BEA WebLogic Server®

Configuring and Managing WebLogic JDBC

Version 9.0 BETA
Revised: December 15, 2004

Contents

Configuring JDBC Multi Data Sources

Using Third-Party JDBC Drivers with WebLogic Server

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Configuring and Managing WebLogic JDBC*.

Document Scope and Audience

This document is a resource for software developers and system administrators who develop and support applications that use the Java Database Connectivity (JDBC) API. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server. The topics in this document are relevant during the evaluation, design, development, pre-production, and production phases of a software project.

This document does not address specific JDBC programming topics. For links to WebLogic Server documentation and resources for these topics, see .

It is assumed that the reader is familiar with J2EE and EJB concepts. This document emphasizes the value-added features provided by WebLogic Server EJBs and key information about how to use WebLogic Server features and facilities to get an EJB application up and running.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide and lists new features in the current release.
- Chapter 2, "Configuring and Deploying WebLogic JDBC Resources," which explains WebLogic JDBC configuration.
- Chapter 3, "Configuring JDBC Data Sources," which describes WebLogic JDBC data source configuration.
- Chapter 4, "Configuring JDBC Multi Data Sources," which describes WebLogic JDBC multi data source configuration.
- Chapter 5, "Using Third-Party JDBC Drivers with WebLogic Server," which describes how to use JDBC driver from other sources in your WebLogic JDBC configuration.
- Appendix A, "Packaged JDBC Modules," which describes how to package a WebLogic JDBC module with your enterprise application.

Related Documentation

This document contains JDBC-specific configuration and administration information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Programming WebLogic JDBC* is a guide to JDBC API programming with WebLogic Server.
- *Developing WebLogic Server Applications* is a guide to developing WebLogic Server applications.
- *Deploying WebLogic Server Applications* is the primary source of information about deploying WebLogic Server applications in development and production environments.
- *WebLogic Server Performance and Tuning* contains information on monitoring and improving the performance of WebLogic Server applications.

JDBC Samples and Tutorials

In addition to this document, BEA Systems provides a variety of JDBC code samples and tutorials that show JDBC configuration and API use, and provide practical instructions on how to perform key JDBC development tasks.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample J2EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and J2EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

JDBC Examples in the WebLogic Server Distribution

WebLogic Server 9.0 optionally installs API code examples in `WL_HOME\samples\server\examples\src\examples`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server 9.0 Start menu.

New and Changed JDBC Features in This Release

The following sections describe new features and changes for JDBC in WebLogic Server 9.0:

- J2EE 1.4 Support:
  - "JDBC 3.0 Support" on page 1-4
  - "RowSets Enhancements" on page 1-5
- Configuration Enhancements:
  - "Modular Deployment of System and Stand-Alone JDBC Resources" on page 1-6
  - "Simplified JDBC Resource Configuration" on page 1-8
  - "Support for the J2EE Management Model (JSR-77)" on page 1-8
- "Support for Logging Last Resource and Other Transaction Options" on page 1-9
- "Credential Mapping for WebLogic Server User IDs to Database User IDs" on page 1-9
- Other Data Source Enhancements:
  - "SQL Statement Timeout Enhancements for Pooled JDBC Connections" on page 1-10

- “Connection Testing Enhancements” on page 1-10
  - “Clean Server Shutdown When Database Connections are In Use” on page 1-10
  - “PinnedToThread Connection Pool Property to Increase Performance” on page 1-10
  - “Support for Oracle Virtual Private Databases” on page 1-11
- Other multi data source/MultiPool Enhancements:
  - “XA Support in Multi Data Sources” on page 1-13
  - “JDBC Multi Data Source Failover Enhancements” on page 1-14
- “JDBC Monitoring and Diagnostics Enhancements” on page 1-14
- “MySQL Support” on page 1-16
- “Updated WebLogic Type 4 JDBC Drivers” on page 1-16
- “Deprecated JDBC Features, Methods, Interfaces, and MBeans” on page 1-17

JDBC 3.0 Support

WebLogic Server 9.0 is compliant with the JDBC 3.0 specification, which includes support for the following new features:

- Savepoints
- Parameter metadata retrieval for prepared statements
- Auto-generated key retrieval
- Multiple open ResultSet objects
- Passing parameters to CallableStatement objects by name
- Holdable cursors
- BOOLEAN data type
- Internal updates to the data in Blob and Clob objects
- Retrieving and updating the object referenced by a Ref object
- Updating columns containing BLOB, CLOB, ARRAY and REF types
- DATALINK/URL data type

- Transform groups and type mapping
- DatabaseMetadata API enhancements

For more information, see the Java JDBC technology page on the Sun Web site at http://java.sun.com/products/jdbc/.

RowSets Enhancements

In WebLogic Server 9.0, the RowSets implementation was enhanced to comply with the new JDBC RowSet Implementations Specification (JSR-114) and to provide extensions to the specification.

Support for JSR-114

The WebLogic Server RowSets implementation fully complies with the JDBC RowSet Implementations Specification (JSR-114), which includes support for the following objects:

- CachedRowSets—disconnected rowsets.
- FilteredRowSets—cached rowsets in which available rows are determined by a filter.
- WebRowSets—cached rowsets that can read and write rowset data in XML format.
- JoinRowSets—rowsets built with a SQL JOIN that joins data from any disconnected rowset.
- JdbcRowSets—connected rowsets.
- SyncResolver—a cached rowset that implements the `javax.sql.rowset.spi.SyncResolver` interface and is used to resolve conflicts that occur when data updates in the database conflict with data updates in the rowset.

For more information about rowsets, see the Java JDBC technology page on the Sun Web site at http://java.sun.com/products/jdbc/. For more information about using rowsets and the WebLogic Server RowSets implementation, see "Using RowSets with WebLogic Server" in *Programming WebLogic JDBC*.

WebLogic RowSet Extensions and Enhancements

WebLogic Server 9.0 includes the following extensions to rowsets:

- WLCachedRowSets—extends CachedRowSets, FilteredRowSets, WebRowSets, and SortedRowSets. WLCachedRowSets can be interchangeably used as any of the standard rowset types that it extends. It includes convenience methods that help make using rowsets easier. WLCachedRowSets also include methods for setting optimistic concurrency options and data synchronization options.

- SharedRowSets—extends CachedRowSets so that additional CachedRowSets can be created for use in other threads based on the data in an original CachedRowSet. Each SharedRowSet is a shallow copy of the original rowset (with references to data in the original rowset instead of a copy of the data) with its own context (cursor, filter, sorter, and pending changes). Using SharedRowSets can increase performance by reducing the number of database round-trips required by an application.
- SortedRowSets—extends CachedRowSets so that rows in a CachedRowSet can be sorted in memory rather than depending on the database management system for sort processing. Using SortedRowSets can increase application performance by reducing the number of database round-trips.
- SQL-style filter—an implementation of the `javax.sql.rowset.Predicate` interface. that you can use to define a filter for a FilteredRowSet using SQL-like WHERE clause syntax.

For more information about using JDBC RowSets and the WebLogic Server RowSets implementation, see "Using RowSets with WebLogic Server" in *Programming WebLogic JDBC*.

Modular Deployment of System and Stand-Alone JDBC Resources

JDBC configuration in WebLogic Server 9.0 is now stored in XML documents that conform to the new `weblogic-jdbc.xsd` schema. You create and manage JDBC resources either as system modules, similar to the way they were managed prior to version 9.0, or as application modules. JDBC application modules are a WebLogic-specific extension of J2EE modules and can be deployed either within a J2EE application or as stand-alone modules.

With modular deployment of JDBC resources, you can migrate your application and the required JDBC configuration from environment to environment, such as from a testing environment to a production environment, without opening an EAR file and without extensive manual JDBC reconfiguration.

JDBC System Modules

An Administrator can create JDBC resources directly using the Administration Console or using WLST, similar to the way resources were created prior to WebLogic Server 9.0. JDBC resources created this way are stored as system resource modules in the `config/jdbc` subdirectory of the domain directory, and are referenced in the domain's `config.xml` file. After they are created, these JDBC resources are also accessible as a `JDBCSystemResourceMBean` through JMX.

System modules are globally available for targeting to servers and clusters configured in the domain, and therefore are available to all applications deployed on the same targets and to client applications. System resource modules are owned by the Administrator, who can delete, modify, or add similar resources at any time.

Application Modules

JDBC resources can also be managed as application modules, similar to standard J2EE modules. A JDBC descriptor can be deployed as a standalone resource using the `weblogic.Deployer` utility, in which case the resource is available to the server or cluster targeted during the deployment process. JDBC resources deployed in this manner can be reconfigured using the Administration Console, but are unavailable through JMX or WLST.

Packaged JDBC Resource Modules

Resource modules can also be included as part of an Enterprise Application as a *packaged module*. Packaged modules are bundled with an EAR or exploded EAR directory, and are referenced in the `weblogic-application.xml` deployment descriptor. The resource module is deployed along with the Enterprise Application, and can be configured to be available only to the enclosing application or to all applications. Using packaged modules ensures that an application always has access to required resources and simplifies the process of moving the application into new environments.

In contrast to system resource modules, packaged modules are owned by the developer who created and packaged the module, rather than the Administrator who deploys the module. This means that the Administrator has more limited control over packaged modules. When deploying a resource module, an Administrator can change resource properties that were specified in the module, but the Administrator cannot add or delete resources. (As with other J2EE modules, deployment configuration changes for a resource module are stored in a deployment plan for the module, leaving the original module untouched.)

JDBC Schema

In support of the new modular deployment model for JDBC resources in WebLogic Server 9.0, BEA now provides a schema for WebLogic JDBC objects. When you create JDBC resource modules (descriptors), the modules must conform to the schema. IDEs and other tools can validate JDBC resource modules based on the schema.

Simplified JDBC Resource Configuration

In WebLogic Server 9.0, the number of JDBC resource types was reduced to simplify JDBC configuration and to reduce the likelihood of configuration errors. Instead of configuring a JDBC connection pool and then configuring a data source or tx data source to point to the connection pool and bind to the JNDI tree, you configure a data source that encompasses a connection pool.

Note: Because of the new configuration design, you can no longer have multiple data sources that point to a single connection pool. Instead, you can create additional data sources, each with its own pool of connections, or you can bind a single data source to the JNDI tree with multiple names. See "Binding a Data Source to the JNDI Tree with Multiple Names" on page 3-11 for more information.

MultiPool configuration has also been simplified in WebLogic Server 9.0. MultiPools are replaced by multi data sources. Like MultiPools, multi data sources include a list of other JDBC resources. However, in this release you do not have to configure a separate data source to point to a multi data source to bind it to the JNDI tree. Also, if you configure the multi data source through the Administration Console, you can configure the multi data source and all encompassed data sources in one step.

As part of the JDBC resource redefinition, some new MBeans have been introduced and others have been deprecated:

- `JDBCSystemResourceMBean`—replaces the `JDBCConnectionPoolMBean`, `JDBCMultiPoolMBean`, and `JDBCTxDataSourceMBean` types, which are now deprecated.
- `JDBCDataSourceRuntimeMBean`—replaces the `JDBCConnectionPoolRuntimeMBean` type, which is now deprecated.

For more information about configuring JDBC resources, see "Configuring and Deploying WebLogic JDBC Resources" on page 2-1.

Support for the J2EE Management Model (JSR-77)

WebLogic Server 9.0 JDBC supports JSR-77, which defines the J2EE Management Model. The J2EE Management Model is used for monitoring the runtime state of a J2EE Web application server and its resources. You can access the J2EE Management Model to monitor resources, including the Weblogic JDBC system as a whole, JDBC drivers loaded into memory, and JDBC data sources.

For more details, see "JDBC Management Objects in the J2EE Management Model (JSR-77 Support)" on page 2-8.

Support for Logging Last Resource and Other Transaction Options

In WebLogic Server 9.0, you can configure a JDBC data source to enable the Logging Last Resource (LLR) transaction optimization, which enables one non-XA resource to participate in a global transaction with improved performance and with the same ACID guarantee as XA.

The LLR optimization improves performance by:

- Removing the need for an XA JDBC driver to connect to the database. XA JDBC drivers are typically inefficient compared to non-XA JDBC drivers.
- Reducing the number of processing steps to complete the transaction, which also reduces network traffic and the number of disk I/Os.
- Removing the need for XA processing at the database level (if the database is the one non-XA resource).

When a connection from a data source configured for LLR participates in a global transaction, the WebLogic Server transaction manager calls prepare on all other (XA-compliant) transaction participants, writes the transaction log on the LLR participant in a database table while committing the transaction on the LLR participant as a one-phase commit, then calls commit on all other transaction participants.

The Logging Last Resource optimization maintains data integrity by writing the transaction log on the LLR participant. If the transaction fails during the one-phase commit, the WebLogic Server transaction manager will roll back the transaction on all other transaction participants.

For more information about the Logging Last Resource option, see "Using LLR Enabled Connection Pools" in *Programming WebLogic JTA*.

Credential Mapping for WebLogic Server User IDs to Database User IDs

Credential mapping for a JDBC data source is the process in which WebLogic Server user IDs are mapped to database user IDs. If credential mapping is enabled on the data source, when an application requests a database connection from the data source, WebLogic Server determines the current WebLogic Server user ID and then sets the mapped database ID as a light-weight client ID on the database connection.

This feature relies on features in the JDBC driver and DBMS. It is only supported for use with Oracle and DB2 databases and with the Oracle Thin and DB2 UDB JDBC drivers, respectively.

For more information, see "Configuring Credential Mapping for a Data Source" on page 3-13.

Other Data Source/Connection Pool Enhancements

SQL Statement Timeout Enhancements for Pooled JDBC Connections

In WebLogic Server 9.0, the following attributes were added to JDBC data source connection pools to enable you to limit the amount of time that a statement can execute on a pooled database connection:

- `StatementTimeout`—The time in seconds after which a statement executing on a pooled JDBC connection times out. When set to -1, (the default) statements do not time out.
- `TestStatementTimeout`—The time in seconds after which a statement executing on a pooled JDBC connection for connection initialization or testing times out. When set to -1, (the default) statements do not time out.

For more information, see "SQL Statement Timeout Enhancements for Pooled JDBC Connections" in *Programming WebLogic JDBC*.

Connection Testing Enhancements

In WebLogic Server 9.0, the following features were added to JDBC data source connection pools to improve the functionality of database connection testing for pooled connections and to minimize delays in connection request handling:

- `PurgePoolUponTestFailure`—Closes all connections in the connection pool after a connection test failure to minimize the delay caused by further database testing.
- `SecondsToTrustAnIdlePoolConnection`—Enables WebLogic Server to skip testing a database connection if the connection was successfully used within the period of time specified. This feature can increase performance by minimizing database connection testing.

Clean Server Shutdown When Database Connections are In Use

In WebLogic Server 9.0, the `IgnoreInUseConnections` attribute was added to JDBC data sources to enable WebLogic Server to ignore database connections that are in use when shutting down a server instance. When this attribute is set to `true`, WebLogic Server ignores any database connections in use and shuts down the server without issue. When set to `false`, WebLogic Server waits for in-use connections to be returned to the pool of connections.

PinnedToThread Connection Pool Property to Increase Performance

To minimize the time it takes for an application to reserve a database connection from a connection pool and to eliminate contention between threads for a database connection, you can add the `PinnedToThread` property in the connection Properties list for the connection pool, and set its value to `true`.

When `PinnedToThread` is enabled, WebLogic Server pins a database connection from the connection pool to an execution thread the first time an application uses the thread to reserve a connection. When the application finishes using the connection and calls `connection.close()`, which otherwise returns the connection to the connection pool, WebLogic Server keeps the connection with the execute thread and does not return it to the connection pool. When an application subsequently requests a connection using the same execute thread, WebLogic Server provides the connection already reserved by the thread. There is no locking contention on the connection pool that occurs when multiple threads attempt to reserve a connection at the same time and there is no contention for threads that attempt to reserve the same connection from a limited number of database connections.

For more details, see *Configuring and Managing WebLogic JDBC*.

Support for Oracle Virtual Private Databases

WebLogic Server 9.0 provides support for Oracle Virtual Private Databases (VPDs). A VPD is means to control access to data based on the user's identity. WebLogic Server uses JDBC extensions in the Oracle thin driver to set the user credentials on a database connection.

See "Programming with Oracle Virtual Private Databases" in *Programming WebLogic JDBC* for more information.

Multiple JNDI Names for Data Sources

You can configure a data source so that it binds to the JNDI tree with multiple names. You can use a multi-named data source in place of configuring multiple data sources that point to a single JDBC connection pool.

For more details, see "Binding a Data Source to the JNDI Tree with Multiple Names" in the Administration Console Online Help.

Connection Profiling for Troubleshooting Transaction Remnants on a JDBC Connection

In WebLogic Server 9.0, the `ConnProfileEnabled` attribute was added to the JDBC data source configuration. When set to true, WebLogic Server stores the stack trace whenever a connection is released back into the data source's connection pool. If an exception is thrown during a subsequent operation on the connection related to global (XA) transactions, WebLogic Server reports this stack trace with the exception.

You can use this feature to detect local transaction work left incomplete by application code, which can interfere with subsequent global (XA) transaction operations on the JDBC connection.

This feature uses more resources than normal connection pool operations and will likely degrade connection pool performance, so it is not recommended for production use. Also, this feature does not apply to connections created with a non-XA JDBC driver.

Support for XAResource Transaction Timeout

The WebLogic Server Transaction Manager now supports setting a transaction branch timeout value on a participating XA resource if the resource manager supports the `javax.transaction.xa.XAResource.setTransactionTimeout()` method. You may want to set a transaction branch timeout if you have long-running transactions that exceed the default timeout value on the XA resource.

For the WebLogic Server Transaction Manager to set the transaction timeout on a JDBC XA resource, specify a value for the following properties in the JDBC connection pool tag in the `config.xml` file:

- `XASetTransactionTimeout`—A boolean property. When set to true, the WebLogic Server Transaction Manager calls `XAResource.setTransactionTimeout()` before calling `XAResource.start`, and passes either the `XATransactionTimeout` or the global transaction timeout in seconds. When set to false, the Transaction Manager does not call `setTransactionTimeout()`. The default value is false.

- `XATransactionTimeout`—The number of seconds to pass as the transaction timeout value in the `XAResource.setTransactionTimeout()` method. When this property is set to `0`, the WebLogic Server Transaction Manager passes the global WebLogic Server transaction timeout in seconds in the method. The default value for this parameter is `0`. If set, this value should be greater than or equal to the global Weblogic Server transaction timeout.

These properties apply to connection pools that use an XA JDBC driver to create database connections only. They are ignored if a non-XA JDBC driver is used.

When these values are set, the WebLogic Server Transaction Manager calls `XAResource.setTransactionTimeout()` as described above. The implementation of the method in the XA resource manager (for example, an XA JDBC driver) or the XA resource determines how the value is used. For example, for Oracle, the `setTransactionTimeout()` method sets the Session Timeout (`SesTm`), which acts as a maximum idle time for a transaction. The behavior may be different for other XA Resources.

use-xa-data-source-interface Property for Data Sources

In WebLogic Server 9.0, the use-xa-data-source-interface data source property was introduced to enable you to specify the use of the XA version of a JDBC driver if the driver uses the same classname for both XA and non-XA use.

Multi Data Source Enhancements

The following multi data source enhancements were added in WebLogic Server 9.0:

- “XA Support in Multi Data Sources” on page 1-13
- “JDBC Multi Data Source Failover Enhancements” on page 1-14
- “HIGH AVAILABILITY Multi Data Source Algorithm Renamed FAILOVER” on page 1-14

XA Support in Multi Data Sources

In WebLogic Server 9.0, JDBC multi data sources are supported for use in XA transactions. You can configure the data sources contained within the multi data source to use XA JDBC drivers.

To configure a multi data source with support for global transactions and two-phase commit, all data sources within the multi data source must have the same transaction-related attribute settings, including:

- SupportsLocalTransaction
- KeepXAConnTillTxComplete
- NeedTxCtxOnClose
- XAEndOnlyOnce
- NewXAConnForCommit
- RollbackLocalTxUponConnClose
- RecoverOnlyOnce
- KeepLogicalConnOpenOnRelease

For more information about configuring multi data sources, see “Configuring JDBC Multi Data Sources” on page 4-1.

JDBC Multi Data Source Failover Enhancements

In WebLogic Server 9.0, the following enhancements were made to JDBC multi data sources:

- Connection request routing enhancements to avoid requesting a connection from a disabled data source within a multi data source.
- Automatic failback on recovery of a failed data sources within a multi data source.
- Failover for busy data sources within a multi data source with the Failover algorithm.
- Failover callbacks for multi data sources with the Failover algorithm.
- Failback callbacks for multi data sources with either algorithm.

See "Multi Data Source Failover Enhancements" on page 4-3 for more details.

HIGH AVAILABILITY Multi Data Source Algorithm Renamed FAILOVER

In WebLogic Server 9.0, the multi data source `HIGH AVAILABILITY` algorithm option was renamed to `FAILOVER`. The new name is more indicative of the behavior of the multi data source with this algorithm selection.

New WLST Script Examples for JDBC Administration

WebLogic Server 9.0 ships with a WLST script example that shows how to perform administrative tasks previously available in the weblogic.Admin utility. For example, creating a data source, resetting the pooled database connections in a data source, and so forth.

JDBC Monitoring and Diagnostics Enhancements

The following enhancements for monitoring JDBC activity and diagnosing JDBC issues were added in WebLogic Server 9.0 to help you easily monitor JDBC resources and diagnose problems:

- "New Monitoring Statistics and Profile Information for JDBC Data Sources" on page 1-15
- "Callbacks for Monitoring Driver-Level Statistics" on page 1-15
- "JDBC Debugging Enhancements" on page 1-16

New Monitoring Statistics and Profile Information for JDBC Data Sources

In WebLogic Server 9.0, new data source usage information was made available through the Administration Console or through JMX to help you monitor and tune your environment, including:

- Cumulative number of requests to reserve a database connection
- Cumulative number of failed requests to reserve a connection
- Average time a connection is in use by an application
- Average time an application waits for a connection
- Percentage of connections currently in use
- Percentage of time that all connections were in use
- Current connection users and how long the application has held the connection
- Current applications waiting for a connection and how long each application has been waiting

In WebLogic Server 9.0, the following new prepared statement cache statistics and usage profile information is available through the Administration Console or through JMX:

- Cumulative number of times the cache is accessed
- Cumulative number of statements added to the cache
- Cumulative number of statements discarded from the cache
- Current number of statements in the cache
- Current statements (actual statement text)

Callbacks for Monitoring Driver-Level Statistics

WebLogic Server 9.0 provides callbacks for methods called on a JDBC driver. You can use these callbacks to monitor and profile JDBC driver usage, including methods being executed, any exceptions thrown, and the time spent executing driver methods.

To enable the callback feature, you specify the fully qualified path of the callback handler for the `driver-interceptor` element in the JDBC data source descriptor (module). Your callback handler must implement the `weblogic.jdbc.extensions.DriverInterceptor` interface. When JDBC driver callbacks are enabled, WebLogic Server calls the `preInvokeCallback()`, `postInvokeExceptionCallback()`, and `postInvokeCallback()` methods of the registered callback handler before and after invoking any method inside the JDBC driver.

JDBC Debugging Enhancements

In Weblogic Server 9.0, the JDBC subsystem uses the new debugging and diagnostics system for centralized debug access and logging. The following JDBC debugging options are available:

- DebugJDBCSQL (weblogic.jdbc.sql scope)—Logs information about all JDBC methods invoked, including their arguments, return values, and thrown exceptions.
- DebugJDBCConn (weblogic.jdbc.connection scope)—Logs all connection reserve and release operations in connection pools as well as all application requests to get or close connections.
- DebugJDBCRMI (weblogic.jdbc.rmi scope)—Similar to JDBCSQL, but at the RMI level.
- DebugJDBCInternal (weblogic.jdbc.internal scope)—Logs low-level debugging information related to the connection pool, the connection environment, the connection pool manager, and the data source manager.
- DebugJDBCDriverLogging (weblogic.jdbc.driverlogging scope)—Enables JDBC driver-level logging (replaces ServerMBean `JDBCLoggingEnabled` and `getJDBCLogFileName`).
- DebugJDBCJTA (weblogic.jdbc.transaction scope)—Logs all XA operations.

See *Understanding the WebLogic Diagnostic Service* for more information.

MySQL Support

WebLogic Server 9.0 is supported for use with a MySQL database, a popular open-source production-ready database management system. The MySQL Connection/J JDBC driver is installed with WebLogic Server. MySQL has been certified for use with JMS, JDBC, and EJB container-managed persistence. MySQL does not support XA or JTA.

Updated WebLogic Type 4 JDBC Drivers

WebLogic Server 8.1 SP3 includes updates to the WebLogic Type 4 JDBC drivers. The updated drivers resolve some important issues and include some notable enhancements. See *WebLogic Type 4 JDBC Drivers* for more information.

Deprecated JDBC Features, Methods, Interfaces, and MBeans

In WebLogic Server 9.0, many changes were made to the JDBC subsystem, including the removal of some drivers and classes and the deprecation of some MBeans.

Removed

The following items were removed in WebLogic Server 9.0:

- WebLogic JDriver for Oracle. Replaced by the WebLogic Type 4 JDBC Driver for Oracle.
- WebLogic JDriver for MS SQL Server. Replaced by the WebLogic Type 4 JDBC Driver for MS SQL Server.

Deprecated Features

The following items were deprecated in WebLogic Server 9.0:

- Legacy application-scoped connection pools. Replaced by JDBC modules. See "Modular Deployment of System and Stand-Alone JDBC Resources" on page 1-6.
- Legacy driver-level logging. Replaced by DebugJDBCDriverLogging. See "JDBC Debugging Enhancements" on page 1-16.
- JDBCXADebugLevel of the JDBCConnectionPoolMBean. Replaced with JTAJDBC on the ServerDebugMBean.
- Legacy connection leak profiling.
- Legacy rowset XML writing and reading. Replaced with functionality in a WebRowSet. See "RowSets Enhancements" on page 1-5.

Deprecated MBeans

The following MBeans were deprecated in WebLogic Server 9.0:

- JDBCConnectionPoolMBean. Functionality from this MBean was added to the JDBCDataSourceMBean.
- JDBCMultiPoolMBean. Functionality from this MBean was added to the JDBCDataSourceMBean.
- JDBCTxDataSourceMBean. Functionality from this MBean was added to the JDBCDataSourceMBean.
- JDBCDataSourceFactoryMBean. No replacement.

- JDBCConnectionPoolRuntimeMBean. Replaced by JDBCDataSourceRuntimeMBean.

Configuring and Deploying WebLogic JDBC Resources

The following sections describe WebLogic JDBC resources, how they are configured, and how those resources apply to a WebLogic domain:

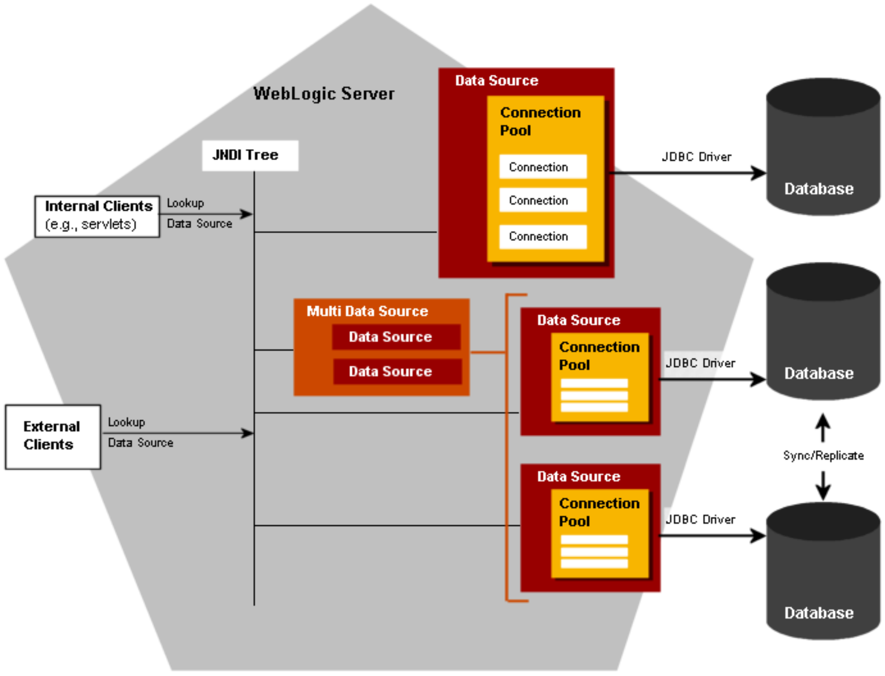
Understanding JDBC Resources in WebLogic Server

In WebLogic Server, you can configure database connectivity by configuring JDBC data sources and multi data sources and then targeting or deploying the JDBC resources to servers or clusters in your WebLogic domain.

Each data source that you configure contains a pool of database connections that are created when the data source instance is created—when it is deployed or targeted, or at server startup. Applications lookup a data source on the JNDI tree or in the local application context (`java:comp/env`), depending on how you configure and deploy the object, and then request a database connection. When finished with the connection, the application calls `connection.close()`, which returns the connection to the connection pool in the data source.

Figure 2-1 shows a data source and a multi data source targeted to a WebLogic Server instance.

Figure 2-1 JDBC Data Source Architecture



For more information about data sources in WebLogic Server, see "Configuring JDBC Data Sources" on page 3-1.

A multi data source is an abstraction around a data sources that provides load balancing or failover processing between the data sources associated with the multi data source. Multi data sources are bound to the JNDI tree or local application context just like data sources are bound to the JNDI tree. Applications lookup a multi data source on the JNDI tree or in the local application context (`java:comp/env`) just like they do for data sources, and then request a database connection. The multi data source determines which data source to use to satisfy the request depending on the algorithm selected in the multi data source configuration: load balancing or failover. For more information about multi data sources, see "Configuring JDBC Multi Data Sources" on page 4-1.

Ownership of Configured JDBC Resources

A key to understanding WebLogic JDBC configuration and management is that *who* creates a JDBC resource or *how* a JDBC resource is created determines how a resource is deployed and modified. Both WebLogic Administrators and programmers can create JDBC resources:

- WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JDBC modules. These JDBC modules are considered *system modules*. See "JDBC System Modules" on page 2-4 for more details.
- Programmers create modules in WebLogic Workshop (not available at beta) or another development tool that supports creating an XML descriptor file, then package the JDBC modules with an application and pass the application to a WebLogic Administrator to deploy. These JDBC modules are considered *application modules*. See "JDBC Application Modules" on page 2-6 for more details.

Table 2-1 lists the JDBC module types and how they can be configured and modified.

Table 2-1 JDBC Module Types and Configuration and Management Options

Module Type	Created with	Add/Remove Modules with Administration Console	Modify with JMX (remotely)	Modify with JSR-88 (non-remotely)	Modify with Administration Console
System	Administration Console or WLST	Yes	Yes	No	Yes—via JMX
Application	WebLogic Workshop, another IDE, or an XML editor	No	No	Yes—via a deployment plan	Yes—via a deployment plan (not at beta)

JDBC Configuration Files

WebLogic JDBC configuration is stored in XML documents that conform to the `weblogic-jdbc.xsd` schema (available at `http://www.bea.com/ns/weblogic/90/weblogic-jdbc.xsd`). You create and manage JDBC resources either as system modules, similar to the way they were managed prior to version

9.0, or as application modules. JDBC application modules are a WebLogic-specific extension of J2EE modules and can be configured either within a J2EE application or as stand-alone modules.

Regardless of whether you are using JDBC system modules or JDBC application modules, each JDBC data source or multi data source is represented by an XML file (a module).

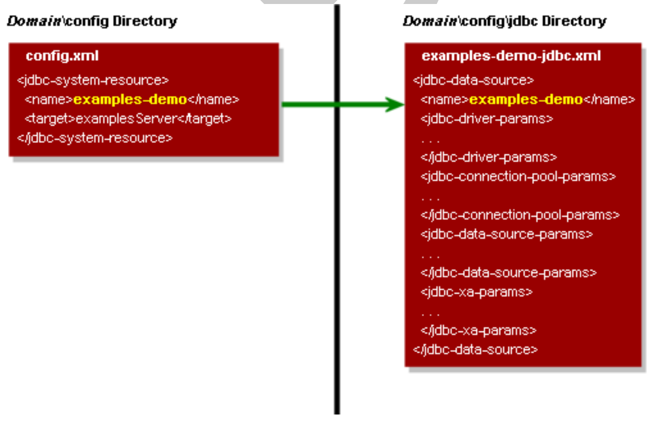
JDBC System Modules

When you create a JDBC resource (data source or multi data source) using the Administration Console or using the WebLogic Scripting Tool (WLST), WebLogic Server creates a JDBC module in the `config/jdbc` subdirectory of the domain directory, and adds a reference to the module in the domain's `config.xml` file. The JDBC module conforms to the `weblogic-jdbc.xsd` schema (available at `http://www.bea.com/ns/weblogic/90/weblogic-jdbc.xsd`).

JDBC resources that you configure this way are considered *system modules*. System modules are owned by an Administrator, who can delete, modify, or add similar resources at any time. System modules are globally available for targeting to servers and clusters configured in the domain, and therefore are available to all applications deployed on the same targets and to client applications. System modules are also accessible through JMX as `JDBCSystemResourceMBeans`.

Data source system modules are included in the domain's `config.xml` file as a `JDBCSystemResource` element, which includes the name of the JDBC module file and the list of target servers and clusters on which the module is deployed. Figure 2-2 shows an example of a data source listing in a `config.xml` file and the module that it maps to.

Figure 2-2 Reference from config.xml to a Data Source System Module



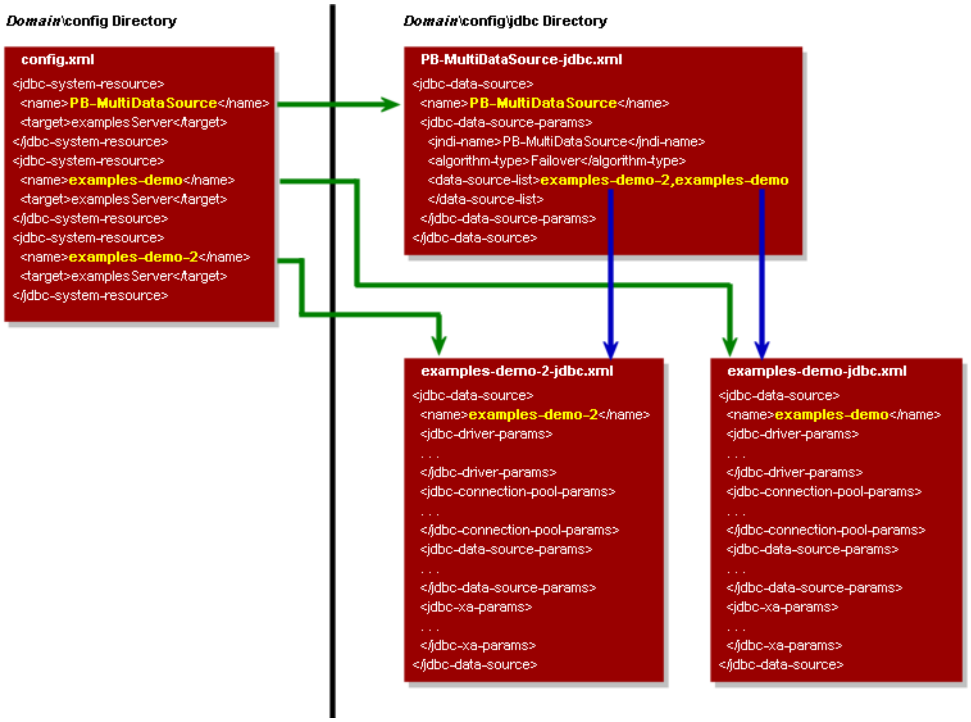
```
<jdbc-system-resource>
  <name>examples-demo</name>
  <target>examplesServer</target>
</jdbc-system-resource>
```

```
<jdbc-data-source>
  <name>examples-demo</name>
  <jdbc-driver-params>
  . . .
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
  . . .
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
  . . .
  </jdbc-data-source-params>
  <jdbc-xa-params>
  . . .
  </jdbc-xa-params>
</jdbc-data-source>
```

In this illustration, the `config.xml` file lists the `examples-demo` data source as a `jdbc-system-resource` element, which maps to the `examples-demo.xml` module in the *`domain`*`\config\jdbc` folder.

Similarly, multi data source system modules are included in the domain's `config.xml` file as a `jdbc-system-resource` element. The multi data source module includes a `data-source-list` parameter that maps to the data source modules used by the multi data source. The individual data source modules are also included in the `config.xml`. Figure 2-3 shows the relationship between elements in the `config.xml` file and the system modules in the `config/jdbc` directory.

Figure 2-3 Reference from config.xml to Multi Data Source and Data Source System Modules



```
<jdbc-system-resource>
  <name>PB-MultiDataSource</name>
  <target>examplesServer</target>
</jdbc-system-resource>
<jdbc-system-resource>
  <name>examples-demo</name>
  <target>examplesServer</target>
</jdbc-system-resource>
<jdbc-system-resource>
  <name>examples-demo-2</name>
  <target>examplesServer</target>
</jdbc-system-resource>
```

```
<jdbc-data-source>
  <name>PB-MultiDataSource</name>
  <jdbc-data-source-params>
    <jndi-name>PB-MultiDataSource</jndi-name>
    <algorithm-type>Failover</algorithm-type>
    <data-source-list>examples-demo-2,examples-demo
    </data-source-list>
  </jdbc-data-source-params>
</jdbc-data-source>
```

```
<jdbc-data-source>
  <name>examples-demo-2</name>
  <jdbc-driver-params>
  . . .
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
  . . .
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
  . . .
  </jdbc-data-source-params>
  <jdbc-xa-params>
  . . .
  </jdbc-xa-params>
</jdbc-data-source>
```

```
<jdbc-data-source>
  <name>examples-demo</name>
  <jdbc-driver-params>
  . . .
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
  . . .
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
  . . .
  </jdbc-data-source-params>
  <jdbc-xa-params>
  . . .
  </jdbc-xa-params>
</jdbc-data-source>
```

In this illustration, the `config.xml` file lists three JDBC modules—one multi data source and the two data sources used by the multi data source, which are also listed within the multi data source module. Your application can look up any of these modules on the JNDI tree and request a database connection. If you look up the multi data source, the multi data source determines which of the other data sources to use to supply the database connection, depending on the data sources in the `data-source-list` parameter, the order in which the data sources are listed, and the algorithm specified in the `algorithm-type` parameter. For more information about multi data sources, see "Configuring JDBC Multi Data Sources" on page 4-1.

JDBC Application Modules

JDBC resources can also be managed as application modules, similar to standard J2EE modules. A JDBC application module is simply an XML file that conforms to the `weblogic-jdbc.xsd` schema and represents a data source or a multi data source. JDBC application modules can be deployed as stand-alone modules or packaged modules.

Stand-Alone JDBC Modules

A JDBC application module can be deployed as a stand-alone resource using the `weblogic.Deployer` utility or the Administration Console (not available at beta), in which case the resource is typically available to the server or cluster targeted during the deployment process. JDBC resources deployed in this manner are called *stand-along modules* and can be reconfigured using the Administration Console or a JSR-88 compliant tool, but are unavailable through JMX or WLST.

Stand-alone JDBC modules promote sharing and portability of JDBC resources. You can create a data source configuration and distribute it to other developers. Stand-alone JDBC modules can also be used to move JDBC configuration between domains, such as between the development domain and the staging domain.

Deploying Stand-Alone JDBC Modules

For information about deploying stand-alone JDBC modules, see "Deploying JDBC and JMS Application Modules."

Packaged JDBC Modules

JDBC modules can also be included as part of an Enterprise Application as a *packaged module*. Packaged modules are bundled with an EAR or exploded EAR directory, and are referenced in all appropriate deployment descriptors, such as the `weblogic-application.xml` and `ejb-jar.xml` deployment descriptors. The JDBC module is deployed along with the enterprise application, and can be configured to be available only to the enclosing application or to all applications. Using packaged modules ensures that an application always has access to required resources and simplifies the process of moving the application into new environments. With packaged JDBC modules, you can migrate your application and the required JDBC configuration from environment to environment, such as from a testing environment to a production environment, without opening an EAR file and without extensive manual JDBC reconfiguration.

In contrast to system resource modules, packaged modules are owned by the developer who created and packaged the module, rather than the Administrator who deploys the module. This means that the Administrator has more limited control over packaged modules. When deploying a resource module, an Administrator can change resource properties that were specified in the module, but the Administrator cannot add or delete modules. (As with other J2EE modules, deployment configuration changes for a resource module are stored in a deployment plan for the module, leaving the original module untouched.)

For more information about packaged modules, see "Packaged JDBC Modules" on page A-1.

Deploying a Packaged JDBC Module

By definition, packaged JDBC modules are included in an enterprise application, and therefore are deployed when you deploy the enterprise application. For more information about deploying applications with packaged JDBC modules, see *Deploying Applications to WebLogic Server*.

JDBC Module File Naming Requirements

All WebLogic JDBC module files must end with the -jdbc.xml suffix, such as `examples-demo-jdbc.xml`. WebLogic Server checks the file name when you deploy the module. If the file does not end in `-jdbc.xml`, the deployment will fail.

JDBC Schema

In support of the new modular deployment model for JDBC resources in WebLogic Server 9.0, BEA now provides a schema for WebLogic JDBC objects: `weblogic-jdbc.xsd`. When you create JDBC resource modules (descriptors), the modules must conform to the schema. IDEs and other tools can validate JDBC resource modules based on the schema.

The schema is available at `http://www.bea.com/ns/weblogic/90/weblogic-jdbc.xsd`.

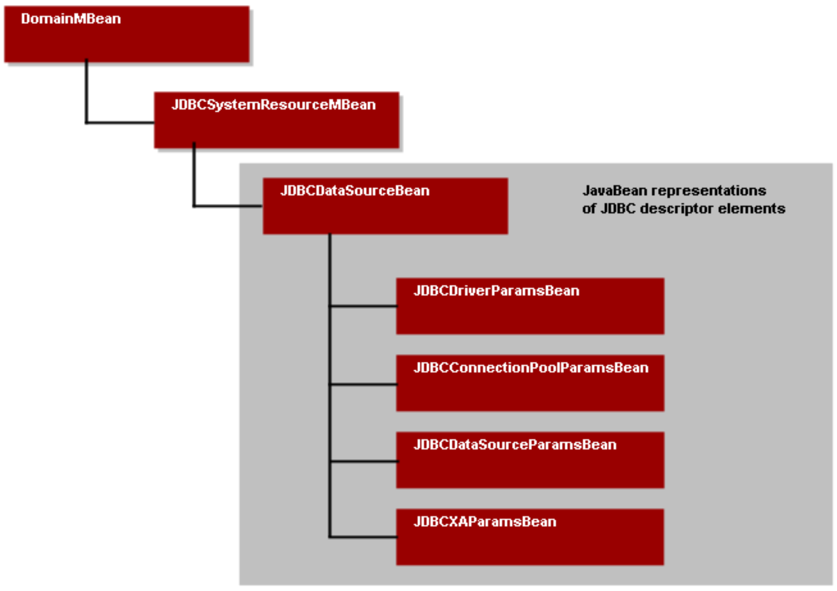
JMX and WLST Access for JDBC Resources

When you create JDBC resources using the Administration Console or WLST, WebLogic Server creates MBeans (management beans) for each of the resources. You can then access these MBeans using JMX or the WebLogic Scripting Tool (WLST). See *Developing Manageable Applications with JMX* and *WebLogic Scripting Tool* for more information.

JDBC MBeans for System Resources

Figure 2-4 shows the hierarchy of the MBeans for JDBC objects in a WebLogic domain.

Figure 2-4 JDBC Bean Tree



The `JDBCSystemResourceMBean` is a container for the JavaBeans created from a data source module. However, all JMX access for a JDBC data source is though the `JDBCSystemResourceMBean`. You cannot directly access the individual JavaBeans created from the data source module.

JDBC Management Objects in the J2EE Management Model (JSR-77 Support)

WebLogic Server 9.0 JDBC supports JSR-77, which defines the J2EE Management Model. The J2EE Management Model is used for monitoring the runtime state of a J2EE Web application server and its resources. You can access the J2EE Management Model to monitor resources, including the WebLogic JDBC system as a whole, JDBC drivers loaded into memory, and JDBC data sources.

To comply with the specification, BEA added the following runtime MBean types for WebLogic JDBC:

- `JDBCServiceRuntimeMBean`—Which represents the JDBC subsystem and provides methods to access the list of `JDBCDriverRuntimeMBeans` and `JDBCDataSourceRuntimeMBeans` currently available in the system.
- `JDBCDriverRuntimeMBean`—Which represents a JDBC driver that the server loaded into memory.
- `JDBCDataSourceRuntimeMBeans`—Which represents a JDBC data source deployed on a server or cluster.

Note: WebLogic JDBC runtime MBeans do *not* implement the optional Statistics Provider interfaces specified by JSR-77.

For more information about using the J2EE management model with WebLogic Server, see *Monitoring and Managing with the J2EE Management APIs*.

Overview of Clustered JDBC

You can target or deploy JDBC resources to a cluster to improve the availability of cluster-hosted applications. For information about JDBC objects in a clustered environment, see "JDBC Connections" in *Using WebLogic Server Clusters* at `http://e-docs.bea.com/wls/docs90/cluster/overview.html#JDBC`.

Multi data sources are supported for use in clusters. However, note that multi data sources can only use data sources in the same JVM. Multi data sources cannot use data sources from other cluster members.

Configuring JDBC Data Sources

This section includes the following information:

Understanding JDBC Data Sources

In WebLogic Server, you configure database connectivity by adding data sources to your WebLogic domain. Each data source contains a pool of database connections that are created when the data source is created and at server startup. Applications reserve a database connection from the data source by looking up the data source on the JNDI tree or in the local application context and then calling `getConnection()`. When finished with the connection, the application should call `connection.close()` as early as possible, which returns the database connection to the pool for other applications to use.

Data sources and their connection pools provide connection management processes that help keep your system running and performant.You can set options in the data source to suit your applications and your environment. The following sections describe these options and how to enable them.

Creating a JDBC Data Source

To create a JDBC data source using the Administration Console, follow these steps:

1. In the Change Center in the upper-left corner of the Administration Console window, click Lock & Edit to start a configuration editing session.
2. In the left pane, expand Services →JDBC, and click Data Sources.
3. On the Summary of JDBC Data Sources page, click New.
4. On the Create a New JDBC Data Source: JDBC Data Source Properties page, enter or select the following information:
   – Name: Enter a name for the data source configuration.
   – JNDI Name: Enter the name or path that applications will use to look up the data source on the JNDI tree.
   – Database Type: Select the vendor of your database management system.
   – Database Driver: Select the JDBC driver that you want to use to create database connections.

   Note: The JDBC driver you select must be listed in the Administration Server's CLASSPATH and in the CLASSPATH on all managed server on which you want to deploy the data source.

   Not all drivers listed in the Administration console are installed with WebLogic Server. They are listed for configuration convenience.
5. On the Transaction Options page, select the transaction options that are most appropriate for your system and DBMS. See "Transaction Options" on page 3-3 for more details.
6. On the Connection Properties page, enter the required information to connect to your database.

7. On the Test Database Connection page, review the connection information and optionally click Test Configuration.

8. On the Select Targets page, select the servers or clusters on which you want to deploy the data source. Select all targets on which you will deploy applications that will use the data source for database access.

   Note: If you are creating a JDBC module that you want to package with an enterprise application, do not select deployment targets.

9. After you finish configuring the data source, click Activate Changes in the Change Center.

Transaction Options

When you configure a JDBC data source using the Administration Console, WebLogic Server automatically selects specific transaction options based on the type of JDBC driver:

- **For XA drivers**, the system automatically selects the **Two-Phase Commit** protocol for global transaction processing. You can optionally select **Supports Local Transactions**, as well. See "Enabling Support for Local Transactions with an XA JDBC Driver" on page 3-4.

- **For non-XA drivers**, local transactions are supported by definition, and WebLogic Server offers the following options

  Supports Global Transactions: (selected by default) Select this option if you want to use connections from the data source in global transactions, even though you have not selected an XA driver. See "Enabling Support for Global Transactions with a Non-XA JDBC Driver" on page 3-4 for more information.

  When you select Supports Global Transactions, you must also select the protocol for WebLogic Server to for the transaction branch when processing a global transaction:

  – **Logging Last Resource**: With this option, the transaction branch in which the connection is used is processed as the last resource in the transaction and is processed as a one-phase commit operation. The result of the operation is written in a log file on the resource itself, and the result determines the success or failure of the prepare phase of the transaction. This option offers some performance benefits with greater data safety than Emulate Two-Phase Commit, but it has some limitations. See "Understanding the Logging Last Resource Transaction Option" on page 3-5.

  – **Emulate Two-Phase Commit**: With this option, the transaction branch in which the connection is used always returns success for the prepare phase of the transaction. It offers performance benefits, but also has risks to data in some failure conditions. Select this option only if your application can tolerate heuristic conditions. See "Understanding the Emulate Two-Phase Commit Transaction Option" on page 3-9.

– **One-Phase Commit**: (selected by default) With this option, a connection from the data source can be the only participant in the global transaction and the transaction is completed using a one-phase commit optimization. If more than one resource participates in the transaction, an exception is thrown. .

Enabling Support for Local Transactions with an XA JDBC Driver

When you configure a JDBC data source with an XA JDBC driver, you can optionally enable support for local transactions. You should select this option if the XA JDBC driver used in the data source to create database connection supports executing SQL code without a global transaction. Some XA drivers prevent you from executing SQL code without a global transaction, while others do not enforce that requirement.

Enabling Support for Global Transactions with a Non-XA JDBC Driver

If you use global transactions in your applications, you should use an XA JDBC driver to create database connections in the JDBC data source. If an XA driver is unavailable for your database, or you prefer not to use an XA driver, you should enable support for global transactions in the data source. You should also enable support for global transaction if your applications meet any of the following criteria:

- Use the EJB container in WebLogic Server to manage transactions
- Include multiple database updates within a single transaction
- Access multiple resources, such as a database and the Java Messaging Service (JMS), during a transaction
- Use the same connection pool on multiple servers

With an EJB architecture, it is common for multiple EJBs that are doing database work to be invoked as part of a single transaction. Without XA, the only way for this to work is if all transaction participants use the exact same database connection. When you enable global transactions and select either Logging Last Resource or Emulate Two-Phase Commit, WebLogic Server internally uses the JTS driver to make sure all EJBs use the same database connection within the same transaction context without requiring you to explicitly pass the connection from EJB to EJB.

If multiple EJBs are participating in a transaction and you do not use an XA JDBC driver for database connections, configure a Data Source with the following options:

- Supports Global Transactions selected
- Logging Last Resource or Emulate Two-Phase Commit selected

This configuration will force the JTS driver to internally use the same database connection for all database work within the same transaction.

With XA (requires an XA driver), EJBs can use a different database connection for each part of the transaction. Weblogic Server coordinates the transaction using the two-phase commit protocol, which guarantees that all or none transaction will be completed.

Understanding the Logging Last Resource Transaction Option

In WebLogic Server 9.0, you can configure a JDBC data source to enable the Logging Last Resource (LLR) transaction optimization, which enables one non-XA resource to participate in a global transaction with improved performance and with the same ACID guarantee as XA.

The LLR optimization improves performance by:

- Removing the need for an XA JDBC driver to connect to the database. XA JDBC drivers are typically inefficient compared to non-XA JDBC drivers.
- Reducing the number of processing steps to complete the transaction, which also reduces network traffic and the number of disk I/Os.
- Removing the need for XA processing at the database level.

When a connection from a data source configured for LLR participates in a global transaction, the WebLogic Server transaction manager completes the transaction by calling prepare on all other (XA-compliant) transaction participants, automatically inserting a commit record to a table on the LLR participant (rather than to the file-based transaction log), committing the LLR participant's local transaction (which includes both the transaction commit record insert and the application's SQL work), then calling commit on all other transaction participants.

The Logging Last Resource optimization maintains data integrity by writing the transaction log on the LLR participant. If the transaction fails during the local transaction commit, the WebLogic Server transaction manager will roll back the transaction on all other transaction participants. For failure recovery, the WebLogic Server transaction manager reads the transaction log on the LLR resource along with other transaction log files in the default store and completes any transaction processing as necessary. Work associated with XA participants is committed if a commit record exists, otherwise their work is rolled back

Advantages to Using the Logging Last Resource Optimization

Depending on your environment, you may want to consider the LLR transaction protocol in place of the two-phase commit protocol for transaction processing because of its performance benefits. The LLR transaction protocol offers the following advantages:

- Allows non-XA JDBC drivers and even non-XA–capable databases to safely participate in 2PC transactions.
- Eliminates the database's use of the XA protocol.
- Performs better than JDBC XA connections.
- Reduces the length of time that database row locks are held.
- Always commits database work prior to other XA work. Full-XA commits these operations in parallel, so, for example, when a JMS send participates, the JMS message may be delivered before database work commits.
- Has no increased risk of heuristic hazards, unlike the Emulate Two-Phase Commit option for a JDBC data source.

Enabling the Logging Last Resource Transaction Optimization

To enable the LLR transaction optimization, you create a JDBC data source with the Logging Last Resource transaction protocol, then use database connections from the data source in your applications. WebLogic Server automatically creates the required table on the database. See "Programming Considerations" on page 3-7 and "Administrative Considerations" on page 3-8 for details about connection request management and limitations with the Logging Last Resource transaction optimization.

To create an LLR-enabled JDBC data source using the Administration Console, follow these steps:

1. In the Change Center in the upper-left corner of the Administration Console window, click Lock & Edit to start a configuration editing session.
2. On the Summary of JDBC Data Sources page, click New.

3. On the Create a New JDBC Data Source page, enter or select the following information:
   – Name: Enter a name for the data source configuration.
   – JNDI Name: Enter the name or path that applications will use to look up the data source on the JNDI tree.
   – Database Type: Select the vendor of your database management system.
   – Database Driver: Select a non-XA JDBC driver for your DBMS.

   Note: You must select a non-XA driver for use with the LLR transaction protocol.
4. On the Transaction Options page, select the following options:
   – Supports Global Transactions
   – Logging Last Resource
5. On the Connection Properties page, enter the required information to connect to your database.
6. On the Select Targets page, select the servers or clusters on which you want to deploy the LLR-enabled data source. You should select all targets on which you will deploy applications that will use the data source for database access.
7. After you finish configuring the LLR-enabled data source, click Activate Changes in the Change Center.

Programming Considerations

You use JDBC connections from an LLR-enabled data source in an application as you would use JDBC connections from any other data source: after beginning a transaction, you look up the data source on the JNDI tree or in the local application context, then request a connection from the data source. However, with the LLR optimization, WebLogic Server internally manages the connection request in the following manner:

- Only one internal JDBC LLR connection is reserved per transaction.
- For additional JDBC connection requests within the transaction from a same-named data source, operations are routed to the reserved connection, even if the subsequent connection request is made on a different instance of the data source (i.e., a data source deployed on a different server than the original data source that supplied the connection for the first request).
- The reserved connection is hosted on the transaction's coordinator server.

Note the following limitations:

- Routed LLR connections may be less capable and less performant than locally hosted XA connections.
- Only one LLR or one "non-XA" resource may participate in a particular transaction, otherwise the transaction will roll back.

Administrative Considerations

- JDBC LLR transaction records are stored in a database table rather than the file-based transaction log (TLog)
- There is one LLR table per WL server
  - Multiple LLR pools may share a table
  - Automatically created if not found
  - Default name is "WL_LLR_<domain>_<server>"
  - Name is configurable via Server option "JDBCLLRTableName." This option is available in the Administration Console.
- Multiple servers must not share the same LLR table. Boot checks to ensure domain and server name match the domain and server name stored in the table when the table is created
- Server will not boot if LLR table is unreachable during boot. LLR records required to correctly resolve in-doubt transactions
- Transaction statistics track LLR resources under "NonXAResource"

To change the table name used to store transaction log records for the resource, follow these steps:

1. In the Change Center in the upper-left corner of the Administration Console window, click Lock & Edit to start a configuration editing session.
2. On the Server →Configuration →General page, click to Advanced to show the advanced configuration options.
3. In JDBC LLR Table Name, enter the name of the table to use to store transaction records for the resource, then click Save.
4. Repeat steps 2 and 3 for each server on which the LLR-enabled data source is deployed.
5. Click Activate Changes in the Change Center.

Note: You must restart all servers for the change to take effect.

Failover Considerations

- The file-based transaction log (TLog) is still required
  - TLog still stores transaction manager "checkpoint" records
  - TLog must still be reachable or copied on failover
- Unlike XA, LLR does not support "migratable targets"
  - In-doubt transactions may get resolved incorrectly, leading to "silent" heuristic hazards (mixed outcome transactions)
  - For server fail-over, use the new "whole-server" migration instead of migrating the transaction recovery service.

Understanding the Emulate Two-Phase Commit Transaction Option

If you need to support distributed transactions with a JDBC data source, but there is no available XA-compliant driver for your DBMS, you can select the Emulate Two-Phase Commit for non-XA Driver option for a data source to emulate two-phase commit for the transactions in which connections from the data source participate. This option is an advanced option on the JDBC Data Source →Configuration →General tab.

When the Emulate Two-Phase Commit for non-XA Driver option is selected (`EnableTwoPhaseCommit` is set to `true`), the non-XA JDBC resource always returns `XA_OK` during the `XAResource.prepare()` method call. The resource attempts to commit or roll back its local transaction in response to subsequent `XAResource.commit()` or `XAResource.rollback()` calls. If the resource commit or rollback fails, a heuristic error results. Application data may be left in an inconsistent state as a result of a heuristic failure.

When the Emulate Two-Phase Commit for non-XA Driver option is not selected in the Console (`EnableTwoPhaseCommit` is set to `false`), the non-XA JDBC resource causes `XAResource.prepare()` to fail. When there is only one resource participating in a transaction, the one phase optimization bypasses `XAResource.prepare()`, and the transaction commits successfully in most instances.

Note: There are risks to data integrity when using the Emulate Two-Phase Commit for non-XA Driver option. BEA recommends that you use an XA-compliant JDBC driver or the

Logging Last Resource option rather than use the Emulate Two-Phase Commit option. Make sure you consider the risks below before enabling this option.

This non-XA JDBC driver support is often referred to as the "JTS driver" because WebLogic Server uses the WebLogic JTS Driver internally to support the feature. For more information about the WebLogic JTS Driver, see "Using the WebLogic JTS Driver" in *Programming WebLogic JDBC*.

Limitations and Risks When Emulating Two-Phase Commit Using a Non-XA Driver

WebLogic Server supports the participation of non-XA JDBC resources in global transactions with the Emulate Two-Phase Commit data source transaction option, but there are limitations that you must consider when designing applications to use such resources. Because a non-XA driver does not adhere to the XA/2PC contracts and only supports one-phase commit and rollback operations, WebLogic Server (through the JTS driver) has to make compromises to allow the resource to participate in a transaction controlled by the Transaction Manager.

Consider the following limitations and risks before using the Emulate Two-Phase Commit for non-XA Driver option.

Heuristic Completions and Data Inconsistency

When Emulate Two-Phase Commit is selected for a non-XA resource, (`enableTwoPhaseCommit = true`), the prepare phase of the transaction for the non-XA resource always succeeds. Therefore, the non-XA resource does not truly participate in the two-phase commit (2PC) protocol and is susceptible to failures. If a failure occurs in the non-XA resource after the prepare phase, the non-XA resource is likely to roll back the transaction while XA transaction participants will commit the transaction, resulting in a heuristic completion and data inconsistencies.

Because of the data integrity risks, the Emulate Two-Phase Commit option should only be used in applications that can tolerate heuristic conditions.

Cannot Recover Pending Transactions

Because a non-XA driver manipulates local database transactions only, there is no concept of a transaction pending state in the database with regard to an external transaction manager. When `XAResource.recover()` is called on the non-XA resource, it always returns an empty set of Xids (transaction IDs), even though there may be transactions that need to be committed or rolled back. Therefore, applications that use a non-XA resource in a global transaction cannot recover from a system failure and maintain data integrity.

Possible Performance Loss with Non-XA Resources in Multi-Server Configurations

Because WebLogic Server relies on the database local transaction associated with a particular JDBC connection to support non-XA resource participation in a global transaction, when the same JDBC data source is accessed by an application with a global transaction context on multiple WebLogic Server instances, the JTS driver will always route JDBC operations to the first connection established by the application in the transaction. For example, if an application starts a transaction on one server, accesses a non-XA JDBC resource, then makes a remote method invocation (RMI) call to another server and accesses a data source that uses the same underlying JDBC driver, the JTS driver recognizes that the resource has a connection associated with the transaction on another server and sets up an RMI redirection to the actual connection on the first server. All operations on the connection are made on the one connection that was established on the first server. This behavior can result in a performance loss due to the overhead associated with setting up these remote connections and making the RMI calls to the one physical connection.

Only One Non-XA Participant

When a non-XA resource (with Emulate Two-Phase Commit selected) is registered with the WebLogic Server Transaction Manager, it is registered with the name of the class that implements the XAResource interface. Since all non-XA resources with Emulate Two-Phase Commit selected use the JTS driver for the XAResource interface, all non-XA resources (with Emulate Two-Phase Commit selected) that participate in a global transaction are registered with the same name. If you use more than one non-XA resource in a global transaction, you will see naming conflicts or possible heuristic failures.

Binding a Data Source to the JNDI Tree with Multiple Names

In WebLogic Server 9.0 and later releases, you can configure a data source so that it binds to the JNDI tree with multiple names. You can use a multi-named data source in place of legacy configurations that included multiple data sources that pointed to a single JDBC connection pool.

To add JNDI names to an existing data source, add names to the JNDI Name attribute separated by semi-colons. You must either restart the system after making your change or undeploy the data source before making the change, and then redeploy after making the change. Follow the instructions below.

1. On the JDBC Data Source →Configuration →General page in the Administration Console, in JNDI Name, enter the names you want to use to bind the data source to the JDNI tree separated by semi-colons (;). For example:

```
name1;name2;name3
```

2. Click Save.

After you activate your changes, you will need to redeploy the data source or restart your server before the changes will take effect.

Connection Pool Features

Database Passwords in a Data Source Connection Pool Configuration

When you create a connection pool, you typically include a password to connect to the database. You can enter the password as a name-value pair in the `Properties` field or you can enter it in the Password field:

- **`Password`**. Use this field to set the database password. This value overrides any `password` value defined in the `Properties` passed to the tier-2 JDBC Driver when creating physical database connections. BEA recommends that you use the Password attribute in place of the password property in the properties string because the value is encrypted in the `config.xml` file (stored as the `Password` attribute in the `JDBCConnectionPool` tag) and is hidden on the administration console.

If you specify a password in the `Properties` field when you first configure the connection pool, WebLogic Server removes the password from the `Properties` string and sets the value as the `Password` value in an encrypted form the next time you start WebLogic Server. If there is already a value for the `Password` attribute for the connection pool, WebLogic Server does not change any values. However, the value for the `Password` attribute overrides the password value in the `Properties` string. For example, if you include the following properties when you first configure a connection pool:

```
user=scott;
password=tiger;
```

The next time you start WebLogic Server, it moves the database password to the Password attribute, and the following value remains for the `Properties` field:

```
user=scott;
```

After a value is established for the `Password` attribute, the values in these attributes override the respective values in the `Properties` attribute. That is, continuing with the previous example, if you specify `tiger2` as the database password in the `Properties` attribute, WebLogic Server ignores the value and continues to use `tiger` as the database password, which is the current encrypted value of the `Password` attribute. To change the database password, you must change the `Password` attribute.

JDBC Connection Pool Statement Timeout Options

The following attributes enable you to limit the amount of time that a statement takes to execute on a database connection from a JDBC connection pool:

- `StatementTimeout`—The time in seconds after which a statement executing on a pooled JDBC connection times out. When set to -1, (the default) statements do not timeout.
- `TestStatementTimeout`—The time in seconds after which a statement executing on a pooled JDBC connection for connection initialization or testing times out. When set to -1, (the default) statements do not timeout.

These attributes rely on underlying JDBC driver support.

Configuring Credential Mapping for a Data Source

Credential mapping for a JDBC data source is the process in which WebLogic Server user IDs are mapped to database user IDs. If credential mapping is enabled on the data source, when an application requests a database connection from the data source, WebLogic Server determines the current WebLogic Server user ID and then sets the mapped database ID as a light-weight client ID on the database connection.

This feature relies on features in the JDBC driver and DBMS. It is only supported for use with Oracle and DB2 databases and with the Oracle Thin and DB2 UDB JDBC drivers, respectively.

To enable this feature, follow these steps:

1. Enable credential mapping on the data source:

   On the JDBC Data Source →Configuration →Connection Pool page in the Administration Console, select Enable Credential Mapping. Note that Enable Credential Mapping is under Advanced settings.

2. Create the mapping entries of WebLogic Server user IDs to database user IDs:

   On the JDBC Data Source →Security →Credential Mapping page in the Administration Console, click New to enter a new credential map entry, then enter the WebLogic Server user ID and the database user ID and password that it maps to. Repeat this process for all WebLogic Server user IDs that you need to map to database user IDs.

Avoiding Server Lockup with the Correct Number of Connections

When your applications attempt to get a connection from a connection pool in which there are no available connections, the connection pool throws an exception stating that a connection is not available in the connection pool. To avoid this error, make sure your connection pool can expand to the size required to accommodate your peak load of connection requests.

JDBC Connection Pool Testing Enhancements

The following attributes help to improve the functionality of database connection testing for pooled connections:

- `CountOfTestFailuresTillFlush`—Closes all connections in the connection pool after the number of test failures that you specify to minimize the delay caused by further database testing. See "Minimizing Connection Test Delay After Database Connectivity Loss."
- `CountOfRefreshFailuresTillDisable`—Disables the connection pool after the number of test failures that you specify to minimize the delay in handling the connection request after a database failure. See "Minimizing Connection Request Delay After Connection Test Failures."
- `SecondsToTrustAnIdlePoolConnection`—Enables skipping a connection test if the connection was recently used or tested successfully. See "Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection" on page 3-15.

Minimizing Connection Test Delay After Database Connectivity Loss

When connectivity to the DBMS is lost, even if only momentarily, some or all of the JDBC connections in a connection pool typically become defunct. If the connection pool is configured to test connections on reserve (recommended), when an application requests a database connection, WebLogic Server tests the connection, discovers that the connection is dead, and tries to replace it with a new connection to satisfy the request. Ordinarily, when the DBMS comes back online, the refresh process succeeds. However, in some cases and for some modes of failure, testing a dead connection can impose a long delay. This delay occurs for each dead connection in the connection pool until all connections are replaced.

To minimize the delay that occurs during the test of dead database connections, you can set the `CountOfTestFailuresTillFlush` attribute on the connection pool. With this attribute set, WebLogic Server considers *all* connections in the connection pool as dead after the number of consecutive test failures that you specify, and closes all connections in the connection pool.

When an application requests a connection, the connection pool creates a connection without first having to test a dead connection. This behavior minimizes the delay for connection requests following the connection pool flush.

If you tend to see small network glitches or have a firewall that may occasionally kill only one socket or connection, you may want to set the number of test failures to 2 or 3, but a value of 1 will provide the best performance after database availability issues have been resolved.

Minimizing Connection Request Delay After Connection Test Failures

If your DBMS becomes and remains unavailable, the connection pool will persistently test and try to replace dead connections while trying to satisfy connection requests. This behavior is beneficial because it enables the connection pool to react immediately when the database becomes available. However, testing a dead database connection can take as long as the network timeout, and can cause a long delay for clients.

To minimize the delay that occurs for client applications while a database is unavailable, you can set the `CountOfRefreshFailuresTillDisable` attribute on the connection pool. With this attribute set, WebLogic Server disables the connection pool after the number of consecutive failures to replace a dead connection. When an application requests a connection from a disabled connection pool, WebLogic Server throws a `ConnectDisabledException` immediately to notify the client that a connection is not available.

For connection pools that are disabled in this manner, WebLogic Server periodically run the refresh process. When the refresh process succeeds in creating a new database connection, WebLogic Server re-enables the connection pool. You can also manually re-enable the connection pool using the administration console or the `weblogic.Admin ENABLE_POOL` command.

If you tend to see small network glitches or have a firewall that may occasionally kill only one socket or connection, you may want to set the number of refresh failures to 2 or 3, but a value of 1 will usually provide the best performance.

Minimizing Connection Request Delay with Seconds to Trust an Idle Pool Connection

Database connection testing during heavy traffic can reduce application performance. To minimize the impact of connection testing, you can set the

SecondsToTrustAnIdlePoolConnection attribute in the JDBC connection pool configuration to trust recently-used or recently-tested database connections and skip the connection test.

If your connection pool is configured to test connections on reserve (recommended), when an application requests a database connection, WebLogic Server tests the database connection before giving it to the application. If the request is made within the time specified for SecondsToTrustAnIdlePoolConnection since the connection was tested or successfully used and returned to the connection pool, WebLogic Server skips the connection test before delivering it to an application.

If your connection pool is configured to periodically test available connections in the connection pool (TestFrequencySeconds is specified), WebLogic Server also skips the connection test if the connection was successfully used and returned to the connection pool within the time specified for SecondsToTrustAnIdlePoolConnection.

SecondsToTrustAnIdlePoolConnection is a tuning feature that can improve application performance by minimizing the delay caused by database connection testing, especially during heavy traffic. However, it can reduce the effectiveness of connection testing, especially if the value is set too high. The appropriate value depends on your environment and the likelihood that a connection will become defunct.

JDBC Data Source Factories (Deprecated)

In previous releases of WebLogic Server, application-scoped JDBC connection pools relied on JDBC data source factories to provide default connection pool values. JDBC data source factories are deprecated in WebLogic Server 9.0 and are included in the release for backward compatibility only. Application-scoped JDBC connection pools are replaced by JDBC application modules. For more information, see "Application or Global Scoping for a Packaged JDBC Module" on page A-6.

Tuning Data Source Connection Pools

By properly configuring the JDBC data source connection pools in your WebLogic Server domain, you can improve application and system performance.

Enabling Connection Requests to Wait for a Connection

On the JDBC→Connection Pool→Configuration→Connections tab, there are two attributes that you can set to enable connection requests to wait for a connection from a connection pool: Connection Reserve Timeout and Maximum Waiting for Connection.

Connection Reserve Timeout

When an application requests a connection from a connection pool, if all connections in the connection pool are in use and if the connection pool has expanded to its maximum capacity, you can configure a Connection Reserve Timeout value (in seconds) so that connection requests will wait for a connection to become available. After the Connection Reserve Timeout has expired, if no connection was has become available, the request will fail.

If you set Connection Reserve Timeout to `-1`, a connection request will wait indefinitely.

Maximum Waiting for Connection

Note that connection requests that wait for a connection block a thread. If too many connection requests concurrently wait for a connection and block threads, your system performance can degrade. To avoid this, you can set the Maximum Waiting for Connection attribute, which limits the number connection requests that can concurrently wait for a connection.

If you set Maximum Waiting for Connection to `0`, the feature is disabled and connection requests will not be able to wait for a connection.

To Enable a Connection Request to Wait for a Connection

1. In the left pane, click to expand the Services, JDBC, and Connection Pool nodes to display the list of connection pools in the current domain.
2. Click the connection pool that you want to configure. A dialog displays in the right pane showing the tabs associated with this instance.
3. Click the Configuration tab, then click the Connections tab.
4. Click Show to show the advanced connection options.
5. In Connection Reserve Timeout, enter the number of seconds that connection requests can wait for a connection.
6. In Maximum Waiting for a Connection, enter the maximum number of connection requests that can wait for a connection from the connection pool while blocking threads.

7. Click Apply.

Initializing Database Connections with SQL Code

When WebLogic Server creates database connections in a connection pool, the server can automatically run SQL code to initialize the database connection. To enable this feature, enter `SQL` followed by a space and the SQL code you want to run in the Init SQL attribute on the JDBC Data Source →Configuration →Connection Pool page in the Administration console. If you leave this attribute blank (the default), WebLogic Server does not run any code to initialize database connections.

WebLogic Server runs this code whenever it creates a database connection for the connection pool, which includes at server startup, when expanding the connection pool, and when refreshing a connection.

To initialize a database connection with SQL code:

1. In the left pane, click to expand the Services, JDBC, and Connection Pool nodes to display the list of connection pools in the current domain.
2. Click the connection pool that you want to configure. A dialog displays in the right pane showing the tabs associated with this instance.
3. Click the Configuration tab, then click the Connections tab.
4. Click Show to show the advanced connection options.
5. In Init SQL, enter `SQL` followed by a space and the SQL code you want to run to initialize database connections.

Connection Testing Options

To make sure that the database connections in a connection pool remain healthy, you should periodically test the connections. WebLogic Server includes two basic types of testing: automatic testing that you configure with options on the JDBC→Connection Pool→Configuration→Connections tab and manual testing that you can do to trouble-shoot a connection pool from the JDBC→Connection Pool→Testing tab. The following section discusses automatic connection testing options.

On the JDBC→Connection Pool→Configuration→Connections, you use the following settings to configure connection testing:

- Test Frequency—Use this attribute to specify the number of seconds between tests of unused connections. The server tests unused connection and reopens any faulty connections. You must also set the Maximum Connections Made Unavailable and a Test Table Name.
- Test Reserved Connections—Select this option to test each connection before giving to a client. This may add a slight delay to the request, but it guarantees that the connection is healthy. You must also set a Test Table Name.
- Test Created Connections—Select this option to test each database connection after it is created. This applies to connections created at server startup and when the connection pool is expanded. You must also set a Test Table Name.
- Test Released Connections—Select this option to test connections when they are returned to the connection pool. You must also set a Test Table Name.
- Maximum Connections Made Unavailable—Use this option to limit the number idle connections that the server will test. For example, if you have 10 connections in your connection pool and five are in use, if the server were to begin testing all five connections that are not in use, there would be no connections available to fill a connection request. If you set the Maximum Connections Made Unavailable attribute to `3`, there would still be two connections available to fill a connection request.
- Test Table Name—Use this attribute to specify a table name to use in a connection test. You can also specify SQL code to run in place of the standard test by entering `SQL` followed by a space and the SQL code you want to run as a test. Test Table Name is required to enable any database connection testing.

You should set connection testing attributes so that they best fit your environment.

To enable a database connection testing:

1. In the left pane, click to expand the Services, JDBC, and Connection Pool nodes to display the list of connection pools in the current domain.
2. Click the connection pool that you want to configure. A dialog displays in the right pane showing the tabs associated with this instance.
3. Click the Configuration tab, then click the Connections tab.
4. Click Show to show the advanced connection options.
5. Select or specify a value for at least one of the following attributes:
   – Test Frequency

  - Test Reserved Connections
  - Test Created Connections
  - Test Released Connections

  If you specify a value for Test Frequency, you must also specify a value for Maximum Connections Made Unavailable.

6. Specify a value for Test Table Name: either a table that exists in your database or `SQL` followed by a space and SQL code.
7. Click Apply.

Default Test Table Name

When you create a connection pool, the Administration Console automatically sets the Test Table Name attribute for a connection pool based on the DBMS of the JDBC driver that you select. The Test Table Name attribute is used in connection testing which is optionally performed periodically or when you create, reserve, or release a connection, depending on how you configure the connection pool. For database tests to succeed, the database user used to create database connections in the connection pool must have access to the database table. If not, you should either grant access to the user (make this change in the DBMS) or change the Test Table Name attribute to the name of a table to which the user does have access (make this change in the WebLogic Server Administration Console).

Table 3-1 Default Test Table Name by DBMS

DBMS	Default Test Table Name (Query)
Cloudscape	SQL SELECT 1
DB2	SQL SELECT COUNT(*) FROM SYSIBM.SYSTABLES
Informix	SQL SELECT COUNT(*) FROM SYSTABLES
Microsoft SQL Server	SQL SELECT COUNT(*) FROM SYSOBJECTS
MySQL	SQL SELECT 1
Oracle	SQL SELECT 1 FROM DUAL
PointBase	SQL SELECT COUNT(*) FROM SYSTABLES
PostgreSQL	SQL SELECT 1

Table 3-1 Default Test Table Name by DBMS

DBMS	Default Test Table Name (Query)
Progress	SQL SELECT COUNT(*) FROM SYSTABLES
Sybase	SQL SELECT COUNT(*) FROM SYSOBJECTS

Increasing Performance with the Statement Cache

When you use a prepared statement or callable statement in an application or EJB, there is considerable processing overhead for the communication between the application server and the database server and on the database server itself. To minimize the processing costs, WebLogic Server can cache prepared and callable statements used in your applications. When an application or EJB calls any of the statements stored in the cache, WebLogic Server reuses the statement stored in the cache. Reusing prepared and callable statements reduces CPU usage on the database server, improving performance for the current statement and leaving CPU cycles for other tasks.

Each connection in a connection pool has its own individual cache of prepared and callable statements used on the connection. However, you configure statement cache options per connection pool. That is, the statement cache for each connection in a connection pool uses the statement cache options specified for the connection pool. Statement cache configuration options include:

- **Statement Cache Type**—The algorithm that determines which statements to store in the statement cache. See "Statement Cache Algorithms" on page 3-22.
- **Statement Cache Size**—The number of statements to store in the cache for each connection. The default value is `10`. See "Statement Cache Size" on page 3-22.

You can use the following methods to set statement cache options for a connection pool:

- Using the Administration Console (preferred). See "Configuring the Statement Cache for a JDBC Data Source" on page 3-24.
- Using the WebLogic management API. See the following methods:
  – `getStatementCacheType()`
  – `setStatementCacheType(string type)`
  – `getStatementCacheSize()`
  – `setStatementCacheSize(int cacheSize)`

Statement Cache Algorithms

The Statement Cache Type (or algorithm) determines which prepared and callable statements to store in the cache for each connection in a connection pool. You can choose from the following options:

- LRU (Least Recently Used)
- Fixed

LRU (Least Recently Used)

When you select LRU (Least Recently Used, the default) as the Statement Cache Type, WebLogic Server caches prepared and callable statements used on the connection until the statement cache size is reached. When an application calls `Connection.prepareStatement()`, WebLogic Server checks to see if the statement is stored in the statement cache. If so, WebLogic Server returns the cached statement (if it is not already being used). If the statement is not in the cache, and the cache is full (number of statements in the cache = statement cache size), Weblogic Server determines which existing statement in the cache was the least recently used and replaces that statement in the cache with the new statement.

The LRU statement cache algorithm in WebLogic Server uses an approximate LRU scheme.

Fixed

When you select FIXED as the Statement Cache Type, WebLogic Server caches prepared and callable statements used on the connection until the statement cache size is reached. When additional statements are used, they are not cached.

With this statement cache algorithm, you can inadvertently cache statements that are rarely used. In many cases, the LRU algorithm is preferred because rarely used statements will eventually be replaced in the cache with frequently used statements.

Statement Cache Size

The Statement Cache Size attribute determines the total number of prepared and callable statements to cache for each connection in each instance of the connection pool. By caching statements, you can increase your system performance. However, you must consider how your DBMS handles open prepared and callable statements. In many cases, the DBMS will maintain a cursor for each open statement. This applies to prepared and callable statements in the statement cache. If you cache too many statements, you may exceed the limit of open cursors on your database server.

For example, if you have a connection pool with 10 connections deployed on 2 servers, if you set the Statement Cache Size to 10 (the default), you may open 200 (10 x 2 x 10) cursors on your database server for the cached statements.

Usage Restrictions for the Statement Cache

Using the statement cache can dramatically increase performance, but you must consider its limitations before you decide to use it. Please note the following restrictions when using the statement cache.

There may be other issues related to caching statements that are not listed here. If you see errors in your system related to prepared or callable statements, you should set the statement cache size to `0`, which turns off statement caching, to test if the problem is caused by caching prepared statements.

Calling a Stored Statement After a Database Change May Cause Errors

Prepared statements stored in the cache refer to specific database objects at the time the prepared statement is cached. If you perform any DDL (data definition language) operations on database objects referenced in prepared statements stored in the cache, the statements may fail the next time you run them. For example, if you cache a statement such as `select * from emp` and then drop and recreate the `emp` table, the next time you run the cached statement, the statement may fail because the exact `emp` table that existed when the statement was prepared, no longer exists.

Likewise, prepared statements are bound to the data type for each column in a table in the database at the time the prepared statement is cached. If you add, delete, or rearrange columns in a table, prepared statements stored in the cache are likely to fail when run again.

These limitations depend on the behavior of your DBMS.

Using setNull In a Prepared Statement

If you cache a prepared statement that uses a `setNull` bind variable, you must set the variable to the proper data type. If you use a generic data type, as in the following example, data may be truncated or the statement may fail when it runs with a value other than null.

```
java.sql.Types.Long sal=null
.
.
.
if (sal == null)
    setNull(2,int)//This is incorrect
```

```
else
    setLong(2,sal)
```

Instead, use the following:

```
if (sal == null)
    setNull(2,long)//This is correct
else
    setLong(2,sal)
```

Statements in the Cache May Reserve Database Cursors

When WebLogic Server caches a prepared or callable statement, the statement may open a cursor in the database. If you cache too many statements, you may exceed the limit of open cursors for a connection. To avoid exceeding the limit of open cursors for a connection, you can change the limit in your database management system or you can reduce the statement cache size for the connection pool.

Configuring the Statement Cache for a JDBC Data Source

On the JDBC Data Source →Configuration →Connection Pool page in the Administration Console, , you can configure statement cache attributes for connection in the connection pool within a JDBC data source. For more information about the statement cache, see "Increasing Performance with the Statement Cache" on page 3-21. To configure the statement cache, follow these steps:

1. On the JDBC Data Source →Configuration →Connection Pool page in the Administration Console, in Statement Cache Type, select one of the following options:
   – LRU - After the `statementCacheSize` is met, the Least Recently Used statement is removed when a new statement is used.
   – Fixed - The first `statementCacheSize` number of statements is stored and stay fixed in the cache. No new statements are cached unless the cache is manually cleared or the cache size is increased.

   See "Statement Cache Algorithms" on page 3-22 for more information.
2. In Statement Cache Size, enter the number of statements to cache per connection per connection pool instance. The default value is `10`. See "Statement Cache Size" on page 3-22 for more information.
3. Click Save to save your changes.

Configuring JDBC Multi Data Sources

A *multi data source* is a pool of data sources. Multi data sources aid in either:

- Load Balancing—data sources are accessed using a round-robin scheme. When switching connections, WebLogic Server selects a connection from the next data source in the order listed.
- Failover—data sources are listed in the order that determines the order in which connection pool switching occurs. That is, WebLogic Server provides database connections from the first connection pool on the list. If that connection pool fails, it attempts to use a database connection from the second, and so forth.

Configuring and Using Multi Data Sources

A multi data source is a "pool of pools." Multi data sources contain a configurable algorithm for choosing which connection pool will return a connection to the client.

You create a multi data source by first creating connection pools, then creating the multi data source using the Administration Console or WebLogic Management API and assigning the data sources to the multi data source.

Multi Data Source Features

A multi data source is a pool of data sources. All the connections in the connection pool in a particular *data source* are created identically with a single database, single user, and the same connection attributes; that is, they are attached to a single database. However, the connection pools within the data sources within a *multi data source* may be associated with different users, databases, or DBMSs.

Choosing the Multi Data Source Algorithm

Before you set up a multi data source, you need to determine the primary purpose of the multi data source—high availability or load balancing. You can choose the algorithm that corresponds with your requirements.

High Availability

The High Availability algorithm provides an ordered list of connection pools. Normally, every connection request to this kind of multi data source is served by the first pool in the list. If a database connection test fails and the connection cannot be replaced, or if the connection pool is suspended, a connection is sought sequentially from the next pool on the list.

Note: This algorithm relies on `TestConnectionsOnReserve` to test to see if a connection in the first connection pool is healthy. If the connection fails the test, the multi data source uses a connection from the next connection pool in the multi data source.

Load Balancing

Connection requests to a load balancing multi data source are served from any connection pool in the list. Pools are accessed using a round-robin scheme. When the multi data source provides a connection, it selects a connection from the connection pool listed just after the last pool that was used to provide a connection. multi data sources that use the Load Balancing algorithm also fail over to the next connection pool in the list if a database connection test fails and the connection cannot be replaced, or if the connection pool is suspended.

Multi Data Source Fail-Over Limitations and Requirements

WebLogic Server provides the High Availability algorithm for multi data sources so that if a connection pool fails (for example, if the database management system crashes), your system can continue to operate. However, you must consider the following limitations and requirements when configuring your system.

Test Connections on Reserve to Enable Fail-Over

Connection pools rely on the `TestConnectionsOnReserve` feature to know when database connectivity is lost. Connections are *not* automatically tested before being reserved by an application. You must set `TestConnectionsOnReserve=true` for the connection pools within the multi data source. After turning on this feature, WebLogic Server will test each connection before returning it to an application, which is crucial to the High Availability algorithm operation. With the High Availability algorithm, the multi data source uses the results from testing connections on reserve to determine when to fail over to the next connection pool in the multi data source. After a test failure, the connection pool attempts to recreate the connection. If that attempt fails, the multi data source fails over to the next connection pool.

Connection Requests are Always Routed to Data Sources in Order

When an application requests a connection from a multi data source, the request is routed to connection pools in the list in order until a connection is available or until the last connection pool in the list, even if the multi data source is aware that the first "n" (1 or more) connection pools are unavailable (dead or disabled). When connection pools in the multi data source are unavailable, there is a performance cost to every application until the first connection pool in the list is restored.

Do Not Enable Connection Creation Retries

Do not enable connection creation retries with connection pools in a High Availability multi data source. Connection requests to the multi data source will fail (not fail-over) when a connection pool in the list is dead and the number of connection requests equals the number of connections in the first connection pool, even if connections are available in subsequent connection pools in the multi data source.

Multi data sources and the connection creation retries feature both attempt to solve the same problem—to gracefully handle database connections when a database is unavailable. If you use these two features together, their functionality will interfere with each other.

No Fail-Over for In-Use Connections

It is possible for a connection to fail after being reserved, in which case your application must handle the failure. WebLogic Server cannot provide fail-over for connections that fail while being used by an application. Any failure while using a connection requires that you restart the transaction and provide code to handle such a failure.

Multi Data Source Failover Enhancements

In WebLogic Server 9.0, the following enhancements were made to multi data sources:

- Connection request routing enhancements to avoid requesting a connection from an automatically disabled (dead) connection pool within a multi data source. See "Connection Request Routing Enhancements When a Connection Pool Fails."
- Automatic failback on recovery of a failed connection pool within a multi data source. See "Automatic Re-enablement on Recovery of a Failed Connection Pool within a Multi Data Source."
- Failover for busy connection pools within a multi data sources. See "Enabling Failover for Busy Connection Pools in a Multi Data Source."
- Failover callbacks for multi data sources with the High Availability algorithm. See "Controlling Multi Data Source Failover with a Callback."
- Failback callbacks for multi data sources with either algorithm. See "Controlling Multi Data Source Failback with a Callback."

Connection Request Routing Enhancements When a Connection Pool Fails

To improve performance when a connection pool within a multi data source fails, WebLogic Server automatically disables the connection pool when a pooled connection fails a connection test. After a connection pool is disabled, WebLogic Server does not route connection requests from applications to the connection pool. Instead, it routes connection requests to the next available connection pool listed in the multi data source.

This feature requires that connection pool testing options are configured for all connection pools in a multi data source, specifically `TestTableName` and `TestConnectionsOnReserve`.

If a callback handler is registered for the multi data source, WebLogic Server calls the callback handler before failing over to the next connection pool in the list. See "Controlling Multi Data Source Failover with a Callback" on page 4-6 for more details.

Automatic Re-enablement on Recovery of a Failed Connection Pool within a Multi Data Source

After a connection pool is automatically disabled because a connection failed a connection test, WebLogic Server periodically tests a connection from the disabled connection pool to determine when the connection pool (or underlying database) is available again. When the connection pool becomes available, WebLogic Server automatically re-enables the connection pool and resumes routing connection requests to the connection pool, depending on the multi data source algorithm and the position of the connection pool in the list of included connection pools.

To control how often WebLogic Server checks automatically disabled connection pools in a multi data source, you add a value for the `HealthCheckFrequencySeconds` attribute to the multi data source configuration in the `config.xml` file. For example:

```
<JDBCMultiDataSource
AlgorithmType="High-Availability"
Name="demoMultiDataSource"
PoolList="demoPool2,demoPool"
HealthCheckFrequencySeconds="240"
Targets="examplesServer" />
```

Note: This attribute is not available in the administration console. To implement this functionality, you must manually add the attribute to the multi data source configuration in the `config.xml` file.

WebLogic Server waits for the period you specify between connection tests for each disabled connection pool. The default value is 300 seconds. If you do not specify a value, WebLogic Server will test automatically disabled connection pools every 300 seconds.

This feature requires that connection pool testing options are configured for all connection pools in a multi data source, specifically `TestTableName` and `TestConnectionsOnReserve`.

WebLogic Server does not test and automatically re-enable connection pools that you manually disable. It only tests connection pools that it automatically disables.

If a callback handler is registered for the multi data source, WebLogic Server calls the callback handler before re-enabling the connection pool. See "Controlling Multi Data Source Failback with a Callback" on page 4-9 for more details.

Enabling Failover for Busy Connection Pools in a Multi Data Source

By default, for multi data sources with the High Availability algorithm, when the number of requests for a database connection exceeds the number of available connections in the current connection pool in the multi data source, subsequent connection requests fail.

To enable the multi data source to failover when all connections in the current connection pool are in use, you must set a value for the `FailoverRequestIfBusy` attribute in the multi data source configuration in the `config.xml` file. If set to `true`, when all connections in the current connection pool are in use, application requests for connections will be routed to the next available connection pool within the multi data source. When set to `false` (the default), connection requests do not failover.[Which exception is thrown?]

After you add the `FailoverRequestIfBusy` attribute to the `config.xml` file, the multi data source entry may look like the following:

```
<JDBCMultiDataSource
AlgorithmType="High-Availability"
Name="demoMultiDataSource"
PoolList="demoPool2,demoPool"
FailoverRequestIfBusy="true"
Targets="examplesServer" />
```

Note: The `FailoverRequestIfBusy` attributes is not available in the administration console. To implement this functionality, you must manually add this attribute to the multi data source configuration in the `config.xml` file.

If a `ConnectionPoolFailoverCallbackHandler` is included in the multi data source configuration, WebLogic Server calls the callback handler before failing over. See "Controlling Multi Data Source Failover with a Callback" on page 4-6 for more details.

Controlling Multi Data Source Failover with a Callback

You can register a callback handler with WebLogic Server that controls when a multi data source with the High-Availability algorithm fails over connection requests from one JDBC connection pool in the multi data source to the next connection pool in the list.

You can use callback handlers to control if or when the failover occurs so that you can make any other system preparations before the failover, such as priming a database or communicating with a high-availability framework.

Callback handlers are registered via an attribute of the multi data source in the `config.xml` file and are registered per multi data source. Therefore, you must register a callback handler for each multi data source to which you want the callback to apply. And you can register different callback handlers for each multi data source.

Callback Handler Requirements

A callback handler used to control the failover and failback within a multi data source must include an implementation of the `weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface. When the multi data source needs to failover to the next connection pool in the list or when a previously disabled connection pool becomes available, WebLogic Server calls the `allowPoolFailover()` method in the `ConnectionPoolFailoverCallback` interface, and passes a value for the three parameters, `currPool`, `nextPool`, and `opcode`, as defined below. WebLogic Server then waits for the return from the callback handler before completing the task.

Your application must return OK, RETRY_CURRENT, or DONOT_FAILOVER as defined below.The application should handle failover and failback cases.

See the Javadoc for the `weblogic.jdbc.extensions.ConnectionPoolFailoverCallback` interface for more details. [Add link]

Note: Failover callback handlers are optional.If no callback handler is specified in the multi data source configuration, WebLogic Server proceeds with the operation (failing over or re-enabling the disabled connection pool).

Callback Handler Configuration

There are two multi data source configuration attributes associated with the failover and failback functionality:

- `ConnectionPoolFailoverCallbackHandler`—To register a failover callback handler for a multi data source, you add a value for this attribute to the multi data source configuration in the `config.xml` file. The value must be an absolute name, such as `com.bea.samples.wls.jdbc.MultiDataSourceFailoverCallbackApplication`.
- `HealthCheckFrequencySeconds`—To control how often WebLogic Server checks disabled (dead) connection pools in a multi data source to see if they are now available, you can add a value for this attribute to the multi data source configuration in the `config.xml` file. See "Automatic Re-enablement on Recovery of a Failed Connection Pool within a Multi Data Source" on page 4-4 for more details.

After you add the attributes to the `config.xml` file, the multi data source entry may look like the following:

```
<JDBCMultiDataSource
AlgorithmType="High-Availability"
Name="demoMultiDataSource"
ConnectionPoolFailoverCallbackHandler="com.bea.samples.wls.jdbc.MultiDataS
ourceFailoverCallbackApplication"
PoolList="demoPool2,demoPool"
HealthCheckFrequencySeconds="120"
Targets="examplesServer" />
```

Note: These attributes are not available in the administration console. To implement this functionality, you must manually add these attributes to the multi data source configuration in the `config.xml` file.

How It Works—Failover

WebLogic Server attempts to failover connection requests to the next connection pool in the list when the current connection pool fails a connection test or, if you enabled `FailoverRequestIfBusy`, when all connections in the current connection pool are busy.

To enable the callback feature, you register the callback handler with Weblogic Server using the `ConnectionPoolFailoverCallbackHandler` attribute in the multi data source configuration in the `config.xml` file.

With the High Availability algorithm, connection requests are served from the first connection pool in the list. If a connection from that connection pool fails a connection test, WebLogic Server marks the connection pool as dead and disables it. If a callback handler is registered, WebLogic Server calls the callback handler, passing the following information, and waits for a return:

- `currPool`—For failover, this is the name of connection pool currently being used to supply database connections. This is the "failover from" connection pool.
- `nextPool`—The name of next available connection pool listed in the multi data source. For failover, this is the "failover to" connection pool.
- `opcode`—A code that indicates the reason for the call:
  - `OPCODE_CURR_POOL_DEAD`—WebLogic Server determined that the current connection pool is dead and has disabled it.
  - `OPCODE_CURR_POOL_BUSY`—All database connections in the connection pool are in use. (Requires `FailoverIfBusy=true` in the multi data source configuration. See "Enabling Failover for Busy Connection Pools in a Multi Data Source" on page 4-5.)

Failover is synchronous with the connection request: Failover occurs only when WebLogic Server is attempting to satisfy a connection request.

The return from the callback handler can indicate one of three options:

- `OK`—proceed with the operation. In this case, that means to failover to the next connection pool in the list.
- `RETRY_CURRENT`—Retry the connection request with the current connection pool.
- `DONOT_FAILOVER`—Do not retry the current connection request and do not failover. WebLogic Server will throw a `weblogic.jdbc.extensions.PoolUnavailableSQLException`.

WebLogic Server acts according to the value returned by the callback handler.

If the secondary connection pools fails, WebLogic Server calls the callback handler again, as in the previous failover, in an attempt to failover to the next available connection pool in the multi data source, if there is one.

Note: WebLogic Server does *not* call the callback handler when you manually disable a connection pool.

For multi data sources with the Load-Balancing algorithm, WebLogic Server does not call the callback handler when a connection pool is disabled. However, it does call the callback handler when attempting to re-enable a disabled connection pool. See the following section for more details.

Controlling Multi Data Source Failback with a Callback

If you register a failover callback handler for a multi data source, WebLogic Server calls the same callback handler when re-enabling a connection pool that was automatically disabled. You can use the callback to control if or when the disabled connection pool is re-enabled so that you can make any other system preparations before the connection pool is re-enabled, such as priming a database or communicating with a high-availability framework.

Callback handlers are registered via an attribute of the multi data source in the `config.xml` file and are registered per multi data source. Therefore, you must register a callback handler for each multi data source to which you want the callback to apply. And you can register different callback handlers for each multi data source.

See the following sections for more details about the callback handler:

- "Callback Handler Requirements" on page 4-6
- "Callback Handler Configuration" on page 4-7

How It Works—Failback

WebLogic Server periodically checks the status of connection pools in a multi data source that were automatically disabled. (See "Automatic Re-enablement on Recovery of a Failed Connection Pool within a Multi Data Source" on page 4-4.) If a disabled connection pool becomes available and if a failover callback handler is registered, WebLogic Server calls the callback handler with the following information and waits for a return:

- `currPool`—For failback, this is the name of the connection pool that was previously disabled and is now available to be re-enabled.
- `nextPool`—For failback, this is null.

- `opcode`—A code that indicates the reason for the call. For failback, the code is always `OPCODE_REENABLE_CURR_POOL`, which indicates that the connection pool named in `currPool` is now available.

Failback, or automatically re-enabling a disabled connection pool, differs from failover in that failover is synchronous with the connection request, but failback is asynchronous with the connection request.

The callback handler can return one of the following values:

- `OK`—proceed with the operation. In this case, that means to re-enable the indicated connection pool. WebLogic Server resumes routing connection requests to the connection pool, depending on the multi data source algorithm and the position of the connection pool in the list of included connection pools.
- `DONOT_FAILOVER`—Do not re-enable the `currPool` connection pool. Continue to serve connection requests from the connection pool(s) in use.

WebLogic Server acts according to the value returned by the callback handler.

If the callback handler returns DONOT_FAILOVER, WebLogic Server will attempt to re-enable the connection pool during the next testing cycle as determined by the `HealthCheckFrequencySeconds` attribute in the multi data source configuration, and will call the callback handler as part of that process.

The order in which connection pools are listed in a multi data source is very important. A multi data source with the High Availability algorithm will always attempt to serve connection requests from the first available connection pool in the list of connection pools in the multi data source. Consider the following scenario:

`MultiDataSource_1` uses the High Availability algorithm, has a registered `ConnectionPoolFailoverCallbackHandler`, and includes three connection pools: `CP1`, `CP2`, and `CP3`, listed in that order.

`CP1` becomes disabled, so `MultiDataSource_1` fails over connection requests to `CP2`.

`CP2` then becomes disabled, so `MultiDataSource_1` fails over connection requests to `CP3`.

After some time, `CP1` becomes available again and the callback handler allows WebLogic Server to re-enable the connection pool. Future connection requests will be served by `CP1` because `CP1` is the first connection pool listed in the multi data source.

If `CP2` subsequently becomes available and the callback handler allows WebLogic Server to re-enable the connection pool, connection requests will continue to be served by `CP1` because `CP1` is listed before `CP2` in the list of connection pools.

Using Third-Party JDBC Drivers with WebLogic Server

Overview of Third-Party JDBC Drivers

WebLogic Server works with third-party JDBC drivers that offer the following functionality:

- Are thread-safe
- Can implement transactions using standard JDBC statements

Third-party JDBC drivers that do not implement Serializable or Remote interfaces cannot pass objects to a remote client application.

This section describes how to set up and use third-party JDBC drivers with WebLogic Server and specific instructions for the following third-party JDBC drivers:

- Oracle Thin Driver 10g (included in the WebLogic Server installation)
- Sybase jConnect and jConnect2 Drivers (included in the WebLogic Server installation)
- MySQL

Third-Party JDBC Drivers Installed with WebLogic Server

The10g (10.1.0.2.0) version of the Oracle Thin driver (`ojdbc14.jar`) and the Sybase jConnect 4.5 (`jConnect.jar`) and 5.5 (`jconn2.jar`) drivers are installed in the *`WL_HOME`*`\server\lib` folder (where *`WL_HOME`* is the folder where WebLogic Platform is installed) with `weblogic.jar`. The manifest in `weblogic.jar` lists these files so that they are loaded when `weblogic.jar` is loaded (when the server starts). Therefore, you do not need to add these JDBC drivers to your

CLASSPATH. If you plan to use a third-party JDBC driver that is not installed with WebLogic Server, you must update your CLASSPATH with the path to the driver files.

Setting the Environment for a Type-4 Third-Party JDBC Driver

If you use a third-party JDBC driver other than the Oracle Thin Driver or Sybase jConnect drivers included in the WebLogic Server installation, you must add the path for the JDBC driver classes to your CLASSPATH. The following sections describe how to set your CLASSPATH for Windows and UNIX when using a third-party JDBC driver.

CLASSPATH for Third-Party JDBC Driver on Windows

Include the path to JDBC driver classes and to weblogic.jar in your CLASSPATH as follows:

```
set CLASSPATH=DRIVER_CLASSES;WL_HOME\server\lib\weblogic.jar;
%CLASSPATH%
```

Where *DRIVER_CLASSES* is the path to the JDBC driver classes and *WL_HOME* is the directory where you installed WebLogic Server.

CLASSPATH for Third-Party JDBC Driver on UNIX

Add the path to JDBC driver classes and to weblogic.jar to your CLASSPATH as follows:

```
export CLASSPATH=DRIVER_CLASSES:WL_HOME/server/lib/weblogic.jar:
$CLASSPATH
```

Where *DRIVER_CLASSES* is the path to the JDBC driver classes and *WL_HOME* is the directory where you installed WebLogic Server.

Using the Oracle Thin Driver

The 10g (10.1.0.2.0) version of the Oracle Thin driver (ojdbc14.jar) is installed in the *WL_HOME*\server\lib folder (where *WL_HOME* is the folder where WebLogic Server is installed) with weblogic.jar. The manifest in weblogic.jar lists this file so that it is loaded when weblogic.jar is loaded (when the server starts). Therefore, you do not need to add the Oracle Thin driver to your CLASSPATH.

Note: The ojdbc14.jar file replaces classes12.zip as the source for Oracle Thin driver classes. This version of the driver is for use with a Java 2 SDK version 1.4.

If you plan to use a different version of the driver, you must replace the ojdbc14.jar file in *WL_HOME*\server\lib with an updated version of the file from Oracle or add the new file to the front of your `CLASSPATH`. You can download driver updates from the Oracle Web site at `http://otn.oracle.com/software/content.html`.

Package Change for Oracle Thin Driver 9.x and 10g

For Oracle 8.x and previous releases, the package that contained the Oracle Thin driver was `oracle.jdbc.driver`. When configuring a JDBC connection pool that uses the Oracle 8.1.7 Thin driver, you specify the DriverName (Driver Classname) as `oracle.jdbc.driver.OracleDriver`. For Oracle 9.x and 10g, the package that contains the Oracle Thin driver is `oracle.jdbc`. When configuring a JDBC connection pool that uses the Oracle 9.x or 10g Thin driver, you specify the DriverName (Driver Classname) as `oracle.jdbc.OracleDriver`. You can use the `oracle.jdbc.driver.OracleDriver` class with the 9.x and 10g drivers, but Oracle may not make future feature enhancements to that class.

See the Oracle documentation for more details about the Oracle Thin driver.

Note: The package change does not apply to the XA version of the driver. For the XA version of the Oracle Thin driver, use `oracle.jdbc.xa.client.OracleXADataSource` as the DriverName (Driver Classname) in a JDBC connection pool.

Note: For Globalization Support with the 10g version of the driver, Oracle supplies the `orai18n.jar` file, which replaces `nls_charset.zip`. If you use character sets other than US7ASCII, WE8DEC, WE8ISO8859P1 and UTF8 with CHAR and NCHAR data in Oracle object types and collections, you must include `orai18n.jar` in your `CLASSPATH`. `orai18n.jar` is *not* installed with WebLogic Server. You can download it from the Oracle Web site.

Updating the Sybase jConnect Driver

WebLogic Server ships with the Sybase jConnect driver versions 4.5 and 5.5 preconfigured and ready to use. To use a different version, you replace `*WL_HOME*\server\lib\jConnect.jar` or `jconn2.jar` with a different version of the file from the DBMS vendor.

To revert to versions installed with WebLogic Server, copy the following files and place them in the `*WL_HOME*\server\lib` folder:

- `*WL_HOME*\server\ext\jdbc\sybase\jConnect.jar`
- `*WL_HOME*\server\ext\jdbc\sybase\jConnect-5_5\classes\jconn2.jar`

MySQL

WebLogic Server 9.0 is supported for use with a MySQL database, a popular open-source production-ready database management system. The MySQL Connection/J JDBC driver is installed with WebLogic Server. MySQL has been certified for use with JMS, JDBC, and EJB container-managed persistence. MySQL does not support XA or JTA.

Packaged JDBC Modules

When you package your enterprise application, you can include JDBC resources in the application by packaging JDBC modules in the archive and adding references to the JDBC modules in all applicable descriptor files. When you deploy the application, the JDBC resources are deployed, too. Depending on how you configure the JDBC modules, the JDBC data sources deployed with the application will either be restricted for use only by the containing application (*application-scoped modules*) or will be available to all applications and clients (*globally-scoped modules*).

The following sections describe the details of packaged JDBC modules:

Packaging a JDBC Module with an Enterprise Application: Main Steps

The main steps for creating, packaging, and deploying a JDBC module with an enterprise application are as follows:

1. Create the module. See "Creating Packaged JDBC Modules" on page A-2.
2. Add references to the module in all applicable descriptor files."Referencing a JDBC Module in J2EE Descriptor Files" on page A-7.
3. Package all application modules in an EAR. See "Packaging an Enterprise Application with a JDBC Module" on page A-9.
4. Deploy the application. See "Deploying an Enterprise Application with a JDBC Module" on page A-10.

Creating Packaged JDBC Modules

You can create JDBC application modules using WebLogic Workshop (not available for Beta) or any another development tool that supports creating an XML descriptor file. You then deploy and manage JDBC modules using JSR 88-based tools, such as the `weblogic.Deployer` utility, the Administration Console, or WebLogic Workshop (not available for Beta).

Note: As a workaround for beta, you can create a JDBC data source using the Administration Console, then copy the module as a template for use in your applications. You must change the `name` and `jndi-name` elements of the module before deploying it with your application to avoid a naming conflict in the namespace.

Each JDBC module represents a data source or a multi data source. Modules that represent a data source include all of the configuration parameters for the data source. Modules that represent a multi data source include configuration parameters for the multi data source, including a list of data source modules used by the multi data source.

Creating a JDBC Data Source Module Using the Administration Console

To create a data source module in the Administration Console that you can re-use as an application module, follow these steps.

1. Create a data source as described in "Creating a JDBC Data Source" on page 3-2. The data source module is created in the `config/jdbc` subdirectory of the domain directory.
2. Copy the `data-source-name.xml` file to a subdirectory within your application and rename the copy to include -jdbc as a suffix, such as `new-data-source-name-jdbc.xml`.
3. Open the file in an editor and change the following elements:
   – `name`—change the `name` to a name that is unique within the domain.

  - `jndi-name`—change the `jndi-name` to a name that you want the enterprise application to use to lookup the data source in the local application context.
  - `scope`—optionally, to limit access to the data source to only the containing application, add a `scope` element to the `jdbc-data-source-params` section of the module. For example, `<scope>Application</scope>`. See "Application or Global Scoping for a Packaged JDBC Module" on page A-6.

4. Continue with adding references to the descriptor files in the enterprise application. See "Referencing a JDBC Module in J2EE Descriptor Files" on page A-7.

JDBC Packaged Module Requirements

A JDBC module must meet the following criteria:

- Conforms to the `weblogic-jdbc.xsd` schema. The schema is available at `http://www.bea.com/ns/weblogic/90/weblogic-jdbc.xsd`.
- Uses a file name that ends in `-jdbc.xml`.
- Includes a `name` element that is unique within the WebLogic domain.

Data source modules must also include the following JDBC driver parameters:

- `url`
- `driver-name`
- `properties`, including any properties required by the JDBC driver to create database connections, such as a user name and password.

Multi data source modules must also include the following data source parameters:

- `data-source-list`, which is a list of data source modules, separated by commas, that the multi data source uses to satisfy database connection requests from applications.

Note: All data sources listed in the data-source-list must have the same transaction XA and transaction protocol settings.

All other configuration parameters are optional or have a default value that WebLogic Server uses if a value is not specified. However, to create a useful JDBC module, you will likely need to specify additional configuration options as required by your applications and your environment.

Creating a JDBC Data Source Module

The main sections within a JDBC data source module are:

- jdbc-driver-params—includes entries for the JDBC driver used to create database connections, including url, driver-name, and individual driver property entries. See the weblogic-jdbc.xsd schema for more valid entries. For an explanation of each element, see "JDBCDriverParamsBean" in the *WebLogic Server MBean Reference*.
- jdbc-connection-pool-params—includes entries for connection pool configuration, including connection testing options, statement cache options, and so forth. This element also inherits connection-pool-params from the weblogic-j2ee.xsd schema, including initial-capacity, max-capacity, and other options common to pooled resources. For more information, see the following:
  - "JDBCConnectionPoolParamsBean" in the *WebLogic Server MBean Reference*
  - weblogic-jdbc.xsd schema
  - weblogic-j2ee.xsd schema
- jdbc-data-source-params—includes entries for data source behavior options and transaction processing options, such as jndi-name, row-prefetch-size, and global-transactions-protocol. See the weblogic-jdbc.xsd schema for more valid entries. For an explanation of each element, see "JDBCDataSourceParamsBean" in the *WebLogic Server MBean Reference*.
- jdbc-xa-params—includes entries for XA database connection handling options, such as keep-xa-conn-till-tx-complete, supports-local-transaction, and xa-transaction-timeout. For an explanation of each element, see "JDBCXAParamsBean" in the *WebLogic Server MBean Reference*.

Listing A-1 shows an example of a JDBC module for a data source with some typical configuration options.

Listing A-1 Sample JDBC Data Source Module

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/90">
  <name>examples-demoXA-2</name>

  <jdbc-driver-params>
    <url>jdbc:pointbase:server://localhost:9092/demo</url>
    <driver-name>com.pointbase.xa.xaDataSource</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>examples</value>
      </property>
      <property>
```

```
        <name>databaseName</name>
        <value>jdbc:pointbase:server://localhost:9092/demo</value>
      </property>
    </properties>
    <password-encrypted>eNEVN9dk4dEDUEVqL1</password-encrypted>
  </jdbc-driver-params>

  <jdbc-connection-pool-params>
    <initial-capacity>3</initial-capacity>
    <max-capacity>10</max-capacity>
    <test-connections-on-reserve>true</test-connections-on-reserve>
    <test-table-name>SQL SELECT COUNT(*) FROM SYSTABLES</test-table-name>
  </jdbc-connection-pool-params>

  <jdbc-data-source-params>
    <global-transactions-protocol>TwoPhaseCommit</global-transactions-protocol>
    <jndi-name>examples-demoXA-2</jndi-name>
    <scope>Application</scope>
  </jdbc-data-source-params>

  <jdbc-xa-params>
    <supports-local-transaction>true</supports-local-transaction>
  </jdbc-xa-params>

</jdbc-data-source>
```

Creating a JDBC Multi Data Source Module

A JDBC multi data source module is much simpler than a data source module. It contains only one main section: `jdbc-data-source-params`. The `jdbc-data-source-params` element in a multi data source differs in that it contains options for multi data source behavior options instead of data source behavior options. Only the following parameters in the `jdbc-data-source-params` are valid for multi data sources:

- `jndi-name` (required)
- `data-source-list` (required)
- `scope`
- `algorithm-type`
- `connection-pool-failover-callback-handler`
- `failover-request-if-busy`

- `health-check-frequency-seconds`

For an explanation of each element, see "JDBCDataSourceParamsBean" in the *WebLogic Server MBean Reference*.

Listing A-2 shows an example of a JDBC module for a data source with some typical configuration options.

Listing A-2 Sample JDBC Multi Data Source Module

```
<jdbc-data-source xmlns="http://www.bea.com/ns/weblogic/90">
  <name>examples-demoXA-multi-data-source</name>
  <jdbc-data-source-params>
    <jndi-name>examples-demoXA -multi-data-source</jndi-name>
    <algorithm-type>Load-Balancing</algorithm-type>
    <data-source-list>examples-demoXA,examples-demoXA-2</data-source-list>
  </jdbc-data-source-params>
</jdbc-data-source>
```

Encrypting Database Passwords in a JDBC Module

BEA recommends that you encrypt database passwords in a JDBC module to keep your data secure. To encrypt a database password, you process the password with the WebLogic Server `encrypt` utility, which returns an encrypted equivalent of the password that you include in the JDBC module as the `password-encrypted` element. For more details about using the WebLogic Server encrypt utility, see "encrypt" in the *WebLogic Server Command Reference*.

Application or Global Scoping for a Packaged JDBC Module

By default, when you package a JDBC module with an application, the JDBC resource is globally scoped—that is, the resource is bound to the global JNDI namespace and is available to all applications and clients. To reserve the resource for use only by the enclosing application, you must include the `<scope>Application</scope>` parameter in the `jdbc-data-source-params` element in the JDBC module, which binds the resource to the local application namespace. For example:

```
<jdbc-data-source-params>
  <jndi-name>examples-demoXA-2</jndi-name>
  <scope>Application</scope>
</jdbc-data-source-params>
```

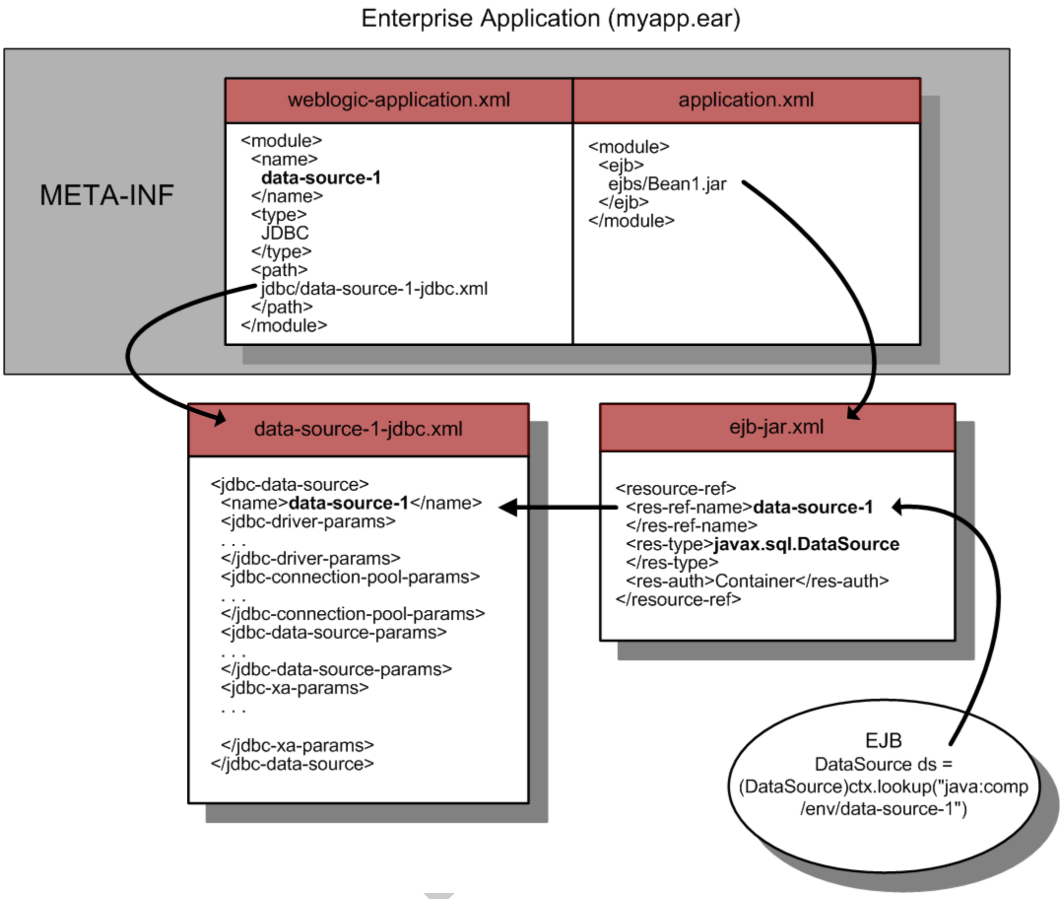
Referencing a JDBC Module in J2EE Descriptor Files

When you package a JDBC module with an enterprise application, you must reference the module in all applicable descriptor files, including among others:

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

Figure A-1 shows the relationship between entries in various descriptor files for an EJB application and how they refer to a JDBC module packaged with the application.

Figure A-1 Relationship Between JDBC Modules and Descriptors in an Enterprise Application



```
<module>
  <name>
    data-source-1
  </name>
  <type>
    JDBC
  </type>
  <path>
    jdbc/data-source-1-jdbc.xml
  </path>
</module>
```

```
<module>
  <ejb>
    ejbs/Bean1.jar
  </ejb>
</module>
```

```
<jdbc-data-source>
  <name>data-source-1</name>
  <jdbc-driver-params>
  . . .
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
  . . .
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
  . . .
  </jdbc-data-source-params>
  <jdbc-xa-params>
  . . .

  </jdbc-xa-params>
</jdbc-data-source>
```

```
<resource-ref>
  <res-ref-name>data-source-1
  </res-ref-name>
  <res-type>javax.sql.DataSource
  </res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

```
DataSource ds =
(DataSource)ctx.lookup("java:comp
/env/data-source-1")
```

Packaged JDBC Module References in weblogic-application.xml

When including JDBC modules in an enterprise application, you must list each JDBC module as a `module` element of type `JDBC` in the `weblogic-application.xml` descriptor file packaged with the application. For example:

```
<module>
  <name>data-source-1</name>
  <type>JDBC</type>
  <path>datasources/data-source-1-jdbc.xml</path>
</module>
```

Packaged JDBC Module References in ejb-jar.xml

If EJBs in your application use database connections through a JDBC module packaged with the application, you must list the JDBC module as a `res-ref` element and include `res-ref-name` and `res-type` parameters in the `ejb-jar.xml` descriptor file packaged with the EJB so that the EJB can lookup the data source in the local context and request a connection. For example:

```
<resource-ref>
  <res-ref-name>data-source-1</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
</resource-ref>
```

The `res-ref-name` element maps the name to a module referenced by an EJB. The `res-type` element specifies the module type, `javax.sql.DataSource` in this case.

Packaged JDBC Module References in weblogic-ejb-jar.xml

By default, if you do not specify a `jndi-name` in a JDBC module, WebLogic Server uses the value for the `name` parameter in the module as the JNDI name for the resource. If your module includes a `jndi-name` parameter in the `jdbc-data-source-params` element that is different than the `name`, you must also reference the JDBC module in the `weblogic-ejb-jar.xml` file packaged with the EJB as a `res-env-ref-name` element and include a `jndi-name` parameter.

For example:

```
<res-env-ref-name>
  <res-ref-name>data-source-1</res-ref-name>
  <jndi-name>point-base-demo</jndi-name>
</res-env-ref-name>
```

Adding this reference to the `weblogic-ejb-jar.xml` descriptor maps the resource name used in the EJB to the JNDI name by which the JDBC module is bound to the JNDI tree, and enables the EJB to look up the data source by using the JNDI name.

Note: The reference in `weblogic-ejb-jar.xml` is optional if you do not specify a `jndi-name` in the JDBC module.

Packaging an Enterprise Application with a JDBC Module

You package an application with a JDBC module as you would any other enterprise application. See "Packaging Applications Using wlpackage" in *Developing Applications with WebLogic Server*.

Deploying an Enterprise Application with a JDBC Module

You deploy an application with a JDBC module as you would any other enterprise application. See "Deploying Applications Using wldeploy" in *Developing Applications with WebLogic Server*.

Getting a Database Connection from a Packaged JDBC Module

To get a connection from JDBC module packaged with an enterprise application, you look up the data source or multi data source defined in the JDBC module in the local environment (`java:comp/env`) or on the JNDI tree and then request a connection from the data source or multi data source. For example:

```
javax.sql.DataSource ds =
    (javax.sql.DataSource) ctx.lookup("java:comp/env/data-source-1");
java.sql.Connection conn = ds.getConnection();
```

When you are finished using the connection, make sure you close the connection to return it to the connection pool in the data source:

```
conn.close();
```