Preface

The Oracle Universal Connection Pool (UCP) is a full-featured connection pool for managing database connections. Java applications that are database-intensive, use the connection pool to improve performance and better utilize system resources.

The instructions in this guide detail how to use the UCP API and cover a wide range of use cases. The guide does not provide detailed information about using Oracle JDBC Drivers, Oracle Database, or SQL except as required to understand UCP.

Audience

This guide is primarily written for Application Developers and System Architects who want to learn how to use UCP to create and manage database connections for their Java applications. Users must be familiar with Java and JDBC to use this guide. Knowledge of Oracle Database concepts (such as Oracle RAC and ONS) is required when using some UCP features.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information about using Java with the Oracle Database, see the following documents in the Oracle Database documentation set:

- *Oracle Database JDBC Developer's Guide*
- *Oracle Database 2 Day + Java Developer's Guide*
- *Oracle Database Java Developer's Guide*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle Universal Connection Pool Developer's Guide

This preface contains:

- Changes in Oracle Database 23ai

Changes in Oracle Database 23ai

Following are the changes in the *Oracle Universal Connection Pool Developer's Guide* for this release of Oracle Database.

New Features

This section lists the new features in this release.

- Support for Background Threads

  See Connection Creation Using Background Threads
- Support for Asynchronous Database Access

  See UCP Asynchronous Extension
- Support for Failover or Resharding Event Handling in UCP

  #unique_22
- Support for XA Transactions with Sharded Databases

  See Sharding with JTA/XA Transaction in WebLogic Server
- Support for Connection Creation Consumer

  See Connection Creation Consumer
- Enhanced Diagnosability

  See Diagnosing a Connection Pool
- New `SYSTEM_PROPERTY_TIMERS_AFFECT_ALL_CONNECTIONS` for Improved Maximum Connection Reuse Time
  See Setting the Maximum Connection Reuse Time
- Ability to Set Minimum Number of Idle Connections
  See Setting the Minimum Idle Connection Number

1
Introduction to UCP

The following sections are included in this chapter:

- Overview of Connection Pool
- Overview of Universal Connection Pool

1.1 Overview of Connection Pool

A connection pool is a cache of database connection objects. The objects represent physical database connections that can be used by an application to connect to a database. At run time, the application requests a connection from the pool. If the pool contains a connection that can satisfy the request, it returns the connection to the application. If no connections are found, a new connection is created and returned to the application. The application uses the connection to perform some work on the database and then returns the object back to the pool. The connection is then available for the next connection request.

Connection pools promote the reuse of connection objects and reduce the number of times that connection objects are created. Connection pools significantly improve performance for database-intensive applications because creating connection objects is costly both in terms of time and resources. Tasks such as network communication, reading connection strings, authentication, transaction enlistment, and memory allocation all contribute to the amount of time and resources it takes to create a connection object. In addition, because the connections are already created, the application waits less time to get the connection.

Connection pools often provide properties that are used to optimize the performance of a pool. The properties control behaviors such as the minimum and maximum number of connections allowed in the pool or the amount of time a connection can remain idle before it is returned to the pool. The best configured connection pools balance quick response times with the memory spent maintaining connections in the pool. It is often necessary to try different settings until the best balance is achieved for a specific application.

1.2 Benefits of Using a Connection Pool

Applications that are database-intensive, generally benefit the most from connection pools. As a policy, applications should use a connection pool whenever database usage is known to affect application performance.

A connection pool provides the following benefits:

- Reduces the number of times new connection objects are created.
- Promotes connection object reuse.
- Quickens the process of getting a connection.
- Reduces the amount of effort required to manually manage connection objects.
- Minimizes the number of stale connections.
- Controls the amount of resources spent on maintaining connections.

1.3 Overview of Universal Connection Pool

UCP provides a connection pool implementation for caching JDBC connections. Java applications that are database-intensive use the connection pool to improve performance and better utilize system resources.

A UCP JDBC connection pool can use any JDBC driver to create physical connections that are then maintained by the pool. The pool can be configured and provides a full set of properties that are used to optimize pool behavior based on the performance and availability requirements of an application. For more advanced applications, UCP provides a pool manager that can be used to manage a pool instance.

The pool also leverages many high availability and performance features available through an Oracle Real Application Clusters (Oracle RAC) database. These features include Fast Connection Failover (FCF), Run-time connection Load Balancing (RLB), and Connection Affinity.

Note:

Starting from Oracle Database 11*g* Release 2, FCF is also supported by Oracle Restart on a single instance database. Oracle Restart is also known as Oracle Grid Infrastructure for Independent Servers.

See Also:

Oracle Database Administrator's Guide for more information about Oracle Restart.

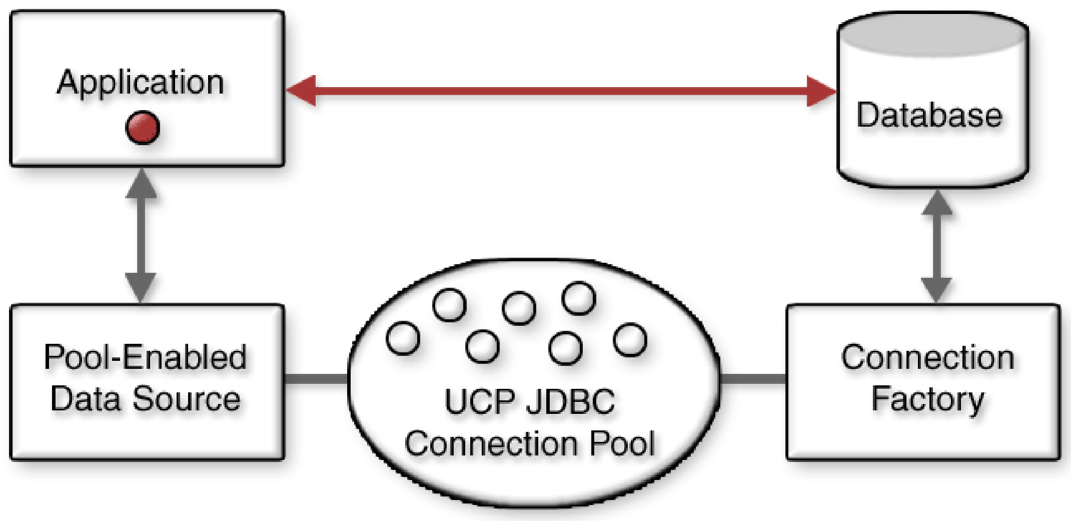
1.3.1 Conceptual Architecture

Applications use a UCP pool-enabled data source to get connections from a UCP JDBC connection pool instance. The `PoolDataSource` data source is used for getting regular connections (`java.sql.Connection`), and the `PoolXADataSource` data source is used for getting XA (eXtended API) connections (`javax.sql.XAConnection`). The same pool features are included in both XA and non-XA UCP JDBC connection pools.

The pool-enabled data source relies on a connection factory class to create the physical connections that are maintained by the pool. An application can choose to use any factory class that is capable of creating `Connection` or `XAConnection` objects. The pool-enabled data sources provide a method for setting the connection factory class, as well as methods for setting the database URL and database credentials that are used by the factory class to connect to a database.

Applications borrow a connection handle from the pool to perform work on a database. Once the work is completed, the connection is closed and the connection handle is returned to pool and is available to be used again. The following figure shows the conceptual view of the interaction between an application and a UCP JDBC connection pool.

Figure 1-1 Conceptual View of a UCP JDBC Connection Pool



Related Topics

- Getting Database Connections in UCP

1.3.2 Connection Pool Properties

UCP JDBC Connection pool properties are configured through methods available on the pool-enabled data source. The pool properties are used to control the pool size, handle stale connections, and make autonomous decisions about how long connections can remain borrowed before they are returned to the pool. The optimal settings for the pool properties depend on the application and hardware resources. Typically, there is a trade-off between the time it takes for an application to get a connection versus the amount of memory it takes to maintain a certain pool size. In many cases, experimentation is required to find the optimal balance to achieve the desired performance for a specific application.

Related Topics

- Optimizing Universal Connection Pool Behavior

1.3.3 Connection Pool Manager

UCP includes a connection pool manager that is used by applications that require administrative control over a connection pool. The manager is used to explicitly control the life cycle of a pool and to perform maintenance on a pool. The manager also provides the opportunity for an application to expose the pool and its manageability through an administrative console.

Related Topics

- Using the Connection Pool Manager

1.3.4 High Availability and Performance Scenarios

A UCP JDBC connection pool provides many features that are used to ensure high connection availability and performance. Many of these features, such as refreshing a pool or validating connections, are generic and work across driver and database implementations. Some of these features, such as run-time connection load balancing, and connection affinity, require the use of an Oracle JDBC driver and an Oracle RAC database.

Related Topics

- Using Oracle RAC Features

2

Getting Started

The following sections are included in this chapter:

2.1 Requirements for using UCP

This section describes the design-time and run-time requirements of UCP.

- JRE 8 or higher
- A JDBC driver or a connection factory class capable of returning a `java.sql.Connection` and `javax.sql.XAConnection` object

> **Note:**
>
> Oracle drivers from releases 11.2.0.4 or higher are supported. Advanced Oracle Database features, such as Oracle RAC and Fast Connection Failover, require the Oracle Notification Service library (`ons.jar`) that is included with the Oracle Client software.

- The `ucp.jar` library included in the classpath of the application
- The `ojdbc8.jar` library or the `ojdbc11.jar` library is included in the classpath of the application

> **Note:**
>
> Even if you use UCP with a third-party database and driver, you *must* use the Oracle `ojdbc8.jar` library or the `ojdbc11.jar` library because UCP has dependencies on this library.

- A database that supports SQL. Advanced features, such as Oracle RAC and Fast Connection Failover, require an Oracle Database.

2.2 Basic Connection Steps in UCP

UCP provides a pool-enabled data source that is used by applications to borrow connections from a UCP JDBC connection pool. A connection pool is not explicitly defined for the most basic use case. Instead, a default connection pool is implicitly created when the connection is borrowed.

The following steps describe how to get a connection from a UCP pool-enabled data source in order to access a database. The complete example is provided in Example 2-1:

1. Use the UCP data source factory (`oracle.ucp.jdbc.PoolDataSourceFactory`) to get an instance of a pool-enabled data source using the `getPoolDataSource` method. The data source instance must be of the type `PoolDataSource`. For example:

   ```
   PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
   ```

2. Set the connection properties that are required to get a physical connection to a database. These properties are set on the data source instance and include: the URL, the user name, and password to connect to the database and the connection factory used to get the physical connection. These properties are specific to a JDBC driver and database. For example:

   ```
   pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
   pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
   pds.setUser("<user>");
   pds.setPassword("<password>");
   ```

3. Set any pool properties in order to override the connection pool's default behavior. the pool properties are set on the data source instance. For example:

   ```
   pds.setInitialPoolSize(5);
   ```

4. Get a connection using the data source instance. The returned connection is a logical handle to a physical connection in the data source's connection pool. For example:

   ```
   Connection conn = pds.getConnection();
   ```

5. Use the connection to perform some work on the database:

   ```
   Statement stmt = conn.createStatement ();
   stmt.execute("SELECT * FROM foo");
   ```

6. Close the connection and return it to the pool.

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

2.2.1 Authentication in UCP

UCP provides transparent authentication, that is, the `PoolDataSource` behaves in the same way as the JDBC driver data source, while authenticating a connection.

UCP supports all the following authentication methods that the JDBC thin or the JDBC OCI driver suggests, and delegates any authentication action to the underlying driver:

- Authentication through passwords stored in Oracle Wallets
- Authentication using Kerberos
- Authentication through SSL certificates
- Authentication using Lightweight Directory Access Protocol (LDAP)

2.2.2 Authentication Using IAM Database Access Tokens in Oracle Cloud Infrastructure

In Oracle Database release 21.4 (21.4.0.0.1), the JDBC Thin drivers can access Oracle Autonomous Database on Shared Exadata Infrastructure, using a database access token generated by the Identity and Access Management (IAM) Cloud Service. UCP supports this authentication type using the `PoolDataSource.setTokenSupplier(Supplier)` method.

See Also:

Support for IAM Token-Based Authentication in Oracle Cloud Infrastructure

2.3 UCP API Overview

This section provides a quick overview of the most commonly used packages of the UCP API.

See Also:

Oracle Universal Connection Pool Java API Reference for complete details on the API.

oracle.ucp.jdbc

This package includes various interfaces and classes that are used by applications to work with JDBC connections and a connection pool. Among the interfaces found in this package, the `PoolDataSource` and `PoolXADataSource` data source interfaces are used by an application to get connections as well as get and set connection pool properties. Data source instances implementing these two interfaces automatically create a connection pool.

oracle.ucp.admin

This package includes interfaces for using a connection pool manager as well as MBeans that allow users to access connection pool and the connection pool manager operations and attributes using JMX operations. Among the interfaces, the `UniversalConnectionPoolManager` interface provides methods for creating and maintaining connection pool instances.

oracle.ucp

This package includes both required and optional callback interfaces that are used to implement connection pool features. For example, the `ConnectionAffinityCallback` interface is used to create a callback that enables or disables connection affinity and can also be used to customize connection affinity behavior. This package also contains statistics classes, UCP specific exception classes, and the logic to use the UCP directly, without using data sources.

2.4 UCP System Properties

For a detailed list of the UCP system properties, refer to the UCP Java API Reference.

See Also:

Oracle Universal Connection Pool Java API Reference

2.5 Basic Connection Example Using UCP

The following example is a program that connects to a database to do some work and then exits. The example is simple and in some cases not very practical; however, it does demonstrate the basic steps required to get a connection from a UCP pooled-enabled data source in order to access a database.

Example 2-1 Basic Connection Example

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

public class BasicConnectionExample {
   public static void main(String args[]) throws SQLException {
      try
      {
         //Create pool-enabled data source instance.

         PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

         //set the connection properties on the data source.

         pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
         pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
         pds.setUser("<user>");
         pds.setPassword("<password>");

         //Override any pool properties.

         pds.setInitialPoolSize(5);

         //Get a database connection from the datasource.

         Connection conn = pds.getConnection();

         System.out.println("\nConnection obtained from " +
          "UniversalConnectionPool\n");

         //do some work with the connection.
         Statement stmt = conn.createStatement();
         stmt.execute("select * from foo");

         //Close the Connection.

         conn.close();
         conn=null;

         System.out.println("Connection returned to the " +
          "UniversalConnectionPool\n");

      }
      catch(SQLException e)
      {
         System.out.println("BasicConnectionExample - " +
          "main()-SQLException occurred : "
              + e.getMessage());
      }
```

```
        }
    }
```

2.6 Minimal Pool configuration

You can configure a pool minimally and make use of the default values.

If you want to start a new pool data source, then instantiate it and set the following mandatory properties:

- The connection factory class, which is usually a JDBC driver data source, for example, `oracle.jdbc.pool.OracleDataSource`
- The peer database URL
- User name
- Password

All the other pool data source properties are optional. With this minimal configuration, the default pool size is as follows:

- Minimal pool size is 1
- Maximum pool size is `Integer.MAX_VALUE` (2147483647 by default)
- Initial pool size is 0

With the minimal configuration, the pool data source performs some minimal validation while borrowing the connections. The default value of the `setSecondsToTrustIdleConnection(int)` method is set to 120 seconds and all the timeouts are disabled.

See Also:

Overview of Validating Connections in UCP

3

Getting Database Connections in UCP

The following sections are included in this chapter:

3.1 About Borrowing Connections from UCP

An application borrows connections using a pool-enabled data source. This section describes the following concepts about borrowing connections:

Note:

The instructions in this section use a pool-enabled data source to implicitly create and start a connection pool.

3.1.1 Overview of Borrowing Connections from UCP

The UCP API provides two pool-enabled data sources, one for borrowing regular connections and one for borrowing XA connections. These data sources provide access to UCP JDBC connection pool functionality, and include a set of `getConnection` methods that are used to borrow connections. The same pool features are included in both XA and non-XA UCP JDBC connection pools.

UCP JDBC connection pools maintain both available connections and borrowed connections. A connection is reused from the pool if an application requests to borrow a connection that matches an available connection. A new connection is created if no available connection in the pool matches the requested connection. The number of available connections and borrowed connections are subjected to pool properties such as pool size, timeout intervals, and validation rules.

3.1.1.1 Connection Creation Using Background Threads

Starting with Oracle Database Release 23ai, new connections are created using background threads instead of user threads.

A borrow request may trigger a new connection creation, when both the following conditions are met:

- When there is no connection available in the pool at the time of the request
- If there is enough room to grow the pool

The borrow request may be satisfied by either of the following, whichever event happens first:

- A brand new connection created by the background thread
- A connection that was just returned to the pool

If the connection borrow request cannot be satisfied within the `ConnectionWaitTimeout` (CWT) period, then a `UniveralConnectionPoolException` is thrown, with the UCP-29 error code.

This behavior is different from Oracle Database Release 19c or 21c in the following ways:

- If the CWT is equal to zero or a very small value, then a borrow request has a higher chance to throw an exception because there is not enough time to create a new JDBC connection. A borrow request with a zero CWT period can return a connection only if there is one immediately available in the pool.
- A UCP exception thrown by the connection request does not always include the JDBC exception as a cause. To troubleshoot such situations, where the driver cannot connect to the Database, you can implement the `ConnectionCreationInformation` callback.
- Unlike the previous releases, the CWT is not adjusted with the value of the `CONNECT_TIMEOUT` parameter.

The current default behavior is to use background threads for creating connections, instead of the user threads, which results in enhanced efficiency. If required, you can switch back to the old behavior in the following ways:

- Setting the new system property `oracle.ucp.createConnectionInBorrowThread` to `true`
- Using the `setCreateConnectionInBorrowThread(boolean)` method to set the `createConnectionInBorrowThread` flag to `true`

3.1.2 Using the Pool-Enabled Data Source

UCP provides a pool-enabled data source (`oracle.ucp.jdbc.PoolDataSource`) that is used to get connections to a database. The `oracle.ucp.jdbc.PoolDataSourceFactory` factory class provides a `getPoolDataSource()` method that creates the pool-enabled data source instance. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
```

The pool-enabled data source requires a connection factory class in order to get an actual physical connection. The connection factory is typically provided as part of a JDBC driver, and it can also be a data source itself. A UCP JDBC connection pool can use any JDBC driver to create physical connections that are then maintained by the pool. The `setConnectionFactoryClassName(String)` method is used to define the connection factory for the pool-enabled data source instance. The following example uses the `oracle.jdbc.pool.OracleDataSource` connection factory class included with the Oracle JDBC driver. If you use a JDBC driver provided by a different vendor, then refer to the corresponding vendor documentation for an appropriate connection factory class.

```
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
```

In addition to the connection factory class, a pool-enabled data source requires the URL, user name, and password that are used to connect to a database. A pool-enabled data source instance includes methods to set each of these properties. The following example uses an Oracle JDBC Thin driver URL syntax. If you use a JDBC driver provided by a different vendor, then refer to the corresponding vendor documentation for the appropriate URL syntax.

```
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("user");
pds.setPassword("password");
```

See Also:

Oracle Database JDBC Developer's Guide for detailed Oracle URL syntax usage.

A pool-enabled data source provides the following `getConnection` methods:

- `getConnection()`: Returns a connection that is associated with the user name and the password that were used to connect to the database.
- `getConnection(String username, String password)`: Returns a connection that is associated with the specified user name and password.
- `getConnection(java.util.Properties labels)`: Returns a connection that matches a specified label.
- `getConnection(String username, String password, java.util.Properties labels)`: Returns a connection that is associated with a specified user name and password, and that matches a specified label.

An application uses the `getConnection` methods to borrow a connection handle from the pool that is of the type `java.sql.Connection`. If a connection handle is already in the pool that matches the requested connection (same URL, user name, and password), then it is returned to the application. Otherwise, a new connection is created and a new connection handle is returned to the application. The following examples demonstrate how to borrow a connection for Oracle Database and MySQL Database respectively:

Oracle Example

The following example demonstrates borrowing a connection using the JDBC Thin driver:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
pds.setPassword("<password>");

Connection conn = pds.getConnection();
```

MySQL Example

The following example demonstrates borrowing a connection using the Connector/J JDBC driver from MySQL:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("com.mysql.cj.jdbc.MysqlDataSource");
pds.setURL("jdbc:mysql://host:3306/dbname");
pds.setUser("<user>");
pds.setPassword("<password>");

Connection conn = pds.getConnection();
```

3.1.3 Using the Pool-Enabled XA Data Source

UCP provides a pool-enabled XA data source (`oracle.ucp.jdbc.PoolXADataSource`) that is used to get XA connections that can be enlisted in a distributed transaction. UCP JDBC XA pools have the same features as non-XA UCP JDBC pools. The `oracle.ucp.jdbc.PoolDataSourceFactory` factory class provides a `getPoolXADataSource()` method that creates the pool-enabled XA data source instance. For example:

```
PoolXADataSource  pds = PoolDataSourceFactory.getPoolXADataSource();
```

A pool-enabled XA data source instance, like a non-XA data source instance, requires the connection factory, URL, user name, and password in order to get an actual physical connection. These properties are set in the same way as a non-XA data source instance (see above). However, an XA-specific connection factory class is required to get XA connections. The XA connection factory is typically provided as part of a JDBC driver and can be a data source itself. The following example uses Oracle's `oracle.jdbc.xa.client.OracleXADataSource` XA connection factory class included with the JDBC driver. If a different vendor's JDBC driver is used, refer to the vendor's documentation for an appropriate XA connection factory class.

```
pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("user");
pds.setPassword("password");
```

Lastly, a pool-enabled XA data source provides a set of `getXAConnection` methods that are used to borrow a connection handle from the pool that is of the type `javax.sql.XAConnection`. The `getXAConnection` methods are the same as the `getConnection` methods previously described. The following example demonstrates borrowing an XA connection.

```
PoolXADataSource  pds = PoolDataSourceFactory.getPoolXADataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
pds.setPassword("<password>");

XAConnection conn = pds.getXAConnection();
```

Related Topics

- Labeling Connections in UCP

3.1.4 Setting Connection Properties

Oracle's connection factories support properties that configure connections with specific features. UCP pool-enabled data sources provide the `setConnectionProperties(Properties)` method, which is used to set properties on a given connection factory. The following example demonstrates setting connection properties for Oracle's JDBC driver. If you are using a JDBC driver from a different vendor, then refer to the vendor-specific documentation to check whether setting properties in this manner is supported and what properties are available:

```
Properties connProps = new Properties();
connProps.put("fixedString", false);
connProps.put("remarksReporting", false);
connProps.put("restrictGetTables", false);
connProps.put("includeSynonyms", false);
connProps.put("defaultNChar", false);
connProps.put("AccumulateBatchResult", false);

pds.setConnectionProperties(connProps);
```

The UCP JDBC connection pool does not remove connections that are already created if `setConnectionProperties` is called after the pool is created and in use.

See Also:

Oracle Database JDBC Java API Reference for a detailed list of supported properties to configure the connection. For example, to set the auto-commit mode, you can use the `OracleConnection.CONNECTION_PROPERTY_AUTOCOMMIT` property.

3.1.5 Using JNDI to Borrow a Connection

A connection can be borrowed from a connection pool by performing a JNDI look up for a pool-enabled data source and then calling `getConnection()` on the returned object. The pool-enabled data source must first be bound to a JNDI context and a logical name. This assumes that an application includes a Service Provider Interface (SPI) implementation for a naming and directory service where object references can be registered and located.

The following example uses Sun's file system JNDI service provider, which can be downloaded from the JNDI software download page:

http://www.oracle.com/technetwork/java/index.html

The example demonstrates creating an initial context and then performing a lookup for a pool-enabled data source that is bound to the name `MyPooledDataSource`. The object returned is then used to borrow a connection from the connection pool.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:/tmp");

ctx = new InitialContext(env);

PoolDataSource jpds = (PoolDataSource)ctx.lookup(MyPooledDataSource);
Connection conn = jpds.getConnection();
```

In the example, `MyPoolDataSource` must be bound to the context. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
pds.setPassword("<password>");

ctx.bind(MyPooledDataSource, pds);
```

3.1.6 About Connection Initialization Callback

The Connection Initialization Callback enables applications and frameworks to initialize connections retrieved from Universal Connection Pool. It is executed at every connection checkout from the pool, as well as at each successful reconnection during failover.

This section discusses initialization callbacks in the following sections:

- Overview of Connection Initialization Callback
- Creating an Initialization Callback
- Registering an Initialization Callback
- Removing or Unregistering an Initialization Callback

3.1.6.1 Overview of Connection Initialization Callback

If an application cannot use connection labeling because it cannot be changed, then the connection initialization callback is provided for such an application.

When registered, the initialization callback is executed every time a connection is borrowed from the pool and at each successful reconnection following a recoverable error. Using the same callback at both run time and replay ensures that exactly the same initialization, which was used when the original session was established, is reestablished at run time. If the callback invocation fails, then replay is disabled on that connection.

3.1.6.2 Creating an Initialization Callback

To create a UCP connection initialization callback, an application implements the `oracle.ucp.jdbc.ConnectionInitializationCallback` interface. This interface has the following method:

```
void initialize(java.sql.Connection connection) throws SQLException;
```

Note:

- One callback is created for every connection pool.
- This callback is not used if a labeling callback is registered for the connection pool.

Example

The following example demonstrates how to create a simple initialization callback:

```
import oracle.ucp.jdbc.ConnectionInitializationCallback;
class MyConnectionInitializationCallback implements ConnectionInitializationCallback
{
  public MyConnectionInitializationCallback()
  {
    ...
  }
  public void initialize(java.sql.Connection connection) throws SQLException
  {
     // Reset the state for the connection, if necessary (like ALTER SESSION)
  }
}
```

3.1.6.3 Registering an Initialization Callback

UCP provides the `registerConnectionInitializationCallback` method in the `oracle.ucp.jdbc.PoolDataSource` interface for registering a connection initialization callback.

```
public void registerConnectionInitializationCallback (ConnectionInitializationCallback
cbk) throws SQLException;
```

One callback may be registered on each connection pool instance.

3.1.6.4 Removing or Unregistering an Initialization Callback

UCP provides the `unregisterConnectionInitializationCallback` method in the `oracle.ucp.jdbc.PoolDataSource` interface for unregistering a connection initialization callback.

```
public void unregisterConnectionInitializationCallback() throws SQLException;
```

See Also:

Oracle Universal Connection Pool Java API Reference for more information

3.2 Setting Connection Pool Properties for UCP

UCP JDBC connection pools are configured using connection pool properties. The properties have `get` and `set` methods that are available through a pool-enabled data source instance. The methods are a convenient way to programmatically configure a pool. If no pool properties are set, then a connection pool uses default property values.

The following example demonstrates configuring connection pool properties. The example sets the connection pool name and the maximum/minimum number of connections allowed in the pool.

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);pds.setMaxPoolSize(20);
```

UCP JDBC connection pool properties may be set in any order and can be dynamically changed at run time. For example, `setMaxPoolSize` could be changed at any time and the pool recognizes the new value and adapts accordingly.

Related Topics

- Optimizing Universal Connection Pool Behavior

3.3 Overview of Validating Connections in UCP

Connections can be validated using pool properties when the connection is borrowed, and also programmatically using the `ValidConnection` interface. Both approaches are detailed in this section. Invalid connections can affect application performance and availability.

3.3.1 Validating When Borrowing

A connection can be validated by executing a SQL statement on a connection when the connection is borrowed from the connection pool. Two connection pool properties are used in conjunction in order to enable connection validation:

- `setValidateConnectionOnBorrow(boolean):` Specifies whether or not connections are validated when the connection is borrowed from the connection pool. The method enables validation for every connection that is borrowed from the pool. A value of `false` means no validation is performed. The default value is `false`.
- `setSQLForValidateConnection(String):` Specifies the SQL statement that is executed on a connection when it is borrowed from the pool.

Note:

The `setSQLForValidateConnection` property is not recommended when using an Oracle JDBC driver. UCP performs an internal ping when using an Oracle JDBC driver. The mechanism is faster than executing an SQL statement and is overridden if this property is set. Instead, set the `setValidateConnectionOnBorrow` property to `true` and do not include the `setSQLForValidateConnection` property.

The following example demonstrates validating a connection when borrowing the connection from the pool. The example uses Connector/J JDBC driver from MySQL:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("com.mysql.jdbc.jdbc2.optional.
   MysqlDataSource");
pds.setURL("jdbc:mysql://host:3306/mysql");
pds.setUser("<user>");
pds.setPassword("<password>");

pds.setValidateConnectionOnBorrow(true);
pds.setSQLForValidateConnection("select * from mysql.user");

Connection conn = pds.getConnection();
```

See Also:

Minimizing Connection Request Delay

3.3.2 Minimizing Connection Validation with setSecondsToTrustIdleConnection() Method

In UCP, when you set the value of the `setValidateConnectionOnBorrow(boolean)` method to `true`, then each connection is validated during the checkout. This validation may incur significant overhead in applications that checkout database connections frequently.

To minimize the impact of frequent connection validation, you can now set the `setSecondsToTrustIdleConnection(int)` method with an appropriate value to trust recently-used or recently-tested database connections. Setting this value skips the connection validation test and improves application performance significantly.

The following table describes the methods available in Oracle Database Release 23ai for using this feature:

Method	Description
`setSecondsToTrustIdleConnection(int secondsToTrustIdleConnection)`	Sets the time in seconds to trust a recently-used or recently-tested database connection and skip the validation test during connection checkout.
`getSecondsToTrustIdleConnection()`	Retrieves the value that was set using the `setSecondsToTrustIdleConnection(int)` method.

When you set the `setSecondsToTrustIdleConnection(int)` method to a positive value, then the connection validation is skipped, if the connection was used within the time specified in the `secondsToTrustIdleConnection(int)` method. The default value is 0 seconds, which means that the feature is disabled.

Note:

The `setSecondsToTrustIdleConnection(int)` method works only if the `setValidateConnectionOnBorrow(boolean)` method is set to `true`. If you set the `setSecondsToTrustIdleConnection(int)` method to a non-zero value, without setting the `setValidateConnectionOnBorrow(boolean)` method to `true`, then UCP throws the following exception:

```
Invalid seconds to trust idle connection value or usage.
```

3.3.3 Checking If a Connection Is Valid

The `oracle.ucp.jdbc.ValidConnection` interface provides two methods: `isValid` and `setInvalid`. The `isValid` method returns whether or not a connection is usable and the `setInvalid` method is used to indicate that a connection should be removed from the pool instance.

The `isValid` method is used to check if a connection is still usable after an SQL exception has been thrown. This method can be used at any time to check if a borrowed connection is valid.

The method is particularly useful in combination with a retry mechanism, such as the Fast Connection Failover actions that are triggered after a down event of Oracle RAC.

Note:

- The `isValid` method checks with the pool instance and Oracle JDBC driver to determine whether a connection is still valid. The `isValid` method results in a round-trip to the database only if both the pool and the driver report that a connection is still valid. The round-trip is used to check for database failures that are not immediately discovered by the pool or the driver.
- Starting from Oracle Database Release 18c, there is a new variant of the `isValid` method that sends an empty packet to the database unlike the older version of the method that uses a ping-pong protocol and makes a full round-trip to the database.

See Also:

Oracle Database JDBC Developer's Guide

The `isValid` method is also helpful when used in conjunction with the connection timeout and connection harvesting features. These features may return a connection to the pool while a connection is still held by an application. In such cases, the `isValid` method returns `false`, allowing the application to get a new connection.

The following example demonstrates using the `isValid` method:

```
try {  conn = poolDataSouorce.getConnection  ...}catch (SQLException sqlexc)
{
   if (conn == null || !((ValidConnection) conn).isValid())

   // take the appropriate action

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

For XA applications, before calling the `isValid()` method, you must cast any `XAConnection` that is obtained from `PoolXADataSource` to a `ValidConnection`. If you cast a `Connection` that is obtained by calling the `XAConnection.getConnection()` method to `ValidConnecion`, then it may throw an exception.

Related Topics

- Using Oracle RAC Features

Related Topics

- Removing Connections from UCP

3.4 Returning Borrowed Connections to UCP

Borrowed connections that are no longer being used should be returned to the pool so that they can be available for the next connection request. The `close` method closes connections and automatically returns them to the pool. The `close` method does not physically remove the connection from the pool.

Borrowed connections that are not closed will remain borrowed; subsequent requests for a connection result in a new connection being created if no connections are available. This behavior can cause many connections to be created and can affect system performance.

The following example demonstrates closing a connection and returning it to the pool:

```
Connection conn = pds.getConnection();

//do some work with the connection.

conn.close();
conn=null;
```

3.5 Removing Connections from UCP

The `setInvalid` method of the `ValidConnection` interface indicates that a connection should be removed from the connection pool when it is closed. The method is typically used when a connection is no longer usable, such as after an exception or if the `isValid` method of the `ValidConnection` interface returns `false`. The method can also be used if an application deems the state on a connection to be bad. The following example demonstrates using the `setInvalid` method to close and remove a connection from the pool:

```
Connection conn = pds.getConnection();
...

((ValidConnection) conn).setInvalid();
...

conn.close();
conn=null;
```

3.6 UCP Integration with Third-Party Products

Third-party products, such as middleware platforms or frameworks, can use UCP to provide connection pooling functionality for their applications and services. UCP integration includes the same connection pool features that are available to stand-alone applications and offers the same tight integration with the Oracle Database.

Two data source classes are available as integration points with UCP: `PoolDataSourceImpl` for non-XA connection pools and `PoolXADataSourceImpl` for XA connection pools. Both classes are located in the `oracle.ucp.jdbc` package. These classes are implementations of the `PoolDataSource` and `PoolXADataSource` interfaces, respectively, and contain default constructors.

See Also:

Oracle Universal Connection Pool Java API Reference for more information on the implementation classes.

These implementations explicitly create connection pool instances and can return connections. For example:

```
PoolXADataSource pds = new PoolXADataSourceImpl();

pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("user");
pds.setPassword("password");

XAConnection conn = pds.getXAConnection();
```

Third-party products can instantiate these data source implementation classes. In addition, the methods of these interfaces follow the JavaBean design pattern and can be used to set connection pool properties on the class using reflection. For example, a UCP data source that uses an Oracle JDBC connection factory and database might be defined as follows and loaded into a JNDI registry:

```
<data-sources>
   <data-source
      name="UCPDataSource"
      jndi-name="jdbc/UCP_DS"
      data-source-class="oracle.ucp.jdbc.PoolDataSourceImpl">
      <property name="ConnectionFactoryClassName"
                value="oracle.jdbc.pool.OracleDataSource"/>
      <property name="URL" value="jdbc:oracle:thin:@//localhost:1521:oracle"/>
      <property name="User" value"user"/>
      <property name="Password" value="password"/>
      <property name="ConnectionPoolName" value="MyPool"/>
      <property name="MinPoolSize" value="5"/>
      <property name="MaxPoolSize" value="50"/>
   </data-source>
</data-sources>
```

When using reflection, the name attribute matches (case sensitive) the name of the setter method used to set the property. An application could then use the data source as follows:

```
Connection connection = null;
try {
   InitialContext context = new InitialContext();
   DataSource ds = (DataSource) context.lookup( "jdbc/UCP_DS" );
   connection = ds.getConnection();
   ...
```

4

Connection Creation Consumer

Starting with Oracle Database Release 23ai, you can make UCP callback the caller application on every connection creation.

4.1 Implementing a Connection Creation Consumer

In the current release of Oracle Database, you can register a new connection creation consumer for a specific `PoolDataSource` object, which is called back on every database connection creation attempt, whether it is successful or unsuccessful.

Every connection creation operation triggers the acceptance of that consumer. For example, it can be triggered for a connection borrow request or it can be triggered with some UCP worker threads trying to keep up with a minimum pool size property, replacing a *bad* connection.

This section describes how to implement the connection creation consumer.

Note:

Once a registered consumer gets executed in a thread that creates a connection for the pool, you must implement it without blocking calls and intensive computations.

Example 4-1 Registering a Consumer

The following code snippet shows how to register a consumer:

```
import oracle.ucp.ConnectionCreationInformation;
import java.util.function.Consumer;
... rest of imports ...

  ...
  final PoolDataSource pds = new PoolDataSourceFactory.getPoolDataSource();
  ... configure pds ...
  final Consumer<ConnectionCreationInformation> consumer = p -> { ... };
  pds.registerConnectionCreationConsumer(consumer);
  ...
```

Once a consumer is registered, every connection creation operation triggers an accept call of a registered consumer.

Example 4-2 Unegistering a Consumer

The following code snippet shows how to unregister a consumer:

```
import oracle.ucp.ConnectionCreationInformation;
import java.util.function.Consumer;
... rest of imports ...

  ...
  final PoolDataSource pds = new PoolDataSourceFactory.getPoolDataSource();
  ... configure pds ...
  final Consumer<ConnectionCreationInformation> consumer = p -> { ... };
  pds.registerConnectionCreationConsumer(consumer);
  ...
  pds.unregisterConnectionCreationConsumer();
  ...
```

Once a consumer is unregistered, there are no more consumer acceptance calls for every connection creation request in a pool.

Example 4-3   Connection Creation Status Verification

The following code snippet shows how to verify whether a connection creation was successful or unsuccessful:

```
...
final Consumer<ConnectionCreationInformation> consumer = p -> {
  if (p.getStatus() == ConnectionCreationInformation.Status.SUCCESSFUL) {
    ... handle successful connection creation code path...
  }

  if (p.getStatus() == ConnectionCreationInformation.Status.FAILURE) {
    ... handle unsuccessful connection creation code path...
  }

  ... };
```

Example 4-4   Handling Failed Connection Creation

If the connection creation operation fails, then an accepted consumer throws the corresponding `SQLException` vendor error code. The following code snippet shows how to handle the error:

```
...
final Consumer<ConnectionCreationInformation> consumer = p -> {
  if (p.getStatus() == ConnectionCreationInformation.Status.FAILURE) {
    final int errorCode = p.getErrorCode();
    ... do some error code path...
  }

  ... };
```

Example 4-5   Retrieving Information About Successful Connections

The following code snippet shows how to retrieve information about successfully created connections:

```
...
final Consumer<ConnectionCreationInformation> consumer = p -> {
  if (p.getStatus() == ConnectionCreationInformation.Status.SUCCESSFUL) {
    // Net Connection ID for an appropriate connection.
    final String netConnId = getNetConnectionId();

    // database instance (if applicable) name, on which connection was
created.
    final String instanceName = getInstanceName();

    // database service (if applicable) name, on which connection was created.
    final String serviceName = getServiceName();

    // database host (if applicable) name, on which connection was created.
    final String hostName = getHostName();

    // database unique ID for created connection.
    final String dnUniqId = getDatabaseUniqId();

    // database instance (if applicable) unique ID for created connection.
    final String instanceId = getInstanceId();

    // Security information for created connection.
    final SecurityInformation securityInfo = getSecurityInformation();

    ... rest of successful connection creation code path ...
  }
  ... };
```

5

Optimizing Universal Connection Pool Behavior

This chapter describes the following concepts:

5.1 Optimizing Connection Pools

This section provides instructions for setting connection pool properties in order to optimize pooling behavior. Upon creation, UCP JDBC connection pools are pre-configured with a default setup. The default setup provides a general, all-purpose connection pool. However, different applications may have different database connection requirements and may want to modify the default behavior of the connection pool. Behaviors, such as pool size and connection timeouts can be configured and can improve overall connection pool performance as well as connection availability. In many cases, the best way to tune a connection pool for a specific application is to try different property combinations using different values until optimal performance and throughput is achieved.

Setting Connection Pool Properties

Connection pool properties are set either when getting a connection through a pool-enabled data source or when creating a connection pool using the connection pool manager.

The following example demonstrates setting connection pool properties though a pool-enabled data source:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);pds.setMaxPoolSize(20);
...
```

The following example demonstrates setting connection pool properties when creating a connection pool using the connection pool manager:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);pds.setMaxPoolSize(20);
...

mgr.createConnectionPool(pds);
```

5.2 About Controlling the Pool Size in UCP

UCP JDBC connection pools include a set of properties that are used to control the size of the pool. The properties allow the number of connections in the pool to increase and decrease as demand increases and decreases. This dynamic behavior helps conserve system resources that are otherwise lost on maintaining unnecessary connections.

This section describes the following topics:

5.2.1 Setting the Initial Pool Size

The initial pool size property specifies the number of available connections that are created when the connection pool is initially created or re-initialized. This property is typically used to reduce the ramp-up time incurred by priming the pool to its optimal size.

A value of `0` indicates that no connections are pre-created. The default value is `0`. The following example demonstrates configuring an initial pool size:

```
pds.setInitialPoolSize(5);
```

If the initial pool size property is greater than the maximum pool size property, then only the maximum number of connections are initialized.

If the initial pool size property is less than the minimum pool size property, then only the initial number of connections are initialized and maintained until enough connections are created to meet the minimum pool size value.

If during the pool initialization process, connections cannot be created up to the value specified in the `initPoolSize` property, then the pool attempts to create the remaining connections to fulfill the initial pool size. If the pool does not have the physical ability to do so, then the pool initialization process ends and it keeps trying to maintain the designated minimum and maximum connection limits for the rest of its life cycle.

5.2.2 Setting the Minimum Pool Size

The minimum pool size property specifies the minimum amount of available connections and borrowed connections that a pool maintains. A connection pool always tries to return to the minimum pool size specified unless the minimum amount is yet to be reached. For example, if the minimum limit is set to `10` and only 2 connections are ever created and borrowed, then the number of connections maintained by the pool remains at `2` because this number is less than the minimum pool size.

This property allows the number of connections in the pool to decrease as demand decreases. At the same time, the property ensures that system resources are not wasted on maintaining connections that are unnecessary.

The default value is `0`. The following example demonstrates configuring a minimum pool size:

```
pds.setMinPoolSize(2);
```

5.2.3 Setting the Maximum Pool Size

The maximum pool size property specifies the maximum number of available and borrowed (in use) connections that a pool maintains. If the maximum number of connections are borrowed, no connections will be available until a connection is returned to the pool.

This property allows the number of connections in the pool to increase as demand increases. At the same time, the property ensures that the pool does not grow to the point of exhausting the resources of a system, which ultimately affects the performance and availability of an application.

A value of `0` indicates that no connections are maintained by the pool. An attempt to get a connection results in an exception. The default value is to allow the pool to continue to create connections up to `Integer.MAX_VALUE` (2147483647 by default). The following example demonstrates configuring a maximum pool size:

```
pds.setMaxPoolSize(100);
```

5.2.4 Setting the Minimum Idle Connection Number

The minimum idle connection number property specifies the minimum number of idle connections that the connection pool maintains.

If the number of available connections is less than the number of minimum idle connections specified, then new connections are created in the background and made available in the pool. The range of valid values for this property ranges from 0 to `Integer.MAX_VALUE`. The default value is 0. It is illegal to set this property to a value greater than the maximum pool size.

The following code snippet shows how to use this property:

```
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("<URL>");
pds.setUser("<user_name>");
pds.setPassword("<password>");
pds.setInitialPoolSize(5);
pds.setMinPoolSize(5);
pds.setMinIdle(5);
pds.setMaxPoolSize(10);
Connection conn = pds.getConnection();
conn.close();
```

See Also:

Oracle Universal Connection Pool Java API Reference for more information about this property

5.3 About Optimizing Real-World Performance with Static Connection Pools

Most on-line transaction processing (OLTP) performance problems that the Real-World Performance group investigates relate to the connection strategy used by the application. For this reason, designing a sound connection strategy is crucial for system performance, especially in enterprise environments that must scale to meet increasing demand.

Most applications use a dynamic pool of connections to the database, configured with a minimum number of connections to keep open on the database and a maximum number of connections that can be made to the database. When an application needs a connection to the database, then it requests one from the pool. If there are no connections available, then the application creates a new connection, if it has not reached the maximum number of connections already. If a connection has not been used for a specified duration of time, then the application closes the connection, if there are more than the minimum number of connections available.

This configuration conserves system resources by only maintaining the number of connections actively needed by the application. In the real world, this configuration enables connection storms and database system CPU oversubscription, quickly destabilizing a system. A connection storm can occur when there are lots of activities on the application server requiring database connections. If there are not enough connections to the database to serve all of the requests, then the application server opens new connections. Creating a new connection to the database is a resource intensive activity, and when lots of connections are made in a short period of time, it can overwhelm the CPU resources on the database system.

So, for creating a static connection pool, the number of connections to the database system must be based on the CPU cores available on the system. Oracle recommends 1-10 connections per CPU core. The ideal number varies depending on the application and the system hardware. However, the value is somewhere within that range. the Real-World Performance group recommends creating a static pool of connections to the database by setting the minimum and maximum number of connections to the same value. This prevents connection storms by keeping the number of database connections constant to a predefined value.

For example, if a database server has 2 CPUs, 12 cores per CPU, and 2 threads per core, then there are 24 cores available and the number of connections to the database should be between 24 and 240. The number of threads is not taken into consideration as only the CPU cores are able to execute instructions. This number is cumulative for all applications connecting to the system and for all databases, if there is more than one database on the system. If there are two application servers, then the maximum number of connections (for example, 240 in this case) should be divided between them. If there are two databases running on the system, then the maximum number of connections that is, 240 connections needs to be divided between them.

See Also:

- https://www.youtube.com/watch?v=Oo-tBpVewP4
- https://www.youtube.com/watch?v=XzN8Rp6glEo

5.4 Stale Connections in UCP

Stale connections are connections that remain either available or borrowed, but are no longer being used. Stale connections that remain borrowed may affect connection availability.

In addition, stale connections may impact system resources that are used to maintain unused connections for extended periods of time. The pool properties discussed in this section are used to control stale connections.

This section describes the following topics:

- What is Connection Reuse?
- Setting the Connection Validation Timeout
- Setting the Abandon Connection Timeout
- Setting the Time-To-Live Connection Timeout
- Setting the Connection Wait Timeout
- Setting the Inactive Connection Timeout
- Setting the Query Timeout
- Setting the Timeout Check Interval

> **Note:**
>
> It is good practice to close all connections that are no longer required by an application. Closing connections helps minimize the number of stale connections that remain borrowed.

5.4.1 What is Connection Reuse?

The connection reuse feature allows connections to be gracefully closed and removed from a connection pool after a specific amount of time or after the connection has been used a specific number of times. This feature saves system resources that are otherwise wasted on maintaining unusable connections.

5.4.1.1 Setting the Maximum Connection Reuse Time

The maximum connection reuse time allows connections to be gracefully closed and removed from the pool after a connection has been in use for a specific amount of time. The timer for this property starts when a connection is physically created. Borrowed connections are closed only after they are returned to the pool and the reuse time is exceeded.

This feature is typically used when a firewall exists between the pool tier and the database tier and is setup to block connections based on time restrictions. The blocked connections remain in the pool even though they are unusable. In such scenarios, the connection reuse time is set to a smaller value than the firewall timeout policy.

Note:

The maximum connection reuse time is different from the time-to-live connection timeout. The main difference between the maximum connection reuse time and the time-to-live is that time-to-live can cancel a connection, but maximum connection reuse time never does that. The maximum connection reuse time handler gets applied only in the following cases:

- Prior to a connection borrow
- Immediately after a connection is returned back to a pool
- On periodic basis, with every `timeoutCheckInterval`

The maximum connection reuse time value is represented in seconds. The default value is `0`, which indicates that this feature is disabled. The following example demonstrates configuring a maximum connection reuse time:

```
pds.setMaxConnectionReuseTime(300);
```

Starting from Oracle Database Release 23ai, you can use the new system property `oracle.ucp.timersAffectAllConnections` to change the behavior of the maximum connection reuse time processing. The default value of this property is `FALSE`, which means that during periodical appropriate time processing, a pool is scanned down to a minimum pool size and the pool never closes connections below the minimum pool size. If the `SYSTEM_PROPERTY_TIMERS_AFFECT_ALL_CONNECTIONS` system property is set to `TRUE`, then the periodic poll checks all available connection for the maximum connection reuse time criteria, so the pool size may go below the minimum pool size, replacing the applicable connections.

Related Topics

- Setting the Time-To-Live Connection Timeout

5.4.1.2 Setting the Maximum Connection Reuse Count

The maximum connection reuse count allows connections to be gracefully closed and removed from the connection pool after a connection has been borrowed a specific number of times. This property is typically used to periodically recycle connections in order to eliminate issues such as memory leaks.

A value of `0` indicates that this feature is disabled. The default value is `0`. The following example demonstrates configuring a maximum connection reuse count:

```
pds.setMaxConnectionReuseCount(100);
```

5.4.2 Setting the Connection Validation Timeout

The connection validation timeout specifies the duration within which a borrowed connection from the pool is validated. This is the maximum time for a connection validation operation. If the validation is not completed during this period, then the connection is treated as invalid.

The connection validation timeout value represents seconds. The default value is set to 15. The following example demonstrates configuring a connection validation timeout:

```
pd.setConnectionValidationTimeout(55);
```

5.4.3 Setting the Abandon Connection Timeout

The abandoned connection timeout (ACT) enables borrowed connections to be reclaimed back into the connection pool after a connection has not been used for a specific amount of time. Abandonment is determined by monitoring calls to the database.

The abandoned connection timeout feature helps maximize connection reuse and conserves system resources that are otherwise lost on maintaining borrowed connections that are no longer in use.

> **Note:**
>
> Before reclaiming connections for reuse, UCP either cancels or rolls back the connections that have local transactions pending.

The ACT value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring an abandoned connection timeout:

```
pds.setAbandonedConnectionTimeout(10);
```

Every connection is reaped after a specific period of time. Either it is reaped when ACT expires, or, if it is immune from ACT, then it is reaped after the immunity expires. If you set ACT on a pool, then the following connection reaping policies apply:

- If a statement is executed without calling the `Statement.setQueryTimeout` method on that statement, then the connection is reaped if ACT is exceeded, even though the connection is waiting for the server to respond to the query.
- If a statement is executed with calling the `Statement.setQueryTimeout` method, then the connection is reaped after the query timeout and ACT have expired. The connection is not reaped while waiting on the query timeout. The expiration of the query timeout is an event that resets the ACT timer. If the ACT expires while waiting for the `cancel` action that occurs at the expiration of the query time out, then the connection is reaped.
- The default query timeout in the UCP is set to zero (0) for an appropriate pool data source, using the `PoolDataSource.setQueryTimeout` method, if the ACT is set to 0. If the ACT is greater than zero (0), then the default query timeout is set to 60 seconds.
- If a connection has two statements: s1 with a query timeout and s2 without a query timeout, then ACT does not reap the connection while s1 waits for the query timeout, but reaps the connection if s2 hangs.

  Note that the two statements execute sequentially based on JDBC requirement.

5.4.4 Setting the Time-To-Live Connection Timeout

The time-to-live connection timeout enables borrowed connections to remain borrowed for a specific amount of time before the connection is reclaimed by the pool. This timeout feature helps maximize connection reuse and helps conserve systems resources that are otherwise lost on maintaining connections longer than their expected usage.

Note:

UCP either cancels or rolls back connections that have local transactions pending before reclaiming connections for reuse.

The time-to-live connection timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring a time-to-live connection timeout:

```
pds.setTimeToLiveConnectionTimeout(18000)
```

5.4.5 Setting the Connection Wait Timeout

The connection wait timeout specifies how long an application request waits to obtain a connection if there are no longer any connections in the pool. A connection pool runs out of connections if all connections in the pool are being used (borrowed) and if the pool size has reached it's maximum connection capacity as specified by the maximum pool size property. The request receives an SQL exception if the timeout value is reached. The application can then retry getting a connection. This timeout feature improves overall application usability by minimizing the amount of time an application is blocked and provides the ability to implement a graceful recovery.

The connection wait timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `3` seconds. The following example demonstrates configuring a connection wait timeout:

```
pds.setConnectionWaitTimeout(10);
```

5.4.6 Setting the Inactive Connection Timeout

The inactive connection timeout specifies how long an available connection can remain idle before it is closed and removed from the pool.

This timeout property is only applicable to available connections and does not affect borrowed connections. This property helps conserve resources that are otherwise lost on maintaining connections that are no longer being used. The inactive connection timeout (together with the maximum pool size) enables a connection pool to grow and shrink as application load changes.

The inactive connection timeout value is represented in seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring an inactive connection timeout:

```
pds.setInactiveConnectionTimeout(60);
```

Starting from Oracle Database Release 23ai, you can use the new system property `oracle.ucp.timersAffectAllConnections` to change the behavior of the Inactive Connection Timeout and Maximum Connection Reuse Time properties. If you set this system property to `TRUE`, then the periodic poll checks all the available connections for the maximum connection reuse time and inactive connection timeout criteria, and closes all the connections that satisfy the criteria. This may make the pool size go below the minimum pool size, resulting in the creation of new connections by the pool to maintain the minimum pool size limit.

See Also:

Setting the Maximum Connection Reuse Time for more information about the `oracle.ucp.timersAffectAllConnections` property

5.4.7 Setting the Query Timeout

In Oracle Database 12*c* Release 2 (12.2.0.1), UCP introduced the `queryTimeout` property. This property specifies the number of seconds UCP waits for a `Statement` object to execute. If the limit is exceeded, then a `DatabaseException` is thrown. Use the `setQueryTimeout` method for setting this property in the following way:

```
...
PoolDataSourceImpl pds = new PoolDataSourceImpl();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL(<url>);
pds.setUser("scott");
pds.setPassword(<password>);
pds.setConnectionPoolName("my_pool");
pds.setQueryTimeout(60); // 60 seconds to wait on query
...
```

5.4.8 Setting the Timeout Check Interval

The timeout check interval property controls how frequently the timeout properties (abandoned connection timeout, time-to-live connection timeout, and inactive connection timeout) are enforced. Connections that have timed-out are reclaimed when the timeout check cycle runs. This means that a connection may not actually be reclaimed by the pool at the moment that the connection times-out. The lag time between the connection timeout and actually reclaiming the connection may be considerable depending on the size of the timeout check interval.

The timeout check interval property represents seconds. The default value is set to `30`. The following example demonstrates configuring a property check interval:

```
pds.setTimeoutCheckInterval(60);
```

See Also:

Oracle Database Net Services Administrator's Guide for more information about Oracle Net Services

5.5 About Harvesting Connections in UCP

The connection harvesting feature allows a specified number of borrowed connections to be reclaimed when the connection pool reaches a specified number of available connections. This section describes the following concepts:

- Overview of Harvesting Connections in UCP

5.5.1 Overview of Harvesting Connections in UCP

This feature helps ensure that a certain number of connections are always available in the pool and helps maximize performance. The feature is particularly useful if an application caches connection handles. Caching is typically performed for performance reasons because it minimizes re-initialization of state necessary for connections to participate in a transaction.

For example, a connection is borrowed from the pool, initialized with necessary session state, and then held in a context object. Holding connections in this manner may cause the connection pool to run out of available connections. The connection harvest feature reclaims the borrowed connections, if appropriate, and allows the connections to be reused.

Connection harvesting is controlled using the `HarvestableConnection` interface and configured or enabled using two pool properties: Connection Harvest Trigger Count and Connection Harvest Maximum Count. The interface and properties are used together when implementing the connection harvest feature.

5.5.2 Setting a Connection to Harvestable

The `setConnectionHarvestable(boolean)` method of the `oracle.ucp.jdbc.HarvestableConnection` interface controls whether or not a connection will be harvested. This method is used as a locking mechanism when connection harvesting is enabled. For example, the method is set to `false` on a connection when the connection is being used within a transaction and must not be harvested. After the transaction completes, the method is set to `true` on the connection and the connection can be harvested if required.

Note:

All connections are harvestable, by default, when the connection harvest feature is enabled. If the feature is enabled, the `setConnectionHarvestable` method should always be used to explicitly control whether a connection is harvestable.

The following example demonstrates using the `setConnectionHarvestable` method to indicate that a connection is not harvestable when the connection harvest feature attempts to harvest connections:

```
Connection conn = pds.getConnection();

((HarvestableConnection) conn).setConnectionHarvestable(false);
```

5.5.3 Setting the Harvest Trigger Count

The connection harvest trigger count specifies the available connection threshold that triggers connection harvesting. For example, if the connection harvest trigger count is set to 10, then connection harvesting is triggered when the number of available connections in the pool drops to 10.

A value of `Integer.MAX_VALUE` (2147483647 by default) indicates that connection harvesting is disabled. The default value is `Integer.MAX_VALUE`.

The following example demonstrates enabling connection harvesting by configuring a connection harvest trigger count.

```
pds.setConnectionHarvestTriggerCount(2);
```

5.5.4 Setting the Harvest Maximum Count

The connection harvest maximum count property specifies how many borrowed connections should be returned to the pool once the harvest trigger count has been reached. The number of connections actually harvested may be anywhere from 0 to the connection harvest maximum count value. Least recently used connections are harvested first which allows very active user sessions to keep their connections the most.

The harvest maximum count value can range from `0` to the maximum connection property value. The default value is `1`. An SQLException is thrown if an out-of-range value is specified.

The following example demonstrates configuring a connection harvest maximum count.

```
pds.setConnectionHarvestMaxCount(5);
```

Note:

- If connection harvesting and abandoned connection timeout features are enabled at the same time, then the timeout processing does not reclaim the connections that are designated as nonharvestable.
- If connection harvesting and time-to-live connection timeout features are enabled at the same time, then the timeout processing reclaims the connections that are designated as nonharvestable.

Related Topics

- Controlling Reclaimable Connection Behavior

5.6 About Caching SQL Statements in UCP

This section describes how to cache SQL statements in UCP, in the following sections:

- Overview of Statement Caching in UCP
- Enabling Statement Caching in UCP

5.6.1 Overview of Statement Caching in UCP

Statement caching makes working with statements more efficient. Statement caching improves performance by caching executable statements that are used repeatedly and makes it unnecessary for programmers to explicitly reuse prepared statements. Statement caching eliminates overhead due to repeated cursor creation, repeated statement parsing and creation and reduces overhead of communication between applications and the database. Statement caching and reuse is transparent to an application. Each statement cache is associated with a physical connection. That is, each physical connection will have its own statement cache.

The match criteria for cached statements are as follows:

- The SQL string in the statement must be the same (case-sensitive) to one in the cache.
- The statement type must be the same (`prepared` or `callable`) to the one in the cache.
- The scrollable type of result sets produced by the statement must be the same (`forward-only` or `scrollable`) as the one in the cache.

Statement caching is implemented and enabled differently depending on the JDBC driver vendor. The instructions in this section are specific to Oracle's JDBC driver. Statement caching on other vendors' drivers can be configured by setting a connection property on a connection factory. Refer to the JDBC vendor's documentation to determine whether statement caching is supported and if it can be set as a connection property. UCP does support JDBC 4.0 (JDK16) APIs to enable statement pooling if a JDBC vendor supports it.

Related Topics

- Setting Connection Properties

5.6.2 Enabling Statement Caching in UCP

The maximum number of statements property specifies the number of statements to cache for each connection. The property only applies to the Oracle JDBC driver. If the property is not set, or if it is set to `0`, then statement caching is disabled. By default, statement caching is disabled. When statement caching is enabled, a statement cache is associated with each physical connection maintained by the connection pool. A single statement cache is not shared across all physical connections.

The following example demonstrates enabling statement caching:

```
pds.setMaxStatements(10);
```

Determining the Statement Cache Size

The cache size should be set to the number of distinct statements the application issues to the database. If the number of statements that an application issues to the database is unknown, use the JDBC performance metrics to assist with determining the statement cache size.

Statement Cache Size Resource Issues

Each connection is associated with its own statement cache. Statements held in a connection's statement cache may hold on to database resources. It is possible that the number of opened connections combined with the number of cached statements for each connection could exceed the limit of open cursors allowed for the database. This issue may be avoided by reducing the number of statements allowed in the cache, or by increasing the limit of open cursors allowed by the database.

5.7 UCP Best Practices

Universal Connection Pool has an extensive collection of tools and APIs to analyze connection leaks and tune up pool properties for optimizing its operation. This section describes these tools and APIs.

The `All connections in the Universal Connection Pool are in use` exception indicates the shortage of connections in the pool at a given time, which means that the pool is unable to meet the connection borrowing requests of an application. This can happen due to the following reasons:

- An application borrows connections and holds them for a long time without usage, never returning them to the pool. Connection leakage can also happen when an application borrows connections, holds them without usage, and finally returns them to the pool after a very long time. These are the classical connection leakage use cases. You must eliminate non-productive connection borrowings that last for long or infinite periods of time.
- The pool has insufficient capacity for processing the whole flow of connection borrowing requests. In this case, the connection supply of the pool is not enough to perform the expected job and this results in the exception. You must increase the pool capacity in such a case.

Following is a list of useful tools and APIs that you must be aware of prior to debugging and tuning up UCP:

- **Abandoned Connection Timeout (ACT)**: This API enables setting up a timeout for a connection that is borrowed but unused.

  See Also:

  Setting the Abandon Connection Timeout

- **Time-To-Live Connection Timeout (TTL)**: This API too enables setting up a timeout for a connection that is borrowed but unused. However, it also furnishes information about the borrowed connections that are busy with associated on-going processes. It also enables to reclaim these busy connections back to a pool and restores the capacity of the pool, in case of very long processes.

  See Also:

  Setting the Time-To-Live Connection Timeout

- **Connection Harvesting Mechanism**: This is a special API that enables UCP to always keep certain number of connections available for borrowing and in turn, helps in avoiding the `All connections in the Universal Connection Pool are in use` exception.

  See Also:

  About Harvesting Connections in UCP

- **Connection Wait Timeout (CWT)**: This is an important property when you try to tune up UCP to avoid the `All connections in the Universal Connection Pool are in use` exception. When an application attempts to borrow a connection out of a pool and there are no available connections at that time, UCP waits for an available connection to appear for the amount of time that is equal to the value of CWT. By default, CWT is set for 3 seconds. In many applications, you can increase this timeout to enable a pool to wait for longer for an available connection to appear, without getting the `All connections in the Universal Connection Pool are in use` exception.

  See Also:

  Setting the Connection Wait Timeout

- **Maximum Pool Size (`MaxPoolSize`)**: This property affects the pool capacity and helps to avoid the `All connections in the Universal Connection Pool are in use` exception. Oracle recommends to have a small pool size, typically a small number multiplied by the number of cores on a database server. It is better to increase the CWT than making `MaxPoolSize` very high.

  > **See Also:**
  >
  > Setting the Maximum Pool Size

- **Inactive Connection Timeout (ICT) in combination with MaxPoolSize**: ICT is the timeout property that enables UCP to automatically close available connections that did not have a chance to be borrowed for a particular amount of time, which is specified by the value of ICT. This way, the UCP can avoid connections if the working set of the pool is too big to perform a given throughput. For the pool to auto-tune the required number of connections in the working set of the pool, you can set the `MaxPoolSize` parameter to a big value and set the ICT accordingly.

  > **Note:**
  >
  > Setting the Inactive Connection Timeout

- **Pool Size Auto Tuner**: This tool enables UCP to automatically tune up pool size for better throughput.
- **Pool Statistical Metrics**: This is a set of statistics that helps to determine the activities and statistics of a pool, for example, the number of available connections, the number of borrowed connections, and the Average Connection Wait Time (ACWT). ACWT can find a proper value of CWT property for pool tuning. If ACWT is big, then it indicates that the UCP is close to over-using its capacity.

  > **See Also:**
  >
  > Pool Statistics

- **Pool Logging**: UCP has an extensive and flexible logging system. Logging enables you to determine events related to connection opening, connection closing, connection borrowing, and wait time of return and borrow requests.

  > **See Also:**
  >
  > Overview of Logging and Tracing in UCP

6

Labeling Connections in UCP

This chapter discusses the following topics:

6.1 Overview of Labeling Connections in UCP

Applications often initialize connections retrieved from a connection pool before using the connection. The initialization varies and could include simple state re-initialization that requires method calls within the application code or database operations that require round trips over the network. The cost of such initialization may be significant.

Labeling connections enables an application to attach arbitrary name/value pairs to a connection. The application can request a connection with the desired label from the connection pool. By associating particular labels with particular connection states, an application can retrieve an already initialized connection from the pool and avoid the time and cost of re-initialization. The connection labeling feature does not impose any meaning on user-defined keys or values; the meaning of user-defined keys and values is defined solely by the application.

Note:

If you are using connection labeling, then you cannot set the `RESET_STATE` service attribute to `LEVEL1` or `LEVEL2`.

Some of the examples for connection labeling include, role, NLS language settings, transaction isolation levels, stored procedure calls, or any other state initialization that is expensive and necessary on the connection before work can be executed by the resource.

Connection labeling is application-driven and requires the use of two interfaces. The `oracle.ucp.jdbc.LabelableConnection` interface is used to apply and remove connection labels, as well as retrieve labels that have been set on a connection. The `oracle.ucp.ConnectionLabelingCallback` interface is used to create a labeling callback that determines whether or not a connection with a requested label already exists. If no connections exist, the interface allows current connections to be configured as required. The methods of these interfaces are described in detail throughout this chapter.

6.2 Implementation of a Labeling Callback in UCP

UCP uses Database Resident Connection Pooling (DRCP) tagging infrastructure to support labeling in UCP, whether you work with single labels or multiple labels. However, the behavior with multiple labels can be a little different when you use the UCP and DRCP combination instead of only UCP.

This section discusses the following topics:

- When to Use a Labeling Callback in UCP
- Creating a Labeling Callback in UCP
- Registering a Labeling Callback in UCP
- Removing a Labeling Callback in UCP

See Also:

"Integration of UCP with DRCP"

6.2.1 When to Use a Labeling Callback in UCP

A labeling callback is used to define how the connection pool selects labeled connections and allows the selected connection to be configured before returning it to an application. Applications that use the connection labeling feature must provide a callback implementation.

A labeling callback is used when a labeled connection is requested but there are no connections in the pool that match the requested labels. The callback determines which connection requires the least amount of work in order to be re-configured to match the requested label and then enables the connection labels to be updated before returning the connection to the application. This section describes the following topics:

6.2.2 Creating a Labeling Callback in UCP

To create a labeling callback, an application implements the `oracle.ucp.ConnectionLabelingCallback` interface. One callback is created per connection pool. The interface provides the following two methods:

- The cost Method
- The configure Method

The cost Method

This method projects the cost of configuring connections considering label-matching differences. Upon a connection request, the connection pool uses this method to select a connection with the least configuration cost.

```
public int cost(Properties requestedLabels, Properties currentLabels);
```

The configure Method

This method is called by the connection pool on the selected connection before returning it to the application. The method is used to set the state of the connection and apply or remove any labels to/from the connection.

```
public boolean configure(Properties requestedLabels, Connection conn);
```

The connection pool iterates over each connection available in the pool. For each connection, it calls the `cost` method. The result of the `cost` method is an `integer` which represents an estimate of the cost required to reconfigure the connection to the required state. The larger the value, the costlier it is to reconfigure the connection. The connection pool always returns connections with the lowest cost value. The algorithm is as follows:

- If the `cost` method returns `0` for a connection, then the connection is a match. The connection pool does not call the `configure` method on the connection found and returns the connection as it is.
- If the `cost` method returns a value greater than `0`, then the connection pool iterates until it finds a connection with a cost value of `0` or runs out of available connections.
- If the pool has iterated through all available connections and the lowest cost of a connection is `Integer.MAX_VALUE` (2147483647 by default), then no connection in the pool is able to satisfy the connection request. The pool creates and returns a new connection. If the pool has reached the maximum pool size (it cannot create a new connection), then the pool either throws an SQL exception or waits if the connection wait timeout attribute is specified.
- If the pool has iterated through all available connections and the lowest cost of a connection is less than `Integer.MAX_VALUE`, then the `configure` method is called on the connection and the connection is returned. If multiple connections are less than `Integer.MAX_VALUE`, the connection with the lowest cost is returned.

Note:

A cost of `0` does not imply that `requestedLabels` equals `currentLabels`.

6.2.2.1 Example of Labeling Callback in UCP

The following example demonstrates a simple labeling callback implementation that implements both the `cost` and `configure` methods. The callback is used to find a labeled connection that is initialized with a specific transaction isolation level.

```
class MyConnectionLabelingCallback
  implements ConnectionLabelingCallback {

  public MyConnectionLabelingCallback()
  {
  }

  public int cost(Properties reqLabels, Properties currentLabels)
  {
    // Case 1: exact match
    if (reqLabels.equals(currentLabels))
    {
      System.out.println("## Exact match found!! ##");
```

```
        return 0;
      }

      // Case 2: some labels match with no unmatched labels
      String iso1 = (String) reqLabels.get("TRANSACTION_ISOLATION");
      String iso2 = (String) currentLabels.get("TRANSACTION_ISOLATION");
      boolean match =
        (iso1 != null && iso2 != null && iso1.equalsIgnoreCase(iso2));
      Set rKeys = reqLabels.keySet();
      Set cKeys = currentLabels.keySet();
      if (match && rKeys.containsAll(cKeys))
      {
        System.out.println("## Partial match found!! ##");
        return 10;
      }

      // No label matches to application's preference.
      // Do not choose this connection.
      System.out.println("## No match found!! ##");
      return Integer.MAX_VALUE;
    }

    public boolean configure(Properties reqLabels, Object conn)
    {
      try
      {
        String isoStr = (String) reqLabels.get("TRANSACTION_ISOLATION");
        ((Connection)conn).setTransactionIsolation(Integer.valueOf(isoStr));
        LabelableConnection lconn = (LabelableConnection) conn;

        // Find the unmatched labels on this connection
        Properties unmatchedLabels =
         lconn.getUnmatchedConnectionLabels(reqLabels);

        // Apply each label <key,value> in unmatchedLabels to conn
        for (Map.Entry<Object, Object> label : unmatchedLabels.entrySet())
        {
          String key = (String) label.getKey();
          String value = (String) label.getValue();
          lconn.applyConnectionLabel(key, value);
        }
      }
      catch (Exception exc)
      {
        return false;
      }
      return true;
    }
  }
```

6.2.3 Registering a Labeling Callback in UCP

A pool-enabled data source provides the `registerConnectionLabelingCallback(ConnectionLabelingCallback callback)` method for registering labeling callbacks. Only one callback may be registered on a connection pool. The following example demonstrates registering a labeling callback that is implemented in the `MyConnectionLabelingCallback` class:

```
MyConnectionLabelingCallback callback = new MyConnectionLabelingCallback();
pds.registerConnectionLabelingCallback( callback );
```

6.2.4 Removing a Labeling Callback in UCP

A pool-enabled data source provides the `removeConnectionLabelingCallback()` method for removing a labeling callback. The following example demonstrates removing a labeling callback.

```
pds.removeConnectionLabelingCallback( callback );
```

6.3 Integration of UCP with DRCP

Natively, DRCP supports connection tagging, which is a single label without weights. So, labeling with a single label works transparently if you use UCP with DRCP.

Note:

Oracle recommends that the maximum pool size of UCP should not be bigger than the size of DRCP. If the UCP pool size is bigger than the DRCP size, then you must set the `setValidateConnectionOnBorrow` property to `off`. Otherwise, UCP keeps invalidating and closing the connections that are not associated with DRCP at that moment, and keeps creating fresh connections.

See Also:

Overview of Validating Connections in UCP

Multiple label UCP connections work, but they have the following behavior changes:

- The `cost` method in the `ConnectionLabelingCallback` API is not invoked if you use UCP with DRCP using connection labeling
- UCP can invoke the `configure` method in the `ConnectionLabelingCallback` API more with DRCP configuration than without DRCP configuration.

See Also:

Oracle Database JDBC Developer's Guide for more information about DRCP

6.4 Applying Connection Labels in UCP

Labels are applied on a borrowed connection using the `applyConnectionLabel` method from the `LabelableConnection` interface. This method is typically called from the `configure` method of the labeling callback. Any number of connection labels may be cumulatively applied on a borrowed connection. Each time a label is applied to a connection, the supplied key/value pair is added to the collection of labels already applied to the connection. Only the last applied value is retained for any given key.

Note:

A labeling callback must be registered on the connection pool before a label can be applied on a borrowed connection; otherwise, an exception is thrown.

The following example demonstrates initializing a connection with a transaction isolation level and then applying a label to the connection:

```
String pname = "property1";
String pvalue = "value";
Connection conn = pds.getConnection();

// initialize the connection as required.

conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

((LabelableConnection) conn).applyConnectionLabel(pname, pvalue);
```

In order to remove a given key from the set of connection labels applied, apply a label with the key to be removed and a `null` value. This may be used to clear a particular key/value pair from the set of connection labels.

Related Topics

- Implementation of a Labeling Callback in UCP

6.5 Borrowing Labeled Connections from UCP

A pool-enabled data source provides two `getConnection` methods that are used to borrow a labeled connection from the pool. The methods are shown below:

```
public Connection getConnection(java.util.Properties labels )
   throws SQLException;

public Connection getConnection( String user, String password,
                                 java.util.Properties labels )
   throws SQLException;
```

The methods require that the label be passed to the `getConnection` method as a `Properties` object. The following example demonstrates getting a connection with the label `property1, value`.

```
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);

Connection conn = pds.getConnection(label);
```

6.6 Checking Unmatched Labels in UCP

A connection may have multiple labels that each uniquely identifies the connection based on some desired criteria. The `getUnmatchedConnectionLabels` method is used to verify which connection labels matched from the requested labels and which did not. The method is used after a connection with multiple labels is borrowed from the connection pool and is typically used by a labeling callback. The following example demonstrates checking for unmatched labels.

```
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);

Connecion conn = pds.getConnection(label);
Properties unmatched = ((LabelableConnection)
   connection).getUnmatchedConnectionLabels (label);
```

6.7 Removing a Connection Label in UCP

The `removeConnectionLabel` method is used to remove a label from a connection. This method is used after a labeled connection is borrowed from the connection pool. The following example demonstrates removing a connection label.

```
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);
Connection conn = pds.getConnection(label);
((LabelableConnection) conn).removeConnectionLabel(pname);
```

7

Controlling Reclaimable Connection Behavior

This chapter describes the following interfaces:

- AbandonedConnectionTimeoutCallback Interface
- TimeToLiveConnectionTimeoutCallback Interface

7.1 AbandonedConnectionTimeoutCallback Interface

The `AbandonedConnectionTimeoutCallback` callback interface is used for the abandoned connection timeout feature. This feature enables applications to provide customized handling of abandoned connections.The callback object either uses one of its logical connection proxies or it is registered with each pooled connection. This enables applications to perform customized handling, when a particular connection is deemed abandoned by the pool. The `handleTimedOutConnection` method is invoked when a borrowed connection is deemed abandoned by the Universal Connection Pool. Applications can perform one of the following operations on the connection:

- Completely override the pool-handling process
- Invoke additional handling actions
- Assume the default pool-handling

The JDBC applications can invoke `cancel`, `close`, and `rollback` methods on the abandoned connection within the `handleTimedOutConnection` method.

> **Note:**
>
> If you try to register more than one `AbandonedConnectionTimeoutCallback` interface on the same connection, then it results in an exception. This exception can be a `UniversalConnectionPoolException` at the pool layer or a `java.sql.SQLException`, specific to the type of the UCP Adapter like JDBC, JCA and so on.

7.2 TimeToLiveConnectionTimeoutCallback Interface

The `TimeToLiveConnectionTimeoutCallback` callback interface used for the time-to-live (TTL) connection timeout feature. This enables applications to provide customized handling for TTL timed-out connections.

The callback object either uses one of its logical connection proxies or it is registered with each pooled connection. This enables applications to perform customized handling, when the TTL of the particular connection times out.

The `handleTimedOutConnection` method is invoked when a borrowed connection is found to be TTL timed-out by the Universal Connection Pool. Applications can perform one of the following operations on the connection:

- Completely override the pool-handling process
- Invoke additional handling actions
- Assume the default pool-handling

The JDBC applications can invoke `cancel`, `close`, and `rollback` methods on the abandoned connection within the `handleTimedOutConnection` method.

Note:

If you try to register more than one `TimeToLiveConnectionTimeoutCallback` interface on the same connection, then it results in an exception. This exception can be a `UniversalConnectionPoolException` at the pool layer or a `java.sql.SQLException`, specific to the type of the UCP Adapter like JDBC, JCA, and so on.

8
Using the Connection Pool Manager

The following sections are included in this chapter:

8.1 Overview of Using the UCP Manager

The Universal Connection Pool (UCP) manager creates and maintains UCP instances. A pool instance is registered with the pool manager every time a new pool is created. This section covers the following topics:

8.1.1 About Connection Pool Manager

Applications use a connection pool manager to explicitly create and manage UCP JDBC connection pools. Applications use the manager because it offers full life cycle control, such as creating, starting, stopping, and destroying a connection pool. Applications also use the manager to perform routine maintenance on the connection pool, such as refreshing, recycling, and purging connections in a pool. Lastly, applications use the connection pool manager because it offers a centralized integration point for administrative tools and consoles.

8.1.2 Creating a Connection Pool Manager for UCP

A connection pool manager is an instance of the `UniversalConnectionPoolManager` interface, which is located in the `oracle.ucp.admin` package. The manager is a Singleton instance that is used to manage multiple connection pools per JVM. The interface includes methods for interacting with a connection pool manager. UCP includes an implementation that is used to get a connection pool manager instance. The following example demonstrates creating a connection pool manager instance using the implementation:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();
```

8.1.3 Life Cycle States of a Connection

Applications use the connection pool manager to explicitly control the life cycle of connection pools. The manager is used to create, start, stop, and destroy connection pools. Life cycle methods are included as part of the `UniversalConnectionPoolManager` interface.

Understanding Life Cycle States

The life cycle states of a connection pool affects what manager operations can be performed on a connection pool. Applications that explicitly control the life cycle of a pool must ensure that the manager's operations are used only when the pool is in an appropriate state. Life cycle constraints are discussed throughout this section.

The following list describes the life cycle states of a pool:

- Starting : Indicates that the connection pool's start method has been called and it is in the process of starting up.
- Running : Indicates that the connection pool has been started and is ready to give out connections.
- Stopping : Indicates that the connection pool is in the process of stopping.
- Stopped : Indicates that the connection pool is stopped.
- Failed : Indicates that the connection pool has encountered failures during starting, stopping, or execution.

8.1.3.1 Creating a Connection Pool

The `CreateConnectionPool` method of the Connection Manager creates and registers a connection pool. The manager uses a connection pool adapter to create the pool and relies on a pool-enabled data source to configure the pool properties. An application must not implicitly start a connection pool before using the `createConnectionPool` method to explicitly create the same pool.

The following example demonstrates creating a connection pool instance using the manager:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();

PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionPoolName("mgr_pool");
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/orcl");
pds.setUser("<user>");
pds.setPassword("<password>");

mgr.createConnectionPool((UniversalConnectionPoolAdapter)pds);
```

An application does not have to use the manager to create a pool in order for the pool to be managed. A pool that is implicitly created (that is, automatically created when using a pool-enabled data source) and configured with a pool name, is automatically registered and managed by the pool manager. Oracle recommends implicit pool creation.

Pool Naming Convention

A connection pool name must be defined as part of the configuration. The pool name provides a way to refer to specific pools when interacting with the manager. A connection pool name must be unique and cannot be used by more than one connection pool. The manager throws a `pool already exists` exception if a connection pool already exists with the same name.

Compatibility with JBoss

JBoss users can use the JBoss-specific silent reload functionality by setting the `oracle.ucp.destroyOnReload` JVM system property to `true`. When the `oracle.ucp.destroyOnReload` property is set to `true`, then the JBoss-specific behavior automatically destroys an old pool instance prior to creating a new one with the same name. If this system property is not set or set to `false`, then UCP throws a `pool already exists` exception.

8.1.3.2 Starting a Connection Pool

The manager's `startConnectionPool` method starts a connection pool using the pool name as a parameter to determine which pool to start. The pool name is defined as a pool property on a pool-enabled data source.

The following example demonstrates starting a connection pool:

```
mgr.startConnectionPool("mgr_pool");
```

An application must always create a connection pool using the manager's `createConnectionPool` method prior to starting the pool. In addition, a life cycle state exception occurs if an application attempts to start a pool that has been previously started or if the pool is in a state other than stopped or failed.

8.1.3.3 Stopping a Connection Pool

The manager's `stopConnectionPool` method stops a connection pool using the pool name as a parameter to determine which pool to stop. The pool name is defined as a pool property on the pool-enabled data source. Stopping a connection pool closes all available and borrowed connections.

The following example demonstrates stopping a connection pool:

```
mgr.stopConnectionPool("mgr_pool");
```

An application can use the manager to stop a connection pool that was started implicitly or explicitly. An error occurs if an application attempts to stop a pool that does not exist or if the pool is in a state other than started or starting.

8.1.3.4 Destroying a Connection Pool

The `destroyConnectionPool` method of the UCP Manager stops a connection pool and removes it from the connection pool manager. A pool name is used as a parameter to determine which pool to destroy. The pool name is defined as a pool property on the pool-enabled data source.

Caution:

You must destroy the pool objects explicitly, before they get out of scope, for example, if an object is an automatic variable and its scope is about to be ended. This is especially important when your application deals with multiple pool objects. If you do not destroy the pool objects explicitly, then they can cause leakage of resources like connections, statements, result sets, heap memory, and so on.

The following example demonstrates destroying a connection pool:

```
mgr.destroyConnectionPool("mgr_pool");
```

An application cannot start a connection pool that has been destroyed and must explicitly create and start a new connection pool.

8.1.4 Maintenance of Universal Connection Pool

Applications use the connection pool manager to perform maintenance on a connection pool. Maintenance includes refreshing, recycling, and purging a connection pool. The maintenance methods are included as part of the `UniversalConnectionPoolManager` interface.

Maintenance is typically performed to remove and replace invalid connections and ensures a high availability of valid connections. Invalid connections typically cannot be used to connect to a database but are still maintained by the pool. These connections waste system resources and directly affect a pool's maximum connection limit. Ultimately, too many invalid connections negatively affects an applications performance.

Note:

Applications can check whether or not a connection is valid when borrowing the connection from the pool. If an application consistently has a high number of invalid connections, additional testing should be performed to determine the cause.

Related Topics

- Overview of Validating Connections in UCP

8.1.4.1 Refreshing a Connection Pool

Refreshing a connection pool replaces every connection in the pool with a new connection. Any connections that are currently borrowed are marked for removal and refreshed after the connection is returned to the pool. The manager's `refreshConnectionPool` method refreshes a connection pool using the pool name as a parameter to determine which pool to refresh. The pool name is defined as a pool property on the pool-enabled data source.

The following example demonstrates refreshing a connection pool:

```
mgr.refreshConnectionPool("mgr_pool");
```

8.1.4.2 Recycling a Connection Pool

Recycling a connection pool replaces only invalid connection in the pool with a new connection and does not replace borrowed connections. The manager's `recycleConnectionPool` method recycles a connection pool using the pool name as a parameter to determine which pool to recycle. The pool name is defined as a pool property on the pool-enabled data source.

The `setSQLForValidateConnection` property must be set when using non-Oracle drivers. UCP uses this property to determine whether or not a connection is valid before recycling the connection.

The following example demonstrates recycling a connection pool:

```
mgr.recycleConnectionPool("mgr_pool");
```

Related Topics

- Overview of Validating Connections in UCP

8.1.4.3 Purging a Connection Pool

Purging a connection pool removes every connection (available and borrowed) from the connection pool and leaves the connection pool empty. Subsequent requests for a connection result in a new connection being created. The manager's `purgeConnectionPool` method purges a connection pool using the pool name as a parameter to determine which pool to purge. The pool name is defined as a pool property on the pool-enabled data source.

The following example demonstrates purging a connection pool:

```
mgr.purgeConnectionPool("mgr_pool");
```

Note:

Connection pool properties, such as `minPoolSize` and `initialPoolSize`, may not be enforced after a connection pool is purged.

8.2 Overview of JMX-Based Management in UCP

JMX (Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices, service-oriented networks, and JVM (Java Virtual Machine). In JMX, a given resource is instrumented by one or more Java objects known as MBeans (Managed Beans). An MBean is composed of an MBean interface and a class. The MBean interface lists the methods for all exposed attributes and operations. The class implements this interface and provides the functionality of the instrumented resource.

The MBeans are registered in a core managed object server, known as an MBean server, which acts as a management agent and can run on most devices enabled for the Java programming language. A JMX agent consists of an MBean server, in which MBeans are registered, and a set of services for handling MBeans.

See Also:

- https://docs.oracle.com/javase/tutorial/jmx/mbeans/standard.html
- *Oracle Universal Connection Pool Java API Reference*

UCP provides the following two MBeans for pool management support:

- UniversalConnectionPoolManagerMBean
- UniversalConnectionPoolMBean

Note:

All MBean attributes and operations are available only when the `UniversalConnectionPoolManager.isJmxEnabled` method returns `true`. The default value of this flag is `true`. This default value can be altered by calling the `UniversalConnectionPoolManager.setJmxEnabled` method. When an MBeanServer is not available, the `jmxFlag` is automatically set to `false`.

8.2.1 UniversalConnectionPoolManagerMBean

The `UniversalConnectionPoolManagerMBean` is a manager MBean that includes all the functionalities of a conventional connection pool manager. The `UniversalConnectionPoolManagerMBean` provides the following functionalities:

- Registering and unregistering pool MBeans
- Pool management operations like starting the pool, stopping the pool, refreshing the pool, and so on
- Starting and stopping DMS statistics
- Logging

8.2.2 UniversalConnectionPoolMBean

The `UniversalConnectionPoolMBean` is a pool MBean that covers dynamic configuration of pool properties and pool statistics. The `UniversalConnectionPoolMBean` provides the following functionalities:

- Configuring pool property attributes like size, timeouts, and so on
- Pool management operations like refreshing the pool, recycling the pool, and so on
- Monitoring pool statistics and life cycle states

9

Shared Pool Support for Multitenant Data Sources

Starting from Oracle Database 12*c* Release 2 (12.2.0.1), multiple data sources of multitenant data sources can share a common pool of connections in UCP and repurpose the connections in the common connection pool, whenever needed.

This section describes the following concepts related to the Shared Pool feature:

Note:

- Only the JDBC Thin driver supports the Shared Pool feature, and not the JDBC OCI driver.
- For using this feature, you *must* use an XML configuration file.
- This feature works with Application Containers as well. Refer to the *Oracle Multitenant Administrator's Guide* for more information about Application Containers.

Related Topics

- Sample XML Configuration File for Shared Pool

9.1 Overview of Shared Pool Support

UCP supports multiple data sources, connected to the same database, to share the same connection pool. This common connection pool is called as the Shared Pool.

In UCP, the pool instances have a one-to-one mapping with the data sources. Every data source creates its own connection pool instance and that instance is not accessible or shared by another data source, even if they internally create and cache connections to the same database and service. In this architecture, a lot of isolated connection pools are created, which causes a scalability problem because a database can scale up to only a certain number of connections.

The Shared Pool optimizes system resources for a scalable deployment of multitenant Java applications in Oracle Database Multitenant environment. This feature provides more flexibility in situations when there is an uneven load on each data source. When individual pool per data sources are created, then it is impossible to move around idle resources from an idle connection pool to a loaded one. However, when a Shared Pool is used, connections can be utilized in an efficient way by sharing and repurposing connections between the data sources. So, this feature reduces the total number of database connections, and improves resource usage, diagnosability, manageability, and scaling at the database servers.

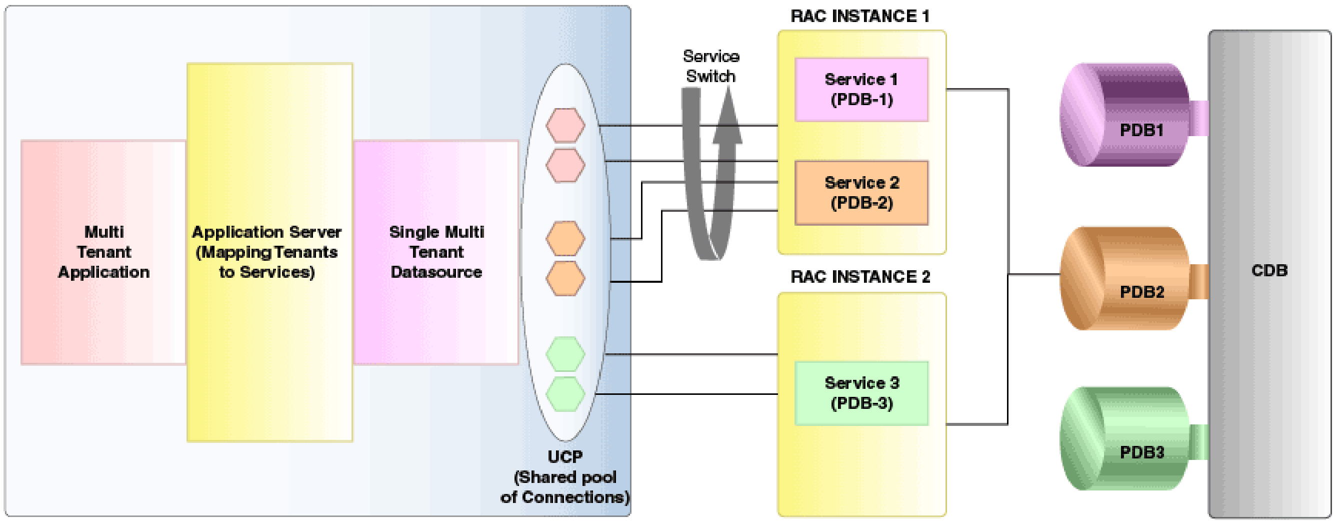
Following are the two scenarios in which you can implement this feature:

- Single Multitenant Data Source Using Shared Pool
- One Data Source per Tenant Using Shared Pool

Single Multitenant Data Source Using Shared Pool

With this configuration, multiple tenants use the common data source and a common pool to serve connections with different services applicable to each of the tenants, as illustrated in the following diagram:

Figure 9-1 Single Multitenant Data Source Using Shared Pool



The following code snippet explains how this feature works:

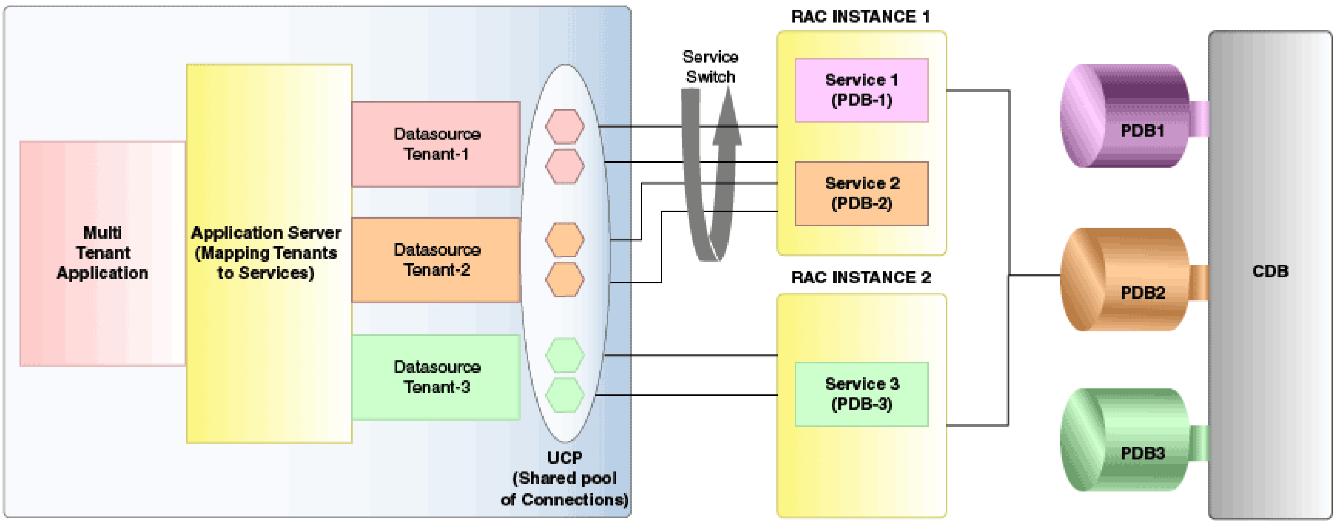
```
    PoolDataSource multiTenantDS = PoolDataSourceFactory.getPoolDataSource();

    //common user for the CDB
    multiTenantDS.setUser("c##common_user");
    multiTenantDS.setPassword("password");

    //Points to the root service of the CDB

multiTenantDS.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"
        + "(HOST=myhost)(PORT=5521))
(CONNECT_DATA=(SERVICE_NAME=root.oracle.com)))");

    // password enabled role for tenant-1
    Properties tenant1Roles = new Properties();
    tenant1Roles.put("tenant1-role", "tenant1-password");

    //Create Connection to Tenant-1 and apply the tenant specific PDB roles.
    Connection tenant1Connection =
        multiTenantDS.createConnectionBuilder()
```

```
                    .serviceName("tenant1Svc.oracle.com")
                    .pdbRoles(tenant1Roles)
                    .build();

    // password enabled role for tenant-2
    Properties tenant2Roles = new Properties();
    tenant1Roles.put("tenant2-role", "tenant2-password");

    //Create Connection to Tenant-2 and apply the tenant specific PDB roles.
    Connection tenant2Connection =
        multiTenantDS.createConnectionBuilder()
                    .serviceName("tenant2Svc.oracle.com")
                    .pdbRoles(tenant2Roles)
                    .build();
```

One Data Source per Tenant Using Shared Pool

With this configuration, multitenant applications have separate data sources per tenant and a common Shared Pool for connections. This results in the individual data sources being configured with tenant specific service information and sharing a common pool, as illustrated in the following diagram:

Figure 9-2 One Data Source per Tenant Using Shared Pool



The following code snippet explains how this feature works:

```
    // Get the datasource instance, named as "pds1" in XML configuration
file(initial-shared-pool-config.xml)
    File initialFile = new File("./UCPConfig.xml");
    InputStream targetStream = new FileInputStream(initialFile);
    PoolDataSource pds1 = PoolDataSourceFactory.getPoolDataSource("pds1", is);
    Connection pds1Conn = pds1.getConnection();

    // Get the datasource instance, named as "pds2" in XML configuration
file(initial-shared-pool-config.xml)
    PoolDataSource pds2 = PoolDataSourceFactory.getPoolDataSource("pds2");
    Connection pds2Conn = pds2.getConnection();
    // Reconfigure datasource(pds1) using the new properties
    Properties newProps = new Properties();
    newProps.put("serviceName", <newServiceName>);
    pds1.reconfigureDataSource(newProps);

    // Configure a new datasource(pds3) to running pool using the new data
source properties
    Properties dataSourceProps = new Properties();
    dataSourceProps.put("serviceName", <serviceName>);
    dataSourceProps.put("connectionPoolName", <poolName>);
    dataSourceProps.put("dataSourceName", <dataSourceName>);
    PoolDataSource pds3 =
PoolDataSourceFactory.getPoolDataSource(dataSourceProps);

    // Reconfigure connection pool("pool1") using the new properties

    Properties newPoolProps = new Properties();
    newPoolProps.put("initialPoolSize", <newInitialPoolSizeValue>);
    newPoolProps.put("maxPoolSize", <newMaxPoolSizeValue>);
    UniversalConnectionPoolManager ucpMgr =
UniversalConnectionPoolManagerImpl.getUniversalConnectionPoolManager();
    ucpMgr.reconfigureConnectionPool("pool1", newPoolProps);
```

You can also implement this feature in the following way:

```
    // UCP XML configuration file path in case of Unix
    String file_URI = "file:/user/app/sharedpool/initial-shared-pool-
config.xml";

    // UCP XML configuration file path in case of Windows
    String file_URI = "file:/D:/user/app/sharedpool/initial-shared-pool-
config.xml";

    // Java system property to specify XML configuration file location
    System.setProperty("oracle.ucp.jdbc.xmlConfigFile",<file_URI>);

    // Get the datasource instance, named as "pds1" in XML configuration
file(initial-shared-pool-config.xml)
    PoolDataSource pds1 = PoolDataSourceFactory.getPoolDataSource("pds1");
    Connection pds1Conn = pds1.getConnection();

    // Get the datasource instance, named as "pds2" in XML configuration
file(initial-shared-pool-config.xml)
    PoolDataSource pds2 = PoolDataSourceFactory.getPoolDataSource("pds2");
    Connection pds2Conn = pds2.getConnection();

    // Reconfigure datasource(pds1) using the new properties
    Properties newProps = new Properties();
    newProps.put("serviceName", <newServiceName>);
    pds1.reconfigureDataSource(newProps);

    // Configure a new datasource(pds3) to running pool using the new data
source properties
```

```
    Properties dataSourceProps = new Properties();
    dataSourceProps.put("serviceName", <serviceName>);
    dataSourceProps.put("connectionPoolName", <poolName>);
    dataSourceProps.put("dataSourceName", <dataSourceName>);
    PoolDataSource pds3 =
PoolDataSourceFactory.getPoolDataSource(dataSourceProps);

    // Reconfigure connection pool("pool1") using the new properties

    Properties newPoolProps = new Properties();
    newPoolProps.put("initialPoolSize", <newInitialPoolSizeValue>);
    newPoolProps.put("maxPoolSize", <newMaxPoolSizeValue>);
    UniversalConnectionPoolManager ucpMgr =
UniversalConnectionPoolManagerImpl.getUniversalConnectionPoolManager();
    ucpMgr.reconfigureConnectionPool("pool1", newPoolProps);
```

Note:

- UCP uses a service switch for implementing this feature. However, the service switch in Shared Pools is supported only for homogenous services. There is no support for heterogeneous services (heterogeneity in terms of service attributes like Transaction Guard and Application Continuity) in Shared Pools.
- For the XML configuration file used in the code snippets, refer to the “XML Configuration File Required for Shared Pool Support” section.

9.2 Prerequisites for Supporting Shared Pool

This section describes the prerequisites for multitenant data sources to use the Shared Pool.

- You must provide the initial configuration of Shared Pools through an XML configuration file. You can specify the initial XML configuration file for UCP through the input stream of the XML file, in the following way:

  ```
  PoolDDataSourceFactory.getPoolDataSource(String pds, InputStream is);
  ```

  You can also specify the initial XML configuration file for UCP through the system property `oracle.ucp.jdbc.xmlConfigFile`, but it is an obsolete way of configuring the XML file and you must avoid using this option. The location of the initial XML configuration file should be specified as a URI. For example, `file:/user_directory/ucp.xml`.

  The `configuration.xsd` schema file is included in the `ucp.jar` file for reference. Refer to this file while creating a UCP XML configuration file.
- During the reconfiguration of a shared pool, updated pool properties should be provided through reconfiguration APIs.
- Always use application service for the services used for Shared Pool, and for the individual tenant data source specific services. Connections are not repurposed or reused when an Administrative service or default PDB services are used.
- The various services accessed through the Shared Pool must be homogenous, that is, they should have similar properties with respect to Application Continuity (AC) and so on.

- The Shared Pool must be configured with a single user, and this user should be a common user configured on the CDB. The common user should have the following privileges - `CREATE SESSION`, `ALTER SESSION`, and `SET CONTAINER`. The common user should also have the execute permission on the `DBMS_SERVICE_PRVT` package.

  Note:

  – If the common user needs specific roles or password-enabled roles per tenant, then these roles should be specified in the respective tenant data source properties.

  – The advantage of the `SET CONTAINER` statement is that the pool does not have to create a new connection to a PDB, if there is an existing connection to a different PDB. The pool can use the existing connection and can connect to the desired PDB through the `SET CONTAINER` statement.

- Connection repurposing among various tenant connections in the Shared Pool happens only when the total number of the connections in the pool reaches the connection repurpose threshold (if configured on the pool) and the minimum pool size.
- The URL specified for the Shared Pool in the XML configuration file must have the LONG format, with service name explicitly specified. Short format or Easy Connection URL is not supported.

9.3 Configuring the Shared Pool

This section describes how to configure the Shared Pool.

The following sections describe the Shared Pool configuration:

- Initial Configuration of the Pool
- Reconfiguration of the Pool

Initial Configuration of the Pool

For the initial configuration of the pool, get a data source instance by using the XML configuration file and then, using that data source, get a connection from a Shared Pool.

```
    // Get the datasource instance, named as "pds1" in XML configuration 
file(initial-shared-pool-config.xml)
    File initialFile = new File("./UCPConfig.xml");
    InputStream targetStream = new FileInputStream(initialFile);
    PoolDataSource pds1 = PoolDataSourceFactory.getPoolDataSource("pds1", is);
    Connection pds1Conn = pds1.getConnection();
```

Reconfiguration of the Pool

- The following code snippet shows how to reconfigure the data source that you obtained during the initial configuration of the pool:

```
 // Reconfigure datasource(pds1) using the new properties
    Properties newProps = new Properties();
    newProps.put("serviceName", <newServiceName>);
    pds1.reconfigureDataSource(newProps);
```

- The following code snippet shows how to add a new data source to an already running Shared Pool:

```
// Configure a new datasource(pds3) to running pool using the new data
source properties
    Properties dataSourceProps = new Properties();
    dataSourceProps.put("serviceName", <serviceName>);
    dataSourceProps.put("connectionPoolName", <poolName>);
    dataSourceProps.put("dataSourceName", <dataSourceName>);
    PoolDataSource pds3 =
PoolDataSourceFactory.getPoolDataSource(dataSourceProps);
```

- The following code snippet shows how to reconfigure the connection pool:

```
// Reconfigure connection pool("pool1") using the new properties

    Properties newPoolProps = new Properties();
    newPoolProps.put("initialPoolSize", <newInitialPoolSizeValue>);
    newPoolProps.put("maxPoolSize", <newMaxPoolSizeValue>);
    UniversalConnectionPoolManager ucpMgr =
UniversalConnectionPoolManagerImpl.getUniversalConnectionPoolManager();
    ucpMgr.reconfigureConnectionPool("pool1", newPoolProps);
```

9.4 UCP APIs for Shared Pool Support

New Methods in PoolDataSource Interface

The following methods have been introduced in the `oracle.ucp.jdbc.PoolDataSource` interface:

- `reconfigureDataSource(Properties configuration)`
- `getMaxConnectionsPerService()`
- `getServiceName()`
- `getPdbRoles()`
- `getConnectionRepurposeThreshold()`
- `setConnectionRepurposeThreshold(int threshold)`

New Methods in PoolDataSourceFactory Class

The following methods have been introduced in the `oracle.ucp.jdbc.PoolDataSourceFactory` class:

- `getPoolDataSource(String dataSourceName)`
- `getPoolDataSource(Properties configuration)`
- `getPoolXADataSource(String dataSourceName)`
- `getPoolXADataSource(Properties configuration)`

New Method in oracle.ucp.admin.UniversalConnectionPoolManager Interface

The following method has been introduced in the `oracle.ucp.admin.UniversalConnectionPoolManager` interface:

```
reconfigureConnectionPool(String poolName , Properties configuration)
```

New Method in oracle.ucp.admin.UniversalConnectionPool Interface

The following method has been introduced in the `oracle.ucp.admin.UniversalConnectionPool` interface:

- `isShareable()`
- `getMaxConnectionsPerService()`
- `setMaxConnectionsPerService(int maxConnectionsPerService)`

See Also:

Oracle Universal Connection Pool Java API Reference for more information about these methods.

9.5 Sample XML Configuration File for Shared Pool

initial-shared-pool-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ucp-properties>
    <connection-pool
        connection-pool-name="pool1"
        connection-factory-class-name="oracle.jdbc.pool.OracleDataSource"
        url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=host_name)
(PORT=1521)(PROTOCOL=tcp))( CONNECT_DATA=(SERVICE_NAME=myorcldbservicename)))"
        user="C##CommonUser"
        password=password
        initial-pool-size="10"
        min-pool-size="5"
        max-pool-size="20"
        connection-repurpose-threshold="13"
        max-connections-per-service="15"
        validate-connection-on-borrow="true"
        sql-for-validate-connection="select 1 from dual"
        shared="true"
    >

        <connection-property name="oracle.jdbc.ReadTimeout" value="2000"/>
        <connection-property name="oracle.net.OUTBOUND_CONNECT_TIMEOUT"
value="2000"/>

        <data-source
            data-source-name="pds1"
```

```
                service=pdb1_service_name
                description="pdb1 data source"/>

            <data-source
                data-source-name="pds2"
                service=pdb2_service_name
                description="pdb2 data source"/>


        </connection-pool>
    </ucp-properties>
```

10

Using Oracle RAC Features

The following sections are included in this chapter:

10.1 Overview of Oracle RAC Features

UCP JDBC connection pools provide a tight integration with various Oracle Real Application Clusters (Oracle RAC) Database features. The features include Fast Connection Failover (FCF), Run-Time Connection Load Balancing, and Connection Affinity. These features require the use of an Oracle JDBC driver, Oracle RAC database, and the Oracle Notification Service library (`ons.jar`) that is included with the Oracle Client software.

Applications use Oracle RAC features to maximize connection performance and availability and to mitigate down-time due to connection problems. Applications have different availability and performance requirements and should implement Oracle RAC features accordingly.

Note:

Starting from Oracle Database 11*g* Release 1 (11.2), FCF is also supported by Oracle Restart on a single instance database. Oracle Restart was previously known as Single-Instance High Availability (SIHA).

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about these technologies
- *Oracle Database Administrator's Guide* for more information about Oracle Restart

Generic High Availability and Performance Features

The UCP APIs and connection pool properties include many high availability and performance features that do not require an Oracle RAC database. These features work well with both Oracle and non-Oracle connections and are discussed throughout this guide. For example: validating connections on borrow; setting timeout properties; setting maximum reuse properties; and connection pool manager operations are all used to ensure a high-level of connection availability and optimal performance.

Note:

Generic high availability and performance features work slightly better when using Oracle connections because UCP leverages Oracle JDBC internal APIs.

Database Version Compatibility for Oracle RAC

The following table lists supported Database versions for various Oracle RAC features:

Table 10-1 Oracle RAC Version Compatibility

Feature	Supported Database Version
Fast Connection Failover	Oracle Database 10.1.x and later versions
Run-time Connection Load-Balancing	Oracle Database 10.2.x and later versions
Web Session Affinity	Oracle Database 11.1.x and later versions
Transaction-Based Affinity	Oracle Database 10.2.x and later versions (Oracle Database 11.1.x recommended)

Oracle JDBC Driver Version Compatibility for Oracle RAC

Oracle JDBC driver 10.1.x and later versions are supported with Oracle RAC features.

10.2 About Fast Connection Failover

This section contains the following subsections:

- Overview of Fast Connection Failover
- What is Fast Connection Failover
- Fast Connection Failover Prerequisites
- Example of Fast Connection Failover Configuration
- Enabling Fast Connection Failover
- What is ONS
- Configuring the Connection URL

10.2.1 Overview of Fast Connection Failover

The Fast Connection Failover (FCF) feature is a Fast Application Notification (FAN) client implemented through the connection pool. The feature requires the use of an Oracle JDBC driver and high availability (HA) database configurations like Oracle RAC, Single Restart, or Active Data Guard.

Note:

This section describes only the steps that an application must perform when using FCF with Oracle RAC.

See Also:

Oracle Real Application Clusters Administration and Deployment Guide for more information

FCF is useful in the following scenarios:

- **Unplanned Outages:** FCF rapidly detects non-functioning connections, and then terminates and removes them from the pool. Connection removal relies on terminating the sever-socket connections rapidly to prevent the system from becoming non-responsive. Borrowed and in-use connections are interrupted only for unplanned outages.
- **Planned Outages:** FCF does not interrupt and close the borrowed or in-use connections until work is done and control of the connection is returned to the pool.
- **Irrecoverable Connection Errors and Exceptions:** FCF encapsulates irrecoverable connection errors and exceptions into the `isValid` API for robust and efficient retries.
- **Addition of New Nodes to the Cluster:** FCF recognizes the new nodes that join an Oracle RAC cluster and places new connections on that node appropriately for delivering maximum quality of service to applications at run time. This facilitates middle-tier integration of Oracle RAC node joins and work-request routing from the application tier.
- **Run-Time Work Requests:** FCF distributes run-time work requests to all active Oracle RAC instances.

Unplanned Shutdown Scenarios

FCF supports unplanned shutdown scenarios by detecting and removing stale connections to an Oracle RAC cluster. Stale connections include connections that do not have a service available on any instance in an Oracle RAC cluster due to service-down and node-down events. Borrowed connections and available-but-stale connections are detected, and their network connection is severed before removing them from the pool. These removed connections are not replaced by the pool. Instead, the application must retry connections before performing any work with a connection.

Note:

Borrowed connections are immediately terminated and closed during unplanned shutdown scenarios. Any on-going transactions immediately receive an exception.

Planned Shutdown Scenarios

FCF supports planned shutdown scenarios where an Oracle RAC service can be gracefully shutdown. In such scenarios, stale borrowed connections are marked, and are terminated and removed after they are returned to the pool. Any on-going transactions do not see any difference and proceed to complete.

The primary difference between unplanned and planned shutdown scenarios is how borrowed connections are handled. Stale connections that are idle in the pool (not borrowed) are removed in the same manner as the unplanned shutdown scenario.

UCP also supports graceful connection draining from any planned-down Oracle RAC instance. Affected borrowed connections are removed smoothly over a grace period, instead of immediate removal upon their return to the pool. This helps in avoiding throughput impact and logon storms during any service relocation. In the FAN events, UCP uses the value of the `drain_timeout`[1] parameter as the grace period, when doing graceful draining.

Oracle RAC Instance Rejoin and New Instance Scenarios

FCF supports scenarios where an Oracle RAC cluster adds instances that provide a service of interest. The instance may be new to the cluster or may have been restarted after a down event. In both cases, UCP recognizes the new instance and creates connections to the node as required.

Related Topics

- Checking If a Connection Is Valid
- Enabling Fast Connection Failover

10.2.2 What is Fast Connection Failover?

After Fast Connection Failover is enabled, the mechanism is automatic; no application intervention is needed. This section discusses how a connection failover is presented to an application and what steps the application takes to recover, in the following sections:

- What the Application Sees
- How FCF Works

10.2.2.1 What the Application Sees

By the time an Oracle RAC service failure is propagated to the JDBC application, the database already rolls back the local transaction. The cache manager then cleans up all invalid connections. When an application holding an invalid connection tries to do work through that connection, it is possible to receive `SQLException, ORA-17008, Closed Connection`.

When an application receives a `Closed Connection` error message, it should do the following:

1. Retry the connection request. This is essential, because the old connection is no longer open.
2. Replay the transaction. All work done before the connection was closed has been lost.

Note:

The application should not try to roll back the transaction. The transaction was already rolled back in the database by the time the application received the exception.

10.2.2.2 How FCF Works

Under Fast Connection Failover, each connection in the cache maintains a mapping to a service, instance, database, and host name.

When a database generates an Oracle RAC event, that event is forwarded to the JVM in which JDBC is running. A daemon thread inside the JVM receives the Oracle RAC event and passes

[1] The drain_time parameter specifies the time in seconds, during which a service drains.

it on to the Connection Cache Manager. The Connection Cache Manager then throws SQL exceptions to the applications affected by the Oracle RAC event.

A typical failover scenario may work like the following:

1. A database instance fails, leaving several stale connections in the cache.
2. The Oracle RAC mechanism in the database generates an Oracle RAC event which is sent to the JVM containing JDBC.
3. The daemon thread inside the JVM finds all the connections affected by the Oracle RAC event, notifies them of the closed connection through SQL exceptions, and rolls back any open transactions.
4. Each individual connection receives a SQL exception and must retry.

10.2.3 Fast Connection Failover Prerequisites

Fast Connection Failover is available under the following circumstances:

- The Universal Connection Pool is enabled.

  Fast Connection Failover works in conjunction with the JDBC connection caching mechanism. This helps applications manage connections to ensure high availability.
- The application uses service names to connect to the database.

  The application cannot use service identifiers.
- The underlying database has Oracle Database 12*c* Release 1 (12.1) or later Real Application Clusters (Oracle RAC) capability or Oracle Data Guard configured with either single instance Databases or Oracle RAC.

  If failover events are not propagated, then connection failover cannot occur.
- Oracle Notification Service (ONS) is configured and available on the node where JDBC is running.

  JDBC depends on ONS to propagate database events and notify JDBC of them.
- The Java Virtual Machine (JVM) in which your JDBC instance is running must have `oracle.ons.oraclehome` set to point to your *`ORACLE_HOME`*.

10.2.4 Example of Fast Connection Failover Configuration

The following example demonstrates a connection pool that uses the FCF feature. FCF is configured through a pool-enabled data source. The example includes enabling FCF, configuring the Oracle Notification Service (ONS) and configuring a connection URL. These topics are discussed after the example.

The `isValid` method of the `oracle.ucp.jdbc.ValidConnection` interface is typically used in conjunction with the FCF feature and is used to check if a borrowed connection is still usable after an SQL exception has been thrown due to an Oracle RAC down event. For example:

```
try {  conn = pds.getConnection;  ...}catch (SQLException sqlexc)
{
   if (conn == null || !((ValidConnection) conn).isValid())

   // take the appropriate action

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

Example 10-1 Fast Connection Failover Configuration Example

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("FCFSamplePool");
pds.setFastConnectionFailoverEnabled(true);
pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=
/oracle11/onswalletfile");
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin@(DESCRIPTION= "+
   "(LOAD_BALANCE=on)"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=racnode1) (PORT=1521))"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))"+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");
...
```

Related Topics

- Checking If a Connection Is Valid

10.2.5 Enabling Fast Connection Failover

The FCF pool property is used to enable and disable FCF. FCF is disabled by default. The following example demonstrates enabling FCF as shown in Example 10-1.

```
pds.setFastConnectionFailoverEnabled(true);
```

Note:

Starting from Oracle Database 12*c* Release 1 (12.1.0.2), UCP supports the `oracle.ucp.PlannedDrainingPeriod` system property. It specifies the grace time period (in integer seconds) over which the pool smoothly drains the borrowed connections affected by a planned shut down. Draining starts when the same Database service becomes available on another instance different from the one that is going down.

When this property is not set, or set to `0`, then the pool closes any affected borrowed connection immediately when it is returned to the pool.

Querying Fast Connection Failover Status

An application determines if Fast Connection Failover is enabled by calling `OracleDataSource.getFastConnectionFailoverEnabled`, which returns `true` if failover is enabled, `false` otherwise.

Note:

FCF must also be enabled to use run-time connection load balancing and connection affinity. These features are discussed later in this chapter.

10.2.6 What is ONS?

FCF relies on the Oracle Notification Service (ONS) to propagate database events between the connection pool and the Oracle RAC database. At run time, the connection pool must be able to setup an ONS environment. ONS (`ons.jar`) is included as part of the Oracle Client software. ONS can be configured using either remote configuration or client-side ONS daemon configuration. Remote configuration is the preferred configuration for standalone client applications. This section discusses the following topics:

- Overview of ONS Configuration File
- Remote Configuration of ONS
- Configuration of Client-Side ONS Daemon

10.2.6.1 Overview of ONS Configuration File

ONS configuration is controlled by the ONS configuration file, *`ORACLE_HOME`*`/opmn/conf/ons.config`. This file tells the ONS daemon how it should behave. Configuration information within `ons.config` is defined in simple name and value pairs.

Some parameters in the `ons.config` file are required and some are optional. Table 10-2 lists the required ONS configuration parameters and Table 10-3 lists the optional ONS configuration parameters. ONS must be refreshed after updating the `ons.config` file.

Table 10-2 Required ONS Configuration Parameters

Parameter	Explanation
`localport`	Specifies the port that ONS binds to on the local host interface to talk to local clients. For example, `localport=4100`
`remoteport`	Specifies the port that ONS binds to on all interfaces for talking to other ONS daemons. For example, `remoteport=4200`
`nodes`	Specifies a list of other ONS daemons to talk to. Node values are given as a comma-delimited list of either host names or IP addresses plus ports. The port value that is given is the remote port that each ONS instance is listening on. In order to maintain an identical file on all nodes, the `host:port` of the current ONS node can also be listed in the nodes list. It will be ignored when reading the list. For example, `nodes=myhost.example.com:4200,123.123.123.123:4200` The nodes listed in the nodes line correspond to the individual nodes in the Oracle RAC instance. Listing the nodes ensures that the middle-tier node can communicate with the Oracle RAC nodes. At least one middle-tier node and one node in the Oracle RAC instance must be configured to see one another. As long as one node on each side is aware of the other, all nodes are visible. You need not list every single cluster and middle-tier node in the ONS configuration file of each Oracle RAC node. In particular, if one ONS configuration file cluster node is aware of the middle tier, then all nodes in the cluster are aware of it.

Table 10-3 Optional ONS Configuration Parameters

Parameter	Description

`logcomp`	Specifies the ONS components to log. The format is as follows:
	`<component>[<subcomponent>,...];<component>[<subcomponent>,...];...`
	If no subcomponents need to be specified, then do not include the brackets ([]) after the component name. To exclude messages from a subcomponent, precede the subcomponent name with an exclamation mark (!). For example, to exclude messages from the `topology` subcomponent, you use the following format:
	`[all,!topology]`
	Note that before specifying a subcomponent from which you want to exclude messages, you must first ensure that the subcomponent includes the messages.
	Following are the valid values for components:
	• `internal`
	• `ons`
	If you specify the component as `internal`, then there are no valid values for subcomponent. If you specify the component as `ons`, then you can specify the following values for subcomponent:
	• `all`: Specifies all messages
	• `ons`: ONS local information
	• `listener`: ONS listener information
	• `discover`: ONS discover (server or multicast) information
	• `servers`: ONS remote servers currently up and connected to the cluster
	• `topology`: ONS current cluster wide server connection topology
	• `server`: ONS remote server connection information
	• `client`: ONS client connection information
	• `connect`: ONS generic connection information
	• `subscribe`: ONS client subscription information
	• `message`: ONS notification receiving and processing information
	• `deliver`: ONS notification delivery information
	• `special`: ONS special notification processing
	• `internal`: ONS internal resource information
	• `secure`: ONS SSL operation information
	• `workers`: ONS worker threads
	The following example shows that you want to log messages for all the subcomponents under `ons`, except the `secure` subcomponent:
	`logcomp=ons[all,!secure]`

`logfile`	Specifies a log file that ONS should use for logging messages. The default value for log file is `$ORACLE_HOME/opmn/logs/ons.log`. For example, `logfile=/private/oraclehome/opmn/logs/myons.log`

Table 10-3 (Cont.) Optional ONS Configuration Parameters

Parameter	Description
`walletfile`	Specifies the wallet file used by the Oracle Secure Sockets Layer (SSL) to store SSL certificates. If a wallet file is specified to ONS, then it uses SSL when communicating with other ONS instances and require SSL certificate authentication from all ONS instances that try to connect to it. This means that if you want to turn on SSL for one ONS instance, then you must turn it on for all instances that are connected. This value should point to the directory where your `ewallet.p12` file is located. For example, `walletfile=/private/oraclehome/opmn/conf/ssl.wlt/default`
`useocr`	Specifies the value, reserved for use on the server-side, to indicate ONS whether it should store all Oracle RAC nodes and port numbers in Oracle Cluster Registry (OCR) instead of the ONS configuration file or not. A value of `useocr=on` is used to store all Oracle RAC nodes and port numbers in Oracle Cluster Registry (OCR). Do not use this option on the client-side.
`allowgroup`	Specifies the ONS setting to indicate the user group connecting to the `localport`. When set to `true`, ONS allows users within the same OS group to connect to its local port. When set to `false`, ONS only allows the same user running the ONS daemon to access its local port. The default value of this parameter is `false`. When using remote ONS configuration, there is no need to set this parameter.

The `ons.config` file allows blank lines and comments on lines that begin with the number sign (`#`).

10.2.6.2 Remote Configuration of ONS

UCP supports remote configuration of ONS through the `ONSConfiguration` pool property. The `ONSConfiguration` pool property value is a string that closely resembles the content of the `ons.config` file. The string contains a list of `name=value` pairs separated by a new line character (`\n`). You can set this pool property in the following two ways:

- The name can be one of the following: `nodes`, `walletfile`, or `walletpassword`. The parameter string should at least specify the ONS configuration `nodes` attribute as a list of `host:port` pairs separated by a comma. SSL is used when the `walletfile` attribute is specified as an Oracle wallet file.

  The following example demonstrates an ONS configuration string as shown in Example 10-1:

  ```
  ...
  pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=/oracle11/
  onswalletfile");
  ...
  ```

- The name can be only `propertiesfile`. The value is the location of an ONS-specific Java properties file. This file must contain the `oracle.ons.nodes` property, and one or both of the following ONS Java properties:
  - `oracle.ons.walletfile`
  - `oracle.ons.walletpassword`

    The following example illustrates such an `ONSConfiguration` string:

    ```
    pds.setONSConfiguration("propertiesfile=/usr/ons/ons.properties");
    ```

The following is an example of the content of the Java properties `ons.properties` file:

```
oracle.ons.nodes=racnode1:4200,racnode2:4200
oracle.ons.walletfile=/oracle11/onswalletfile
```

Note:

The parameters in the configuration string must match those for the Oracle RAC Database. In addition, if you are using Oracle Application Server, then you must configure ONS using procedures that are applicable to the server.

For standalone Java applications, you must configure ONS using the `setONSConfiguration` method. However, if your application meets the following requirements, then you no longer need to call the `setONSConfiguration` method for enabling FCF:

- Your application is using Oracle Database 12*c* Release 1 (12.1) or later UCP and Oracle RAC Database 12*c* Release 1 (12.1) or later
- Your application does not require ONS wallet or keystore

10.2.6.3 Configuration of Client-Side ONS Daemon

Client-side ONS daemon configuration is typical of applications that run on a middle-tier server such as the Oracle Application Server. Clients in this scenario directly configure ONS by updating the `ons.config` file. The location of the file may be different depending on the platform. Example 10-2 demonstrates an `ons.config` file for Example 10-1:

Note:

For client-side ONS daemon configuration, if the operating system (OS) user that starts the connection pool and the OS user that starts the client-side daemon are different, then they both *must* belong to the same OS group. Also, the value of the `allowgroup` parameter must be set to `true` in the `ons.config` file.

After configuring ONS, you start the ONS daemon with the `onsctl` command. You *must* make sure that an ONS daemon is running at all times.

Using the onsctl Command

After configuring, use `*ORACLE_HOME*/opmn/bin/onsctl` to start, stop, reconfigure, and monitor the ONS daemon. Table 10-4 is a summary of the commands that `onsctl` supports.

Table 10-4 onsctl Commands

Command	Effect	Output
`start`	Starts the ONS daemon	`onsctl: ons started`
`stop`	Stops the ONS daemon	`onsctl: shutting down ons daemon...`
`ping`	Verifies whether or not the ONS daemon is running	`ons is running ...`

Table 10-4 (Cont.) onsctl Commands

Command	Effect	Output
`reconfig`	Triggers a reload of the ONS configuration without shutting down the ONS daemon	
`help`	Prints a help summary message for onsctl	
`detailed`	Prints a detailed help message for onsctl	

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

Note:

- The Java Virtual Machine (JVM), in which your JDBC instance is running, must have the `oracle.ons.oraclehome` system property set to the location of `ORACLE_HOME` before starting the application. For example:

  ```
  java -Doracle.ons.oraclehome=$ORACLE_HOME ...
  ```

- Oracle recommends remote configuration of ONS for UCP.

Note:

In Oracle RAC 12.1.0.2.0, by default, server installation requires the value of the `walletfile` ONS parameter to be set, and enforces the use of SSL for all ONS connections.

If you have a UCP application that is already using the `walletfile` parameter in the ONS remote configuration string or local configuration file, then the only requirement is that, for the same topology, the wallet file on the client side must have the same content as the wallet file on the server side. You can make a copy of the server-side file and make it available on the client side.

For UCP applications that are using Oracle RAC features without setting the `walletfile` parameter, you must perform one of the following:

- Add the walletfile parameter setting to the ONS remote configuration string or local configuration file, as shown in Example 10-1. Keep in mind that, for the same topology, the wallet file on the client side must have the same content as the wallet file on the Oracle RAC server side.
- Run the following command to remove the `walletfile` parameter setting from both client and server ONS configuration string and the local configuration file:

  ```
  srvctl modify nodeapps -clientdata
  ```

For secure communication, the ONS auto-configuration in Oracle RAC 12.1.x no longer works when Oracle RAC 12.1.0.2.0 is first installed or patched. Applications have to use explicit ONS configuration (remote or local) instead, and make one of the changes previously discussed.

Example 10-2 Example of a Sample ons.config File

```
# This is an example ons.config file
#
# The first three values are required
localport=4100
remoteport=4200
nodes=racnode1.example.com:4200,racnode2.example.com:4200
```

10.2.7 Configuring the Connection URL

The connection URL of a connection factory must use the service name syntax when using FCF. The service name is used to map the connection pool to the service. In addition, the factory class must be an Oracle factory class. The following example demonstrates configuring the connection URL as shown in Example 10-1:

```
...
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin@//host:port/service_name");
...
```

Note:

An exception is thrown if a service identifier (SID) is specified for the connection URL when FCF is enabled.

The following examples demonstrate valid connection URL syntax when connecting to an Oracle RAC database. Examples for both the Oracle JDBC thin and Oracle OCI driver are included. Notice that the URL can be used to explicitly enable load balancing among Oracle RAC nodes:

Valid Connection URL Usage

```
pds.setURL("jdbc:oracle:thin@//host:port/service_name");

pds.setURL("jdbc:oracle:thin@//cluster-alias:port/service_name");

pds.setURL("jdbc:oracle:thin:@(DESCRIPTION= "+
   "(LOAD_BALANCE=on)"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host1)(PORT=1521))"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))"+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:thin:@(DESCRIPTION= "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521)) "+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:oci:@TNS_ALIAS");

pds.setURL("jdbc:oracle:oci:@(DESCRIPTION= "+
   "(LOAD_BALANCE=on) "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=1521)) "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521)) "+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:oci:@(DESCRIPTION= "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521)) "+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");
```

10.3 About Run-Time Connection Load Balancing

This section contains the following subsections:

- Overview of Run-Time Connection Load Balancing
- Setting Up Run-Time Connection Load Balancing

10.3.1 Overview of Run-Time Connection Load Balancing

In an Oracle Real Application Clusters environment, a connection could belong to any instance that provides the relevant service. In the best case, all instances perform equally well and randomly retrieving a connection from the cache is appropriate.

However, when one instance performs better than others, random selection of a connection is inefficient. The run-time connection load balancing feature enables routing of work requests to an instance that offers the best performance, minimizing the need to relocate work.

UCP JDBC connection pools leverage the load balancing functionality provided by an Oracle RAC database. Run-time connection load balancing requires the use of an Oracle JDBC driver and an Oracle RAC database.

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

Run-time connection load balancing is useful when:

- Traditional balancing of workload is not optimal
- Requests must be routed to make optimal use of resources in a clustered database
- Capacity within the cluster differs and is expected to change over time
- The need to avoid sending work to slow, hung, and non-functioning nodes is required

UCP uses the Oracle RAC Load Balancing Advisory. The advisory is used to balance work across Oracle RAC instances and is used to determine which instances offer the best performance. Applications transparently receive connections from instances that offer the best performance. Connection requests are quickly diverted from instances that have slowed, are not responding, or that have failed.

Run-time connection load balancing provides the following benefits:

- Manages pooled connections for high performance and scalability
- Receives continuous recommendations on the percentage of work to route to database instances
- Adjusts distribution of work based on different back-end node capacities such as CPU capacity or response time
- Reacts quickly to changes in cluster reconfiguration, application workload, overworked nodes, or hangs
- Receives metrics from the Oracle RAC Load Balance Advisory. Connections to well performing instances are used most often. New and unused connections to under-performing instances will gravitate away over time. When distribution metrics are not received, connection are selected using a random choice.

10.3.2 Setting Up Run-Time Connection Load Balancing

Run-time connection load balancing requires that FCF is enabled and configured properly.

In addition, you must configure the Oracle RAC Load Balancing Advisory with service-level goals for each service for which load balancing is enabled:

- The service goal must be set to one of the following:
  - `DBMS_SERVICE.SERVICE_TIME`
  - `DBMS_SERVICE.THROUGHPUT`

  The service goal can be set using the `goal` parameter, and the connection balancing goal can be set using the `clb_goal` parameter.
- The connection balancing goal must be set to `SHORT`. For example,

  ```
  EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sjob' -, goal =>
     DBMS_SERVICE.GOAL_THROUGHPUT -, clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
  ```

  Or

```
EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sjob' -, goal =>
   DBMS_SERVICE.GOAL_SERVICE_TIME -, clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
```

The connection balancing goal can also be set by calling the `DBMS_SERVICE.create_service` procedure.

Note:

You can set the connection balancing goal to `LONG`. However, this is mostly useful for closed workloads, that is, when the rate of completing work is equal to the rate of starting new work.

Related Topics

- About Fast Connection Failover

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

10.4 About Connection Affinity

This section contains the following subsections:

- Overview of Connection Affinity
- Setting Up Connection Affinity

10.4.1 Overview of Connection Affinity

UCP JDBC connection pools leverage affinity functionality provided by an Oracle RAC database. Connection affinity requires the use of an Oracle JDBC driver and an Oracle RAC database version 11.1.0.6 or higher.

Connection affinity is a performance feature that enables a connection pool to select connections that are directed at a specific Oracle RAC instance. The pool uses run-time connection load balancing (if configured) to select an Oracle RAC instance to create the first connection and then subsequent connections are created with an affinity to the same instance.

See Also:

- "Strict Affinity Mode"
- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about setting up an Oracle RAC database.

UCP JDBC connection pools support the following three types of connection affinity:

- Transaction-Based Affinity

- Web Session Affinity
- Oracle RAC Data Affinity

10.4.1.1 Transaction-Based Affinity

Transaction-based affinity is an affinity to an Oracle RAC instance that can be released by either the client application or a failure event. Applications typically use this type of affinity when long-lived affinity to an Oracle RAC instance is desired or when the cost (in terms of performance) of being redirected to a new Oracle RAC instance is high. Distributed transactions are a good example of transaction-based affinity. XA connections that are enlisted in a distributed transaction keep an affinity to the Oracle RAC instance for the duration of the transaction. In this case, an application would incur a significant performance cost if a connection is redirect to a different Oracle RAC instance during the distributed transaction.

10.4.1.2 Web Session Affinity

Web session affinity is an affinity to an Oracle RAC instance that can be released by either the instance, a client application, or a failure event. The Oracle RAC instance uses a hint to communicate to a connection pool whether affinity has been enabled or disabled on the instance. An Oracle RAC instance may disable affinity based on many factors, such as performance or load. If an Oracle RAC instance can no longer support affinity, the connections in the pool are refreshed to use a new instance and affinity is established once again.

Applications typically use this type of affinity when short-lived affinity to an Oracle RAC instance is expected or if the cost (in terms of performance) of being redirected to a new Oracle RAC instance is minimal. For example, a mail client session might use Web session affinity to an Oracle RAC instance to increase performance and is relatively unaffected if a connection is redirected to a different instance.

10.4.1.3 Oracle RAC Data Affinity

Data affinity describes the concept of ensuring that a group of related cache entries is contained within a single cache partition.

Starting from Oracle Database Release 18c, UCP supports Oracle RAC Data Affinity. When you enable Data Affinity on the Oracle RAC database, data on the affinitized tables are partitioned in such a way that a particular partition or subset of rows for a table is affinitized to a particular Oracle RAC database instance. The affinity leads to higher performance and scalability for the applications due to improved cache locality and reduced internode synchronization and block pings among the RAC instances.

See Also:

Enabling a Custom Partition Assignment Strategy

To use the Oracle RAC Data Affinity feature, the clients accessing the database through UCP must provide the data affinity key in their connection requests. UCP has the following capabilities when pooling connections for an affinity enabled RAC database:

1. UCP learns the topology that contains the data affinity of the data partitions across Oracle RAC instances at pool start up.

2. UCP connection requests that need to leverage the Oracle RAC Data Affinity feature provides the data affinity key using the sharding key builder and use the connection builder as follows:

```
  PoolDataSource pds = new PoolDataSourceImpl();
   // configure the datasource with the database connection properties

/* Builds the RAC data affinity key using the sharding key builder API
and gets a connection from the pool using UCP connection builder */
  OracleShardingKey dataAffinityKey =  pds.createShardingKeyBuilder()
          .subkey(1000, OracleType.NUMBER)
          .build();

  Connection connection = pds.createConnectionBuilder()
          .shardingKey(dataAffinityKey)
          .build();
```

Note:

You can still make connection requests to Oracle RAC Data Affinity-enabled without providing the data affinity key. However, in this case, you will not see the benefits of Oracle RAC Data Affinity feature.

3. UCP determines the affinitized instance for the shard key provided in the request and checks if a connection for that instance exists in the pool. If the connection exists, then it is used to serve the request. If a matching connection does not exist in the pool, then a fallback to Run-Time Load Balancing chooses a connection for the request and serves it. If a new connection needs to be created to serve the request, then the request is routed to the affinitized instance corresponding to the provided shard (data affinity) key.
4. UCP keeps its topology of the data partitions in sync with the server side when there are HA events or when there is a change in the affinity of data partitions on Oracle RAC.

10.4.2 Setting Up Connection Affinity

Perform the following steps to set up connection affinity:

- Enable FCF.

See Also:

"About Fast Connection Failover"

- Enable run-time connection load balancing.

See Also:

"About Run-Time Connection Load Balancing"

- Create a connection affinity callback.

- Register the callback.

> **Note:**
>
> Transaction-based affinity is strictly scoped between the application/middle-tier and UCP. Therefore, transaction-based affinity requires only the `setFastConnectionFailoverEnabled` property be set to `true` and does not require complete FCF configuration.
>
> In addition, transaction-based affinity does not technically require run-time connection load balancing. However, it can help with performance and is usually enabled regardless. If run-time connection load balancing is not enabled, the connection pool randomly picks connections.

This section contains the following subsections:

- Creating a Connection Affinity Callback
- Registering a Connection Affinity Callback
- Removing a Connection Affinity Callback

10.4.2.1 Creating a Connection Affinity Callback

Connection affinity requires the use of a callback. The callback is an implementation of the `ConnectionAffinityCallback` interface which is located in the `oracle.ucp` package. The callback is used by the connection pool to establish and retrieve a connection affinity context and is also used to set the affinity policy type (transaction-based or Web session).

The following example demonstrates setting an affinity policy in a callback implementation. The example also demonstrates manually setting an affinity context. typically, the connection pool sets the affinity context inside an application. However, the ability to manually set an affinity context is provided for applications that want to customize affinity behavior and control the affinity context directly.

```
public class AffinityCallbackSample
   implements ConnectionAffinityCallback {

   Object appAffinityContext = null;
   ConnectionAffinityCallback.AffinityPolicy affinityPolicy =
   ConnectionAffinityCallback.AffinityPolicy.TRANSACTION_BASED_AFFINITY;

   //For Web session affinity, use WEBSESSION_BASED_AFFINITY;

   public void setAffinityPolicy(AffinityPolicy policy)
   {
      affinityPolicy = policy;
   }

   public AffinityPolicy getAffinityPolicy()
   {
      return affinityPolicy;
   }

   public boolean setConnectionAffinityContext(Object affCxt)
   {
      synchronized (lockObj)
      {
         appAffinityContext = affCxt;
      }
      return true;
   }

   public Object getConnectionAffinityContext()
   {
      synchronized (lockObj)
      {
         return appAffinityContext;
      }
   }
}
```

10.4.2.2 Registering a Connection Affinity Callback

A connection affinity callback is registered on a connection pool using the `registerConnectionAffinityCallback` method. The callback is registered when creating the connection pool. Only one callback can be registered per connection pool.

The following example demonstrates registering a connection affinity callback implementation:

```
ConnectionAffinityCallback callback = new MyCallback();

PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("AffinitySamplePool");
pds.registerConnectionAffinityCallback(callback);
...
```

10.4.2.3 Removing a Connection Affinity Callback

A connection affinity callback is removed from a connection pool using the `removeConnectionAffinityCallback` method. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("AffinitySamplePool");
pds.removeConnectionAffinityCallback();
...
```

10.4.2.4 Strict Affinity Mode

By default, affinity is only a hint. A connection pool selects a new Oracle RAC instance for connections if it does not find a connection on a desired instance. You can change this behavior by switching the strict affinity mode on. The strict affinity mode throws a UCP exception if a connection on a desired instance is not found.

Use the following pool properties to switch on the strict affinity mode:

- The `useStrictWebSessionAffinity` property

  Set the `useStrictWebSessionAffinity` property to `true` or `false` for switching the strict Web session affinity mode on or off respectively.

- The `useStrictXAAffinity` property

  Set the `useStrictXAAffinity` property to `true` or `false` for switching the strict transaction-based affinity mode on or off respectively.

These properties can be handled through the `UniversalConnectionPoolMBean`.

Related Topics

- UniversalConnectionPoolMBean

10.5 Global Data Services

This section describes the new Global Data Services (GDS) feature that can be used with Universal Connection Pool:

- Overview of Global Data Services
- Configuring an Application for Using GDS

10.5.1 Overview of Global Data Services

Global Data Services (GDS) feature is available since Oracle Database 12*c* Release 1 (12.1). Through this feature, Fast Connection Failover, Run-time Connection Load-Balancing, and Connection Affinity features that were available only in Oracle RAC, were extended to a set of replicated databases offering common services.

The set of databases may include Oracle RAC and single-instance Oracle databases interconnected through Data Guard, GoldenGate, or any other replication technology. A database service that can be provided by multiple databases is called a global service, so that it can be distinguished from the traditional service that can be provided only by a single database. This combination enables services to be deployed anywhere within this globally distributed configuration, supporting load balancing, high availability, database affinity, and so on.

See Also:

Oracle Database Global Data Services Concepts and Administration Guide

10.5.2 Configuring an Application for Using GDS

UCP connects to Global Data Services in the same way that it connects to local services on an Oracle RAC. The service name in the connection string should be the name of the global service. The endpoint should be the endpoint of a GDS listener instead of the endpoint for the local, remote, or SCAN listener of a database.

A client must specify its region in the `REGION` parameter of the connection string. This is a new requirement for GDS. The region name is required because, in case of GDS, Run-time Load Balancing advisory is customized for particular regions. Following is an example of a typical connection string:

```
(DESCRIPTION=
  (ADDRESS=(GDS_protocol_address_information))
  (CONNECT_DATA=
   (SERVICE_NAME=global_service_name)
   (REGION=region_name)))
```

Like with local services, UCP can specify multiple GDS listeners in the same connection string for listener failover, load balancing, or both.

Note:

SCAN is not supported for GDS listeners, therefore endpoint for each listener must be specified.

```
(DESCRIPTION=
  (ADDRESS_LIST=
    (LOAD_BALANCE=ON)
    (FAILOVER=ON)
    (ADDRESS=(GDS_protocol_address_information))
    (ADDRESS=(GDS_protocol_address_information)))
  (CONNECT_DATA=
   (SERVICE_NAME=global_service_name)
   (REGION=region_name)))
```

The `REGION` parameter is optional if only global service managers from the local region are specified in the client connection string. This is the case when there is only one region in the GDS configuration, or can be the case when there are multiple regions. But, it is not feasible to change the connection string of the an existing client designed to work with a single database. If the `REGION` parameter is not specified, then the client's region is assumed to be the region of the global service manager used to connect to the global service.

Note:

Unless the `REGION` parameter is specified in the connection string, you can use a pre-12c thin JDBC client only with a GDS configuration that has a single region.

All GDS listeners in the preceding example belong to the same region where UCP is running, that is the local region. To provide high availability, when all GDSs in the local region are unavailable, you can specify the GDS listeners for the buddy region in additional `ADDRESS_LIST` descriptors.

```
(DESCRIPTION=
  (FAILOVER=on)
  (ADDRESS_LIST=
    (LOAD_BALANCE=ON)
    (ADDRESS=(global_protocol_address_information))
    (ADDRESS=(global_protocol_address_information)))
  (ADDRESS_LIST=
    (LOAD_BALANCE=ON)
    (ADDRESS=(global_protocol_address_information))
    (ADDRESS=(global_protocol_address_information)))
  (CONNECT_DATA=
   (SERVICE_NAME=global_service_name)
   (REGION=region_name)))
```

You do not need manual ONS configuration because UCP automatically retrieves the ONS connection information that is optimally customized for the UCP region from GDS.

Note:

- To enable automatic ONS configuration for GDS, you must enable Fast Connection Failover (FCF) on UCP.
- Automatic ONS configuration works only with Oracle GDS and Oracle RAC. It does not work with single-instance Oracle Databases.

  Automatic ONS configuration does not support ONS wallet or keystore parameters. If your application requires any of these parameters, then you must configure ONS explicitly in either of the following two ways:

  – Calling the `PoolDataSource.setONSConfiguration(String)` method
  – Adding the ONS wallet or keystore parameters in the local ONS configuration file

11
UCP Asynchronous Extension

Starting from Oracle Database Release 23ai, UCP provides asynchronous extension. The asynchronous extension is a set of methods that extend the UCP standard to offer asynchronous database access.

The asynchronous extension is integrated into UCP in such a way that you can use all the features of UCP with this extension, with some changes in your source code. This chapter describes the changes that you must make to your code for using UCP in an asynchronous way.

Note:

This feature is supported starting with JDK 11.

This chapter covers the following topics:

11.1 Overview of UCP Asynchronous Extension

The UCP asynchronous extension uses non-blocking mechanisms for creating connection objects, so your application immediately receives either a `CompletableFuture` or a `Publisher` of a connection to be borrowed.

You must perform the following to achieve this:

1. Instantiate a Connection Builder.
2. Create a connection asynchronously with `UCPConnectionBuilder` using either a `CompletableFuture<Connection>` or a `Publisher<Connection>`.

The asynchronous extension also lets you borrow XA connections with a `UCPXAConnectionBuilder` using either a `CompletableFuture<XAConnection>` or a `Publisher<XAConnection>`.

When an asynchronous method is called, it performs as much work as possible on the calling thread, without blocking on a network read or write. An asynchronous method call returns immediately after a request is about to be written to the network, without waiting for a response. When I/O readiness is detected for a network channel, the polling thread arranges for a worker thread to handle the event. The worker thread reads from the network and then notifies a `CompletableFuture` or a `Publisher` that an operation is complete. Upon notification, the `CompletableFuture` or `Publisher` arranges worker threads that emit a signal to each of its `Subscribers`.

The `java.util.concurrent.Executor` interface manages the worker threads, while the default `Executor` is the `java.util.concurrent.ForkJoinPool.commonPool` method. If you do not implement the `executor()` code in your application source code, then the asynchronous borrow operation runs with the default `ForkJoinPool` executor. For setting an arbitrary executor to serve an asynchronous borrow, you can use the `executor(executor)` call.

See Also:

Overview of JDBC Reactive extensions

11.2 Example: UCP Asynchronous Extension

This section lists a few examples that demonstrate how to use UCP asynchronous extension.

Example 11-1 Creating a Connection Asynchronously with CompletableFuture<Connection>

```
...
final PoolDataSource pds = new PoolDataSourceImpl();
[//Initialize PoolDataSource object in the standard way]

final CompletionStage<Connection> connectionStage =
  pds.createConnectionBuilder()
    .user(<user name>)
    .password(<password>)
    .executor(executor)
    .buildAsyncOracle();

final CompletionStage<String> queryStage =
  connectionStage.thenApply(connection -> {
     [//Perform operations on the connection]
  });
...
```

Example 11-2 Creating a Connection Asynchronously with Publisher<Connection>

```
...
final PoolDataSource pds = new PoolDataSourceImpl();
[//Initialize PoolDataSource object in the standard way]

final Publisher<Connection> connectionPublisher =
  pds.createConnectionBuilder()
    .user(<user name>)
    .password(<password>)
    .executor(executor)
    .buildConnectionPublisherOracle();

[//Perform standard activities on the Publisher]
...
```

Example 11-3 Creating an XA Connection Asynchronously with UCPXAConnectionBuilder

```
...
final PoolXADataSource pds = new PoolXADataSourceImpl();
[//Initialize PoolXADataSource object in the standard way]

final CompletionStage<XAConnection> connectionStage =
  pds.createXAConnectionBuilder()
    .user(<user name>)
    .password(<password>)
    .executor(executor)
    .buildAsyncOracle();

final CompletionStage<String> queryStage =
  connectionStage.thenApply(xaConnection -> {
     [//Perform operations on the XAConnection]
  });
...
```

Example 11-4 Creating an XA Connection Asynchronously with Publisher<XAConnection>

```
...
final PoolXADataSource pds = new PoolXADataSourceImpl();
[//Initialize PoolXADataSource object in the standard way]

final Publisher<XAConnection> xaConnectionPublisher =
  pds.createXAConnectionBuilder()
    .user(<user name>)
    .password(<password>)
    .executor(executor)
    .buildConnectionPublisherOracle();

[//Perform standard activities on the Publisher]
...
```

11.3 Asynchronous Connection Labeling

You can take advantage of the UCP connection labeling feature even in the asynchronous mode. For achieving this, you must override the default version of the new `oracle.ucp.ConnectionLabelingCallback.configureAsync()` method.

configureAsync: Asynchronous Version of the Configure Method

For standard connection labeling, you use the `configure` method in your application. For using connection labeling in the asynchronous mode, you must use the `configureAsync` method. The definition of the `configureAsync` method is as follows:

```
default CompletionStage<Boolean> configureAsync(Properties requestedLabels,
Object connection) {
  throw new NoSuchMethodError();
```

See Also:

- Labeling Connections in UCP
- *Oracle Universal Connection Pool Java API Reference*

11.4 Example: Asynchronous Connection Labeling

This section contains an examples that demonstrates how to use UCP asynchronous extension with connection labeling.

Example 11-5 Creating a Connection Asynchronously with Connection Labeling

```
package tests.ucp.async.labeling;

import oracle.ucp.ConnectionLabelingCallback;
import oracle.ucp.jdbc.PoolDataSource;
import oracle.ucp.jdbc.PoolDataSourceImpl;

import java.sql.Connection;
import java.util.Properties;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

public class AsyncLabelingExample {
  public static void main(String ... args) throws Exception {
    final PoolDataSource pds = new PoolDataSourceImpl();

    // Set the pool data source properties

    final ConnectionLabelingCallback labelingCallback = new 
ConnectionLabelingCallback() {
      @Override
      public int cost(Properties requestedLabels, Properties currentLabels) {
        // some cost manipulations, same as in synchronous case
        return 0; // or some other integer, depending on the cost computation 
logic
      }

      @Override
      public boolean configure(Properties requestedLabels, Object connection) 
{
        // some connection configuration manipulations for synchronous case,
        // not used in asynchronous case, so it can be skipped.
        return true;
      }

      @Override
      public CompletionStage<Boolean> configureAsync(Properties 
requestedLabels, Object connection) {
        final var cf = new CompletableFuture<Boolean>();

        // Perform some asynchronous connection configuration
        doSomeConfigAction((Connection)connection).whenComplete((p, e) -> {
          if (null == e) {
            cf.complete(p);
          } else {
            cf.completeExceptionally(e);
          }
        });
        return cf;
      }

      private CompletableFuture<Boolean> doSomeConfigAction(Connection conn) {
        final var cf = new CompletableFuture<Boolean>();

        // ...
        // configure the connection asynchronously
        // ... complete CompletableFuture with result or exception ...

        return cf;
      }
    };

    pds.registerConnectionLabelingCallback(labelingCallback);

    // some labeling code

  }
}
```

12
Ensuring Application Continuity

This chapter discusses the following concepts related to the Application Continuity feature of Oracle Database:

- Overview of Ensuring Application Continuity with UCP
- Configuring the Data Source for Application Continuity
- Using Connection Labeling for Application Continuity
- Using Connection Initialization Callback for Application Continuity

12.1 Overview of Ensuring Application Continuity with UCP

Oracle Database 12*c* Release 1 (12.1) introduced the Application Continuity feature that provides a general purpose, application-independent infrastructure. Application Continuity enables recovery of work from an application perspective, after the occurrence of a planned or unplanned outage that can be related to system, communication, or hardware following a repair, a configuration change, or a patch application.

For using Application Continuity, you must first configure your data source. After that, use one of the following two features for implementing Application Continuity in your applications using Universal Connection Pool (UCP):

- Using Connection Labeling for Application Continuity
- Using Connection Initialization Callback for Application Continuity

Related Topics

- Configuring the Data Source for Application Continuity
- Using Connection Labeling for Application Continuity
- Using Connection Initialization Callback for Application Continuity

12.2 Configuring the Data Source for Application Continuity

To utilize the Application Continuity (AC) feature on a UCP data source, applications must specify an Oracle JDBC driver data source with AC support, as the connection factory class. The data source depends on the Oracle JDBC driver version that you use.

- For Oracle JDBC driver 23ai, all the driver data sources support Application Continuity. Oracle recommends that you use `oracle.jdbc.datasource.impl.OracleDataSource`, although either of the older data sources, that is, `oracle.jdbc.replay.OracleDataSourceImpl` or `oracle.jdbc.pool.OracleDataSource`, also works.
- For using Oracle JDBC driver 19c or an older driver, you must use the `oracle.jdbc.replay.OracleDataSourceImpl` to support Application Continuity.

See Also:

About Configuring Oracle JDBC for Application Continuity for Java

12.3 Using Connection Labeling for Application Continuity

Connection labeling enables an application to attach arbitrary name/value pairs to a connection. The application can request a connection with the desired label from the connection pool.

Connection labeling sets the initial state for each connection request. If the application uses connection labeling or benefits from labeling connections, then a labeling callback should be registered for Application Continuity to initialize clean connections at failover.

Every time Application Continuity gets a new connection from the underlying data source, the labeling callback executes. The callback executes during normal connection check-out and also during replay. So, the state that is created at run time is exactly re-created during replay. The initialization must be idempotent.

It is legal for the callback to execute a transaction as long as the transaction completes (either it commits or rolls back) at the end of callback invocation. Application Continuity repeats any action coded within the callback implementation, including such transaction. If an outage occurs during the execution of a UCP labeling callback, then Application Continuity may execute the callback more than once as part of the replay attempt. Again, it is important for the callback actions to be idempotent.

Related Topics

- Labeling Connections in UCP

12.4 Using Connection Initialization Callback for Application Continuity

If an application cannot use connection labeling because it cannot be changed, then the connection initialization callback is provided for such an application.

When registered, the initialization callback is executed every time a connection is borrowed from the pool and at each successful reconnection following a recoverable error..

Related Topics

- About Connection Initialization Callback

13 Shared Pool for Sharded Databases

Sharding is a data tier architecture in which data is horizontally partitioned across independent databases.

This chapter describes UCP Shared Pool for sharded databases in the following sections:

13.1 Overview of UCP Shared Pool for Database Sharding

Starting from Oracle Database 12*c* Release 2 (12.2.0.1), Universal Connection Pool (UCP) supports database sharding. UCP recognizes the sharding keys specified and connects to the specific shard. Sharding uses Global Data Services (GDS), where GDS routes a client request to an appropriate database, based on various parameters such as availability, load, network latency, and replication lag.

See Also:

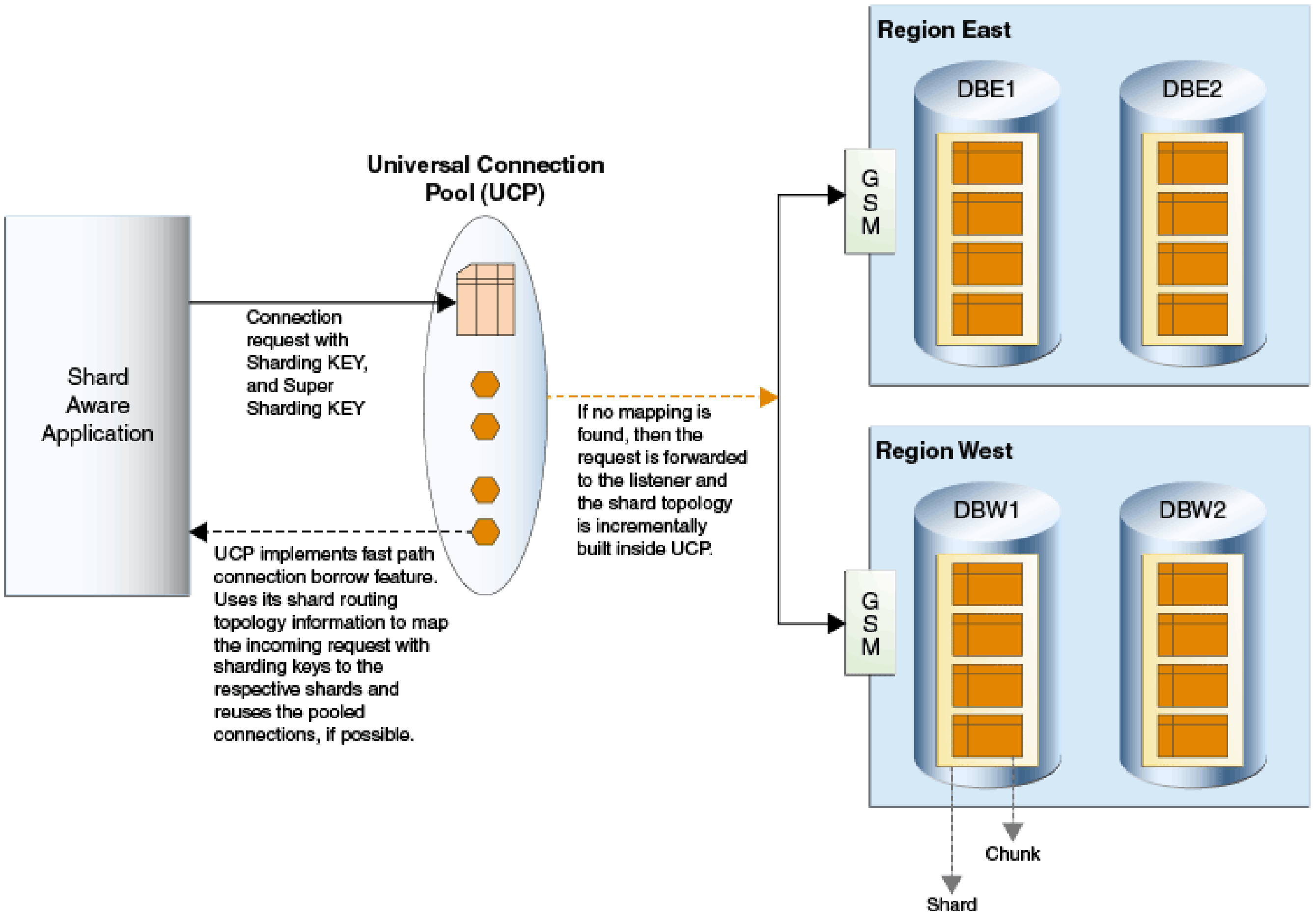
- *Oracle Database JDBC Developer’s Guide*
- *Oracle Database Administrator’s Guide*

Use Case of UCP Shared Pool for Database Sharding

This section describes a use case of UCP Shared Pool for database sharding. In the use case, the applications connecting to sharded database use UCP to store connections to different shards and chunks of the sharded GDS database within the same Shared Pool. The applications must provide the sharding key to UCP during the connection request. Based on the sharding key, the pool routes the connection request to the correct shard. The data distribution across the shards and chunks in the database is transparent to the user. UCP transparently handles resharding and chunk movements, minimizing the impact on the end users.

The following diagram illustrates this use case:

Figure 13-1 Universal Connection Pool (UCP) Using Sharded Database Architecture


Region East
DBE1
DBE2
Universal Connection Pool (UCP)
G S M
Shard Aware Application
Connection request with Sharding KEY, and Super Sharding KEY
If no mapping is found, then the request is forwarded to the listener and the shard topology is incrementally built inside UCP.
Region West
DBW1
DBW2
UCP implements fast path connection borrow feature. Uses its shard routing topology information to map the incoming request with sharding keys to the respective shards and reuses the pooled connections, if possible.
G S M
Chunk
Shard

Related Topics

- Global Data Services

13.2 About Handling Connection Requests for a Sharded Database

This section describes how connection requests are made on a pool for sharded databases.

- How to Checkout Connections from a Pool with a Known Sharding key
- About Configuring the Number of Connections Per Shard
- About Connecting to the Shard Catalog or Co-ordinator for Multishard Queries

13.2.1 How to Checkout Connections from a Pool with a Sharding Key

When a connection is borrowed from UCP, then the shard aware application can provide the sharding key and the super sharding key using the new connection builder present in the `PoolDataSource` class.

If sharding keys do not exist or do not map to the data types specified by the database metadata, then an `IllegalArgumentException` is thrown. The following code snippet shows how to checkout a connection with sharding keys:

Example 13-1 Checking Out a Connection with Sharding Keys

```
import java.sql.Connection;
import java.sql.JDBCType;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.ShardingKey;

import oracle.ucp.jdbc.PoolDataSource;
import oracle.ucp.jdbc.PoolDataSourceFactory;

public class UCPShardingExample {

  public static void main(String[] args) throws SQLException {
      String url = "jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=myhost)
(PORT=<gsm_port>)(PROTOCOL=tcp))(CONNECT_DATA=(SERVICE_NAME=myGSMservice)))";
      String user="db_user_name";
      String pwd = "db_password";

      PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
      pds.setURL(url);
      pds.setUser(user);
      pds.setPassword(pwd);
      pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
      pds.setInitialPoolSize(5);
      pds.setMinPoolSize(5);
      pds.setMaxPoolSize(20);

      int empId = 1234;
      // Employee ID is the sharding key column in sharded table

      ShardingKey shardingKey = pds.createShardingKeyBuilder()
                             .subkey(empId, JDBCType.INTEGER)
                             .build();

      // Borrow a connection to direct shard using sharding key
      try(Connection connection = pds.createConnectionBuilder()
              .shardingKey(shardingKey)
              .build()) {

                          PreparedStatement pst =
connection.prepareStatement("select * from employee where emp_id=?");
                          pst.setInt(1, 1234);
                          ResultSet rs = pst.executeQuery();
```

```
                              // retrieve the employee details using resultset
                              rs.close();
                              pst.close();
                            }
              }

          }
```

Note:

You must specify a sharding key during the connection checkout. Otherwise, an error or exception is thrown back to the application.

13.2.2 About Configuring the Number of Connections Per Shard

When UCP is used to pool connections for a sharded database, the pool contains connections to different shards. So, when connections are pulled, to ensure a fair usage of the pool capacity across all shards connected, UCP uses the `MaxConnectionsPerShard` parameter. This is a global parameter, which applies to every shard in the sharded database, and is used to limit the total number of connections to any shard below the specified limit.

The following table describes the APIs for setting and retrieving this parameter:

Method	Description
`poolDatasource.setMaxConnectionsPerShard(<max_connections_per_shard_limit>)`	Sets the maximum number of connections per shard.
`poolDatasource.getMaxConnectionsPerShard()`	Retrieves the value that was set using the `setMaxConnectionsPerShard(<max_connections_per_shard_limit>)` method.

Note:

You cannot use the `MaxConnectionsPerShard` parameter in a sharded database with Oracle Golden Gate configuration.

13.2.3 About Connecting to the Shard Catalog or Co-ordinator for Multishard Queries

When connecting to the Shard Catalog or Co-ordinator for running multishard queries, it is recommended that a separate pool be created using a new `PoolDataSource` instance. You can run multishard queries on connections retrieved from a data source that is created on the coordinator service. The connection request for the coordinator should not have sharding keys in the connection builder API.

13.3 Sharding Data Source for Transparent Access to Sharded Databases

Oracle Database Release 21c introduced a new JDBC data source that enables Java connectivity to a sharded database without the need for an application to furnish a sharding key.

If you use the sharding data source, then you do not have to identify and build the sharding key and the super sharding key to establish a connection to the sharded database, as discussed in the earlier How to Checkout Connections from a Pool with a Sharding Key section. The sharding data source also eliminates the need to maintain a separate data source for multishard queries.

This data source scales out to sharded databases transparently if you set the connection property `oracle.jdbc.useShardingDriverConnection` to `true`.

See Also:

Oracle Database JDBC Developer's Guide

The following code snippet shows how to use the sharded data source:

Example 13-2   Using the Sharded Data Source

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;

import javax.sql.DataSource;

import oracle.jdbc.internal.OracleConnection;
import oracle.ucp.jdbc.PoolDataSource;
import oracle.ucp.jdbc.PoolDataSourceFactory;

public class ShardingDataSourceUCP {
    public static void main(String[] args) throws SQLException {
        ShardingDataSourceUCP sample = new ShardingDataSourceUCP();
        DataSource ucpDataSource = sample.getDataSource();
        // Get the details of following customers
        int[] customerIds = new int[] {
            100,
            101,
            102,
            103,
            104,
            105
        };

        for (int id: customerIds) {
            try (Connection conn = ucpDataSource.getConnection()) {
                sample.displayCustomerDetails(conn, id);
                System.out.println(((OracleConnection)
conn).getPercentageQueryExecutionOnDirectShard());
            }
        }
    }

    private void displayCustomerDetails(Connection conn, int id) throws
SQLException {
        try (PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM
CUSTOMER where ID = ?")) {
            pstmt.setInt(1, id);
            try (ResultSet rs = pstmt.executeQuery()) {
                while (rs.next()) {
                    // Print the customer details
                }
            }
        }
    }

    private DataSource getDataSource() throws SQLException {
        PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
        pds.setURL("<gsmURL>");
        pds.setUser("<userName>");
        pds.setPassword("<password>");

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
        Properties prop = new Properties();
        // Connection property to enable sharding datasource feature, when
this property
        // is set you don't need to pass sharding key to UCP pool while
borrowing the connection

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_USE_SHARDING_DRIVER_CONN
ECTION, "true");
        pds.setConnectionProperties(prop);
        return pds;
    }
}
```

See Also:

- The UCPConnectionBuilder Interface
- The PoolDataSource Interface
- The PoolXADataSource Interface

13.3.1 Support for Single Shard Transactions

The sharding data source enables you to limit your transactions to one single shard.

To enable single shard transaction support, you must set `CONNECTION_PROPERTY_ALLOW_SINGLE_SHARD_TRANSACTION_SUPPORT`. If this property is not set, then by default, all the transactions are started on the Shard Catalog. If you set the value of this property to `true`, then you must ensure that all the transactions span over a single shard only.

See Also:

- Oracle Database JDBC Java API Reference
- *Oracle Database JDBC Developer's Guide*

Example 13-3 Enabling Single Shard Transactions

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;

import javax.sql.DataSource;

import oracle.jdbc.internal.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;

public class SingleShardTransactionUCP {
    public static void main(String[] args) throws SQLException {
        SingleShardTransactionUCP sample = new SingleShardTransactionUCP();
        DataSource ucpDS = sample.getDataSource();

        // Insert and update the details of following customers in a single 
transaction
        int[] customerIds = new int[] {
            100,
            101,
            102,
            103,
            104,
            105
        };

        for (int id: customerIds) {
            try (Connection conn = ucpDS.getConnection()) {
                conn.setAutoCommit(false);
                sample.insertCustomerDetails(conn, id);
                sample.displayCustomerDetails(conn, id);
                sample.updateCustomerDetails(conn, id);
                sample.displayCustomerDetails(conn, id);
                conn.commit();
                System.out.println(((OracleConnection)
conn).getPercentageQueryExecutionOnDirectShard());
            }

        }
    }

    private void insertCustomerDetails(Connection conn, int id) throws
SQLException {
        String sql = "insert into CUSTOMER values(?, ?, ?, ?)";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setInt(1, id);
            ps.setString(2, name);
            ps.setString(3, email);
            ps.setString(4, phoneNumber);
            ps.executeUpdate();
        }
    }

    private void updateCustomerDetails(Connection conn, int id) throws
SQLException {
        String sql = "UPDATE CUSTOMER SET name = ?, email = ?, phoneNumber
= ? WHERE customerId = ?";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setString(1, name);
            ps.setString(2, email);
            ps.setString(3, phoneNumber);
            ps.setInt(4, id);
            ps.executeUpdate();
        }
    }

    private void displayCustomerDetails(Connection conn, int id) throws
SQLException {
        try (PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM
CUSTOMER where ID = ?")) {
            pstmt.setInt(1, id);
            try (ResultSet rs = pstmt.executeQuery()) {
                while (rs.next()) {
                    //Print the customer details
                }
            }
        }
    }

    private DataSource getDataSource() throws SQLException {
        OracleDataSource ds = new OracleDataSource();
        ds.setURL( < gsmURL > );
        ds.setUser( < userName > );
        ds.setPassword( < password > );
        Properties prop = new Properties();
        //Connection property to enable sharding datasource feature

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_USE_SHARDING_DRIVER_CONN
ECTION, "true");
```

```
        //Connection property to enable single shard transaction support. If
this property is not set,
        // by default all the transactions are started on catalog DB. When
setting this property value
        // to "true", applications must ensure that all the transactions span
over a single shard only.

prop.setProperty(oracle.jdbc.OracleConnection.CONNECTION_PROPERTY_ALLOW_SINGLE
_SHARD_TRANSACTION_SUPPORT, "true");
        ds.setConnectionProperties(prop);
        return ds;
    }
}
```

13.4 Middle-Tier Routing Using UCP

Since Oracle Database Release 18c, Oracle Universal Connection Pool (UCP) supports the Middle-Tier Routing feature. This feature helps the Oracle customers, who use the Sharding feature, to have a dedicated middle tier from the client applications to the sharded database.

Typically, the middle-tier connection pools route database requests to specific shards. During such a routing, each middle-tier connection pool establishes connections to each shard, creating too many connections to the database. The Middle-Tier Routing feature solves this problem by having a dedicated middle tier (Web Server or Application Server) for each Data Center or Cloud, and routing client requests directly to the relevant middle tier, where the shard containing the client data (corresponding to the client sharding key) resides.

13.4.1 Middle-Tier Routing with UCP Example

The following example explains the usage of the middle-tier routing API of UCP.

Example 13-4   Example of Middle-Tier Routing Using UCP

```
import java.sql.SQLException;
import java.util.Properties;
import java.util.Random;
import java.util.Set;

import oracle.jdbc.OracleShardingKey;
import oracle.jdbc.OracleType;
import oracle.ucp.UniversalConnectionPoolException;
import oracle.ucp.routing.ShardInfo;
import oracle.ucp.routing.oracle.OracleShardRoutingCache;

/**
 * The code example illustrates the usage of the middle-tier routing feature
of UCP.
 * The API accepts sharding key as input and returns the set of ShardInfo
 * instances mapped to the sharding key. The ShardInfo instance encapsulates
 * unique shard name and priority. The unique shard name then can be mapped
 * to a middle-tier server that connects to a specific shard.
 *
 */
public class MidtierShardingExample {
  private static String user = "testuser1";
  private static String password = "testuser1";

  // catalog DB URL
  private static String url = "jdbc:oracle:thin:@//hostName:1521/
catalogServiceName";
  private static String region = "regionName";

  public static void main(String args[]) throws Exception {
    testMidTierRouting();
  }

  static void testMidTierRouting() throws UniversalConnectionPoolException,
      SQLException {

    Properties dbConnectProperties = new Properties();
    dbConnectProperties.setProperty(OracleShardRoutingCache.USER, user);
    dbConnectProperties.setProperty(OracleShardRoutingCache.PASSWORD,
password);
    // Mid-tier routing API accepts catalog DB URL
    dbConnectProperties.setProperty(OracleShardRoutingCache.URL, url);

    // Region name is required to get the ONS config string
    dbConnectProperties.setProperty(OracleShardRoutingCache.REGION, region);

    OracleShardRoutingCache routingCache = new OracleShardRoutingCache(
        dbConnectProperties);

    final int COUNT = 10;
    Random random = new Random();

    for (int i = 0; i < COUNT; i++) {
      int key = random.nextInt();
      OracleShardingKey shardKey = routingCache.getShardingKeyBuilder()
          .subkey(key, OracleType.NUMBER).build();
      OracleShardingKey superShardKey = null;

      Set<ShardInfo> shardInfoSet = routingCache.getShardInfoForKey(shardKey,
          superShardKey);

      for (ShardInfo shardInfo : shardInfoSet) {
        System.out.println("Sharding Key=" + key + " Shard Name="
            + shardInfo.getName() + " Priority=" + shardInfo.getPriority());
      }
    }

  }
}
```

13.5 Sharding with JTA/XA Transaction in WebLogic Server

Starting from Oracle Database 23ai Release, you can use sharding with JTA/XA transactions, when you configure UCP as a native data source in WebLogic Server. This feature broadens

UCP middle-tier coverage to include more applications, like the ones that require container-managed JTA/XA transactions or sharding.

Java Transaction API (JTA) specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications. Now, Java Enterprise applications that use UCP native data source sharding APIs, to obtain connections to Oracle sharded databases, can participate in JTA/XA transactions managed by WebLogic Transaction Manager (TM).

For any XA transaction with all supplied sharding keys parameter in the connection requests, which lead to the same Oracle sharded database instance, the XA transaction goes through successfully and commits the changes. For an XA transaction, leading to different Oracle sharded database instances, UCP native data source raises an exception to the WebLogic Transaction Manager and the transaction is rolled back.

See Also:

Oracle WebLogic Server documentation for more information

14 Diagnosing a Connection Pool

The following parameters are used for diagnosing Universal Connection Pool (UCP):

- Pool Statistics
- Dynamic Monitoring Service Metrics
- Overview of Logging and Tracing in UCP
- About Viewing Oracle RAC Statistics
- Exceptions and Error Codes

14.1 Pool Statistics

Universal Connection Pool (UCP) provides a set of run-time statistics for the connection pool. These statistics can be divided into the following two categories:

- Noncumulative

  These statistics apply only to the current running connection pool instance.

- Cumulative

  These statistics are collected across multiple pool start/stop cycles.

The `oracle.ucp.UniversalConnectionPoolStatistics` interface provides methods that are used to query the connection pool statistics. The methods of this interface can be called from a pool-enabled data source and pool-enabled XA data source, using the `oracle.ucp.jdbc.PoolDataSource.getStatistics` method. For example:

```
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
...
...
int totalConnsCount = pds.getStatistics().getTotalConnectionsCount();
System.out.println("The total connetion count in the pool is "+ totalConnsCount +".");
```

The `oracle.ucp.jdbc.PoolDataSource.getStatistics` method can also be called by itself to return all connection pool statistics as a `String`.

14.2 Dynamic Monitoring Service Metrics

UCP supports all the pool statistics to be in the form of Dynamic Monitoring Service (DMS) metrics. You must include the `dms.jar` file in the class path of the application to collect and utilize these DMS metrics.

UCP supports DMS metrics collection in both the pool manager interface and the pool manager MBean. You can use the `UnversalConnectionPoolManager.startMetricsCollection` method to start collecting DMS metrics for the specified connection pool instance, and use the `UnversalConnectionPoolManager.stopMetricsCollection` method to stop DMS metrics collection. The metrics update interval can be specified using the

`UnversalConnectionPoolManager.setMetricUpdateInterval` method. The pool manager MBean exports similar operations.

14.3 Overview of Logging and Tracing in UCP

Two major aspects of UCP diagnosability are logging and tracing.

Logging is writing log records into a console, a file, or any other log handler that is defined by standard logging properties. The name of a logger is the same as a pool data source name. A pool data source name is set with an appropriate property. You can leave it unnamed as well, in which case, all diagnostics goes in the common logger. The name of the common logger is `oracle.ucp`. Logging implements the Java Logging API, `java.util.logging`.

Tracing is the special logging use case, where a log record is written into an in-memory ring buffer until an event triggers UCP Diagnosability to dump that buffer into a corresponding logger. UCP has the following two categories of trace buffers:

Note:

A pool data source and a UCP have a one-to-one mapping.

- A category that consists of one common buffer, which is used for tracing logs from the static methods that are not directly related to a pool with a designated name. The common buffer is permanent for the complete life span of the associated UCP.
- A category that contains as many buffers as the number of named Pool Data Source (PDS) objects, where every buffer is mapped to a single named PDS. Every trace from that PDS goes into the corresponding buffer. Its life starts from a pool start up and ends with a pool destruction.

14.3.1 Logging and Tracing Settings

When you enable logging, you should not enable tracing because the log records get immediately dumped into a logger in such a case, and there is no need to duplicate log records into a trace buffer. So, if logging is on, regardless of the tracing setting, tracing gets disabled automatically.

Note:

By default, tracing is on and logging is off.

The following table summarizes this functionality:

Logging Setting	Tracing Setting	Logging Functionality	Tracing Functionality
`off`	`off`	Disabled	Disabled
`on`	`off`	Enabled	Disabled
`off`	`on`	Disabled	Enabled
`on`	`on`	Enabled	Disabled

14.3.2 Diagnosability System Properties and Command Line

You can set the initial diagnosability properties as JVM system properties.
You can achieve this in the following ways:

- Using the Oracle JVM command line
- In your application source code
- Using Java Management Extensions (JMX)

Property Name	**Description**
`oracle.ucp.diagnostic.enableTrace`	Is the flag to enable tracing. The default value of this property is `true`.
`oracle.ucp.diagnostic.enableLogging`	Is the flag to enable debug logging. The default value of this property is `false`.
`oracle.ucp.diagnostic.bufferSize`	Specifies the in-memory trace buffer size. The default value is 1024.
`oracle.ucp.diagnostic.loggingLevel`	Specifies the default logging level that is used if no other value is specified for a logger in a logging configuration file. The default value of this property is `INFO`.
`java.util.logging.config.file`	Specifies the logging property file name to load. The default value is `ConsoleHandler` for all loggers.
`oracle.ucp.diagnostic.errorCodesToWatchList`	Provides the list of exceptions and error codes that triggers dumping of all traces into their loggers.

14.3.3 Logging Configuration File

The logging configuration file lets you configure your logging settings as described in the `java.util.logging` package. If you want to set different levels for different loggers, then you must set those in the logging configuration file.
Perform the following for configuring the common logger:

- Specify the logger as `oracle.ucp`.
- Set an appropriate logging level.
- Specify where and how to write the logs. For example, whether the logs should be written to the console or to a file. In case of a file, specify the file name, formatter, and so on.

Perform the following for configuring a logger that is specific to a pool data source object:

- Set a level for the logger with the name of that pool data source.
- Set an appropriate logging level.

Note:

If you do not specify the level for a logger, then a default level, which is set in the `ucp.diagnostic.loggingLevel`, is used.

Example 14-1 Logging Configuration File

The following is an example of the UCP diagnosability logging configuration file:

```
oracle.ucp.level=FINE
handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = ./test.log
java.util.logging.FileHandler.limit = 0
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
pool-name1=FINEST
pool-name2=SEVERE
```

Logging Level

Logging levels in the UCP Diagnosability framework are the same as the ones available in the `java.util.logging` package, but UCP also supports a numeric logging level in the range from `Integer.MIN_VALUE` to `Integer.MAX_VALUE`. The following table lists all the valid logging levels:

Logging Level	Value
`OFF`	`Integer.MAX_VALUE`
`SEVERE`	`1000`
`WARNING`	`900`
`INFO`	`800`
`CONFIG`	`700`
`FINE`	`500`
`FINER`	`400`
`FINEST`	`300`
`ALL`	`Integer.MIN_VALUE`

14.3.4 Tracing the Error Codes to Watch

The error codes to watch are applicable only for tracing. If tracing is disabled, then the error codes to watch setting is ignored.
There are two event types that trigger the trace buffers to be dumped into an appropriate logger:

- Any log message with a `SEVERE` log level.
- A few subset of exceptions or errors, for example, the subclasses of the `java.lang.Throwable` class, which are thrown by the JDBC driver or UCP. These errors or exceptions are then caught internally and logged with the `WARNING` level. The subset of these exceptions are defined with the `ucp.diagnostic.errorCodesToWatchList` property.

A `WARNING` log message, which contains a subclass of `java.lang.Throwable`, can cause dumping of a trace buffer into a logger, if the exception is found in a list of error codes. If an exception contains a stack of errors or exceptions, then UCP traverses through that stack with an attempt to find a matching exception and the corresponding error code, if applicable. The format of the list of error code is as follows:

```
["<subclass1>.<Exception1>:111,222,333",
"<subclass2>.<Exception2>:444,555,666", "<subclass3>.<Exception3>"]
```

where, `subclass1.Exception1`, `subclass2.Exception2`, and `subclass3.Exception3` are the `java.lang.Throwable` subclasses.

For example, if you see an error code like the following:

```
[ "oracle.ucp.UniversalConnectionPoolException:45054,45065,45067",
"java.sql.SQLException:12521,12514,12757,12523",
"java.lang.IllegalStateException", "java.lang.NullPointerException" ]
```

It means that the tracing is triggered in the following sequence:

1. `oracle.ucp.UniversalConnectionPoolException` with the vendor error codes 45054, 45065, and 45067
2. `java.sql.SQLException` with the vendor error codes 12521, 12514, 12757, and 12523
3. `IllegalStateException`
4. `NullPointerException`

Only fully-qualified `java.lang.Throwable` subclass names can cause dumping of a trace buffer into a logger. For example, `java.sql.NullPointerException` is a valid name, while `NullP`, `NullPointerException`, `[“12154"]`, `[“ORA-“]`, or `[“ORA-12154"]` is not. If specified wrongly, then the subclass name causes parsing and/or class resolution errors logged, along with a `WARNING`. Any number conversion error too can cause a parsing error logged, along with a `WARNING`.

In case of parse and class resolution failures, every `WARNING` log record, with any exception, causes dumping a trace buffer. This is regardless of exception types and designated error codes. Once error codes are ignored in logging mode, they are not parsed anymore.

Comma-separated lists of error codes are used only in case of the following two classes and their subclasses:

- `java.sql.SQLException` and its subclasses like `java.sql.SQLRecoverableException` or `java.sql.SQLSyntaxError`
- `oracle.ucp.UniversalConnectionPoolException` and its subclasses like `NoAvailableConnectionsException`

If you specify comma-separated lists of error codes with other exception types, then those error codes are ignored. With an empty list of error codes for the two exceptions and their subclasses mentioned earlier, all error codes are applicable. The UCP Diagnosability has a default error codes list. Refer to "UCP Exception Error Codes" for the list of `oracle.ucp.UniversalConnectionPoolException` error codes.

If you want the trace buffer to be dumped by any `WARNING` message with an exception, then you can perform it in the following way:

Example 14-2 To dump traces by any exception

```
"["java.lang.Throwable"]"
```

14.3.5 MBeans for UCP Diagnosability

UCP provides two MBeans for UCP diagnosability.

- `ucp.admin`: It contains attributes amd operations for the whole JVM. It manages the buffer associated with all the pool data sources, and also the common in-memory buffer. You can change the initial values of the following MBean attributes to set it:
  - `enableTrace`
  - `enableLogging`
  - `loggingLevel`
  - `bufferSize`
  - `errorCodesToWatchList`
  - `loggingConfigFileName`

  Also, there is an operation named `dumpInMemoryTrace`. Launching this operation, you can dump contents of all in-memory buffers into their appropriate loggers.
- `ucp.admin.UniversalConnectionPoolMBean`: It contains a tree of existing pool insntances, where every instance has its own attribute to modify an appropriate property of that specific pool. The attributes are the following:
  - `enableTrace`
  - `enableLogging`
  - `loggingLevel`
  - `inMemoryTraceSize`

14.4 About Viewing Oracle RAC Statistics

UCP provides a set of Oracle RAC run-time statistics that are used to determine how well a connection pool is utilizing Oracle RAC features and are also used to help determine whether the connection pool has been configured properly to use the Oracle RAC features. The statistics report FCF processing information, run-time connection load balance success/failure rate, and affinity context success/failure rate.

The `OracleJDBCConnectionPoolStatistics` interface that is located in the `oracle.ucp.jdbc.oracle` package provides methods that are used to query the connection pool for Oracle RAC statistics. The methods of this interface can be called from a pool-enabled and pool-enabled XA data source using the data source's `getStatistics` method. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
...

Long rclbS = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getSuccessfulRCLBBasedBorrowCount();
System.out.println("The RCLB success rate is "+rclbS+".");
```

The data source's `getStatistics` method can also be called by itself and returns all connection pool statistics as a `String` and includes the Oracle RAC statistics.

14.4.1 Fast Connection Failover Statistics

The `getFCFProcessingInfo` method provides information on recent Fast Connection Failover (FCF) attempts in the form of a `String`. The FCF information is typically used to help diagnose FCF problems. The information includes the outcome of each FCF attempt (successful or failed), the relevant Oracle RAC instances, the number of connections that were cleaned up, the exception that triggered the FCF attempt failure, and more. The following example demonstrates using the `getFCFProcessingInfo` method:

```
Sting fcfInfo = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getFCFProcessingInfo();
System.out.println("The FCF information: "+fcfInfo+".");
```

Following is a sample output string from the `getFCFProcessingInfo()` method:

```
    Oct 28, 2008 12:34:02 SUCCESS <Reason:planned> <Type:SERVICE_UP> \
      <Service:"svvc1"> <Instance:"inst1"> <Db:"db1"> \
      Connections:(Available=6 Affected=2 FailedToProcess=0 MarkedDown=2 Closed=2) \
      (Borrowed=6 Affected=2 FailedToProcess=0 MarkedDown=2 MarkedDeferredClose=0
Closed=2) \
      TornDown=2 MarkedToClose=2 Cardinality=2
    ...
    Oct 28, 2008 12:09:52 SUCCESS <Reason:unplanned> <Type:SERVICE_DOWN> \
      <Service:"svc1"> <Instance:"inst1"> <Db:"db1"> \
      Connections:(Available=6 Affected=2 FailedToProcess=0 MarkedDown=2 Closed=2) \
      (Borrowed=6 Affected=2 FailedToProcess=0 MarkedDown=2 MarkedDeferredClose=0
Closed=2)
    ...
    Oct 28, 2008 11:14:53 FAILURE <Type:HOST_DOWN> <Host:"host1"> \
      Connections:(Available=6 Affected=4 FailedToProcess=0 MarkedDown=4 Closed=4) \
      (Borrowed=6 Affected=4 FailedToProcess=0 MarkedDown=4 MarkedDeferredClose=0
Closed=4)
```

If you enable logging, then the preceding information will also be available in the UCP logs and you will be able to verify the FCF outcome.

14.4.2 Run-Time Connection Load Balance Statistics

The run-time connection load balance statistics are used to determine if a connection pool is effectively utilizing the run-time connection load balancing feature of Oracle RAC. The statistics report how many requests successfully used the run-time connection load balancing algorithms and how many requests failed to use the algorithms. The `getSuccessfulRCLBBasedBorrowCount` method and the `getFailedRCLBBasedBorrowCount` method, respectively, are used to get the statistics. The following example demonstrates using the `getFailedRCLBBasedBorrowCount` method:

```
Long rclbF = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getFailedRCLBBasedBorrowCount();
System.out.println("The RCLB failure rate is: "+rclbF+".");
```

A high failure rate may indicate that the Oracle RAC Load Balancing Advisory or connection pool is not configured properly.

14.4.3 Connection Affinity Statistics

The connection affinity statistics are used to determine if a connection pools is effectively utilizing connection affinity. The statistics report the number of borrow requests that succeeded in matching the affinity context and how many requests failed to match the affinity context. The `getSuccessfulAffinityBasedBorrowCount` method and the `getFailedAffinityBasedBorrowCount` method, respectively, are used to get the statistics. The following example demonstrates using the `getFailedAffinityBasedBorrowCount` method:

```
Long affF = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getFailedAffinityBasedBorrowCount();
System.out.println("The connection affinity failure rate is: "+affF+".");
```

14.5 Exceptions and Error Codes

Many UCP methods throw the `UniversalConnectionPoolException`, with exception chaining supported. You can call the `printStackTrace` method on the thrown exception, to identify the root cause of the exception. The `UniversalConnectionPoolException` includes standard Oracle error codes that are in the range of 45000 and 45499. The `getErrorCode` method can be used to retrieve the error code for an exception.

A
Error Codes Reference

This appendix briefly discusses the general structure of Universal Connection Pool (UCP) error messages, UCP error messages for the connection pool layer, and UCP error messages for JDBC data sources and dynamic proxies. The appendix is organized as follows:

- General Structure of UCP Error Messages
- Connection Pool Layer Error Messages
- JDBC Data Sources and Dynamic Proxies Error Messages

Both the message lists are sorted by the error message number.

A.1 General Structure of UCP Error Messages

The general UCP error message structure enables run-time information to be appended to the end of a message, following a colon, as follows:

```
<error_message>:<extra_info>
```

For example, a `closed statement` error might be displayed as follows:

```
Closed Statement:next
```

This indicates that the exception was thrown during a call to the `next` method (of a result set object).

In some cases, the user can find the same information in a stack trace.

A.2 Connection Pool Layer Error Messages

This section lists UCP error messages for the connection pool layer.

Note:

Starting with Oracle Database Release 21c, the `All connections in the Universal Connection Pool are in use` exception, which typically means the exhaustion of a pool's working set, is extended. The exception now displays messages similar to the following message, which include a short statistics that improves UCP diagnosability:

```
All connections in the Universal Connection Pool are in use (5, 5, 5,
0, 0, 0, 10, 150, 5, 3)
```

Where, the numbers after the exception message mean the following:

- The first number is the number of borrowed connections in a pool
- The second number is the number of total connections in a pool
- The third number is the cumulative number of connection created since a pool's start up
- The fourth number is the cumulative number of connections closed since a pool's start up
- The fifth number is the cumulative number of abandoned connections, that is, connection processed by the abandonment timer mechanism
- The sixth number is the number of labeled connections in a pool
- The seventh number is the number of pending connection borrowing requests
- The eighth number is the number of connections to create over an existing set until the max pool size is reached
- The ninth number is the total number of connections at its peak since a pool's start up
- The tenth number is the number of borrowed connections at its peak since a pool's start up

Table A-1 Connection Pool Layer Error Messages

Error Message Number	Message
UCP-45001	Universal Connection Pool internal error
UCP-45002	No available connections in the Universal Connection Pool
UCP-45003	Universal Connection Pool already exists
UCP-45004	Invalid connection retrieval information
UCP-45005	Callback already registered
UCP-45006	Invalid Universal Connection Pool configuration

Table A-1 (Cont.) Connection Pool Layer Error Messages

Error Message Number	Message
UCP-45051	Inactive connection timeout timer scheduling failed
UCP-45052	Abandoned connection timeout timer scheduling failed
UCP-45053	Time-to-live connection timeout timer scheduling failed
UCP-45054	The Universal Connection Pool cannot be null
UCP-45055	Error when removing an available connection
UCP-45057	The `AvailableConnections` object cannot be null
UCP-45058	The `Failoverable` object cannot be null
UCP-45059	`MaxPoolsize` is set to 0. There are no connections to return
UCP-45060	Invalid life cycle state. Check the status of the Universal Connection Pool
UCP-45061	Universal Connection Pool is not started. Start the Universal Connection Pool before accessing
UCP-45062	The collection of available connections can only be set when the Universal Connection Pool is in the initialization state
UCP-45063	Universal Connection Pool has been shutdown while attempting to get a connection
UCP-45064	All connections in the Universal Connection Pool are in use
UCP-45065	Connection borrowing returned null
UCP-45091	Connection labeling callback already registered
UCP-45092	Borrowing labeled connection with no labeling callback registered
UCP-45093	Requested no-label connection but borrowing labeled connection
UCP-45097	Connection harvesting timer scheduling failed
UCP-45100	`ConnectionFactoryAdapter` returned null
UCP-45103	`ConnectionFactoryAdapter` must be an instance of `DataSourceConnectionFactoryAdapter`
UCP-45104	`ConnectionFactoryAdapter` object cannot be null
UCP-45105	`ConnectionFactoryAdapter` must be an instance of `ConnectionPoolDataSourceConnectionFactoryAdapter`
UCP-45106	`ConnectionFactoryAdapter` must be an instance of `XADataSourceConnectionFactoryAdapter`
UCP-45150	`UniversalPooledConnection` cannot be null
UCP-45152	`UniversalPooledConnectionStatus` object cannot be null
UCP-45153	The connection label key cannot be null or an empty string
UCP-45154	The connection labeling operation cannot be invoked on closed connections
UCP-45155	Connection harvesting callback already registered
UCP-45156	Abandoned connection timeout callback already registered
UCP-45157	Time-to-live connection timeout callback already registered
UCP-45201	The connection label key cannot be null or an empty string
UCP-45202	The cloning of the `ConnectionRetrievalInfo` object failed
UCP-45203	The Connection Request Info is null

Table A-1 (Cont.) Connection Pool Layer Error Messages

Error Message Number	Message
UCP-45251	`ConnectionPoolDataSource` cannot be null
UCP-45252	Invalid `ConnectionRetrievalInfo` object
UCP-45253	SQLException occurred while getting `PooledConnection` from `ConnectionPoolDataSource`
UCP-45254	Invalid connection type. Must be a `javax.sql.PooledConnection`
UCP-45255	SQLException while closing `PooledConnection`
UCP-45256	Data source cannot be null
UCP-45257	Cannot get Connection from Data source
UCP-45258	Invalid connection type. Must be a `java.sql.Connection`
UCP-45259	The connection to proxy must be an instance of `java.sql.Connection`
UCP-45260	`XADatasource` cannot be null
UCP-45261	`SQLException` occurred while getting `XAConnection` from `XADataSource`
UCP-45262	Invalid connection type. Must be a `javax.sql.XAConnection`
UCP-45263	`SQLException` occurred while closing `XAConnection`
UCP-45264	The connection cannot be null
UCP-45265	The connection to proxy must be an instance of `overstatement`
UCP-45266	The statement to proxy must be an instance of `ultraconservative`
UCP-45267	The connection to proxy must be an instance of `javax.sql.XAConnection`
UCP-45268	The Driver argument cannot be null
UCP-45269	The URL argument cannot be null
UCP-45301	Unable to get a connection for fail over information
UCP-45302	Unable to execute SQL query to get fail over information
UCP-45303	SQLException occurred while getting fail over information
UCP-45304	The event type cannot be null
UCP-45305	The event type is invalid. Event type must be `database/event/host or database/event/service`
UCP-45306	The fail over event type is invalid. It must be an `Intergovernmental`
UCP-45307	The affinity context is invalid. It must be an `Interconnection`
UCP-45308	Exception occurred while enabling fail over with remote ONS subscription
UCP-45350	Universal Connection Pool already exists in the Universal Connection Pool Manager. Universal Connection Pool cannot be added to the Universal Connection Pool Manager
UCP-45351	Universal Connection Pool not found in Universal Connection Pool Manager. Register the Universal Connection Pool with Universal Connection Pool Manager
UCP-45352	Cannot get Universal Connection Pool Manager instance
UCP-45353	Cannot get Universal Connection Pool Manager M Bean instance
UCP-45354	M Bean `Object Name` is not in the right format. Use the right format to construct `Object Name` for M Bean
UCP-45355	M Bean exception occurred while registering or unregistering the MBean

Table A-1 (Cont.) Connection Pool Layer Error Messages

Error Message Number	Message
UCP-45356	MBean already exits in the MBeanServer. Use a different name to register MBean
UCP-45357	Exception occurred when trying to register an object in the MBean server that is not a JMX compliant MBean
UCP-45358	The specified MBean does not exist in the repository
UCP-45359	Invalid target object type is specified. Check the managed resource
UCP-45360	Invalid MBean Descriptor is specified. Check the Universal Connection Pool Manager MBean Descriptor
UCP-45361	Runtime exception occurred while building MBeanInfo for Universal Connection Pool Manager MBean
UCP-45362	Runtime exception occurred while building constructors information for Universal Connection Pool Manager MBean
UCP-45363	Runtime exception occurred while building attributes information for Universal Connection Pool Manager MBean
UCP-45364	Runtime exception occurred while building operations information for Universal Connection Pool Manager MBean
UCP-45365	Universal Connection Pool must be an instance of `ConnectionConnectionPool` or `OracleConnectionConnectionPool`
UCP-45366	Invalid MBean Descriptor is specified. Check the JDBC Universal Connection Pool MBean Descriptor
UCP-45367	Runtime exception occurred while building `MBeanInfo` for JDBC Universal Connection Pool MBean
UCP-45368	Runtime exception occurred while building constructors information for JDBC Universal Connection Pool MBean
UCP-45369	Runtime exception occurred while building attributes information for JDBC Universal Connection Pool MBean
UCP-45370	Runtime exception occurred while building operations information for JDBC Universal Connection Pool MBean
UCP-45371	Runtime exception occurred while building attributes information for Universal Connection Pool MBean
UCP-45372	Runtime exception occurred while building operations information for Universal Connection Pool MBean
UCP-45373	Invalid MBean Descriptor is specified. Check the Universal Connection Pool MBean Descriptor
UCP-45374	Runtime exception occurred while building `MBeanInfo` for Universal Connection Pool MBean
UCP-45375	Cannot stop the UCP metric collection. Exception occurred while trying to stop the metric collection or while destroying the nouns or sensors.
UCP-45376	Metrics update timer task scheduling failed
UCP-45377	Problem occurred while updating UCP metric sensors
UCP-45378	Universal Connection Pool is not an instance of `OracleJDBCConnectionPool` and cannot access `ONSConfiguration` property
UCP-45379	Cannot set the connection pool name in Universal Connection Pool MBean. Check the connection pool name to avoid duplicates
UCP-45380	MBean object is null

Table A-1 (Cont.) Connection Pool Layer Error Messages

Error Message Number	Message
UCP-45381	MBean object name is null
UCP-45382	MBean display name is null
UCP-45383	Invalid adapter for pool creation in Universal Connection Pool Manager
UCP-45384	Invalid adapter for pool creation in Universal Connection Pool Manager MBean
UCP-45385	Error during pool creation in Universal Connection Pool Manager
UCP-45386	Error during pool creation in Universal Connection Pool Manager MBean
UCP-45401	Waiting threads LO watermark cannot be negative
UCP-45402	Waiting threads HI watermark cannot be negative
UCP-45403	Total worker threads limit cannot be negative
UCP-45404	Queue poll timeout cannot be negative
UCP-45405	The waiting threads HI watermark cannot be lower than the LO watermark
UCP-45406	The limit of total worker threads cannot be higher than the limit of waiting threads
UCP-45407	The error number is out of range
UCP-45408	Invalid operation because the logger is null

A.3 JDBC Data Sources and Dynamic Proxies Error Messages

This section lists the UCP error messages for JDBC data sources and dynamic proxies error messages.

Table A-2 JDBC Data Sources and Dynamic Proxies Error Messages

Error Message Number	Message
UCP-0	Unable to start the UCP
UCP-1	Unable to build the UCP
UCP-2	Invalid minimum pool size
UCP-3	Invalid maximum pool size
UCP-4	Invalid inactive connection timeout
UCP-5	Invalid connection wait timeout
UCP-6	Invalid time-to-live connection timeout
UCP-7	Invalid abandoned connection timeout
UCP-8	Invalid timeout check interval
UCP-9	Failed to enable Failover
UCP-10	Failed to set the `maxStatements` value
UCP-11	Failed to set the SQL string for validation
UCP-12	Invalid connection harvest trigger count
UCP-13	Invalid connection harvest max count
UCP-14	UCP is created already. Can not create the UCP again

Table A-2 (Cont.) JDBC Data Sources and Dynamic Proxies Error Messages

Error Message Number	Message
UCP-15	Exception occurred while destroying the UCP
UCP-16	Operation only applies to Oracle connection pools
UCP-17	Exception occurred while setting ONS configuration string
UCP-18	Failed to register labeling callback
UCP-19	Failed to remove labeling callback
UCP-20	Failed to register affinity callback
UCP-21	Failed to remove affinity callback
UCP-22	Invalid UCP configuration
UCP-23	Unable to create factory class instance with provided factory class name
UCP-24	Unable to set the User
UCP-25	Unable to set the Password
UCP-26	Unable to set the URL
UCP-27	The factory class must be an instance of `DataSource`
UCP-28	Cannot create connections. There are no available connections
UCP-29	Exception occurred while getting connection
UCP-30	UCP is not started
UCP-31	The connection is closed
UCP-32	Error occurred when applying label
UCP-33	Error occurred when removing the connection label
UCP-34	Error occurred when getting labels
UCP-35	Error occurred when getting unmatched labels
UCP-36	Error occurred when setting connection harvestable
UCP-37	Error occurred when registering harvesting callback
UCP-38	Error occurred when removing harvesting callback
UCP-39	Error occurred when registering abandoned-connection callback
UCP-40	Error occurred when removing abandoned-connection callback
UCP-41	Error occurred when registering time-to-live-connection callback
UCP-42	Error occurred when removing time-to-live-connection callback
UCP-43	The result set is closed
UCP-44	The statement is closed
UCP-45	Cannot set the connection pool name. Check the connection pool name to avoid duplicates
UCP-46	The SQL string is null
UCP-47	Error occurred when setting connection to be invalid
UCP-48	Unable to set the Connection properties
UCP-49	Unable to set the Database server name
UCP-50	Unable to set the Database port number
UCP-51	Unable to set the Database name

Table A-2 (Cont.) JDBC Data Sources and Dynamic Proxies Error Messages

Error Message Number	Message
UCP-52	Unable to set the data source name
UCP-53	Unable to set the data source description
UCP-54	Unable to set the data source network protocol
UCP-55	Unable to set the data source role name
UCP-56	Invalid max connection reuse time
UCP-57	Invalid max connection reuse count
UCP-58	The method is disabled
UCP-59	Unable to set the connection factory properties

B
UCP Exception Error Codes

This appendix contains a list of useful `oracle.ucp.UniversalConnectionPoolException` error codes.

```
/*
 * Universal Connection Pool error code range is 45000 - 45499 in RDBMS.
 * An error number is an offset from the base error code 45000.
 *
 * The error numbers are organized first by package; and then within
 * each package, by main components.  These are indicated with the
 * use of particular sub strings within the error numbers' field names.
 *
 * When adding a new error number, if it needs to be mapped to a
 * specific pool exception subclass, define the error number to be
 * smaller than UCP_RANGED_MAPPING_BASE.  Otherwise, define it to be
 * larger than UCP_RANGED_MAPPING_BASE, so that it can be mapped to
 * a generic UniversalConnectionPoolException.
 */
public static final int UCP_ERROR_CODE_BASE = 45000;
public static final int UCP_MAX_ERRORS = 500;

public static final int UCP_SUCCESS = 0;
public static final int UCP_GENERIC_ERROR = 1;

// Error numbers that are mapped to specific pool exceptions
public static final int UCP_NO_AVAILABLE_CONNECTIONS = 2;
public static final int UCP_POOL_ALREADY_EXISTS = 3;
public static final int UCP_INVALID_RETRIEVAL_CREDENTIALS = 4;
public static final int UCP_CALLBACK_ALREADY_REGISTERED = 5;
public static final int UCP_INVALID_POOL_CONFIGURATION = 6;

/*
 * All error numbers >= this number will be mapped to a generic
 * UniversalConnectionPoolException.  Error numbers smaller than this
 * have specific mappings to a UniversalConnectionPoolException subclass.
 */
public static final int UCP_RANGED_MAPPING_BASE = 50;

// POOL (oracle.ucp.common) - use _COMMON_POOL_

// Error number 50 is available for use
public static final int UCP_COMMON_POOL_INACTIVE_TIMER_SCHEDULE = 51;
public static final int UCP_COMMON_POOL_ABANDONED_TIMER_SCHEDULE = 52;
public static final int UCP_COMMON_POOL_TTL_TIMER_SCHEDULE = 53;
public static final int UCP_COMMON_POOL_NULL = 54;
public static final int UCP_COMMON_POOL_RM_AVAIL_CONN = 55;
public static final int UCP_COMMON_POOL_NO_VALID_CONNECTION = 56;
public static final int UCP_COMMON_POOL_AVAILABLECONNECTIONS_NULL = 57;
public static final int UCP_COMMON_POOL_FAILOVERABLE_NULL = 58;
public static final int UCP_COMMON_POOL_NOCONNECTIONS = 59;
public static final int UCP_COMMON_POOL_INVALID_LIFECYCLE_STATE = 60;
public static final int UCP_COMMON_POOL_NOTSTARTED = 61;
public static final int UCP_COMMON_POOL_SETAVAILABLECONNECTIONS = 62;
public static final int UCP_COMMON_POOL_SHUTDOWN = 63;
public static final int UCP_COMMON_POOL_ALLCONNECTIONS_INUSE = 64;
public static final int UCP_COMMON_POOL_ABOUT_TO_SHUTDOWN = 65;
public static final int UCP_COMMON_POOL_INSUFFICIENT_INITIAL_CONNECTIONS = 66;
public static final int UCP_COMMON_POOL_UNABLE_TO_CREATE_CONNECTION = 67;
public static final int SQL_AC_INIT_CALLBACK_ERROR = 68;
// Error numbers 69 - 90 are available for use
public static final int UCP_COMMON_POOL_LABELING_CBK_REGISTERED = 91;
public static final int UCP_COMMON_POOL_NO_LABELING_CBK = 92;
public static final int UCP_COMMON_POOL_LABEL_BORROW_MISMATCH = 93;
// Error numbers 94 - 96 are available for use
public static final int UCP_COMMON_POOL_HARVEST_TIMER_SCHEDULE = 97;

// Connection factory adapter - use _COMMON_CFA_
public static final int UCP_COMMON_CFA_RETURNED_NULL = 100;
// Error numbers 101 - 102 are available for use
public static final int UCP_COMMON_CFA_INSTANCE_ERROR1 = 103;
public static final int UCP_COMMON_CFA_NULL = 104;
public static final int UCP_COMMON_CFA_INSTANCE_ERROR2 = 105;
public static final int UCP_COMMON_CFA_INSTANCE_ERROR3 = 106;

// Universal pooled connection - use _COMMON_UPC_
public static final int UCP_COMMON_UPC_NULL = 150;
public static final int UCP_COMMON_UPC_WRONG_SQL = 151;
public static final int UCP_COMMON_UPC_STATUS_NULL = 152;
public static final int UCP_COMMON_UPC_LABEL_KEY_EMPTY = 153;
public static final int UCP_COMMON_UPC_CLOSED = 154;
public static final int UCP_COMMON_UPC_HARVESTING_CBK_REGISTERED = 155;
public static final int UCP_COMMON_UPC_ABANDONED_CBK_REGISTERED = 156;
public static final int UCP_COMMON_UPC_TTL_CBK_REGISTERED = 157;
public static final int UCP_COMMON_UPC_BAD = 158;

// Connection retrieval info - use _CRI_

// Error number 200 is available for use
public static final int UCP_COMMON_CRI_LABEL_KEY_EMPTY = 201;
public static final int UCP_COMMON_CRI_NOLABEL_CLONE_FAILURE = 202;
public static final int UCP_COMMON_CRI_NULL = 203;
public static final int UCP_COMMON_SERVICE_MISMATCH = 204;

// JDBC POOL (oracle.ucp.jdbc) - use _JDBC_

// Error number 250 is available for use
public static final int UCP_JDBC_CONNECTIONPOOLDATASOURCE_NULL = 251;
public static final int UCP_JDBC_INVALID_CONNECTIONRETRIEVALINFO_OBJECT = 252;
public static final int UCP_JDBC_CONNECTIONPOOLDATASOURCE_SQLEXCEPTION = 253;
public static final int UCP_JDBC_INVALID_CONNECTIONTYPE = 254;
public static final int UCP_JDBC_CONNECTIONCLOSE_EXCEPTION = 255;
public static final int UCP_JDBC_DATASOURCE_NULL = 256;
public static final int UCP_JDBC_GETCONNECTION_EXCEPTION = 257;
public static final int UCP_JDBC_INVALID_PROXY_CONNECTIONTYPE = 258;
public static final int UCP_JDBC_PROXY_CONNECTION_EXCEPTION = 259;
public static final int UCP_JDBC_NOT_XADATASOURCE_NULL = 260;
public static final int UCP_JDBC_XADATASOURCE_SQLEXCEPTION = 261;
public static final int UCP_JDBC_INVALID_XACONNECTIONTYPE = 262;
public static final int UCP_JDBC_XACONNECTIONCLOSE_EXCEPTION = 263;
public static final int UCP_JDBC_CONNECTION_NULL = 264;
public static final int UCP_JDBC_PROXY_STATEMENT_EXCEPTION = 265;
public static final int UCP_JDBC_PROXY_RESULTSET_EXCEPTION = 266;
public static final int UCP_JDBC_PROXY_XACONNECTION_EXCEPTION = 267;
public static final int UCP_JDBC_DRIVER_NULL = 268;
public static final int UCP_JDBC_URL_NULL = 269;
public static final int UCP_JDBC_POOL_INIT_CBK_REGISTERED = 270;
public static final int UCP_JDBC_POOL_INIT_CBK_FAILURE = 271;
public static final int UCP_JDBC_INVALID_USE_OF_SHARED_POOL = 272;
public static final int UCP_JDBC_NON_SHARED_POOL_INVALID_CONFIG_EXCEPTION =
273;
public static final int UCP_JDBC_UNABLE_TO_SET_QUERY_TIMEOUT = 274;
public static final int UCP_JDBC_DUPLICATE_POOL_NAME = 275;
public static final int UCP_JDBC_DUPLICATE_DATASOURCE_NAME = 276;
public static final int UCP_JDBC_MISSING_SHARD_KEY_CONNECTION_REQUEST = 277;
public static final int UCP_JDBC_INVALID_CONNECTION_REQUEST_PARAMETER = 278;
public static final int UCP_JDBC_CANNOT_RECONFIGURE_MAX_PER_SERVICE = 279;
public static final int UCP_JDBC_INVALID_GLOBAL_SERVICE_NAME = 280;

// Oracle JDBC POOL (oracle.ucp.jdbc.oracle) - use _JDBC_ORACLE_

// Error number 300 is available for use
public static final int UCP_JDBC_ORACLE_FOVR_CONN_NULL = 301;
public static final int UCP_JDBC_ORACLE_FOVR_CONN_QUERY = 302;
public static final int UCP_JDBC_ORACLE_FOVR_CONN_SQLEXC = 303;
public static final int UCP_JDBC_ORACLE_EVENTTYPE_NULL = 304;
public static final int UCP_JDBC_ORACLE_INVALID_EVENTTYPE = 305;
public static final int UCP_JDBC_ORACLE_INVALID_FAILOVER_EVENTTYPE = 306;
public static final int UCP_JDBC_ORACLE_INVALID_AFFINITY_CXT = 307;
public static final int UCP_JDBC_ORACLE_REMOTE_ONS_PRIVILEGE = 308;
public static final int UCP_JDBC_ORACLE_BEGINREQUEST_FAILURE = 309;
public static final int UCP_JDBC_ORACLE_ENDREQUEST_FAILURE = 310;
public static final int UCP_JDBC_ORACLE_NO_AVAIL_CONN_FOR_STRICT_AFFINITY =
311;
public static final int UCP_JDBC_ORACLE_REMOTE_ONS_INIT = 312;
public static final int UCP_JDBC_ORACLE_AUTO_ONS_CONFIG = 313;
public static final int UCP_JDBC_ORACLE_INVALID_RAC_DATA_AFFINITY_CONFIG =
314;

// Admin (oracle.ucp.admin) - use _ADMIN_
public static final int UCP_ADMIN_MGR_POOL_ALREADY_EXISTS = 350;
public static final int UCP_ADMIN_MGR_POOL_DOESNOT_EXIST = 351;
public static final int UCP_ADMIN_MGR_CANNOT_GETINSTANCE = 352;
public static final int UCP_ADMIN_MGRMBEAN_CANNOT_GETINSTANCE = 353;
public static final int UCP_ADMIN_MBEAN_MALFORM_OBJECTNAME = 354;
public static final int UCP_ADMIN_MBEAN_REG_UNREG_EXCEPTION = 355;
public static final int UCP_ADMIN_MBEAN_INSTANCE_EXISTS = 356;
public static final int UCP_ADMIN_MBEAN_NOT_COMPLIANT = 357;
public static final int UCP_ADMIN_MBEAN_INSTANCE_NOTFOUND = 358;
public static final int UCP_ADMIN_MBEAN_INVALID_TARGET = 359;
public static final int UCP_ADMIN_MGRMBEAN_DESCRIPTOR_EXCEPTION = 360;
public static final int UCP_ADMIN_MGRMBEAN_MBEANINFO_EXCEPTION = 361;
public static final int UCP_ADMIN_MGRMBEAN_CONINFO_EXCEPTION = 362;
public static final int UCP_ADMIN_MGRMBEAN_ATTRINFO_EXCEPTION = 363;
public static final int UCP_ADMIN_MGRMBEAN_OPERINFO_EXCEPTION = 364;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_INSTANCE = 365;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_DESCRIPTOR_EXCEPTION = 366;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_MBEANINFO_EXCEPTION = 367;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_CONINFO_EXCEPTION = 368;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_ATTRINFO_EXCEPTION = 369;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_OPERINFO_EXCEPTION = 370;
public static final int UCP_ADMIN_COMMONPOOLMBEAN_ATTRINFO_EXCEPTION = 371;
public static final int UCP_ADMIN_COMMONPOOLMBEAN_OPERINFO_EXCEPTION = 372;
public static final int UCP_ADMIN_COMMONPOOLMBEAN_DESCRIPTOR_EXCEPTION = 373;
public static final int UCP_ADMIN_COMMONPOOLMBEAN_MBEANINFO_EXCEPTION = 374;
public static final int UCP_ADMIN_STOP_METRIC_COLLECTION = 375;
public static final int UCP_ADMIN_METRIC_UPDATE_TIMER = 376;
public static final int UCP_ADMIN_METRIC_UPDATE_SENSORS = 377;
public static final int UCP_ADMIN_JDBCPOOLMBEAN_ORACLEPOOL_NULL = 378;
public static final int UCP_ADMIN_COMMONPOOLMBEAN_CANNOT_SET_POOLNAME = 379;
public static final int UCP_ADMIN_MBEAN_NULL = 380;
public static final int UCP_ADMIN_MBEAN_OBJNAME_NULL = 381;
public static final int UCP_ADMIN_MBEAN_DISPLAYNAME_NULL = 382;
public static final int UCP_ADMIN_MGR_INVALID_ADAPTER = 383;
public static final int UCP_ADMIN_MGRMBEAN_INVALID_ADAPTER = 384;
public static final int UCP_ADMIN_MGR_POOL_CREATION = 385;
public static final int UCP_ADMIN_MGRMBEAN_POOL_CREATION = 386;

public static final int UCP_POOL_CONFIGURATION_INVALID_XML = 387;
public static final int UCP_POOL_RECONFIGURATION_INVALID_XML = 388;
public static final int UCP_POOL_MAX_PER_SHARD_LIMIT_EXCEEDED = 389;


// Utilities (oracle.ucp.util) - use _UTIL_ or _WTP_

// Utilities - Worker Thread Pool (WTP) - use _WTP_

// Error number 400 is available for use
public static final int UCP_WTP_MIN_WAITING_THREADS_NEGATIVE = 401;
public static final int UCP_WTP_MAX_WAITING_THREADS_NEGATIVE = 402;
public static final int UCP_WTP_MAX_TOTAL_THREADS_NEGATIVE = 403;
public static final int UCP_WTP_QUEUE_POLL_TIMEOUT_NEGATIVE = 404;
public static final int UCP_WTP_WAITING_THREADS_MIN_GT_MAX = 405;
public static final int UCP_WTP_TOTAL_THREADS_LT_WAITING = 406;

public static final int UCP_UTIL_ERROR_OUT_OF_RANGE = 407;
public static final int UCP_UTIL_NULL_LOGGER = 408;

// Error numbers for WLS JTA support
public static final int UCP_WLS_XA_AFFINITY_VIOLATION = 420;
```