate NIC.

To configure a custom channel for administrative traffic, configure the channel as described in "Configuring a Channel" on page 4-9, and select "admin" as the channel protocol. Note the configuration and usage guidelines described in:

- "Administration Port Requires SSL" on page 4-7
- "Booting Managed Servers to Use Administration Port" on page 4-7

Using Internal Channels

Previous version of WebLogic Server allowed you to configure multiple channels for external traffic, but required you to use the default channel for internal traffic between server instances. WebLogic Server allows you to create network channels to handle administration traffic or communications between clusters. This can be useful in the following situations:

- internal administration traffic needs to occur over a secure connection, separate from other network traffic.
- other types of network traffic, for example replication data, need to occur over a separate network connection.
- certain types of network traffic need to be monitored.

Channel Selection

All internal traffic is handled via a network channel. If you have not created a custom network channel to handle adinstrative or clustered traffic, WebLogic Server automatically selects a default channel based on the protocol required for the connection. For more information on default channels, see "The Default Network Channel" on page 4-5.

Internal Channels Within a Cluster

Within a cluster, internal channels can be created to handle to the following types of server-to-server connections:

- multicast traffic
- replication traffic
- administration traffic

For more information on configuring channels within a cluster, see "Configuring Network Channels For a Cluster" on page 4-11.

Configuring a Channel

You can configure a network channel using Servers-->Protocols-->Channels tab in the Administration Console or using the `NetworkAccessPointMBean`.

To configure a channel for clustered Managed Servers see, "Configuring Network Channels For a Cluster" on page 4-11.

For a summary of key facts about network channels, and guidelines related to their configuration, see "Guidelines for Configuring Channels" on page 4-9.

Guidelines for Configuring Channels

Follow these guidelines when configuring a channel.

Channels and Server Instances

- Each channel you configure for a particular server instance must have a unique combination of listen address, listen port, and protocol.
- A channel can be assigned to a single server instance.
- You can assign multiple channels to a server instance.

- If you assign non-SSL and SSL channels to the same server instance, make sure that they do not use the same combination of address and port number.

Dynamic Channel Configuration

- In WebLogic Server, you can configure a network channel without restarting the server. Additionally, you can start and stop dynamically configured channels while the server is running. However, when you shutdown a channel while the server is running, the server does not attempt to gracefully terminate any work in progress.

Channels and Protocols

- Some protocols do not support particular features of channels. In particular the COM protocol does not support SSL or tunneling.
- You must define a separate channel for each protocol you wish the server instance to support, with the exception of HTTP.

  HTTP is enabled by default when you create a channel, because RMI protocols typically require HTTP support for downloading stubs and classes. You can disable HTTP support on the Advanced Options portion of Servers-->Protocols-->Channels tab in the Administration Console.

Reserved Names

- WebLogic Server uses the internal channel names `.WLDefaultChannel` and `.WLDefaultAdminChannel` and reserves the `.WL` prefix for channel names. do not begin the name of a custom channel with the string `.WL.`

Channels, Proxy Servers, and Firewalls

If your configuration includes a a firewall between a proxy web server and a cluster (as described in Firewall Between Proxy Layer and Cluster, in *Using WebLogicServer Clusters*, and the clustered servers are configured with two custom channels for segregating https and http traffic, those channels must share the same listen address. Furthermore if both http and https traffic needs to be supported there must be a custom channel for each—it is not possible to use the default configuration for one or the other.

If either of those channels has a `PublicAddress` defined, as is likely given existence of firewall both channels must define `PublicAddress`, and they both must define the same `PublicAddress`.

Configuring Network Channels For a Cluster

To configure a channel for clustered Managed Servers, note the information in "Guidelines for Configuring Channels" on page 4-9, and follow the guidelines described in the following sections.

Create the Cluster

If you have not already configured a cluster you can:

- Use the Configuration Wizard to create a new, clustered domain, following the instructions in Create a Clustered Domain in *Using WebLogic Clusters*, or
- Use the Administration Console to create a cluster in an existing domain, following the instructions Configuring a Cluster in *Administration Console Online Help*.

For information and guidelines about configuring a WebLogic Server cluster, see Before You Start in *Using WebLogic Clusters*.

Create and Assign the Network Channel

Use the instructions in Configuring a Network Channel in *Administration Console Online Help* to create a new network channel for each Managed Server in the cluster. When creating the new channels:

- For each channel you want to use in the cluster, configure the channel identically, including its name, on each Managed Server in the cluster.
- Make sure that the listen port and SSL listen port you define for each Managed Server's channel are different than the Managed Server's default listen ports. If the custom channel specifies the same port as a Managed Server's default port, the custom channel and the Managed Server's default channel will each try to bind to the same port, and you will be unable to start the Managed Server.
- If a cluster address has been explicitly configured for the cluster, it will be appear in the Cluster Address field on the Server-->Protocols-->Channels-->Configuration tab.

  If you are using dynamic cluster addressing, the cluster address field will be empty, and you do not need to supply a cluster address. For information about the cluster address, how WebLogic Server can dynamically generate the cluster address, see Cluster Address in *Using WebLogic Clusters*.

  Note: If you want to use dynamic cluster addressing, do not supply a cluster address on the Server-->Protocols-->Channels-->Configuration tab. If you supply a cluster address

explicitly, that value will take precedence and WebLogic Server will not generate the cluster address dynamically.

Configuring a Replication Channel

A replication channel is a network channel that is designated to transfer replication information between clusters. If a replication channel is not explicitly defined, WebLogic Server uses a default network channel to communicate replication information.

When WebLogic Server uses a default network channel as the replication channel, it does not use SSL encryption by default. You must enable SSL encryption using the `secureReplicationEnabled` attribute of the `ClusterMBean`. You can also update this setting from the Administration Console.

Enabling SSL encryption can have a direct impact on clustered application throughput as session replication is blocking by design. in other words, no response is sent to the client until replication is completed. You should consider this when deciding to enable SSL on the replication channel.

If a replication channel is explicitly define, the channel's protocol is used to transmit replication traffic.

Increase Packet Size When Using Many Channels

Use of more than about twenty channels in a cluster can result in the formation of multicast header transmissions that exceed the default maximum packet size. The `MTUSize` attribute in the `Server` element of `config.xml` sets the maximum size for packets sent using the associated network card to 1500. Sending packets that exceed the value of `MTUSize` can result in a `java.lang.NegativeArraySizeException`. You can avoid exceptions that result from packet sizes in excess of `MTUSize` by increasing the value of `MTUSize` from its default value of 1500.

Assigning a Custom Channel to an EJB

You can assign a custom channel to an EJB. After you configure a custom channel, assign it to an EJB using the `network-access-point` element in `weblogic-ejb-jar.xml`. For more information, see `network-access-point` in *Programming WebLogic Server EJBs*.

CHAPTER 5

Configuring Web Server Functionality

The following sections describe how to configure a Java EE Web Application hosted on WebLogic Server to function as a standard HTTP Web Server hosting static content. Web Applications also can host dynamic content such as JSPs and Servlets. See Developing Web Applications, Servlets, and JSPs for WebLogic Server.

Overview of Configuring Web Server Components

In addition to hosting dynamic Java-based distributed applications, WebLogic Server functions as a Web server that handles high-volume Web sites, serving static files such as HTML files and

image files, as well as servlets and JavaServer Pages (JSP). WebLogic Server supports the HTTP 1.1 standard.

Configuring the Server

You can specify the port that each WebLogic Server listens on for HTTP requests. Although you can specify any valid port number, if you specify port `80`, you can omit the port number from the HTTP request used to access resources over HTTP. For example, if you define port `80` as the listen port, you can use the form `http://hostname/myfile.html` instead of `http://hostname:portnumber/myfile.html`.

On UNIX systems, binding a process to a port lower than 1025 must be done from the account of a privileged user, usually root. Consequently, if you want WebLogic Server to listen on port 80, you must start WebLogic Server as a privileged user; yet it is undesirable from a security standpoint to allow long-running processes like WebLogic Server to run with more privileges than necessary. WebLogic needs root privileges only until the port is bound.

By setting the weblogic.system.enableSetUID property (and, if desired, the weblogic.system.enableSetGID property) to true, you enable an internal process by which WebLogic Server switches its UNIX user ID (UID) after it binds to port 80. The companion properties, weblogic.system.nonPrivUser and weblogic.system.nonPrivGroup, identify a non-privileged UNIX user account (and optionally a groupname) under which WebLogic Server will run after startup.

You can switch to the UNIX account "nobody," which is the least privileged user on most UNIX systems. If desired, you may create a UNIX user account expressly for running WebLogic Server. Make sure that files needed by WebLogic Server, such as log files and the WebLogic classes, are accessible by the non-privileged user. Once ownership of the WebLogic process has switched to the non-privileged user, WebLogic will have the same read, write, and execute permissions as the non-privileged user.

You define a separate listen port for non-SSL and secure (using SSL) requests. For additional information on configuring Listen Ports, see “Understanding Network Channels.”

Configuring the Listen Port

1. Use the Administration Console to set the listen port to port 80. See Configure Listen Ports.
2. If the machine hosting WebLogic Server is running Windows, skip to step 8 in “Configuring the Listen Port”.

3. Use the Administration Console to create a new Unix Machine. See Configure Machines.
4. Check the Enable Post-Bind UID field.
5. Enter the user name you want WebLogic Server to run as in the Post-Bind UID field.
6. Check the Enable Post-Bind GID fields.
7. Enter the group name you want WebLogic Server to run as in the Post-Bind GID field.
8. Click Save.
9. To activate these changes, in the Change Center of the Administration Console, click Activate Changes.

Web Applications

HTTP and Web Applications are deployed according to the Servlet 2.4 and JSP 2.0 specifications from Sun Microsystems, which describe *Web Applications* as a standard for grouping the components of a Web-based application. These components include JSP pages, HTTP servlets, and static resources such as HTML pages or image files. In addition, a Web Application can access external resources such as EJBs and JSP tag libraries. Each server can host any number of Web Applications. You normally use the name of the Web Application as part of the URI you use to request resources from the Web Application.

For more information, see Developing Web Applications, Servlets, and JSPs for WebLogic Server.

Web Applications and Clustering

Web Applications can be deployed to a WebLogic Server cluster. When a user requests a resource from a Web Application, the request is routed to one of the servers in the cluster that host the Web Application. If an application uses a session object, then sessions must be replicated across the nodes of the cluster. Several methods of replicating sessions are provided.

For more information, see Using WebLogic Server Clusters.

Designating a Default Web Application

Every server instance and virtual host in your domain can declare a *default Web Application*. The default Web Application responds to any HTTP request that cannot be resolved to another deployed Web Application. In contrast to all other Web Applications, the default Web

Application does *not* use the Web Application name as part of the URI. Any Web Application targeted to a server or virtual host can be declared as the default Web Application. (Targeting a Web Application is discussed later in this section. For more information about virtual hosts, see "Configuring Virtual Hosting" on page 5-4).

The examples domain that is shipped with WebLogic Server has a default Web Application already configured. The default Web Application in this domain is named `DefaultWebApp` and is located in the `applications` directory of the domain.

For example, if your Web Application is called `shopping`, you would use the following URL to access a JSP called `cart.jsp` from the Web Application:

```
http://host:port/shopping/cart.jsp
```

If, however, you declared `shopping` as the default Web Application, you would access `cart.jsp` with the following URL:

```
http://host:port/cart.jsp
```

(Where `host` is the host name of the machine running WebLogic Server and `port` is the port number where the WebLogic Server is listening for requests.)

To designate a default Web Application for a server or virtual host, set the context root in the application.xml or weblogic.xml file to "".

If you declare a default Web Application that fails to deploy correctly, an error is logged and users attempting to access the failed default Web Application receive an HTTP `404` error message.

Configuring Virtual Hosting

Virtual hosting allows you to define host names that servers or clusters respond to. When you use virtual hosting you use DNS to specify one or more host names that map to the IP address of a WebLogic Server instance or cluster, and you specify which Web Applications are served by the virtual host. When used in a cluster, load balancing allows the most efficient use of your hardware, even if one of the DNS host names processes more requests than the others.

For example, you can specify that a Web Application called `books` responds to requests for the virtual host name `www.books.com`, and that these requests are targeted to WebLogic Servers A,B and C, while a Web Application called `cars` responds to the virtual host name `www.autos.com` and these requests are targeted to WebLogic Servers D and E. You can configure a variety of combinations of virtual host, WebLogic Server instances, clusters, and Web Applications, depending on your application and Web server requirements.

For each virtual host that you define you can also separately define HTTP parameters and HTTP access logs. The HTTP parameters and access logs set for a *virtual host* override those set for a *server*. You may specify any number of virtual hosts.

You activate virtual hosting by targeting the virtual host to a server or cluster of servers. Virtual hosting targeted to a cluster will be applied to all servers in the cluster.

Virtual Hosting and the Default Web Application

You can also designate a *default Web Application* for each virtual host. The default Web Application for a virtual host responds to all requests that cannot be resolved to other Web Applications deployed on the same server or cluster as the virtual host.

Unlike other Web Applications, a default Web Application does not use the Web Application name (also called the *context path*) as part of the URI used to access resources in the default Web Application.

For example, if you defined virtual host name `www.mystore.com` and targeted it to a server on which you deployed a Web Application called `shopping`, you would access a JSP called `cart.jsp` from the `shopping` Web Application with the following URI:

```
http://www.mystore.com/shopping/cart.jsp
```

If, however, you declared `shopping` as the default Web Application for the virtual host `www.mystore.com`, you would access `cart.jsp` with the following URI:

```
http://www.mystore.com/cart.jsp
```

For more information, see "How WebLogic Server Resolves HTTP Requests" on page 5-6.

When using multiple Virtual Hosts with different default web applications, you can not use single sign-on, as each web application will overwrite the JSESSIONID cookies set by the previous web application. This will occur even if the CookieName, CookiePath, and CookieDomain are identical in each of the default web applications.

Setting Up a Virtual Host

1. Use the Administration Console to define a virtual host. See Virtual Host.
2. Add a line naming the virtual host to the `etc/hosts` file on your server to ensure that the virtual host name can be resolved.

How WebLogic Server Resolves HTTP Requests

When WebLogic Server receives an HTTP request, it resolves the request by parsing the various parts of the URL and using that information to determine which Web Application and/or server should handle the request. Table 5-1 demonstrates various combinations of requests for Web Applications, virtual hosts, servlets, JSPs, and static files and the resulting response.

Note: If you package your Web Application as part of an Enterprise Application, you can provide an alternate name for a Web Application that is used to resolve requests to the Web Application. For more information, see Developing Web Applications, Servlets, and JSPs for WebLogic Server.

Table 5-1 provides some sample URLs and the file that is served by WebLogic Server. The Index Directories Checked column refers to the Index Directories attribute that controls whether or not a directory listing is served if no file is specifically requested.

Table 5-1 Examples of How WebLogic Server Resolves URLs

URL	Index Directories Checked?	This file is served in response
`http://host:port/apples`	No	Welcome file* defined in the `apples` Web Application.
`http://host:port/apples`	Yes	Directory listing of the top level directory of the `apples` Web Application.
`http://host:port/oranges/naval`	Does not matter	Servlet mapped with `<url-pattern>` of `/naval` in the `oranges` Web Application. There are additional considerations for servlet mappings. For more information, see Configuring Servlets.

Table 5-1 Examples of How WebLogic Server Resolves URLs

URL	Index Directories Checked?	This file is served in response
`http://host:port/naval`	Does not matter	Servlet mapped with `<url-pattern>` of `/naval` in the `oranges` Web Application and `oranges` is defined as the default Web Application. For more information, see Configuring Servlets.
`http://host:port/apples/pie.jsp`	Does not matter	`pie.jsp`, from the top-level directory of the `apples` Web Application.
`http://host:port`	Yes	Directory listing of the top level directory of the *default* Web Application
`http://host:port`	No	Welcome file* from the *default* Web Application.
`http://host:port/apples/myfile.html`	Does not matter	`myfile.html`, from the top level directory of the `apples` Web Application.
`http://host:port/myfile.html`	Does not matter	`myfile.html`, from the top level directory of the *default* Web Application.
`http://host:port/apples/images/red.gif`	Does not matter	`red.gif`, from the images subdirectory of the top-level directory of the `apples` Web Application.
`http://host:port/myFile.html` Where `myfile.html` does not exist in the `apples` Web Application and a *default servlet* has not been defined.	Does not matter	Error 404

Table 5-1 Examples of How WebLogic Server Resolves URLs

URL	Index Directories Checked?	This file is served in response
`http://www.fruit.com/`	No	Welcome file from the default Web Application for a virtual host with a host name of `www.fruit.com`.
`http://www.fruit.com/`	Yes	Directory listing of the top level directory of the `default` Web Application for a virtual host with a host name of `www.fruit.com`.
`http://www.fruit.com/oranges/myfile.html`	Does not matter	`myfile.html`, from the `oranges` Web Application that is targeted to a virtual host with host name `www.fruit.com`.

Setting Up HTTP Access Logs

WebLogic Server can keep a log of all HTTP transactions in a text file, in either *common log format* or *extended log format*. Common log format is the default. Extended log format allows you to customize the information that is recorded. You can set the attributes that define the behavior of HTTP access logs for each server instance or for each virtual host that you define.

To set up HTTP logging for a server or a virtual host, refer to the following topics in the *Administration Console Online Help*:

- Enabling and Configuring HTTP Access Logs
- Specifying HTTP Log File Settings for a Virtual Host

Log Rotation

You can rotate the log file based on either the size of the file or after a specified amount of time has passed. When either criterion is met, the current access log file is closed and a new access log

file is started. If you do not configure log rotation, the HTTP access log file grows indefinitely. You can configure the name of the access log file to include a time and date stamp that indicates when the file was rotated. If you do not configure a time stamp, each rotated file name includes a numeric portion that is incremented upon each rotation. Separate HTTP access logs are kept for each Virtual Host you have defined.

Common Log Format

The default format for logged HTTP information is the common log format. This standard format follows the pattern:

```
host RFC931 auth_user [day/month/year:hour:minute:second
    UTC_offset] "request" status bytes
```

where:

host
: Either the DNS name or the IP number of the remote client

RFC931
: Any information returned by IDENTD for the remote client; WebLogic Server does not support user identification

auth_user
: If the remote client user sent a userid for authentication, the user name; otherwise "-"

day/month/year:hour:minute:second UTC_offset
: Day, calendar month, year and time of day (24-hour format) with the hours difference between local time and GMT, enclosed in square brackets

"request"
: First line of the HTTP request submitted by the remote client enclosed in double quotes

status
: HTTP status code returned by the server, if available; otherwise "-"

bytes
: Number of bytes listed as the content-length in the HTTP header, not including the HTTP header, if known; otherwise "-"

Setting Up HTTP Access Logs by Using Extended Log Format

WebLogic Server also supports extended log file format, version 1.0, an emerging standard defined by the draft specification from W3C. The current definitive reference is on the W3C Technical Reports and Publications page

The extended log format allows you to specify the type and order of information recorded about each HTTP communication. To enable this format, set the Format attribute on the HTTP tab in the Administration Console to `Extended`. (See "Creating Custom Field Identifiers" on page 5-12).

You specify what information should be recorded in the log file with directives, included in the actual log file itself. A directive begins on a new line and starts with a # sign. If the log file does not exist, a new log file is created with default directives. However, if the log file already exists when the server starts, it must contain legal directives at the head of the file.

Creating the Fields Directive

The first line of your log file must contain a directive stating the version number of the log file format. You must also include a `Fields` directive near the beginning of the file:

```
#Version: 1.0
#Fields: xxxx xxxx xxxx ...
```

Where each `xxxx` describes the data fields to be recorded. Field types are specified as either simple identifiers, or may take a prefix-identifier format, as defined in the W3C specification. Here is an example:

```
#Fields: date time cs-method cs-uri
```

This identifier instructs the server to record the date and time of the transaction, the request method that the client used, and the URI of the request for each HTTP access. Each field is separated by white space, and each record is written to a new line, appended to the log file.

Note: The `#Fields` directive must be followed by a new line in the log file, so that the first log message is not appended to the same line.

Supported Field identifiers

The following identifiers are supported, and do not require a prefix.

`date`
: Date at which transaction completed, field has type <date>, as defined in the W3C specification.

`time`
: Time at which transaction completed, field has type <time>, as defined in the W3C specification.

`time-taken`
: Time taken for transaction to complete in seconds, field has type <fixed>, as defined in the W3C specification.

`bytes`
: Number of bytes transferred, field has type <integer>.

Note that the `cached` field defined in the W3C specification is not supported in WebLogic Server.

The following identifiers require prefixes, and cannot be used alone. The supported prefix combinations are explained individually.

IP address related fields:

These fields give the IP address and port of either the requesting client, or the responding server. These fields have type <address>, as defined in the W3C specification. The supported prefixes are:

`c-ip`
: The IP address of the client.

`s-ip`
: The IP address of the server.

DNS related fields

These fields give the domain names of the client or the server and have type <name>, as defined in the W3C specification. The supported prefixes are:

`c-dns`
: The domain name of the requesting client.

`s-dns`
: The domain name of the requested server.

`sc-status`
: Status code of the response, for example (404) indicating a “File not found” status. This field has type <integer>, as defined in the W3C specification.

`sc-comment`
: The comment returned with status code, for instance “File not found”. This field has type <text>.

`cs-method`
: The request method, for example GET or POST. This field has type <name>, as defined in the W3C specification.

`cs-uri`
: The full requested URI. This field has type <uri>, as defined in the W3C specification.

`cs-uri-stem`
: Only the stem portion of URI (omitting query). This field has type <uri>, as defined in the W3C specification.

`cs-uri-query`
: Only the query portion of the URI. This field has type <uri>, as defined in the W3C specification.

Creating Custom Field Identifiers

You can also create user-defined fields for inclusion in an HTTP access log file that uses the extended log format. To create a custom field you identify the field in the ELF log file using the `Fields` directive and then you create a matching Java class that generates the desired output. You can create a separate Java class for each field, or the Java class can output multiple fields. For a sample of the Java source for such a class, see "Java Class for Creating a Custom ELF Field" on page 5-16.

To create a custom field:

1. Include the field name in the `Fields` directive, using the form:

   `x-`*`myCustomField`*`.`

   Where *`myCustomField`* is a fully-qualified class name.

   For more information on the `Fields` directive, see "Creating the Fields Directive" on page 5-10.

2. Create a Java class with the same fully-qualified class name as the custom field you defined with the `Fields` directive (for example `myCustomField`). This class defines the information you want logged in your custom field. The Java class must implement the following interface:

   `weblogic.servlet.logging.CustomELFLogger`

   In your Java class, you must implement the `logField()` method, which takes a `HttpAccountingInfo` object and `FormatStringBuffer` object as its arguments:

   – Use the `HttpAccountingInfo` object to access HTTP request and response data that you can output in your custom field. Getter methods are provided to access this information. For a complete listing of these get methods, see "Get Methods of the HttpAccountingInfo Object" on page 5-13.

   – Use the `FormatStringBuffer` class to create the contents of your custom field. Methods are provided to create suitable output. For more information on these methods, see the Javadocs for FormatStringBuffer.

3. Compile the Java class and add the class to the `CLASSPATH` statement used to start WebLogic Server. You will probably need to modify the `CLASSPATH` statements in the scripts that you use to start WebLogic Server.

   Note: Do not place this class inside of a Web Application or Enterprise Application in exploded or jar format.

4. Configure WebLogic Server to use the extended log format. For more information, see "Setting Up HTTP Access Logs by Using Extended Log Format" on page 5-9.

Note: When writing the Java class that defines your custom field, do not execute any code that is likely to slow down the system (For instance, accessing a DBMS or executing significant I/O or networking calls.) Remember, an HTTP access log file entry is created for *every* HTTP request.

Note: If you want to output more than one field, delimit the fields with a tab character. For more information on delimiting fields and other ELF formatting issues, see Extended Log Format.

Get Methods of the HttpAccountingInfo Object

The following methods return various data regarding the HTTP request. These methods are similar to various methods of `javax.servlet.ServletRequest`, `javax.servlet.http.Http.ServletRequest`, and `javax.servlet.http.HttpServletResponse`.

For details on these methods see the corresponding methods in the Java interfaces listed in the following table, or refer to the specific information contained in the table.

Table 5-2 Getter Methods of HttpAccountingInfo

HttpAccountingInfo Methods	Method Information
`Object getAttribute(String name);`	javax.servlet.ServletRequest
`Enumeration getAttributeNames();`	javax.servlet.ServletRequest
`String getCharacterEncoding();`	javax.servlet.ServletRequest
`int getResponseContentLength();`	javax.servlet.ServletResponse. setContentLength() This method *gets* the content length of the response, as set with the `set`ContentLength() method.
`String getContentType();`	javax.servlet.ServletRequest

Table 5-2 Getter Methods of HttpAccountingInfo

HttpAccountingInfo Methods	Method Information
`Locale getLocale();`	javax.servlet.ServletRequest
`Enumeration getLocales();`	javax.servlet.ServletRequest
`String getParameter(String name);`	javax.servlet.ServletRequest
`Enumeration getParameterNames();`	javax.servlet.ServletRequest
`String[] getParameterValues(String name);`	javax.servlet.ServletRequest
`String getProtocol();`	javax.servlet.ServletRequest
`String getRemoteAddr();`	javax.servlet.ServletRequest
`String getRemoteHost();`	javax.servlet.ServletRequest
`String getScheme();`	javax.servlet.ServletRequest
`String getServerName();`	javax.servlet.ServletRequest
`int getServerPort();`	javax.servlet.ServletRequest
`boolean isSecure();`	javax.servlet.ServletRequest
`String getAuthType();`	javax.servlet.http.HttpServletRequest
`String getContextPath();`	javax.servlet.http.HttpServletRequest
`Cookie[] getCookies();`	javax.servlet.http.HttpServletRequest
`long getDateHeader(String name);`	javax.servlet.http.HttpServletRequest
`String getHeader(String name);`	javax.servlet.http.HttpServletRequest
`Enumeration getHeaderNames();`	javax.servlet.http.HttpServletRequest
`Enumeration getHeaders(String name);`	javax.servlet.http.HttpServletRequest
`int getIntHeader(String name);`	javax.servlet.http.HttpServletRequest
`String getMethod();`	javax.servlet.http.HttpServletRequest
`String getPathInfo();`	javax.servlet.http.HttpServletRequest

Table 5-2 Getter Methods of HttpAccountingInfo

HttpAccountingInfo Methods	Method Information
`String getPathTranslated();`	javax.servlet.http.HttpServletRequest
`String getQueryString();`	javax.servlet.http.HttpServletRequest
`String getRemoteUser();`	javax.servlet.http.HttpServletRequest
`String getRequestURI();`	javax.servlet.http.HttpServletRequest
`String getRequestedSessionId();`	javax.servlet.http.HttpServletRequest
`String getServletPath();`	javax.servlet.http.HttpServletRequest
`Principal getUserPrincipal();`	javax.servlet.http.HttpServletRequest
`boolean isRequestedSessionIdFromCookie();`	javax.servlet.http.HttpServletRequest
`boolean isRequestedSessionIdFromURL();`	javax.servlet.http.HttpServletRequest
`boolean isRequestedSessionIdFromUrl();`	javax.servlet.http.HttpServletRequest
`boolean isRequestedSessionIdValid();`	javax.servlet.http.HttpServletRequest
`byte[] getURIAsBytes();`	Returns the URI of the HTTP request as byte array, for example: If `GET /index.html HTTP/1.0` is the first line of an HTTP Request, `/index.html` is returned as an array of bytes.
`long getInvokeTime();`	Returns the length of time it took for the service method of a servlet to write data back to the client.
`int getResponseStatusCode();`	javax.servlet.http.HttpServletResponse
`String getResponseHeader(String name);`	javax.servlet.http.HttpServletResponse

Listing 5-1 Java Class for Creating a Custom ELF Field

```
import weblogic.servlet.logging.CustomELFLogger;
import weblogic.servlet.logging.FormatStringBuffer;
import weblogic.servlet.logging.HttpAccountingInfo;

/* This example outputs the User-Agent field into a
 custom field called MyCustomField
*/

public class MyCustomField implements CustomELFLogger{

public void logField(HttpAccountingInfo metrics,
  FormatStringBuffer buff) {
  buff.appendValueOrDash(metrics.getHeader("User-Agent"));
  }
}
```

Preventing POST Denial-of-Service Attacks

A Denial-of-Service attack is a malicious attempt to overload a server with phony requests. One common type of attack is to send huge amounts of data in an HTTP `POST` method. You can set three attributes in WebLogic Server that help prevent this type of attack. These attributes are set in the Console, under *Servers* or *Virtual Hosts*. If you define these attributes for a virtual host, the values set for the virtual host override those set under *Servers*.

`PostTimeoutSecs`
: Amount of time that WebLogic Server waits between receiving chunks of data in an HTTP POST.

  The default value for `PostTimeoutSecs` is 30.

`MaxPostTimeSecs`
: Maximum time that WebLogic Server spends receiving post data. If this limit is triggered, a `PostTimeoutException` is thrown and the following message is sent to the server log:

  `Post time exceeded MaxPostTimeSecs.`

  The default value for `MaxPostTimeSecs` is 30.

`MaxPostSize`
: Maximum number of bytes of data received in a POST from a single request. If this limit is triggered, a `MaxPostSizeExceeded` exception is thrown and the following message is sent to the server log:

  ```
  POST size exceeded the parameter MaxPostSize.
  ```

  An HTTP error code 413 (Request Entity Too Large) is sent back to the client.

  If the client is in listening mode, it gets these messages. If the client is not in listening mode, the connection is broken.

  The default value for `MaxPostSize` is -1.

Setting Up WebLogic Server for HTTP Tunneling

HTTP tunneling provides a way to simulate a stateful socket connection between WebLogic Server and a Java client when your only option is to use the HTTP protocol. It is generally used to *tunnel* through an HTTP port in a security firewall. HTTP is a stateless protocol, but WebLogic Server provides tunneling functionality to make the connection appear to be a regular T3Connection. However, you can expect some performance loss in comparison to a normal socket connection.

Configuring the HTTP Tunneling Connection

Under the HTTP protocol, a client may only make a request, and then accept a reply from a server. The server may not voluntarily communicate with the client, and the protocol is stateless, meaning that a continuous two-way connection is not possible.

WebLogic HTTP tunneling simulates a T3Connection via the HTTP protocol, overcoming these limitations. There are two attributes that you can configure in the Administration Console to tune a tunneled connection for performance. It is advised that you leave them at their default settings unless you experience connection problems. These properties are used by the server to determine whether the client connection is still valid, or whether the client is still alive.

`Enable Tunneling`
: Enables or disables HTTP tunneling. HTTP tunneling is disabled by default.

  Note that the server must also support both the HTTP and T3 protocols in order to use HTTP tunneling.

`Tunneling Client Ping`
: When an HTTP tunnel connection is set up, the client automatically sends a request to the server, so that the server may volunteer a response to the client. The client may also include instructions in a request, but this behavior happens regardless of whether the client application needs to communicate with the server. If the server does not respond (as part of the application code) to the client request within the number of seconds set in this attribute, it does so anyway. The client accepts the response and automatically sends another request immediately.

  Default is 45 seconds; valid range is 20 to 900 seconds.

`Tunneling Client Timeout`
: If the number of seconds set in this attribute have elapsed since the client last sent a request to the server (in response to a reply), then the server regards the client as dead, and terminates the HTTP tunnel connection. The server checks the elapsed time at the interval specified by this attribute, when it would otherwise respond to the client's request.

  Default is 40 seconds; valid range is 10 to 900 seconds.

Connecting to WebLogic Server from the Client

When your client requests a connection with WebLogic Server, all you need to do in order to use HTTP tunneling is specify the HTTP protocol in the URL. For example:

```
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "http://wlhost:80");
Context ctx = new InitialContext(env);
```

On the client side, a special tag is appended to the `http` protocol, so that WebLogic Server knows this is a tunneling connection, instead of a regular HTTP request. Your application code does not need to do any extra work to make this happen.

The client must specify the port in the URL, even if the port is 80. You can set up your WebLogic Server instance to listen for HTTP requests on any port, although the most common choice is port 80 since requests to port 80 are customarily allowed through a firewall.

You specify the listen port for WebLogic Server in the Administration Console under the "Servers" node, under the "Network" tab.

Using Native I/O for Serving Static Files (Windows Only)

When running WebLogic Server on Windows NT/2000/XP you can specify that WebLogic Server use the native operating system call TransmitFile instead of using Java methods to serve

static files such as HTML files, text files, and image files. Using native I/O can provide performance improvements when serving larger static files.

To use native I/O, add two parameters to the web.xml deployment descriptor of a Web Application containing the files to be served using native I/O. The first parameter, `weblogic.http.nativeIOEnabled` should be set to TRUE to enable native I/O file serving. The second parameter, `weblogic.http.minimumNativeFileSize` sets the minimum file size for using native I/O. If the file being served is larger than this value, native I/O is used. If you do not specify this parameter, a value of 4K is used.

Generally, native I/O provides greater performance gains when serving larger files; however, as the load on the machine running WebLogic Server increases, these gains diminish. You may need to experiment to find the correct value for `weblogic.http.minimumNativeFileSize`.

The following example shows the complete entries that should be added to the web.xml deployment descriptor. These entries must be placed in the web.xml file after the `<distributable>` element and before `<servlet>` element.

```
<context-param>
  <param-name>weblogic.http.nativeIOEnabled</param-name>
  <param-value>TRUE</param-value>
</context-param>
<context-param>
  <param-name>weblogic.http.minimumNativeFileSize</param-name>
  <param-value>500</param-value>
</context-param>
```

`weblogic.http.nativeIOEnabled` can also be set as a context parameter in the FileServlet.

CHAPTER 6

Using the WebLogic Persistent Store

The following sections explain how to configure and monitor the persistent store, which provides a built-in, high-performance storage solution for WebLogic Server subsystems and services that require persistence.

Overview of the Persistent Store

The persistent store provides a built-in, high-performance storage solution for WebLogic Server subsystems and services that require persistence. For example, it can store persistent JMS messages or temporarily store messages sent using the Store-and-Forward feature. The persistent store supports persistence to a file-based store or to a JDBC-enabled database.

Table 6-1 defines many of the WebLogic services and subsystems that can create connections to the persistent store. Each subsystem that uses the persistent store specifies a unique connection ID that identifies that subsystem.

Table 6-1 Persistent Store Users

Subsystem/Service	What It Stores	More Information
Diagnostic Service	Log records, data events, and harvested metrics.	Understanding WLDF Configuration in *Configuring and Using the WebLogic Diagnostic Framework*
JMS Messages	Persistent messages and durable subscribers.	Understanding the Messaging Models in *Programming WebLogic JMS*
JTA Transaction Log (TLOG)	Information about committed transactions coordinated by the server that may not have been completed.	Managing Transactions in *Programming WebLogic JTA*
Path Service	The mapping of a group of messages to a messaging resource.	Using the WebLogic Path Service in *Configuring and Managing WebLogic JMS*
Store-and-Forward (SAF) Service Agents	Messages for a sending SAF agent for retransmission to a receiving SAF agent	Understanding the Store-and-Forward Service *in Configuring and Managing WebLogic Store-and-Forward*
Web Services	Request and response SOAP messages from an invocation of a reliable WebLogic Web Service.	Using Reliable SOAP Messaging in *WebLogic Web Services: Advanced Programming*
EJB Timer Services	EJB Timer objects.	Understanding Enterprise JavaBeans in *Programming WebLogic Enterprise JavaBeans*

For more information about the store connection IDs, see "Monitoring Store Connections" on page 6-22.

Features of the Persistent Store

The key features of the persistent store include:

- Default file store for each server instance that requires no configuration.
- One store is shareable by multiple subsystems, as long as they are all targeted to the same server instance or migratable target.
- High-performance throughput and transactional support.
- Modifiable parameters that let you create customized file stores and JDBC stores.
- Monitoring capabilities for persistent store statistics and open store connections.
- In a clustered environment, a customized store can be migrated from an unhealthy server to a backup server, either on the whole-server level or on the service-level.

High-Performance Throughput and Transactional Support

Throughput is the main performance goal of the persistent store. Multiple subsystems can share the same persistent store, as long as they are all targeted to the same server instance or migratable target.

This is a performance advantage because the persistent store is treated as a single resource by the transaction manager for a particular transaction, even if the transaction involves multiple services that use the same store. For example, if JMS and EJB timers share a store, and a JMS message and an EJB timer are created in a single transaction, the transaction will be one-phase and incur a single resource write, rather than two-phase, which incurs four resource writes (two on each resource), plus a transaction entry write (on the transaction log).

Both the file store and the JDBC store can survive a process crash or hardware power failure without losing any committed updates. Uncommitted updates may be retained or lost, but in no case will a transaction be left partially complete after a crash.

Comparing File Stores and JDBC Stores

The following are some similarities and differences between file stores and JDBC stores:

- The default persistent store can only be a file store. Therefore, a JDBC store cannot be used as a default persistent store.
- The transaction log (TLOG) can only be stored in a default store.

- Both have the same transaction semantics and guarantees. As with JDBC store writes, file store writes are guaranteed to be persisted to disk and are not simply left in an intermediate (that is, unsafe) cache.
- Both have the same application interface (no difference in application code).
- All things being equal, file stores generally offer better throughput than a JDBC store.

  Note: If a database is running on high-end hardware with very fast disks, and WebLogic Server is running on slower hardware or with slower disks, then you may get better performance from the JDBC store.

- File stores are generally easier to configure and administer, and do not require that WebLogic subsystems depend on any external component.
- File stores generate no network traffic; whereas, JDBC stores will generate network traffic if the database is on a different machine from WebLogic Server.
- JDBC stores may make it easier to handle failure recovery since the JDBC interface can access the database from any machine on the same network. With the file store, the disk must be shared or migrated.

Securing File Store Data

In order to properly secure file store data, you must set appropriate directory permissions on all your file store directories. If you require data encryption, you must use appropriate third-party encryption software.

High Availability For Persistent Stores

For high availability, a persistent file-based store (default or custom) can be migrated along with its parent server as part of the "whole server-level" migration feature, which provides both automatic and manual migration at the server-level, rather than on the service level. For more information, see Server Migration section of *Using WebLogic Server Clusters*. However, file-based stores must be configured on a shared disk that is available to the migratable target servers in the cluster.

Persistent Store Migration

File-based stores *and* JDBC-accessible stores can also be migrated as part of a "service-level" migration for JMS-related services, such as stores, JMS servers, SAF agents, and the path service, which rely on stores to maintain data. Service-level migration is controlled by a *migratable*

target, which serves as a grouping of JMS-related services, and which is hosted on only one physical server in a cluster. JMS services hosted by a migratable target can be manually migrated, either in response to a server failure or as part of regularly scheduled server maintenance. When the migratable target is migrated, all pinned services hosted by that target are also migrated. For more information on service-level migration, see Service-level Migration in *Using WebLogic Server Clusters*.

Migratable JMS-related services cannot use the default file store, so you must configure a custom file store or JDBC store and target it to the same migratable target as the JMS server, SAF agent, or path service associated with the store.

Tip: As a best practice, a path service should use its own custom store and migratable target.

Migratable file stores must also either be configured on a shared disk that is available to the migratable targets in the cluster or you can use pre/post-migration scripts to migrate a file store's data to a backup server. See Custom Store Availability for JMS Services in *Configuring WebLogic Server Environments*.

High Availability Storage Solutions

If you have applications that need access to persistent stores that reside on remote machines after the migration of a JMS server or JTA transaction log, then you should implement one of the following highly-available storage solutions:

- File-based stores (default or custom) — Implement a hardware solution, such as a dual-ported SCSI disk or Storage Area Network (SAN) to make a file store available from shareable disks or remote machines.

  Note: If a file store is disconnected and re-connected again, its host server instance must be rebooted to successfully continue sending/receiving persistent JMS messages. For example, if for some reason the file system containing a file store is unmounted and then remounted, attempts to send persistent JMS messages will generate JMS exceptions until the host server is rebooted.

- JDBC-accessible stores — Configure a JDBC store and use JDBC to access this store, which can be on yet another server. Applications can then take advantage of any high-availability or failover solutions offered by your database vendor. In addition, JDBC stores support multi data sources, which provide failover between nodes of a highly available database system, such as redundant databases or Oracle Real Application Clusters (RAC). For more information about using JDBC multi data sources, see Configuring JDBC Multi Data Sources in *Configuring and Managing WebLogic JDBC*.

- Any persistent store — Use high-availability clustering software such as VERITAS™ Cluster Server, which provides an integrated, out-of-the-box solution for BEA WebLogic Server-based applications. Some other recommended high-availability software solutions include SunCluster, IBM HACMP, or the equivalent.

Using the Default Persistent Store

Each server instance, including the administration server, has a default persistent store that requires no configuration. The default store is a file-based store that maintains its data in a group of files in a server instance's `data\store\default` directory. A directory for the default store is automatically created if one does not already exist. This default store is available to subsystems that do not require explicit selection of a particular store and function best by using the system's default storage mechanism. For example, a JMS Server with no persistent store configured will use the default store for its Managed Server and will support persistent messaging.

The default store can be configured by directly manipulating the `DefaultFileStoreMBean` parameters. If this MBean is not defined in the domain's configuration file, then the configuration subsystem ensures that the `DefaultFileStoreMBean` always exists with the default values.

Also, the Administration Console enables you to change the default store parameters, such as its default directory location and Synchronous Write Policy, as described in Modify the Default Store Settings in the *Administration Console Online Help*.

In addition to using the default file store, you can also configure a custom file store or JDBC store to suit your specific needs, as explained in "Using Custom File Stores and JDBC Stores" on page 6-7. One exception, however, is the JTA transaction log (TLOG), which always uses the default store. This is because the transaction log must always be available early in the server boot process. For more information about the TLOG, see Transaction Log Files in *Programming WebLogic JTA*.

Default Store Location

The default store maintains its data in a `data\store\default` directory inside the *`servername`* subdirectory of a domain's root directory

For example, if no directory name is specified for the default file store, it defaults to:

`bea_home``\user_projects\domains\`*`domain-name`*`\servers\`*`server-name`*`\data\store\default`

where *domainname* is the root directory of your domain, typically `c:\bea\user_projects\domains\`*domainname*, which is parallel to the directory in which WebLogic Server program files are stored, typically `c:\bea\wlserver_10.0`.

You can, however, specify another location for the default store by directly manipulating the `DefaultFileStoreMBean` parameters or by using the Administration Console, as described in Modify the Default Store Settings in the *Administration Console Online Help*.

Example of a Default File Store

Here's an example of how a default file store may look in a domain's configuration file, with the default directory location and Synchronous Write Policy settings overridden:

```
<server>
  <name>myserver</name>
  <default-file-store>
  <directory>C:/store</directory>
  <synchronous-write-policy>Disabled</synchronous-write-policy>
  </default-file-store>
</server>
```

Using Custom File Stores and JDBC Stores

In addition to using the default file store, you can also configure a file store or JDBC store to suit your specific needs. A custom file store, like the default file store, maintains its data in a group of files in a directory. However, you may want to create a custom file store so that the file store's data is persisted to a particular storage device or when you want a JMS service that accesses a file store to be able to migrate with the store to another server member in a cluster. When configuring a file store directory, the directory must be accessible to the server instance on which the file store is located.

A JDBC store can be configured when a relational database is used for storage. A JDBC store enables you to store persistent messages in a standard JDBC-capable database, which is accessed through a designated JDBC data source. The data is stored in the JDBC store's database table, which has a logical name of `WLStore`. It is up to the database administrator to configure the database for high availability and performance. JDBC stores also support migratable targets for automatic or manual JMS service migration.

For more information about configuring a persistent store, see "When to Use a Custom Persistent Store" on page 6-8.

When to Use a Custom Persistent Store

WebLogic Server provides configuration options for creating a custom file store or JDBC store. For example, you may want to:

- Place a file store's files on a particular device.
- Use a JDBC store rather than a file store for a particular server instance.
- Allow all physical stores in a cluster to share the same logical name.
- Logically separate different services to use different files or tables. (This may simplify administration and maintenance at the expense of reduced performance.)
- Migratable JMS-related services cannot use the default persistent store, so you must configure a custom store and target it to the same migratable target as the migratable JMS service. For more information, see Service-Level Migration in *Using WebLogic Server Clusters*.

Methods of Creating a Persistent Store

A customized persistent store can be configured in the following ways:

- Use the Administration Console to configure a custom file store or JDBC store, as described in Configure Persistent Stores in the *Administration Console Online Help*.
- Directly edit the configuration file (`config.xml`) of the server instance that is hosting the default persistent store.
- Use the WebLogic Java Management Extensions (JMX) to create persistent stores. JMX is the Java EE solution for monitoring and managing resources on a network. For more information see, Developing Custom Management Utilities with JMX.
- Use the WebLogic Scripting Tool (WLST) to create persistent stores. WLST is a command-line scripting interface that you use to interact with and configure WebLogic Server instances and domains. For more information, see WebLogic Scripting Tool.
- Use the WebLogic Configuration Wizard to change the options of the default persistent store. For detailed information on how to use the Configuration Wizard to configure a persistent store, see Creating a New WebLogic Domain.

Modifying Custom Persistent Store Parameters

Modifying certain custom store configuration options, such as a JDBC store's prefix or a file store's directory, do not necessarily require a server restart if you do the following:

1. Set the targets of any dependent services (such as a JMS server that uses the custom store) to null, and then setting the custom store target to null. (Setting a service's target to null implicitly shuts down the service.)
2. Reverse the process by setting the custom store target back to its original value and then setting the dependent resource targets back to their original values.

In cases where the custom store and JMS servers share a migratable target, you can administratively restart the migratable target.

Creating a Custom (User-Defined) File Store

The following sections provide an example of a custom file store and configuration guidelines for choosing a synchronous write policy.

To create a custom file store, you can directly modify the default `FileStoreMBean` parameters. For instructions on using the Administration Console to create a custom file store, see Create File Stores in the *Administration Console Online Help*.

Main Steps for Configuring a Custom File Store

The main steps for creating a custom file store are as follows:

1. Create a directory where the file store's data will be persisted.
2. Create a custom file store and specify the directory location that you created.
3. Associate the custom file store with the subsystem(s) or migratable target that will be accessing it, such as:
   - For JMS servers, select the custom file store on the General Configuration page.
   - For Store-and-Forward agents, select the custom file store on the General Configuration page.
   - For a Path Service, select the custom file store on the General Configuration page.

Example of a Custom File Store

Here's an example of how a custom file store may look in a domain's configuration file with its files kept in a `/disk1/jmslog` directory.

```
<file-store>
  <name>SampleFileStore</name>
  <target>myserver</target>
  <directory>/disk1/jmslog</directory>
</file-store>
```

Table 6-2 briefly describes the file store configuration parameters that can be modified.

Table 6-2 File Store Configuration Options

Option	Required	What it does. . .
Name	Yes	The name of the file store, which must be unique across all stores in the domain.
Targets	Yes	The server instance or migratable target where a file store is targeted. Multiple subsystems can share the same file store, as long as they are all targeted to the same server instance or migratable target. **Note:** When using migratable targets for JMS services, you must target the file store to the same migratable target used by the JMS service. See Service-Level Migration in *Using WebLogic Server Clusters*.
Directory	Yes	The path name to the directory on the file system where the file store is kept. Modifying an existing file store's directory does not necessarily require a server restart, as described in "Modifying Custom Persistent Store Parameters" on page 6-9.
Logical Name	No	Optionally used with subsystems, like EJBs, when deploying a module to an entire cluster, but also require a different physical store on each server instance in the cluster. In such a configuration, each physical store would have its own name, but all the persistent stores would share the same logical name.
Synchronous Write Policy	No	Optionally defines how hard a file store will try to flush records to the disk. Values are: Direct-Write (default), Cache-Flush, and Disabled. For more information, see "Guidelines for Configuring a Synchronous Write Policy" on page 6-11.

For instructions on configuring a custom file store using the Administration Console, see Create file stores in the *Administration Console Online Help*.

Guidelines for Configuring a Synchronous Write Policy

The default Direct-Write policy is supported on most platforms, and this policy generally performs faster than the Cache-Flush option. For a potential performance boost, stores in direct I/O mode will automatically load a native I/O `wlfileio2` driver. This driver is available on Windows, Solaris, HP, AIX, and Linux platforms.

Changing the default policy to Disabled generally improves file store performance, often quite dramatically, but at the expense of possibly losing sent messages or generating duplicate received messages (even if messages are transactional) in the event of an operating system crash or a hardware failure. This is because transactions are complete as soon as their writes are cached in memory, instead of waiting for the writes to successfully reach the disk. Simply shutting down an operating system does not generate these failures, as an OS flushes all outstanding writes during a normal shutdown. Instead, these failures can be emulated by abruptly shutting the power off to a busy server.

Tip: To view a running file store's synchronous write policy and driver, check the `BEA-280050` message in the server log.

Creating a JDBC Store

The following sections provide an example of a JDBC store, and information about creating a database table for a JDBC store, either using existing DDL or It also includes instruction on enabling Oracle blob record columns in a DDL file.

To create a JDBC store, you can directly modify the default `JDBCStoreMBean` parameters. For instructions on using the Administration Console to create a JDBC store, see Create JDBC Stores in the *Administration Console Online Help*.

For configuration guidelines on using prefixes with JDBC stores and recommended JDBC data source settings, see "Guidelines for Configuring a JDBC Store" on page 6-18.

Main Steps for Configuring a JDBC Store

The main steps for creating a JDBC store are as follows:

1. Create a JDBC data source or multi data source to interface with the JDBC store.

2. Create a JDBC store and associate it with the JDBC data source or multi data source.
3. It is highly recommended that you configure the Prefix option to a unique value for each configured JDBC store table.
4. Associate the JDBC store with the subsystem(s) that will be using it, such as:
   - For JMS servers, select the JDBC store on the General Configuration page.
   - For Store-and-Forward agents, select the JDBC store on the General Configuration page.
   - For a Path Service, select the custom file store on the General Configuration page.

Example of a JDBC Store

Here's an example of how a JDBC store may look in the configuration file, using the JDBC data source `MyDataSource`, and with a logical name specified:

```
<jdbc-store>
  <name>SampleJDBCStore</name>
  <target>yourserver</target>
  <data-source>MyDataSource</data-source>
  <logical-name>Baz</logical-name>
</jdbc-store>
```

Table 6-3 describes the JDBC store configuration parameters that can be modified.

Table 6-3 JDBC Store Configuration Option

Option	Required	What it does. . .
Name	Yes	The name of the JDBC store, which must be unique across all stores in the domain.
Targets	Yes	The server instance or migratable target where a JDBC store is targeted. Multiple subsystems can share the same JDBC store, as long as they are all targeted to the same server instance or migratable target. **Note:** When using migratable targets for JMS services, you must target the JDBC store to the same migratable target used by the JMS service. See Service-Level Migration in Using WebLogic Server Clusters.

Table 6-3 JDBC Store Configuration Option

Option	Required	What it does. . .
Data Source	Yes	The JDBC data source or multi data source used by this JDBC store to access the store's database table (`WLStore`). This data source or multi data source must be targeted to the same server instance as the JDBC store. **Note:** You cannot specify a JDBC data source that is configured to support global transactions. Therefore, the specified JDBC data source must use a non-XA JDBC driver. In addition, you cannot enable Logging Last Resource or Emulate Two-Phase Commit in the data source. This limitation does not remove the XA capabilities of layered subsystems that use JDBC stores. For example, WebLogic JMS is fully XA-capable regardless of whether it uses a file store or any JDBC store.
Prefix Name	No	The prefix for the JDBC store's table is generally entered in the following format: `[[[catalog.]schema.]prefix]` When using multiple JDBC stores, it is required to set this option to a unique value for each configured JDBC store. When no prefix is specified, the JDBC store table name is simply `WLStore` and the database implicitly determines the schema according the current user of the JDBC connection. Also, two JDBC stores cannot share the same database table. For more information, see "Using Prefixes with a JDBC Store" on page 6-18. Modifying an existing JDBC store's prefix does not necessarily require a server restart, as described in "Modifying Custom Persistent Store Parameters" on page 6-9.
Logical Name	No	Optionally used with WebLogic Server subsystems, like EJBs, when deploying a module to an entire cluster, but also require a different physical store on each server instance in the cluster. In such a configuration, each physical store would have its own name, but all the persistent stores would share the same logical name.
Create Table from DDL File	No	Optionally used with supported DDL (data definition language) files to create the JDBC store's database table (`WLStore`). This option is ignored when the JDBC store's database table already exists. For more information, see "Automatically Creating a JDBC Store Table Using Default and Custom DDL Files" on page 6-15.

For instructions on configuring a JDBC store using the Administration Console, see "Create JDBC stores" in the *Administration Console Online Help*.]

Supported JDBC Drivers

When using a JDBC store, the backing database can be any database that is accessible through a JDBC driver. WebLogic Server detects some drivers for supported databases.

For each of these databases, there are corresponding DDL (data definition language) files within the *WL_HOME*`\server\lib\weblogic.jar` file, in the `weblogic/store/io/jdbc/ddl` directory, where *WL_HOME* is the top-level directory of your WebLogic Server installation.

Table 6-4 Supported JDBC Drivers and Corresponding DDL Files

Database	DDL Files
IBM DB2	`db2.ddl` `db2v6.ddl`
Informix	`informix.ddl`
Microsoft SQL (MSSQL) Server	`msssql.ddl`
MySQL	`mysql.ddl`
Oracle	`oracle.ddl` `oracle_blob.ddl`
Pointbase	`pointbase.ddl`
Sybase	`sysbase.ddl`

The DDL files are actually text files containing the SQL commands (terminated by semicolons) that create the JDBC store's database table (`WLStore`). Therefore, if you are using a database that is not included in this list, you can copy and edit any one of the existing DDL files to suit your specific database, as described in "Creating a JDBC Store Table Using a Custom DDL File" on page 6-15.

Automatically Creating a JDBC Store Table Using Default and Custom DDL Files

The JDBC Store Configuration page provides an optional Create Table from DDL File option, through which you can access a pre-configured DDL file that is used to create the JDBC store's backing table (`WLStore`). This option is ignored when the JDBC store's backing table already exists. It is mainly used to specify a custom DDL file created for an unsupported database, or when upgrading JMS JDBC store tables from a prior release to a current JDBC Store table.

If a DDL file name is *not* specified in the Create Table from DDL File field, and the JDBC store detects that its backing table does not already exist, the JDBC store automatically creates the table by executing a pre-configured DDL file that is specific to the database vendor (see Supported JDBC Drivers).

If a DDL file name is specified in the Create Table from DDL File field, and the JDBC store detects that its backing table does not already exist, the JDBC store searches for the specified DDL file in the file path first, and then, if not found, searches for the DDL file in the `CLASSPATH`. Once found, the SQL within the DDL file is executed to create the JDBC store's backing table. If the configured file is not found and the table doesn't already exist, the JDBC store will fail to boot.

Creating a JDBC Store Table Using a Custom DDL File

To use a different database from those listed in "Supported JDBC Drivers" on page 6-14, you can copy and edit any one of the existing DDL template files to suit your specific database.

1. Use the JAR utility supplied with the JDK to extract the DDL files to the `/weblogic/store/io/jdbc/ddl` directory using the following command:

   ```
   jar xf weblogic.jar /weblogic/store/io/jdbc/ddl
   ```

   Note: If you omit the `weblogic/store/io/jdbc/ddl` parameter, the entire jar file is extracted.

2. Edit the DDL file for your database. An SQL command can span several lines and is terminated with a semicolon (;). Lines beginning with pound signs (#) are comments.

3. Save your changes and rename the new DDL appropriately (for example, `mydatabase.ddl`)

4. Create a JDBC store, as explained in Create JDBC Stores in the *Administration Console Online Help*.

5. Use the Create Table from DDL File option on the General Configuration page to specify your custom DDL file (for example, `/mydatabase.ddl`

   Note: On Windows systems, for full path names always include the drive letter.

Enabling Oracle BLOB Record Columns

For Oracle databases, you can use the `oracle_blob.ddl` file to create a JDBC store table with a BLOB record column type rather than the default LONG RAW record column type. The `oracle_blob.ddl` file is pre-configured and supplied in the WebLogic `CLASSPATH`, as described in "Supported JDBC Drivers" on page 6-14.

To use the Oracle BLOB DDL with a JDBC store:

1. Shut down the server instance that uses the JDBC store.
2. Delete the current JDBC table, as explained in "Managing JDBC Store Tables" on page 6-16.
3. Reboot the server instance.
4. Create a new JDBC store, as explained in Create JDBC Stores in the *Administration Console Online Help*.
5. In the Create Table from DDL File field on the General Configuration page, enter the location of the `oracle_blob.ddl` file, as follows: `/oracle_blob.ddl`
6. Click Finish to create the JDBC store's backing table.

If you need to preserve data already in a Oracle LONG RAW column, but still want to switch the column to BLOB, *do not* use this method. Instead, consult the Oracle documentation for the SQL ALTER TABLE command.

Managing JDBC Store Tables

The JDBC `utils.Schema` utility allows you to regenerate a new JDBC store database table (`WLStore`) by deleting the existing version. Running this utility is usually not necessary, since WebLogic Server automatically creates this table for you. However, if your existing JDBC store database table somehow becomes corrupted, you can delete it using the `utils.Schema` utility.

The `utils.Schema` utility is a Java program that takes command-line arguments to specify the following:

- JDBC driver
- Database connection information

- Name of a file containing the SQL Data Definition Language (DDL) commands that create the database table

Using the utils.Schema Utility to Delete a JDBC Store Table

Enter the `utils.Schema` command, as follows:

```
$ java utils.Schema url JDBC_driver [options] DDL_file
```

Note: To execute `utils.Schema`, your `CLASSPATH` must contain the `weblogic.jar` file.

Table 6-5 lists the `utils.Schema` command-line arguments.

Table 6-5 Command-line arguments for utils.Schema

Argument	Description
url	Database connection URL. This value must be a colon-separated URL as defined by the JDBC specification.
JDBC_driver	Full package name of the JDBC Driver class.
options	Optional command options. If required by the database, you can specify: • The user name and password as follows: `-u <username> -p <password>` • The Domain Name Server (DNS) name of the JDBC database server as follows: `-s <dbserver>` You can also specify the `-verbose` option, which causes `utils.Schema` to echo SQL commands as they are executed.
DDL_file	The full pathname of the DDL text file containing the SQL commands that you want to execute. For more information, see "Supported JDBC Drivers" on page 6-14.

For example, the following command deletes a JDBC table named `MYWLStore` in an Oracle server named `DEMO`, with the user name `user1` and password `foobar`:

```
$ echo "drop MYWLStore;" > drop.ddl

$ java utils.Schema
jdbc:weblogic:oracle:DEMO \
```

```
weblogic.jdbc.oci.Driver -u user1 -p foobar -verbose \
drop.ddl
```

Guidelines for Configuring a JDBC Store

The following sections provide guidelines for using JDBC store prefixes, recommended WebLogic JDBC data source settings for JDBC stores, and handling JMS transactions with JDBC stores.

Using Prefixes with a JDBC Store

The JDBC store database contains a database table, named `WLStore`, that is generated automatically and is used internally by WebLogic Server. The JDBC store provides an optional Prefix Name parameter, which can be used to provide more precise access to the database table.

It is always a best practice to configure a prefix for the JDBC `WLStore` table name, especially when:

- The database requires fully-qualified names. (You should verify this with your database administrator.)
- There is more than one JDBC store instance sharing a database, since no two JDBC stores can share the same table.
- There are many tables in the database. Setting the prefix reduces the number of tables the JDBC store must search through to find its table during boot.

JDBC Store Table Rules

To avoid potential data loss, follow these rules:

- Each JDBC store table name must be unique.
- If multiple JDBC stores share a table, the behavior is undefined and data loss is likely.
- There is no procedure for combining two database tables into a single table.

Prefix Name Format Guidelines

For most databases, the Prefix Name option for the JDBC store's backing database table should be set in the following format for each configured JDBC store, which will result in a valid table name when prepended to the JDBC store table name:

```
[[[catalog.]schema.]prefix]
```

Note that each period in the `[[[catalog.]schema.]prefix]` format is significant. Generally, *`catalog`* identifies the set of system tables being referenced by the DBMS, and *`schema`* generally corresponds to ID of the table owner (*`username`*). When no prefix is specified, the JDBC store table name is simply `WLStore` and the database implicitly determines the schema according the current user of the JDBC connection.

For example, in a production database, the database administrator could maintain a unique table for the Sales department, as follows:

```
[[[Production.]JMSAdmin.]Sales]
```

The resulting table will be created in the Production catalog, under the JMSAdmin schema, and will be named `SalesWLStore`.

For some DBMS vendors, such as Oracle, there is no catalog to set or choose, so the format simplifies to `[[schema.]prefix]`. For more information, refer to your DBMS documentation for instructions on fully-qualified table names, but note that the syntax specified by the DBMS may differ from the format required for this option.

Caution: If the Prefix Name setting is changed, but the `WLStore` database table already exists in the database, take care to preserve existing table data. In this case, the existing database table must be renamed by a database administrator to match the new configured table name.

Recommended JDBC Data Source Settings for JDBC Stores

The following settings are recommended when you use a JDBC data source or multi data source for JDBC stores.

Automatic Reconnection to Failed Databases

WebLogic Server provides robust JDBC data sources that can automatically reconnect to failed databases after they come back online, without requiring you to restart WebLogic Server. To take advantage of this capability, and make your use of JDBC stores more robust, configure the following options on the JDBC data source associated with the JDBC store:

```
TestConnectionsOnReserve="true"
TestTableName="SYSTABLES"
ConnectionCreationRetryFrequencySeconds="600"
```

For more information about JDBC default Test Table Names, see Connection Testing Options for a Data Source in the *Configuring and Managing WebLogic JDBC*. For more information about setting the number of database reconnection attempts, see the Enabling Connection Creation Retries section in *Configuring and Managing WebLogic JDBC*.

Required Setting for WebLogic Type 4 JDBC DB2 Drivers

For data sources used as a JDBC store that use the WebLogic Type 4 JDBC driver for DB2, the `BatchPerformanceWorkaround` property must be set to "true" due to internal JMS batching requirements.

For more information, see the Performance Workaround for Batch Inserts and Updates section in the *WebLogic Type 4 JDBC Drivers* documentation.

Handling JMS Transactions with JDBC Stores

You cannot configure a JDBC store to use a JDBC data source that is configured to support global transactions. The JDBC store must use a JDBC data source that uses a non-XA JDBC driver. In addition, you cannot enable Logging Last Resource or Emulate Two-Phase Commit in the data source. This limitation does not remove the XA capabilities of layered subsystems that use JDBC stores. For example, WebLogic JMS is fully XA-capable regardless of whether it uses a file store or any JDBC store.

Because the JDBC store implements the XAResource interface, it acts as it's own resource manager and handles the transactions above the JDBC driver level. That is, the store itself implements the `XAResource` and handles the transactions without depending on the database (even when the messages are stored in the database).

This means that whenever you are using a JDBC store and a database (even if it is the same database where the JMS messages are stored), then it is two-phase commit transaction.

For more information about using JMS transactions with a JDBC store, see Using Transactions with WebLogic JMS in *Programming WebLogic JMS*.

From a performance perspective, you may also boost your performance as follows:

- Ensure that the JDBC data source used for the database work exists on the same server instance as the JMS destination—the transaction will still be two-phase, but it will be handled with less network overhead.
- Use file stores rather than JDBC stores.

- Configure multiple services to share the same store if they will commonly be invoked within the same transaction.
- If an application directly performs database operations in addition to invoking store services (such as JMS) within the same transaction, consider using a JDBC data source with Logging Last Resource (LLR) enabled for the database operations.

  With the LLR optimization, the transaction will follow the two-phase commit protocol, but the database operations will be handled in a single local transaction, which may improve overall transaction performance. For more information on using the LLR optimization, see Understanding the Logging Last Resource Transaction Option in *Configuring and Managing WebLogic JDBC*.

Monitoring a Persistent Store

You can monitor statistics for each existing persistent store and for each open store connection.

Monitoring Stores

Each persistent store is represented at runtime by an instance of the `PersistentStoreRuntimeMBean`, which provides the following options.

Table 6-6 Persistent Store Run-time Options

Option	What it does. . .
CreateCount	Number of create requests issued to this persistent store.
ReadCount	Number of read requests issued to this persistent store.
UpdateCount	Number of update requests issued by this persistent store.
DeleteCount	Number of delete requests issued by this persistent store.
ObjectCount	Number of objects contained in the persistent store.
Connections	Number of active connections in the persistent store.
PhysicalWriteCount	Number of times the persistent store flushes its data to durable storage.

Monitoring Store Connections

For each open persistent store connection, the persistent store also registers a `PersistentStoreConnectionRuntimemMBean`, which provides the following options.

Table 6-7 Persistent Store Connection Runtime Options

Option	What it does. . .
CreateCount	Number of create requests issued to this connection.
ReadCount	Number of read requests issued to this connection.
UpdateCount	Number of update requests issued by this connection.
DeleteCount	Number of delete requests issued by this connection.
ObjectCount	Number of objects contained in the connection.

Table 6-8 defines most of the runtime prefix names of the WebLogic services and subsystems that can create a connection to the persistent store.

Table 6-8 Persistent Store Runtime Prefix Names

Subsystem/Service	Runtime Prefix Name
Deployment	`weblogic.deploy.`*`internal`* where *`internal`* is the name of the deployment connection
Diagnostic Service	`weblogic.diagnostics.`*`internal`* where *`internal`* is the logical name of the diagnostic archive connection
EJB Timer Services	`weblogic.ejb.timer.`*`internal`* where *`internal`* uniquely identifies EJB deployments in a server instance
JMS Service	JMS server: `weblogic.messaging.jmsServer.`*`internal`* where *`internal`* is the name of the JMS server connection JMS durable subscriber: `weblogic.messaging.jmsServer.durablesubs.`*`internal`* where *`internal`* is the name of the durable subscriber connection
JTA Transaction Log (TLOG)	`weblogic.transaction.`*`internal`* where *`internal`* is the name of the TLOG connection
Path Service	`weblogic.messaging.PathService.`*`internal`* where *`internal`* is the name of the path service connection
SAF Service	SAF agent `weblogic.messaging.SAFAgent@server1.`*`internal`* where *`internal`* is the name of the SAF agent's connection SAF durable subscriber: `weblogic.messaging.SAFAgent@server1.durablesubs.`*`internal`* where *`internal`* is the name of the durable subscriber connection
Web Services	`weblogic.wsee.server.store.`*`internal`* where *`internal`* is the name of the Web Service's connection

Administering a Persistent Store

The WebLogic Store administration utility enables administrators to troubleshoot a WebLogic persistent store. The store utility operates only on a store that is not currently opened by a running server instance. This utility can be run from a Java command-line or from WLST (WebLogic Scripting Tool), as described in "Store Administration Using a Java Command-line" on page 6-25 and "Store Administration Using WLST" on page 6-27.

The most common uses-cases for store administration are for compacting a file store to reduce its size and for dumping the contents of a file store of JDBC store to an XML file for troubleshooting purposes. Examples of these use cases are provided later in this section.

Table 6-9 defines the available store administration commands for Java and WLST.

Table 6-9 Persistent Store Administration Options

Java Command	WLST Method	What it does. . .
`help`	`helpstore`	Displays available commands, usage, and examples.
`compact`	`compactstore`	Compacts and defragments the space occupied by a file store. This command only works offline and does not work for JDBC stores. **Note:** Compacting a file store is usually not necessary if you know that file store will likely grow to the current size again. File stores automatically re-use space freed by deleted records and expand only when there is insufficient internal space for new records. Also, file stores do not normally become fragmented as most persistent records are short-lived.
`openfile`	`openfilestore`	Opens an existing file store for further operations. If a file store does not exist, a new one is created in an open state using the `-create` parameter.
`openjdbc`	`openjdbcstore`	Opens an existing JDBC store for further operations. If a JDBC store does not exist, a new one is created in an open state
`dump`	`dumpstore`	Dumps store or connection contents in a human-readable format to user-specified XML file. The XML file format is the same format used by the diagnostic image of the persistent store.
`list`	`liststore`	Lists store names, open stores, or connections in a store.

Table 6-9 Persistent Store Administration Options

Java Command	WLST Method	What it does. . .
n/a	getstoreconns	Returns a list of connections in the specified store (for script access)
n/a	getopenstores	Returns a list of opened stores (for script access).
opts	n/a	Lists invocation options for the store administration tool.
verbose	n/a	Controls display of additional information, such as stack traces.
close	closestore	Closes a previously opened store.
quit	exit	Ends the store administration session.

A persistent store can be backed by the file system (file store) or by a JDBC-capable database (JDBC store). Except for the `openfile`/`openfilestore()` and `openjdbc`/`openjdbcstore()` options, there is no difference in the options to operate on these two different types of stores.

Most commands and methods work in terms of store names, while others also work in terms of connection names. Store connections are logical groups of records within persistent stores. For example, the JMS and JTA subsystems persist their respective records in different connections in the same store.

Store Administration Using a Java Command-line

To open the persistent store administration utility from a Java command-line, type the following:

```
> java weblogic.store.Admin
> storeadmin->
```

Accessing Store Administration Help

Type `help` for detailed descriptions on available store administration commands, as well as examples of typical command usage. For example, the following comprehensive help is provided for the `list` command, which lists store names, open stores, or connections in a store.

```
storeadmin->help list
  Command:
    list
```

```
Description:
  lists store names, open stores, or connections in a store
Usage:
  list [-store storename|-dir dir]
Examples:
  list #lists all opened stores by storename
  list -store store1 #lists all connections in store1
  list -dir dir1 #lists all storenames found in dir1
```

Dumping the Contents of a File Store

Here's an example of using a series of store administration commands to ultimately export the contents of a file store named `myfilestore` into a human-readable XML file format in a temporary directory. This does not include store connection names or the actual record contents, which require the optional `-conn` and `-deep` parameters.

```
> storeadmin-> list -dir .
> storeadmin-> openfile -store myfilestore -dir .
> storeadmin-> dump -store myfilestore -out d:\tmp\filestore1-out
> storeadmin-> close -store myfilestore
```

The `list` command shows all the store names in the current directory. The `openfile` and `openjdbc` commands must be used to open and/or create a file or JDBC store first before calling certain administration functions, like `dump` and `list` (only when listing open stores). After administering an open store, you must close it using the `close` command.

Compacting a File Store

Here's an example of using the `compact` command to compact the space occupied by a file store in the `mystores` directory.

```
> storeadmin->compact -dir c:\mystores -tempdir c:\tmp
```

Since the compact command can only be used on an unopened file store, none of the stores that have files in the source `-dir` directory should be open. Also, the temporary `-tempdir` directory should have at least enough extra space as the source directory and should also not be under the source directory. When compact successfully completes, the newly compacted store files will be in the `mystores` directory. In addition, a new, uniquely-named directory will be created under `tmp` containing the original uncompacted store files.

Store Administration Using WLST

The WLST interface has a couple of additional methods (compared to the Java command-line) such as `getopenstores` and `getstoreconns`, that return relevant Java objects and can be used for scripting in WLST.

Accessing Store Administration Help

To access the persistent store administration utility from WLST, type the following command:

```
> java weblogic.WLST
```

Type `helpstore()` for detailed descriptions on available store administration commands, as well as examples of typical command usage. For example, the following help is provided for the `list` command, which lists store names, open stores, or connections in a store.

```
> wls:/offline> helpstore(liststore)
   lists storenames, opened stores, or connections (for interactive access)
    Parameters store and dir cannot both be specified concurrently.

    Usage: liststore(store='null',dir='null')

    @param store [optional] a previously opened JDBC or File store's name.
          If store is specified, all connections in the store are listed.
    @param dir [optional] directory for which to list available store names
          If dir is specified, all store names in the directory are listed.

    If neither store nor dir are specified, all open store names are listed.
    @return 1 on success, 0 on failure
```

Note that the parameters with an equal sign "=" are optional. For example, the `compactstore` method can be invoked as either `compactstore(dir='storename', tempdir='/tmp')` or `compactstore(store='storename')`, where `tempdir` takes the default value. Default values for optional parameters are listed in the command-specific help.

Dumping the Contents of a JDBC Store Using WLST

Here's an example of using the `dumpstore` method (`store, outfile, conn='null', deep='false'`) to export the contents of a JDBC store named `myJDBCStore` in a human-readable XML file format out to a file named `mystoredump-out.xml`. This does not include store connection names or the actual record contents, which require the optional `conn` and `deep` parameters.

```
> wls:/offline>
  (openjdbcstore('myJDBCStore', 'oracle.jdbc.OracleDriver',
 'jdbc:oracle:thin:@test2k31:1521:test120a', './wlstoreadmin-dump.props',
  'jmstest', 'jmstest', '', 'jdbcstoreprefix')
  dumpstore('myJDBCStore', 'mystoredump-out')
  closestore('myJDBCStore')
```

The `openjdbcstore` and `openfilestore` methods must be used to open and/or create a store first before calling certain administration functions, like `dumpstore` and `liststore` (only when listing open stores). After administering an open store, you must close it using the `closestore` method.

Compacting a File Store Using WLST

Here's an example of a WLST script that uses the `compactstore` method `(dir,tempdir='null')` to compact the space occupied by a file store files in the `mystores` directory.

```
> wls:/offline> compactstore('c:\mystores','c:\tmpmystore.dir')
```

Since the `compactstore()` method can only be used on unopened file stores, none of the stores that have files in the source `'dir'` directory should be open. Also, the temporary `'tempdir'` directory should have at least enough extra space as the source directory and should also not be under the source directory. When compact successfully completes, the newly compacted store files will be in the `mystores` directory. In addition, a new, uniquely-named directory will be created under `tmpmystore` containing the original uncompacted store files.

Limitations of the Persistent Store

The following limitations apply to the persistent store:

- A persistent file store should not be opened simultaneously by two server instances; otherwise, there is no guarantee that the data in the file will not be corrupted. If possible, the persistent store will attempt to return an error in this case, but it will not be possible to detect this condition in every case. It is the responsibility of the administrator to ensure that the persistent store is being used in an environment in which multiple servers will not try to access the same store at the same time. (Two file stores are considered the "same store" if they have the same name and the same directory.)
- Two JDBC stores must not share the same database table, because this will result in data corruption.

- A persistent store may not survive arbitrary corruption. If the disk file is overwritten with arbitrary data, then the results are undefined. The store may return inconsistent data in this case, or even fail to recover at all.
- A file store may return exceptions when its disk is full. However, it will resume normal operation by no longer throwing an exception when disk space has been made available. Also, the data in the persistent store must remain intact as described in the previous points.