Spring Data JDBC Extensions Reference Documentation

1.1.0.RELEASE

ThomasRisberg

Table of Contents

Preface

The Spring Data JDBC Extensions project provides advanced JDBC features that extends the support provided by the "spring-jdbc" module in the Spring Framework project.

The bulk of the features in the Spring Data JDBC Extensions project is made up of code ported from the SpringSource project "Advanced Pack for Oracle Database" that was available for support subscription customers. We are now making this code available to all Spring users and any new developments will be made in the Spring Data JDBC Extensions project.

There is also support for using the Querydsl SQL module to provide type-safe query, insert, update and delete functionality.

Part I. Core JDBC Extensions

This part of the reference documentation details the core extended JDBC support that can be used for any supported SQL database.

Overview

The JDBC support in the Spring Framework is extensive and covers the most commonly used features, but there are some new usage scenarios like type-safe queries that warrants some extension to be provided. The core part of the *Spring Data JDBC Extensions* project provides this type of extension and it can be used together with any supported SQL database.

1 Features provided

The following lists the various features that are covered. Each feature is documented in more detail in the following chapters

- Core support
  The core support provides extensions to the Spring Framework JDBC support.
- Querydsl
  The Querydsl project provides a way to work with many datastore in a type-safe manner. This includes support for working with SQL databases. We provide the "glue" that let's you easily work with Querydsl in a Spring based project.

2 Requirements

The requirements for using the features provided in the `core` module of the "Spring Data JDBC Extensions" project are listed below.

- Java 6 or later
  The minimum Java version is now 1.6.
- Spring Framework 3.0
  All Spring Framework features that are needed are provided in Spring Framework version 3.0 or later.
- Apache Commons Logging
  Apache Commons Logging is used by the Spring Framework but it can be replaced by the jcl-over-slf4j bridge provided by the SLF4J project.
- Querydsl
  The Querydsl support requires the use of Querydsl SQL module version 3.0.0 or later.

1. Support classes that extend Spring features

The Spring Framework projects JDBC support is excellent but every now and then there are some features that seem useful, but might not warrant inclusion in the framework project itself. The Spring Data JDBC Extensions project provides a home for these type of extensions.

1.1 Mapping a one-to-many relationship

We often have to map one-to-many relationships in our database projects. A customer can have many addresses, an order can contain many line items and so on. We are now providing a `ResultSetExtractor` implementation to deal with this common task.

Let's look at the schema definition first:

```
CREATE TABLE customer(
  id BIGINT IDENTITY PRIMARY KEY,
  name VARCHAR(255));
CREATE TABLE address (
  id BIGINT IDENTITY PRIMARY KEY,
  customer_id BIGINT CONSTRAINT address_customer_ref
    FOREIGN KEY REFERENCES customer (id),
  street VARCHAR(255),
  city VARCHAR(255));
```

Two tables linked by a foreign key constraint. To map this we need two domain classes - `Customer` and `Address` where `Customer` has a Set of `Addresses`.

```
public class Customer {

  private Integer id;

  private String name;

  private Set<Address> addresses = new HashSet<Address>();

  public Set<Address> getAddresses() {
    return addresses;
  }

  public void addAddress(Address address) {
    this.addresses.add(address);
  }

// other setters and getters

}
```

```
public class Address {

  private Integer id;

  private String street;

  private String city;

// setters and getters

}
```

Executing the following query we would potentially get multiple rows returned for each customer.

```
List<Customer> result = template.query(
  "select customer.id, customer.name, address.id, " +
    "address.customer_id, address.street, address.city " +
    "from customer " +
    "left join address on customer.id = address.customer_id " +
    "order by customer.id",
  resultSetExtractor);
```

To be able to handle the multiple rows we create a new `CustomerAddressExtractor` that extends the abstract class `OneToManyResultSetExtractor`. We parameterize the `OneToManyResultSetExtractor` with the root class (`Customer`), the child class (`Address`), and the class for the primary and foreign key (`Integer`).

```
public class CustomerAddressExtractor extends
    OneToManyResultSetExtractor<Customer, Address, Integer> {

  public CustomerAddressExtractor() {
    super(new CustomerMapper(), new AddressMapper());
  }

  @Override
  protected Integer mapPrimaryKey(ResultSet rs) throws SQLException {
    return rs.getInt("customer.id");
  }

  @Override
  protected Integer mapForeignKey(ResultSet rs) throws SQLException {
    if (rs.getObject("address.customer_id") == null) {
      return null;
    }
    else {
      return rs.getInt("address.customer_id");
    }
  }

  @Override
  protected void addChild(Customer root, Address child) {
    root.addAddress(child);
  }
}
```

We need a way to match the primary key of the Customer with the foreign key of the Address so we provide mappings for these via the abstract methods `mapPrimaryKey` and `mapForeignKey`. We have to take into account that there might not be an address record for every customer so the foreign key could be `null`. We also need to add the mapped `Address` instances to the `Customer` instance. We do this

by implementing the abstract method `addChild`. We simply call the `addAddress` on the `Customer` class here.

Looking at the constructor of the CustomerAddressExtractor we see that we call the super constructor providing `RowMapper` implementations for the `Customer` and the `Address` classes. These are standard `RowMappers` that we in this example provide as static inner classes.

```
private static class CustomerMapper implements RowMapper<Customer> {

  public Customer mapRow(ResultSet rs, int rowNum) throws SQLException {
    Customer c = new Customer();
    c.setId(rs.getInt("customer.id"));
    c.setName(rs.getString("customer.name"));
    return c;
  }
}
```

```
private static class AddressMapper implements RowMapper<Address> {

  public Address mapRow(ResultSet rs, int rowNum) throws SQLException {
    Address a = new Address();
    a.setId(rs.getInt("address.id"));
    a.setStreet(rs.getString("address.street"));
    a.setCity(rs.getString("address.city"));
    return a;
  }
}
```

We now have a complete solution for this common problem.

2. Querydsl Support

The Querydsl project provides a framework that let's you write type-safe queries in Java rather than constructing them using strings. This has several advantages like code completion in your IDE, domain types and properties can be accessed in a type-safe manner reducing the probability of query syntax errors during run-time. Querydsl has modules that support JPA, JDO, SQL, MongoDB and more. It is the SQL support that is used for the JDBC Extensions project. You can read more about Querydsl at their website http://www.querydsl.com.

2.1 Introduction to Querydsl

Before you can use the Spring support for Querydsl you need to configure your application to use the Querydsl SQL support. See the instruction in the Mysema blog post on how this is done. Once you have generated your Querydsl query types then you can start using the Spring support as outlined bellow.

2.2 QueryDslJdbcTemplate

The central class in the Querydsl support is the `QueryDslJdbcTemplate`. Just like the `NamedParameterJdbcTemplate` it wraps a regular `JdbcTemplate` that you can get access to by calling the `getJdbcOperations` method. One thing to note is that when you use the `QueryDslJdbcTemplate`, there is no need to specify the SQL dialect to be used since the template will auto-detect this when it is created.

You can create a QueryDslJdbcTemplate by passing in a JdbcTemplate or a DataSource in the constructor. Here is some example code showing this:

```
private QueryDslJdbcTemplate template;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.template = new QueryDslJdbcTemplate(dataSource);
}
```

At this point the template is ready to be used and we give examples for various uses below.

2.3 Queries

For queries you need to have a reference to the query type. For the examples in this document we define the query type as follows:

```
private final QProduct qProduct = QProduct.product;
```

Now we are ready to create the first query. Instead of directly creating an instance of `SQLQueryImpl` we ask the template for a managed instance.

```
SQLQuery sqlQuery = template.newSqlQuery()
```

The managed part here refers to the managing of the connection and the SQL dialect. The `QueryDslJdbcTemplate` will provide both of these. The dialect is set when the `SQLQuery` is created and the connection is provided when the `SQLQuery` is executed using the clone feature of the `SQLQuery` implementation class.

We continue to build this query providing from and where clauses:

```
SQLQuery sqlQuery = template.newSqlQuery().from(qProduct)
        .where(qProduct.category.eq(categoryId));
```

Here *`categoryId`* is a parameter that is passed in to the method.

The final step is to execute the query. Depending on how you want to map the results, there are two flavors of the query methods. You can

- use the method taking a regular Spring `RowMapper` together with a projection in the form of a Querydsl `Expression`.

or

- use the method that takes a Querydsl class implementing `Expression` like an extension of the handy `MappingProjection`, a `QBean` implementation or a Querydsl query type combined with a Querydsl bean type to specify the mapping.

Here is an example using the query created above together with a `MappingProjection` for mapping the query results:

```
public List<Product> getProductListByCategory(final String categoryId)
        throws DataAccessException {

    SQLQuery sqlQuery = template.newSqlQuery().from(qProduct)
            .where(qProduct.category.eq(categoryId));

    return template.query(sqlQuery, new MappingProductProjection(qProduct);
}

private static class MappingProductProjection extends MappingProjection<Product> {

    public MappingProductProjection(QProduct qProduct) {
        super(Product.class, qProduct.productid,
            qProduct.name, qProduct.descn, qProduct.category));
    }

    @Override
    protected Product map(Tuple tuple) {
        Product product = new Product();

        product.setProductId(tuple.get(qProduct.productid));
        product.setName(tuple.get(qProduct.name));
        product.setDescription(tuple.get(qProduct.descn));
        product.setCategoryId(tuple.get(qProduct.category));

        return product;
    }
}
```

2.4 Inserts

For inserts we need to call the template's `insert` method and implement an `SqlInsertCallback` to handle the mapping of data from the domain object values to the insert. Here is an example:

```
public void insertProduct(final Product product) throws DataAccessException {
    template.insert(qProduct, new SqlInsertCallback() {
        public long doInSqlInsertClause(SQLInsertClause sqlInsertClause) {
            return sqlInsertClause.columns(qProduct.productid, qProduct.name,
                        qProduct.descn, qProduct.category)
                    .values(product.getProductId(), product.getName(),
                        product.getDescription(), product.getCategoryId())
                    .execute();
        }
    });
}
```

2.5 Updates

Updates are similar to the inerts but we of course call the `update` method and implement an `SqlUpdateCallback` to provide the where clause and handle the mapping of update parameter values.

```
public void updateProduct(final Product product) throws DataAccessException {
    template.update(qProduct, new SqlUpdateCallback() {

        public long doInSqlUpdateClause(SQLUpdateClause sqlUpdateClause) {
            return sqlUpdateClause.where(qProduct.productid.eq(product.getProductId()))
                    .set(qProduct.name, product.getName())
                    .set(qProduct.descn, product.getDescription())
                    .set(qProduct.category, product.getCategoryId())
                    .execute();
        }
    });
}
```

2.6 Deletes

Deletes are also very similar except we don't need to do any value mapping. We simply call the `delete` method and implement an `SqlDeleteCallback` with a where clause.

```
public void deleteProduct(final Product product) {
    template.delete(qProduct, new SqlDeleteCallback() {

        public long doInSqlDeleteClause(SQLDeleteClause sqlDeleteClause) {
            return sqlDeleteClause.where(qProduct.productid.eq(product.getProductId()))
                    .execute();
        }
    });
}
```

Part II. JDBC Extensions for the Oracle Database

This part of the reference documentation details the extended JDBC support provided for the Oracle database.

Overview

The Oracle Database is a powerful relational database that continues to lead the market in several areas. It has advanced support for replication and clustering, powerful support for stored procedures and support for advanced data types and XML data types. Many of these features aren't directly supported by the JDBC standard and Oracle has developed several extensions to their JDBC implementation to make access to these features possible.

The *Spring Data JDBC Extensions* project's support for the Oracle Database provides a simplified approach to gain access to these advanced features and at the same time provide this support in a fashion that is in line with the JDBC support provided by the Spring Framework and its JDBC abstraction.

1 Features provided

The majority of features provided by the Oracle Database and their JDBC implementation are already well supported by the core Spring Framework. There are however some advanced features not covered by the JDBC specification that provide some interesting functionality. The Spring Data JDBC Extension project provides explicit support for some of these features and it also provides documentation and examples how to take advantage of some of Oracle's JDBC extensions using standard Spring Framework APIs.

The following lists the various features that are covered. Each feature is documented in more detail in the following chapters

- RAC "Fast Connection Failover"
  The RAC "Fast Connection Failover" provides the ability to have a Spring application transparently failover when a database node fails.
- Streams AQ (Advanced Queueing)
  The AQ support provides the option of using a single local transaction manager for both database and message access without resorting to expensive distributed 2-phase commit transaction management.
- XML Types
  Custom classes, examples and documentation on how to use Oracle JDBC extensions for their native XML Type.
- Advanced Data Types
  Custom classes, examples and documentation on how to use Oracle JDBC extensions for their advanced data types like STRUCT and ARRAY.
- Custom DataSource Connection Preparer
  This feature provides an API for customizing the connection environment with Oracle specific session settings etc.

2 Requirements

The requirements for using the features provided in the `oracle` module of the "Spring Data JDBC Extensions" project are listed below.

- Java 6 or later
  The minimum Java version is now 1.6.

- Spring Framework 3.0
  All Spring Framework features that are needed are provided in Spring Framework version 3.0 or later.
- JDBC driver for Oracle 10g R2
  All features are supported using the Oracle JDBC driver 10.2.0.2 or later. Using a recent 11gR2 or later driver is recommended.
- Apache Commons Logging
  Apache Commons Logging is used by the Spring Framework but it can be replaced by the jcl-over-slf4j bridge provided by the SLF4J project.
- Spring Retry
  The Fast Connection Failover support requires using the Spring Retry project (https://github.com/spring-projects/spring-retry).

3. Oracle Pooling DataSource

Oracle provides an advanced DataSource implementation that has some unique features. It provides connection pooling and it is required when using "Fast Connection Failover" for RAC.

3.1 Configuration using the traditional <bean> element

We'll start by looking at a very basic DataSource configuration using the traditional <bean> element.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="dataSource" class="oracle.jdbc.pool.OracleDataSource" destroy-
method="close"> ❶
        <property name="URL" value="${jdbc.url}" /> ❷
        <property name="user" value="${jdbc.username}"/> ❸
        <property name="password" value="${jdbc.password}"/> ❹
        <property name="connectionCachingEnabled" value="true"/> ❺
    </bean>

    <context:property-placeholder location="classpath:jdbc.properties"/> ❻

</beans>
```

❶ Here we specify the `DataSource` implementation class as the `OracleDataSource`.

❷ We specify the URL using the URL property. Note that it is upper case in this implementation while it is lower case in most other `DataSource` implementations.

❸ The user name is specified using the `user` property.

❹ The password is specified using the `password` property.

❺ The connection caching must be enabled explicitly using the `connectionCachingEnabled` property.

❻ The property place holders will be filled in using this `<context:property-placeholder>` element from the context namespace.

3.2 Using the "orcl" namespace to configure the OracleDataSource

The new "orcl" namespace contains a `pooling-data-source` element used for easy configuration of the `OracleDataSource`. We will show several ways this element can be used and we will start with a basic one that can replace the traditional <bean> element configuration used above.

When using the `pooling-data-source` element connection caching is enabled by default and must explicitly be turned off using the `connection-caching-enabled` attribute if you don't want to use this pooling support.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl" ❶
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl ❷
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        url="${jdbc.url}" username="${jdbc.username}" password="${jdbc.password}"/> ❸

    <context:property-placeholder location="classpath:jdbc.properties"/> ❹

</beans>
```

❶ Here we specify the reference to the `orcl` schema.

❷ We also specify the location for the `orcl` schema.

❸ The properties needed to connect to the database in this example are `url`, `username` and `password`. The `url` property could also be specifiec as URL and the `username` property could be specifed as `user`.

❹ Just as in the previous example, the property place holders will be filled in using this `<context:property-placeholder>` element from the context namespace.

3.3 Using a properties file directly for connection properties

We used a `property-placeholder` in the previous example to provide connection properties. We can also read the properties directly from a properties file without using placeholders. This is done by using a `properties-location` attribute specifying the location of the properties file.

Note

When you specify properties using a property file there are two basic properties, url and username, where you can use the Oracle property name or the name traditionally used by Spring developers. For url we also accept URL and for username we also accept user.

We will use the following property file named `orcl.properties` and we will place it at the root of the classpath.

```
url=jdbc:oracle:thin:@//maui:1521/xe
username=spring
password=spring
```

Once we have this file in place we can reference it from our `pooling-data-source` entry and omit the property plceholder declarations for any properties provided in this file.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        properties-location="classpath:orcl.properties"/> ❶

</beans>
```

❶ The `pooling-datasource` with the `properties-location` specified. The `URL`, `user` and `password` properties will be read from the provided properties file.

You can even remove the `properties-location` attribute as long as you use the default location and name which is a file named `orcl.properties` at the root of the classpath.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"/> ❶

</beans>
```

❶ The `pooling-datasource` without properties or the `properties-location` specified. We are relying on the default properties file name and location.

3.4 Additional connection and cache properties

It's sometimes necessary to provide additional connection properties to control how the database access is configured. There are several ways you can provide these properties and they are outlined below.

Using the property file for additional connection properties

We can provide additional connection properties by ust adding them to the properties file we used in the example above.

```
url=jdbc:oracle:thin:@//maui:1521/xe
username=spring
password=spring
processEscapes=false
```

Any properties specified in addition to the standard URL/url, user/username and password will be used for configuring the `OracleDataSource`.

We can also use a prefix for the connection properties. This can be useful if the properties file contain other properties like connection cache properties. We will see how these additional properties are used later on.

```
conn.url=jdbc:oracle:thin:@//maui:1521/xe
conn.username=spring
conn.password=spring
conn.processEscapes=false
```

The prefix must be specified in the `pooling-data-source` element configuration. It is specified using the `connection-properties-prefix` attribute.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        connection-properties-prefix="conn" ❶
        properties-location="classpath:orcl.properties"/>

</beans>
```

❶ The `connection-properties-prefix` is specified here.

Using the property file for additional cache properties

We can also specify connection cache properties in the properties file. We must use a prefix for these connection cache properties to distinguish them from the regular connection properties. In this example we are using "cache" as the prefix.

```
conn.url=jdbc:oracle:thin:@//maui:1521/xe
conn.username=spring
conn.password=spring
conn.processEscapes=false
cache.InitialLimit=10
```

The connection cache prefix must be specified using the `connection-cache-properties-prefix` attribute.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        connection-properties-prefix="conn"
        connection-cache-properties-prefix="cache" ❶
        properties-location="classpath:orcl.properties"/>

</beans>
```

❶ The `connection-cache-properties-prefix` is specified here.

Using "connection-properties" element for additional connection properties

The connection properties can be specified using the `connection-properties` element.

Note

If you specify a `connection-properties` element then any connection properties specified in a property file other than the basic url, username and password will not be used.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        properties-location="classpath:orcl.properties">
        <orcl:connection-properties>
            processEscapes=false ❶
        </orcl:connection-properties>
    </orcl:pooling-datasource>

</beans>
```

❶ The connection properties are specified here.

Using "connection-cache-properties" element for additional cache properties

The connection cache properties can be specified using the `connection-cache-properties` element.

Note

If you specify a `connection-cache-properties` element then any connection cache properties specified in a property file will not be used.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        properties-location="classpath:orcl.properties">
        <orcl:connection-properties>
            processEscapes=false
        </orcl:connection-properties>
        <orcl:connection-cache-properties>
            InitialLimit=10 ❶
        </orcl:connection-cache-properties>
    </orcl:pooling-datasource>

</beans>
```

❶ The connection cache properties are specified here.

Using "username-connection-proxy" element for proxy connections

The Oracle JDBC driver provides proxy authentication. This means that you can configure a connection pool using a proxy user account with limited rights. Then during the connection process you would specify the actual username for the end user. This username must be configured to allow a proxy connection through the user proxy ("grant connect through proxyuser"). See ??? for more details on this usage.

Connection proxy authentication is configured using the `username-connection-proxy` element. You also need to provide a user name provider that implements the `ConnectionUsernameProvider` interface. This interface has a single method named `getUserName` that should return the username for the current end user to be connected via the proxy user.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        properties-location="classpath:orcl.properties">
        <orcl:username-connection-proxy connection-context-provider="usernameProvider"/> ❶
    </orcl:pooling-datasource>

    <bean id="usernameProvider"
 class="org.springframework.data.jdbc.test.CurrentUsernameProvider"/>

</beans>
```

❶ The connection proxy username provider is specified here.

3.5 Summary of configuration options for the "pooling-data-source"

The `pooling-data-source` element has the following attributes:

Table 3.1. <`pooling-data-source`> attribute settings

Attribute	Required	Default	Description
`url`	Yes		The url to be used for connecting to the database. Can be provided in a property file. Alternate property name is URL
`username`	No		The user name used to connect. Can be provided in a property file. Alternate property name is `user`
`password`	No		The password used to connect. Can be provided in a property file.
`connection-caching-enabled`	No	`true`	Is connection caching enabled?
`fast-connection-failover-enabled`	No	`false`	Is the fast connection failover feature enabled?
`ONS-configuration`	No		The ONS configuration string.
`properties-location`	No		The location of a properties file containing key-value pairs for the connection and connection

Attribute	Required	Default	Description
			cache environment using a specific prefix to separate connection cache properties from connection properties (in standard Properties format, namely 'key=value' pairs). If no location is specified a properties file located at `classpath:orcl.properties` will be used if found.
`connection-properties-prefix`	No		The prefix that is used for connection properties provided in the property file.
`connection-cache-properties-prefix`	No		The prefix that is used for connection cache properties provided in the property file.

The `pooling-data-source` element has the following child elements:

Table 3.2. <`pooling-data-source`> child elements

Element	Description
`connection-properties`	The newline-separated, key-value pairs for the connection properties (in standard Properties format, namely 'key=value' pairs)
`connection-cache-properties`	The newline-separated, key-value pairs for the connection-cache-properties (in standard Properties format, namely 'key=value' pairs)
`username-connection-proxy`	The configuration of a proxy authentication using a connection context provider

4. Fast Connection Failover

Oracle's RAC (Real Application Clusters) is an option that supports deployment of a single database across a cluster of servers, providing fault tolerance from hardware failures or other outages. Since a single database is served by a number of nodes, any node failure can be detected and subsequent operations can be directed to other nodes in the cluster. This support is provided by the "Fast Connection Failover" feature (FCF). When the failover occurs the current transaction is rolled back and a new transaction has to be initiated.

Spring's FCF support detects the transaction failure and attempts to retry the entire transaction. If this retry is successful it means that the client of the failed application will be unaware of this failover and it will look like the transaction completed after a brief delay.

The configuration for the FCF support is a two step configuration. First you need to configure a `DataSource` for RAC and second you need to configure an AOP advisor with a failover interceptor to handle the retries.

4.1 DataSource Configuration

We are going to need a DataSource that is capable of participating in a "Fast Connection Failover" scenario. The only one we have available is the `oracle.jdbc.pool.OracleDataSource` implementation that we will configure using the "orcl" namespace. This `DataSource` configured with some additional properties used for RAC.

We will be using the following property file to specify the username and password for the following example.

```
username=spring
password=spring
```

The url used in this example is a two node RAC configuration using the thin driver. It is probably too long to fit on the screen or on the page so if you would like to see the entire url it's listed in the callout notes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="racDataSource"
        url="jdbc:oracle:thin:@(description=(address_list=
            (address=(host=rac1)(protocol=tcp)(port=1521))
            (address=(host=rac2)(protocol=tcp)(port=1521)))
            (connect_data=(service_name=racdb1)))"
        properties-location="classpath:orcl.properties"
        fast-connection-failover-enabled="true" ❶
        ONS-configuration="rac1:6200,rac2:6200"/> ❷

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="racDataSource"/>
    </bean>

</beans>
```

❶ The fast connection failover is enabled here.

❷ The ONS (Oracle Notification Service) configuration is defined here since we are using the thin driver for this example.

4.2 AOP Configuration for Fast Connection Failover Retry

In order for the Fast Connection Failover to be transparent to the end user you need to consider the overall impact of this failover. The original transaction will fail and another transaction will be started to retry the same operation. You need to consider any non-transactional side effects that the failed transaction might have caused. You also need to consider work done while the transaction is suspended. This could happen if a method with a transactional attribute of "REQUIRES_NEW" is executed within the original transaction.

Once you have considered any possible side effects, you can proceed to configure a RacFailoverInterceptor together with the AOP advisor and pointcut. The failover advisor must be before or at the same pointcut where the transaction advisor is applied. If the pointcuts for the failover advisor and the transaction advisor are at the same pointcut then the failover advisor must have a higher priority than the transaction advisor that it should wrap.

For the AOP advisor configuration we use the "aop" namespace and for the `RacFailoverInterceptor` we use the `rac-failover-interceptor` tag from the "orcl" namespace.

Configuration when defining transactions using a <tx:advice> and an <aop:advisor>

When using a `<tx:advice>` combined with an `<aop:advisor>` you simply add an additional `<aop:advisor>` for the RAC failover Interceptor referencing the `<orcl:rac-failover-`

`interceptor>` element. You must make sure that the RAC failover interceptor comes before the transaction advice and you can do that by specifying the order attribute on the advisor for the RAC failover interceptor. Any advisor specified without an order automatically gets the lowest priority, so by specifying `order="1"` for the RAC failover interceptor we are assured this advice will come before the transaction advice.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <aop:config>
        <aop:advisor pointcut="execution(* *..PetStoreFacade.insertOrder(..))" ❶
            advice-ref="racFailoverInterceptor" order="1"/>
        <aop:advisor pointcut="execution(* *..PetStoreFacade.*(..))" ❷
            advice-ref="txAdvice"/>
    </aop:config>

    <orcl:rac-failover-interceptor id="racFailoverInterceptor"/> ❸

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="insert*"/>
            <tx:method name="update*"/>
            <tx:method name="*" read-only="true"/>
        </tx:attributes>
    </tx:advice>

</beans>
```

❶ The advisor defined for the RAC failover interceptor. This must have a higher order than the transaction advisor. We do use the same pointcut

❷ The standard transaction advice defined here.

❸ The RAC failover interceptor is defined using the `rac-failover-interceptor` element of the "orcl" namespace.

Configuration when defining transactions using @Transactional annotation

When using a <tx:annotation-driven> configuration you add `<aop:config>` entry with an `<aop:advisor>` element for the RAC failover Interceptor referencing the `<orcl:rac-failover-interceptor>` element. You must make sure that the RAC failover interceptor comes before the transaction advice and you can do that by specifying the order attribute on the advisor for the RAC failover interceptor. Any <tx:annotation-driven> specified without an order automatically gets the lowest priority, so by specifying `order="1"` for the RAC failover interceptor we are assured this advice will come before the transaction advice.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <aop:config>
        <aop:advisor ❶

 pointcut="@annotation(org.springframework.transaction.annotation.Transactional)"
            advice-ref="racFailoverInterceptor" order="1"/>
    </aop:config>

    <orcl:rac-failover-interceptor id="racFailoverInterceptor"/> ❷

    <tx:annotation-driven/> ❸

</beans>
```

❸ The advisor defined for the RAC failover interceptor. This must have a higher order than the transaction advisor. We use an `@annotation` pointcut referencing the `@Transactional` annotation.

❶ The RAC failover interceptor is defined using the `rac-failover-interceptor` element of the "orcl" namespace.

fcf.1.adoc.standard??? The standard transaction annotation-driven element defined here.

4.3 Configuration options for <rac-failover-interceptor>

There is a number of optional attributes you can use to configure the `rac-failover-interceptor`.

Table 4.1. <`rac-failover-interceptor`> attribute settings

Attribute	Required	Default	Description
`recoverable-error-codes`	No	3113, 3114, 1033, 1034, 1089, 17002, 17008, 17410	A comma separated list of integer error codes that will be used instead of the default set.
`max-number-of-retries`	No	5	The maximum number of times the retry will be performed.
`back-off-policy`	No	`NoBackOffPolicy`	A specific back off policy to be used. This is a reference to a bean that implements `BackOffPolicy` *

* `org.springframework.batch.retry.backoff.BackOffPolicy`

5. Oracle's Streams AQ (Advanced Queueing)

Oracle Streams is a feature that enables the propagation and management of data, transactions and events in a data stream either within a database, or from one database to another. This can be used both for replication and for messaging purposes. The Advanced Queuing (AQ) feature provides the messaging support. This messaging support will integrate with the standard JMS API provided with Java. Since the AQ support runs in the database it is possible to use the same transaction for both messaging and database access. This eliminates the need for expensive 2-phase commit processing that would be necessary when integrating database access with a traditional JMS solution.

Most of the JMS support we discuss in this chapter is provided directly by the Spring Framework. See the *Spring Framework Reference Documentation* for the details regarding this JMS support.

In addition to this standard support, The Advance Pack for Oracle Database provides easier configuration of a connection factory using the `<orcl>` namespace. It also provides support for some payload types not directly supported by the Spring JMS support like the `XMLType` and custom Advanced Data Types.

5.1 Supported payload types

JMS and Oracle Streams AQ can support a variety of payloads. These payloads are stored in the database and need to be converted to a Java representation in order for our programs to manipulate them. The following table outlines what payloads are supported and the corresponding classes that will work with these payloads and that will be able to convert them to and from a Java representation.

Table 5.1. supported payload types

Payload Type	Support Notes
`SYS.AQ $_JMS_TEXT_MESSAGE`, `SYS.AQ $_JMS_MAP_MESSAGE`, `SYS.AQ $_JMS_OBJECT_MESSAGE`, `SYS.AQ $_JMS_BYTES_MESSAGE`	Directly supported by `SimpleMessageConverter` which is the default for the `JmsTemplate` and the `DefaultMessageListenerContainer`. When configuring a message listener container the `DefaultMessageListenerContainer` is the class that supports the Oracle AQ JMS features.
`SYS.XMLType`	This payload type requires a custom message listener container named `XmlMessageListenerContainer`. This listener container also needs a `MessageListenerAdapter` with an Oracle AQ XML specific message converter specified as `XmlMessageConverter`. See below for configuration details.
`custom Advanced Data Type (ADT) (CREATE TYPE xxx AS OBJECT)`	This payload type requires a custom message listener container named `AdtMessageListenerContainer`. This listener container also can use a `MessageListenerAdapter` with a Oracle AQ ADT specific message converter specified as `MappingAdtMessageConverter`. This converter works with an implementation of the `DatumMapper` interface. See below for configuration details.

5.2 Configuration of the Connection Factory using the "orcl" namespace

When you use the JmsTemplate together with the Oracle AQ JMS support you can use the `aq-jms-connection-factory` entry to provide a connection factory to the JmsTemplate.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl" ❶
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd"> ❷


    <orcl:aq-jms-connection-factory id="connectionFactory"
            data-source="dataSource"/> ❸

    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
        <property name="sessionTransacted" value="true"/>
        <property name="connectionFactory" ref="connectionFactory"/>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" ... />

</beans>
```

❶ Here we specify the reference to the `orcl` schema.
❷ We also specify the location for the `orcl` schema.
❸ The connection factory is configured using a reference to the data source to be used.

The configuration for a Message-Driven POJO with a `MessageListenerContainer` is very similar. You use the same type of connection factory configuration. This is passed in to the listener container configuration. Here is an example using the JMS namespace support.

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl" ❶
       xmlns:jms="http://www.springframework.org/schema/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl ❷
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd
       http://www.springframework.org/schema/jms
       http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">

    <context:annotation-config/>

    <tx:annotation-driven/>

    <bean id="messageDelegate" class="spring.test.MessageDelegate"/>

    <jms:listener-container connection-factory="connectionFactory"
            transaction-manager="transactionManager">
        <jms:listener destination="jmsadmin.jms_text_queue"
                ref="messageDelegate" method="handleMessage"/>
    </jms:listener-container> ❸

    <orcl:aq-jms-connection-factory id="connectionFactory"
            data-source="dataSource"/> ❹

    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" ... />

</beans>
```

❶ Here we specify the reference to the `orcl` and `jms` schemas.
❷ We also specify the location for the `orcl` and `jms` schemas.
❹ The listener container is configured using a reference to the connection factory.
❹ The connection factory is configured using a reference to the data source to be used.

See the next section for how to configure the transaction support and use a the same local transaction as the JDBC or ORM data access.

5.3 Configuring the Connection Factory to use the same local transaction as your data access code.

The configurations in the previous section will take advantage of the transaction synchronization provided by Spring, but there will be two transactions. One transaction for the data access and one for the JMS messaging. They will be synchronized, so if the data access transaction commits then the

messaging transaction will also commit while if the data access transaction roll back then the messaging transaction will also roll back.

There is always a chance that the commit for the messaging transaction could fail after the data access transaction has committed successfully. This is of course a problem that you would have to account for in your code by checking for duplicate delivery of a message.

A better solution is to configure both data access and the messaging to share a transaction. Most often this is done using JTA, and that works, but has some impact on performance. For JTA you need to use distributed transactions and XA capable resources designed for two-phase commits. This comes at an extra cost that we try to avoid if possible.

Another option is to have the data access and the messaging share a local data access transaction. This is possible since the Oracle AQ implementation consists of a set of tables and stored procedures running in the database accessed through a standard JDBC connection. If you use the same database for data access and messaging with AQ, then you can configure the connection factory to share the database connection and the local transaction. You configure this connection and transaction sharing by setting the attribute `use-local-data-source-transaction` to `true`.

```
    <orcl:aq-jms-connection-factory id="connectionFactory"
        use-local-data-source-transaction="true" ❶
        data-source="dataSource"/>
```

❶ Setting the attribute `use-local-data-source-transaction`.

Configuring the connection factory to share a local data source transaction with the data access code has some implications for JMS connection and session caching. You can still configure a MessageListenerContainer to cache the JMS connection since each JMS session will be created as it's needed inside a data source transaction. However, if you cache the JMS session, then the database connection for it is established when the container starts up and it will not be possible to have this cached JMS session participite in the local data source transaction.

In many application server environments the JDBC connection is wrapped in an implementation specific class that delegates to the underlying native JDBC connection. Oracle's AQ connection factory needs the native Oracle connection and will throw an "oracle.jms.AQjmsException: JMS-112: Connection is invalid" exception if the connection is wrapped by a foreign class. To solve this problem you can specify a `NativeJdbcExtractor` that can be used to unwrap the connection. Spring provides a number of implementations to match the application server environment. Here is an example for specifying a `NativeJdbcExtractor`.

```
    <orcl:aq-jms-connection-factory id="connectionFactory"
        use-local-data-source-transaction="true"
        native-jdbc-extractor="dbcpNativeJdbcExtractor" ❶
        data-source="dataSource" />

    <bean id="dbcpNativeJdbcExtractor"

  class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"/>

    <bean id="dbcpDataSource" class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>
```

❶ Here we specify the reference to the native JDBC extractor.

For some use cases the default plain ConnectionFactory does not work and you need to explicitly use a QueueConnectionFactory or a TopicConnectionFactory. To support this requirement it is possible to specify this using the `connection-factory-type` attribute. The default is `CONNECTION` but you can specify `QUEUE_CONNECTION` or `TOPIC_CONNECTION` instead. Here is an example for specifying the connection factory type.

```
    <orcl:aq-jms-connection-factory id="connectionFactory"
        use-local-data-source-transaction="true"
        connection-factory-type="QUEUE_CONNECTION" ❶
        data-source="dataSource" />

    <bean id="dbcpDataSource" class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>
```

❶ Here we specify the type of connection factory to be used.

5.4 Configuration when using a SYS.XMLType payload

When you use a SYS.XMLType as payload there a few additional configuration settings are needed.

Enqueuing XML messages

When enqueuing messages the JmsTemplate can be configured with a message converter. This message converter should be of a type `XmlMessageConverter` configured with a specific `XmlTypeHandler` that you would like to use. The following handlers are available:

Table 5.2. xml handlers

XML Handler	Usage
`StringXmlTypeHandler`	Handles converting XMLTypes values to and from String representation.

XML Handler	Usage
`DocumentXmlTypeHandler`	Handles converting XMLTypes values to and from Document representation.
`StreamXmlTypeHandler`	Handles converting XMLTypes values to and from an InputStream.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory"/> ❶
    <property name="messageConverter">
        <bean id="messageConverter"❷

 class="org.springframework.data.jdbc.jms.support.converter.oracle.XmlMessageConverter">
            <constructor-arg>

  <bean class="org.springframework.data.jdbc.support.oracle.StringXmlTypeHandler"/> ❸
            </constructor-arg>
        </bean>
    </property>
</bean>
```

❶ A reference to the configured connection factory.

❷ Declaration of an `XmlMessageConverter` to convert from `XMLType` to desired representation.

❸ Declaration of the specific `XmlTypeHandler` that should be used. In this case a `StringXmlTypeHandler`.

Once the JmsTemplate is configured the XML value can be sent using the `convertAndSend` method. In this example we are passing in a String containing the value.

```
        String xmlval = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
                "<product id=\"10\">\n" +
                " <description>Foo</description>\n" +
                " <price>2.05</price>\n" +
                "</product>";

        jmsTemplate.convertAndSend("jmsadmin.jms_xml_queue", xmlval);
```

Dequeuing XML messages

When you want to dequeue messages using a message listener container you need to configure an `XmlMessageListenerContainer` that can dequeue the messages and convert the `XMLType` payload.

```
<bean id="messageDelegate" class="org.springframework.data.jdbc.test.xml.MessageDelegate"
 />

<jms:listener-container connection-factory="connectionFactory" ❶
    transaction-manager="transactionManager"
    message-converter="messageConverter"
    container-
class="org.springframework.data.jdbc.jms.listener.oracle.XmlMessageListenerContainer"> ❷
    <jms:listener destination="jmsadmin.jms_xml_queue"
        ref="messageDelegate" method="handleMessage">
    </jms:listener>
</jms:listener-container>

<bean id="messageConverter" ❸
   class="org.springframework.data.jdbc.jms.support.converter.oracle.XmlMessageConverter">
    <constructor-arg>
        <bean class="org.springframework.data.jdbc.support.oracle.DocumentXmlTypeHandler"/
> ❹
    </constructor-arg>
</bean>
```

❶ A reference to the configured connection factory.

❷ Configuring the class to use for the container - this is a custom class `XmlMessageListenerContainer` that dequeues the Oracle XMLType messages.

❸ The `XmlMessageConverter` is defined here.

❹ The `DocumentXmlTypeHandler` is used to retrieve XML value as a Document.

Here is an example of the message delegate used in the above message listener container:

```
public class MessageDelegate {

    @Autowired
    private DomainService domainService;

    public void handleMessage(Document xmlDoc)
            throws MessageConversionException, JMSException {
        domainService.processXmlMessage(xmlDoc);
    }

}
```

As you can see the method that handles the message takes a `Document` as its parameter. The conversion from the XMLType to a Document representation is handled by the `MessageListenerAdapter` since we specified a message converter.

5.5 Configuration when using a custom ADT payload

When you use a custom ADT as payload there are certain configuration settings that are needed. When creating the queue and its queue table you specify the custom type as the "queue_payload_type". This custom type is defined using a regular "CREATE TYPE" statement. In the code example that follow we have defined a PRODUCT type:

```
create or replace TYPE PRODUCT_TYPE AS OBJECT
(
  id INTEGER,
  description VARCHAR(50),
  price DECIMAL(12,2)
);
```

Enqueuing ADT messages

When enqueuing messages the JmsTemplate can be configured with a message converter. This message converter should be of a type `MappingAdtMessageConverter` configured with a specific `DatumMapper` that you would like to use. This `DatumMapper` can be a custom implementation or the provided `StructDatumMapper` that will map between bean properties and STRUCT attributes of the same name.

The `DatumMapper` interface has the following methods declared:

```
public interface DatumMapper {

    public Datum toDatum(Object object, Connection conn) throws SQLException;

    public Object fromDatum(Datum datum) throws SQLException;

}
```

The `toDatum` method will be called with the Object to convert to a STRUCT as the first parameter and the current connection as the second. It's up to the mapping implementation to extract the object properties and to create the STRUCT. For the `fromDatum` method the STRUCT is passed in and the implementation is responsible for retrieving the attributes and constructing and instance of the required class.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory"/> ❶
  <property name="messageConverter">
    <bean id="messageConverter"

  class="org.springframework.data.jdbc.jms.support.converter.oracle.MappingAdtMessageConverter">❷
       <constructor-arg>

  <bean class="org.springframework.data.jdbc.jms.support.oracle.StructDatumMapper"> ❸
           <constructor-arg index="0" value="JMSADMIN.PRODUCT_TYPE"/>
           <constructor-
arg index="1" value="org.springframework.data.jdbc.test.domain.Product"/>
         </bean>
       </constructor-arg>
    </bean>
  </property>
</bean>
```

❶ A reference to the configured connection factory.

❷ Declaration of an `MappingAdtMessageConverter` to convert from custom type to corresponding JavaBean.

❸ Declaration of the specific `DatumMapper` that should be used. In this case the provided `StructDatumMapper`.

Once the JmsTemplate is configured the XML value can be sent using the `convertAndSend` method. In this example we are passing in a String containing the value.

```
        Product product = new Product();
        product.setId(22L);
        product.setDescription("Foo");
        product.setPrice(new BigDecimal("42.95"));

        jms.convertAndSend("jmsadmin.jms_product_queue", product);
```

Dequeuing ADT messages

When you want to dequeue messages using a message listener container you need to configure an `AdtMessageListenerContainer` that can dequeue the messages and convert the ADT payload.

```
<bean id="messageDelegate" class="org.springframework.data.jdbc.test.adt.MessageDelegate"
 />

<jms:listener-container connection-factory="connectionFactory" ❶
    transaction-manager="transactionManager"
    message-converter="messageConverter"
    container-
class="org.springframework.data.jdbc.jms.listener.oracle.AdtMessageListenerContainer"> ❷
  <jms:listener destination="jmsadmin.jms_product_queue"
      ref="messageDelegate" method="handleMessage">
  </jms:listener>
</jms:listener-container>

<bean id="messageConverter"

  class="org.springframework.data.jdbc.jms.support.converter.oracle.MappingAdtMessageConverter">❸
  <constructor-arg>
    <bean class="org.springframework.data.jdbc.jms.support.oracle.StructDatumMapper"> ❹
      <constructor-arg index="0" value="JMSADMIN.PRODUCT_TYPE"/>
      <constructor-
arg index="1" value="org.springframework.data.jdbc.test.domain.Product"/>
    </bean>
  </constructor-arg>
</bean>
```

❶ A reference to the configured connection factory.

❶ Configuring the class to use for the container - this is a custom class `AdtMessageListenerContainer` that dequeues the ADT messages.

❸ The `MappingAdtMessageConverter` is defined here.

❹ The `StructDatumMapper` is used to map the attributes of the STRUCT retrieved for the ADT to properties of the bean class specified as the second constructor argument.

Here is an example of the message delegate used in the above message listener container:

```
public class MessageDelegate {

    @Autowired
    private DomainService domainService;

    public void handleMessage(Product product)
            throws MessageConversionException, JMSException {
        domainService.saveProduct(product);
    }

}
```

As you can see the method that handles the message takes a `Product` as its parameter. The conversion from the STRUCT to a Product is handled by the `MessageListenerAdapter` since we specified a message converter.

6. XML Types

Oracle has some advanced XML Type support built into the database. XML data stored in the database is accessible wia JDBC using some Oracle specific classes.

The JDBC framework provided with the Spring Framework supports most of this already via SqlTypeValue and SqlReturnType. There is however a need for documentation and examples which are specifically targeted for an Oracle environment so teams can take advantage of this support and have a reference for best practices for the use of these features in an Oracle/Spring environment.

6.1 Dependencies

To use the Oracle XML support you need to use a couple of jar files available in the *Oracle XML Developers Kit* download available from Oracle. You need `xdb.jar` and also the `xmlparserv2.jar` since the XMLType depends on this parser library.

There is optional support for Spring's Object/XML Mapping (OXM) support. If you use this support then you would also need a dependency for the Spring Framework OXM sub-project. The jar files needed is `spring-oxm.jar`.

All samples in this chapter access a table named "xml_table". Here is the DDL to create this table:

```
CREATE TABLE xml_table (
    id NUMBER(10),
    xml_text XMLTYPE,
    PRIMARY KEY (id));
```

6.2 Writing XML to an XMLTYPE column

To write XML data to a table you need to pass in the XML using a custom `SqlTypeValue`. In this implementation you would be responsible for setting the parameter value for the XML column in accordance with the API provided by the database driver.

For Oracle we provide a database specific implementation of an `SqlXmlValue`, which is an extension of the `SqlTypeValue`, that is easier to use. It works together with an `SqlXmlHandler` and adds an abstraction layer on top of the database specific APIs provided by the database vendors. There is a new `SQLXML` datatype in JDBC 4.0 that provides an abstraction, but so far it is not widely implemented.

In this example we have an XML value that we pass in as the second parameter. This XML value can be in the form of a `String` or an `org.w3c.dom.Document`. We use an `SqlXmlHandler` instance to gain access to a new instance of the `SqlXmlValue`. For the Oracle support the implementation classes are `OracleXmlHandler` and `OracleXmlTypeValue` respectively.

```
    simpleJdbcTemplate.update(
            "INSERT INTO xml_table (id, xml_text) VALUES (?, ?)",
            id,
            sqlXmlHandler.newSqlXmlValue(xml));❶
```

❶ We instantiate a new `SqlXmlValue` that will handle setting the parameter value for the XML.

The implementation of the SqlXmlHandler is chosen in the data access configuration file and should be injected into the DAO or Repository class.

```
<bean id="sqlXmlHandler"
    class="org.springframework.data.jdbc.support.oracle.OracleXmlHandler"/>
```

Oracle's `XMLType` supports passing in an `java.io.InputStream` but since this is not supported by the JDBC 4.0 `SQLXML` datatype you will have to use the Oracle specific `OracleXmlTypeValue` directly.

```
    simpleJdbcTemplate.update(
            "INSERT INTO xml_table (id, xml_text) VALUES (?, ?)",
            id,
            new OracleXmlTypeValue(is));
```

6.3 Reading XML from an XMLTYPE column

Running a query against a table with an XMLTYPE column requires a RowMapper that can handle retrieval of the XMLType and the corrsponding XML data. The `OracleXmlHandler` provides several methods that supports easy access to the XML data. It is typically used in a RowMapper.

```
    String s = simpleJdbcTemplate.queryForObject(
            "SELECT xml_text FROM xml_table WHERE id = ?",
            new ParameterizedRowMapper<String>() {
                public String mapRow(ResultSet rs, int i) throws SQLException {
                    String s = sqlXmlHandler.getXmlAsString(rs, 1);❶
                    return s;
                }
            },
            id);
```

❶ We use the `OracleXmlHandler` to retreive the XML value as a String.

The XML data can be retreived as a String, a java.io.InputStream, a java.io.Reader or a javax.xml.transform.Source.

6.4 Marshalling an object to an XMLTYPE column

To map an object to XML and write this XML to a table you first need to use marshalling support available from the Spring Web Services project. Once the object data is marshalled to XML we can write the XML to a column in a database table. The latter part is very similar to the the XML support discussed above. We need to pass in the XML using a custom `SqlTypeValue`. In the object mapping implementation you would be responsible for marshalling the object to XML before setting the parameter value.

In this example we have an object that needs to be marshalled to XML. We are using a `Marshaller` provided by the Spring Web Services project. The marshaller is typically configured and then injected into the DAO or Repository. Here is an example configuration using the JAXB 2.0 support. In addition to JAXB 2.0, there is also support for JAXB 1.0, Castor, XML Beans, JiBX and XStream.

```
    <bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>org.springframework.data.jdbc.samples.Item</value>
            </list>
        </property>
    </bean>
```

The JAXB 2.0 class that we are marshalling is a typical javaBean and it uses annotations for the meta data so there is no additional configuration needed.

```
package org.springframework.data.jdbc.samples;

import javax.xml.bind.annotation.*;
import java.math.BigDecimal;

@XmlRootElement(name = "item")
@XmlType(propOrder = {"name", "price"})
public class Item{
    private Long id = 0L;
    private String name;
    private BigDecimal price;

    @XmlAttribute(name="id")
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @XmlElement(name = "item-name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlElement(name = "price")
    public BigDecimal getPrice() {
        return price;
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
    }


    public String toString() {
        return "[" + id + "] " + name + " " + price;
    }
}
```

For Oracle we provide a database specific implementation of an `SqlXmlMarshallingValue`, which is an extension of the `SqlXmlValue`, that is easier to use. It works together with an `SqlXmlObjectMappingHandler` similar to the `SqlXmlHandler` that we used in the previous example. The object to be marshalled is passed in when the new instance of the `SqlXmlValue` is created.

For our dabase insert we pass in the marshalled value as the second parameter. The first parameter is the id of the object, and this will be use as the primary key for the row. We use an `SqlXmlHandler` instance to gain access to a new instance of the `SqlXmlMappingValue`. For the Oracle support the implementation classes are `OracleXmlObjectMappingHandler` and `OracleXmlMarshallingValue` respectively.

```
    simpleJdbcTemplate.update(
            "INSERT INTO xml_table (id, xml_text) VALUES (?, ?)",
            item.getId(),
            sqlXmlObjectMappingHandler
                    .newMarshallingSqlXmlValue(item));❶
```

❶ We instantiate a new marshalling `SqlXmlValue` that will handle mapping the parameter object to XML using a marshaller.

The implementation of the SqlXmlObjectMappingHandler is chosen in the data access configuration file and should be injected into the DAO or Repository class.

```
<bean id="sqlXmlHandler"
    class="org.springframework.data.jdbc.support.oracle.OracleXmlObjectMappingHandler">
    <property name="marshaller" ref="marshaller"/>
</bean>
```

6.5 Unmarshalling an object from an XMLTYPE column

Last piece we need is reading the XML from the database and have it unmarshalled to an Item object. We will perform this work in a `RowMapper` together with the `SqlXmlObjectMappingHandler`.

```
    Item i = simpleJdbcTemplate.queryForObject(
            "SELECT xml_text FROM xml_table WHERE id = ?",
            new ParameterizedRowMapper<Item>() {
                public Item mapRow(ResultSet rs, int i) throws SQLException {
                    return (Item) sqlXmlObjectMappingHandler
                            .getXmlAsObject(rs, 1);❶
                }
            },
            id);
```

❶ We use the `SqlXmlObjectMappingHandler` to retreive the XML value and have it unmarshalled to an Item instance.

The XML data is unsmarshalled using an `Unmarshaller` which in the JAXB 2.0 case is also implemented by the `Jaxb2Marshaller` class. It must be injected into the `unmarshaller` property of the SqlXmlObjectMappingHandler. Since marshalling and unmarshalling is performed by the same object we pass in the bean named `marshaller` for the `unmarshaller` property.

```
<bean id="sqlXmlHandler"
    class="org.springframework.data.jdbc.support.oracle.OracleXmlObjectMappingHandler">
    <property name="unmarshaller" ref="marshaller"/>
</bean>
```

7. Advanced Data Types

The Oracle database and the PL/SQL language used for stored procedures in Oracle has built in support for some advanced data types. These data types can't easily be accessed using standard JDBC APIs, so it is necessary to rely on Oracle's JDBC extensions like ARRAY and STRUCT and the APIs that are used to access them.

The JDBC framework provided with the Spring Framework supports most of this already via `SqlTypeValue` and `SqlReturnType`. The `SqlTypeValue` interface is used to pass IN parameter values. This is easiest accomplished by extending the `AbstractSqlTypeValue` class. Here you need to implement the createTypeValue method. In this method you have access to the current connection, the `SqlType` and the type name for any custom processing that is necessary. When you retrieve advanced data types you need to implement the `SqlReturnType` interface and pass that implementation into the `SqlOutParameter` constructor. The `SqlReturnType` interface has one method named `getTypeValue` that must be implemented. Here you have access to the CallableStatement that is currently executing as well as the parameterIndex, the `SqlType` and the type name for customizing the processing.

When implementing these interfaces there is some boilerplate type code that is necessary and it makes your data access code look unnecessarily complex. That is the reason why we are providing a number of Oracle specific implementations that can handle the Oracle advanced types. The usage of these type handlers is documented in this chapter. These features are specifically targeted for an Oracle environment so teams can take advantage of this support and have a reference for best practices for the use of these features in an Oracle/Spring environment.

We will use the following simple table for all the examples in this chapter. This table is used to store some basic information about actors.

```
CREATE TABLE actor (
    id NUMBER(10),
    name VARCHAR2(50),
    age NUMBER,
    PRIMARY KEY (id));
```

7.1 Using a STRUCT parameter

When your stored procedures has parameters that are declared using custom object types that aren't part of the standard JDBC types they are managed using JDBC `Struct` objects. When working with Oracle it's easier to work with Oracle's extension to `Struct` which is `oracle.sql.STRUCT`.

For the `STRUCT` examples we will use the following type declaration.

```
CREATE OR REPLACE TYPE actor_type
    AS OBJECT (id NUMBER(10), name VARCHAR2(50), age NUMBER);
```

The data contained in a STRUCT parameter can be accessed in two ways. Either using the `SQLData` interface which is part of the JDBC specification, or by using Oracle specific calls accessing the attributes directly. We will cover both methods.

Now we will look at the sample procedures used for this example. First one is the procedure we use to add the actor data.

```
CREATE OR REPLACE PROCEDURE add_actor (in_actor IN actor_type)
AS
BEGIN
  INSERT into actor (id, name, age) VALUES(in_actor.id, in_actor.name, in_actor.age);
END;
```

This procedure has one IN parameter (in_actor) of object type `actor_type`.

Next we show the procedure used to retrieve the actor data.

```
CREATE OR REPLACE PROCEDURE get_actor (in_actor_id IN NUMBER, out_actor OUT actor_type)
AS
BEGIN
  SELECT actor_type(id, name, age) INTO out_actor FROM actor WHERE id = in_actor_id;
END;
```

This procedure has two parameters, one IN parameter (`in_actor_id`) that is the id of the actor to retrieve and one OUT parameter (`out_actor`) of type `actor_type` to pass back the data retrieved.

The last piece we will cover here is the Java class that represents the type we are accessing. Here is the `Actor` implementation used in this example. It has the Java equivalent of the variables we defined for the type in the database. We also have setters and getters for all fields.

```
package org.springframework.data.jdbc.samples;

import java.sql.SQLData;
import java.sql.SQLException;
import java.sql.SQLInput;
import java.sql.SQLOutput;
import java.math.BigDecimal;

public class Actor {

    private Long id;
    private String name;
    private int age;


    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String toString() {
    return "Actor: [" + id + "] " + name + " " + age;
    }

}
```

Using an SQLData implementation for a STRUCT IN parameter

For the examples that use `SQLData` we first need a Java class that implements the SQLData interface. For this example we create an `SqlActor` class that extends the `Actor` class shown earlier and provides the `SQLData` implementation for our `ACTOR_TYPE`.

```
package org.springframework.data.jdbc.samples;

import java.sql.SQLData;
import java.sql.SQLException;
import java.sql.SQLInput;
import java.sql.SQLOutput;
import java.math.BigDecimal;

public class SqlActor extends Actor implements SQLData {

    public String getSQLTypeName() throws SQLException {
        return "ACTOR_TYPE"; ❶
    }

    public void readSQL(SQLInput sqlInput, String string) throws SQLException { ❷
        setId(Long.valueOf(sqlInput.readLong()));
        setName(sqlInput.readString());
        setAge(sqlInput.readInt());
    }

    public void writeSQL(SQLOutput sqlOutput) throws SQLException { ❸
        sqlOutput.writeLong(getId().longValue());
        sqlOutput.writeString(getName());
        sqlOutput.writeInt(getAge());
    }

}
```

❶ Here we specify the ACTOR_TYPE advanced data type as the type supported by this implementation. Note: Since Oracle's metadata is stored using all caps, unless the name was explicitly defined as mixed case using quotes, we define the type name here as all caps.

❷ Here we specify the implementation used to map data between the ACTOR_TYPE advanced data type and the `Actor` class during a read operation.

❸ Here we specify the implementation used to map data between the `Actor` class and the ACTOR_TYPE advanced data type during a write operation.

As you can see, the `SQLData` implementation is fairly straightforward. We implemented the three methods required in the `SQLData` interface. These methods are getSQLTypeName, readSQL and writeSQL.

Now we can move on to actually call the stored procedure. First example is using the newer `SimpleJdbcCall` API but the `SqlParameter` would be the same if you used the classic `StoredProcedure` approach. We create the `SimpleJdbcCall` and in the `declareParameters` method call we pass in an `SqlParameter` that defines the parameter with the type as OracleTypes.STRUCT and a type name of ACTOR_TYPE to match what it is defined in the database. Note that the type name is defined here as all caps since that is how it is stored in the database metadata. Any type names declared here are case sensitive and must match what is actually stored in the database metadata.

```
    this.addSqlActorCall =
            new SimpleJdbcCall(dataSource).withProcedureName("add_actor")
                .declareParameters(
                    new SqlParameter("in_actor", OracleTypes.STRUCT, "ACTOR_TYPE")); ❶
```

❶ We define the `SqlParameter` with parameter name, the Oracle type and the type name as it is declared in the database.

Next we look at the code that executes this `SimpleJdbcCall`.

```
    public void addSqlActor(final SqlActor actor) {
        Map in = Collections.singletonMap("in_actor", actor); ❶
        addSqlActorCall.execute(in);
    }
```

❶ We execute the call by passing in a `Map` containing any in parameters - in this case the actor object.

What happens here is that the JDBC driver access the data in the passed in Actor instance via the `SQLData` interface and maps that data to the ACTOR_TYPE passed in to the stored procedure. There is no need to implement anything else since this is all handled by the JDBC layer.

Please note that since `SimpleJdbcCall` is relying on database metadata, the parameter names used for the input must match the names used when declaring the stored procedure. They are however not case sensitive, only the type names are case sensitive.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class AddSqlActorProc extends StoredProcedure {

        public AddSqlActorProc(DataSource dataSource) {
            super(dataSource, "add_actor");
            declareParameter(new SqlParameter("in_actor",
OracleTypes.STRUCT, "ACTOR_TYPE"));
        }

        public void execute(Actor actor) {
            Map in = Collections.singletonMap("in_actor", actor);
            this.execute(in);
        }

    }
```

Using SqlReturnSqlData with an SQLData implementation from a STRUCT OUT parameter

Now we will call the stored procedure that retrieves actor data. We are still using the newer `SimpleJdbcCall` API. We create the `SimpleJdbcCall` and in the `declareParameters` call we pass in an `SqlOutParameter` that uses an `SqlReturnType` to handle the configuration necessary for the mapping between the Oracle type and the Java type which is still the `SqlActor`. We also need to link between the `Actor` class and the `ACTOR_TYPE` since the JDBC driver is not aware of this relationship when we are reading data from the database. This is done by declaring a SqlReturnSqlData class and passing in the target class in the constructor.

```
        this.getSqlActorCall =
                new SimpleJdbcCall(dataSource).withProcedureName("get_actor")
                    .declareParameters(
                        new SqlOutParameter("out_actor",
OracleTypes.STRUCT, "ACTOR_TYPE", ❶
                                new SqlReturnSqlData(SqlActor.class)) ❷
                    );
```

❶ We define the parameter name and the Oracle type and the type name as it is declared in the database.

❷ Here we define the `SqlReturnSqlData` and the desired target class.

Next we look at the code that executes this `SimpleJdbcCall`.

```
    public SqlActor getSqlActor(int id) {
        Map in = Collections.singletonMap("in_actor_id", id);
        return getSqlActorCall.executeObject(SqlActor.class, in); ❶
    }
```

❶ We execute the call by passing in a `Map` containing any in parameters. The `executeObject` method returns an `SqlActor` containing the data returned by the stored procedure call.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class GetSqlActorProc extends StoredProcedure {

        public GetSqlActorProc(DataSource dataSource) {
            super(dataSource, "get_actor");
            declareParameter(new SqlParameter("in_actor_id", Types.NUMERIC));
            declareParameter(
                new SqlOutParameter("out_actor", OracleTypes.STRUCT, "ACTOR_TYPE",
                    new SqlReturnSqlData(SqlActor.class))
            );
        }

        public SqlActor execute(Long id) {
            Map in = Collections.singletonMap("in_actor_id", id);
            Map out = this.execute(in);
            return (SqlActor) out.get("out_actor");
        }

    }
```

Setting STRUCT attribute values using SqlStructValue for an IN parameter

An alternate access technique is to use the `Struct` interface to access a generic collection of attributes representing the type. The `SqlStructValue` implementation will map properties in a JavaBean to the corresponding attributes of the STRUCT so there is no need to provide custom mapping code. The following example will perform the same operations as the previous example using this alternate technique.

The SimpleJdbcCall declaration for the "add_actor" call looks the same.

```
    this.addActorCall =
            new SimpleJdbcCall(dataSource).withProcedureName("add_actor")
                .declareParameters(
                    new SqlParameter("in_actor", OracleTypes.STRUCT, "ACTOR_TYPE")); ❶
```

❶ We define the `SqlParameter` with parameter name, the Oracle type and the type name as it is declared in the database.

Next we'll look at the code used to execute this procedure call. The difference is in the execution and the mapping of attributes. Instead of relying on the `SqlActor` class to do the mapping, we create a `SqlStructValue` and pass in the `Actor` instance in the constructor. The `SqlStructValue` class will do the mapping between the bean properties of the Actor class and the attributes of the STRUCT. This `SqlStructValue` is then passed in as the data value in the input map for the execute call.

```
    public void addActor(final Actor actor) {
        Map in = Collections.singletonMap("in_actor", new SqlStructValue(actor)); ❶
        addActorCall.execute(in); ❷
    }
```

❶ We create an `SqlStructValue` that will handle the type creation and mapping and add it to the `Map` containing the in parameters.
❷ We execute the call by passing in the input `Map`.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class AddActorProc extends StoredProcedure {

        public AddActorProc(DataSource dataSource) {
            super(dataSource, "add_actor");
            declareParameter(new SqlParameter("in_actor",
OracleTypes.STRUCT, "ACTOR_TYPE"));
        }

        public void execute(Actor actor) {
            Map in = Collections.singletonMap("in_actor", new SqlStructValue(actor));
            this.execute(in);
        }

    }
```

Using SqlReturnStruct to access STRUCT data from an OUT parameter

You can use the `SqlReturnStruct` class to map between the attributes of a `STRUCT` object and properties of a `JavaBean`. This is more convenient than providing this mapping yourself. This example will show how this can be done using an SqlOutParameter combined with the `SqlReturnStruct` class.

```
    this.getActorCall =
            new SimpleJdbcCall(dataSource).withProcedureName("get_actor")
                .declareParameters(
                    new SqlOutParameter("out_actor", OracleTypes.STRUCT, "ACTOR_TYPE", ❶
                            new SqlReturnStruct(Actor.class)) ❷
            );
```

❶ We define the `SqlParameter` with parameter name, the Oracle type and the type name as it is declared in the database.
❷ The `SqlReturnStruct` will retrieve the STRUCT and access the array of objects representing the attributes and then map them to the properties of the `JavaBean` instance provided n the constructor.

Next we look at the code that executes this `SimpleJdbcCall`.

```
    public Actor getActor(int id) {
        Map in = Collections.singletonMap("in_actor_id", id);
        return getActorCall.executeObject(Actor.class, in); ❶
    }
```

❶ We execute the call by passing in a `Map` containing any in parameters. The `executeObject` method returns an `Actor` containing the data returned by the stored procedure call.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class GetActorProc extends StoredProcedure {

        public GetActorProc(DataSource dataSource) {
            super(dataSource, "get_actor");
            declareParameter(new SqlParameter("in_actor_id", Types.NUMERIC));
            declareParameter(
                new SqlOutParameter("out_actor", OracleTypes.STRUCT, "ACTOR_TYPE",
                        new SqlReturnStruct(Actor.class))
            );
        }

        public Actor execute(Long id) {
            Map in = Collections.singletonMap("in_actor_id", id);
            Map out = this.execute(in);
            return (Actor) out.get("out_actor");
        }

    }
```

7.2 Using an ARRAY parameter

Sometimes your stored procedures has parameters that are declared as arrays of some type. These arrays are managed using JDBC `Array` objects. When working with Oracle it's sometimes easier to work with Oracle's extension to `Array` which is `oracle.sql.`ARRAY.

For the ARRAY examples we will use the following type declarations.

```
CREATE OR REPLACE TYPE actor_name_array
    AS VARRAY(20) OF VARCHAR2(50);
CREATE OR REPLACE TYPE actor_id_array
    AS VARRAY(20) OF NUMBER;
```

We will show how to access parameters using these declarations in two JDBC calls. The first one is a procedure call that deletes actor entries based on ids provided in an `actor_id_array`. The second example calls a function to retrieve an array of the names for all actors in the table.

Setting ARRAY values using SqlArrayValue for an IN parameter

We are using the SimpleJdbcCall for this example and when we configure this call its important to note that we can't rely on the database metadata. Whenever a collection type is used the metadata reported back from the JDBC driver contains entries bot for the collection type and for the type contained in the collection so it looks like there are additional parameters. Because of this it is best to turn off the metadata processing by calling the`withoutProcedureColumnMetaDataAccess` method.

This example calls a procedure that deletes actors based on the ids provided in an array. Here is the source for this procedure:

```
CREATE OR REPLACE PROCEDURE delete_actors (in_actor_ids IN actor_id_array)
AS
BEGIN
  FOR i IN 1..in_actor_ids.count loop
    DELETE FROM actor WHERE id = in_actor_ids(i);
  END LOOP;
END;
```

The declaration of the ARRAY parameter follows the same pattern as we used previously for the STRUCT parameters. We are simply providing the `OracleTypes.ARRAY` SQL type along with the type name as it is specified in the database metadata.

```
    this.deleteActorsCall =
            new SimpleJdbcCall(dataSource).withProcedureName("delete_actors")
                .withoutProcedureColumnMetaDataAccess()
                .declareParameters(
                    new SqlParameter("in_actor_ids",
OracleTypes.ARRAY, "ACTOR_ID_ARRAY")); ❶
```

❶ We define the `SqlParameter` with parameter name, the Oracle type and the type name as it is declared in the database.

Next we look at the code that executes this `SimpleJdbcCall`. For IN parameters the arrays are managed using an `SqlArrayValue` implementation that will handle the ArrayDescriptor creation and the mapping of the array to an oracle.sql.ARRAY instance.

```
    public void deleteActors(final Long[] ids) {
        Map in = Collections.singletonMap("in_actor_ids", new SqlArrayValue(ids)); ❶
        deleteActorsCall.execute(in);
    }
```

❶ We declare an `SqlArrayValue` instance that will handle creating the `ArrayDesriptor` and the ARRAY to be passed in as the parameter value.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class DeleteActorsProc extends StoredProcedure {

        public DeleteActorsProc(DataSource dataSource) {
            super(dataSource, "delete_actors");
            declareParameter(
                new SqlParameter("in_actor_ids", OracleTypes.ARRAY, "ACTOR_ID_ARRAY"));
        }

        public void execute(Long[] ids) {
            Map in = Collections.singletonMap("in_actor_ids", new SqlArrayValue(ids));
            Map out = this.execute(in);
        }

    }
```

Using SqlReturnArray to handle the ARRAY from an OUT parameter

Now it is time to handle the OUT parameter scenario. Here it is an SqlOutParameter combined with an SqlReturnArray instance that is responsible for handling the `Array`.

```
    this.getActorNamesCall =
            new SimpleJdbcCall(dataSource).withFunctionName("get_actor_names")
                .withoutProcedureColumnMetaDataAccess()
                .declareParameters(
                     new SqlOutParameter("return", Types.ARRAY, "ACTOR_NAME_ARRAY", ❶
                         new SqlReturnArray())); ❷
```

❶ We declare an `SqlOutParameter` with parameter name, the Oracle type and the type name as it is declared in the database.

❷ The `SqlReturnArray` accesses the ARRAY parameter using the JDBC calls and creates the `String` array that is the return value for this example.

Next we look at the code that executes this `SimpleJdbcCall`.

```
    public String[] getActorNames() {
        Map in = Collections.emptyMap();
        return getActorNamesCall.executeFunction(String[].class, in); ❶
    }
```

❶ Here we just have to call executeFunction passing in the expected output class and an empty map since there are no IN parameters.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class GetActorNamesProc extends StoredProcedure {

        public GetActorNamesProc(DataSource dataSource) {
            super(dataSource, "get_actor_names");
            setFunction(true);
            declareParameter(new SqlOutParameter("return",
 Types.ARRAY, "ACTOR_NAME_ARRAY",
                    new SqlReturnArray()));
        }

        public String[] execute() {
            Map in = Collections.emptyMap();
            Map out = this.execute(in);
            return (String[]) out.get("return");
        }

    }
```

7.3 Handling a REF CURSOR

The Spring Framework already contains implementations that simplify the handling of REF CURSORS but we include an example here just to complete the coverage of the handling of Oracle specific advanced data types. The procedure we are calling is declared as follows:

```
CREATE OR REPLACE PROCEDURE read_actors (out_actors_cur OUT sys_refcursor)
AS
BEGIN
  OPEN out_actors_cur FOR 'select * from actor';
END;
```

Retrieving data using a ParameterizedBeanPropertyRowMapper from a REF CURSOR

First we'll look at a `SimpleJdbcCall` implementation where we use the `returningResultSet` method to declare the `RowMapper` we want to use. We have an Actor class that is a JavaBean and the properties match the column names so we can use the ParameterizedBeanPropertyRowMapper to automatically map data from the `ResultSet` to the bean properties. Here is the code used to declare this `SimpleJdbcCall`:

```
    this.readActorsCall =
            new SimpleJdbcCall(dataSource).withProcedureName("read_actors")
                .returningResultSet("out_actors_cur", ❶
                        ParameterizedBeanPropertyRowMapper.newInstance(Actor.class)); ❷
```

❶ We declare a `returningResultSet` with parameter name and the `RowMapper` we would like to use.

❷ The `ParameterizedBeanPropertyRowMapper` accesses the `ResultSetMetaData` and maps the row columns to corresponding bean properties in the class specified as parameter to the `newInstace` method call.

To execute this call we use the following code:

```
    public List<Actor> getActors() {
        return readActorsCall.executeObject(List.class, Collections.emptyMap()); ❶
    }
```

❶ Here we just have to call executeObject passing in the expected output class which is a `List` and an empty map since there are no IN parameters.

When using the StoredProcedure class we would need to use an `SqlOutParameter` that accepts a `RowMapper`. Here is an example of an `SqlOutParameter` configured with a `ParameterizedBeanPropertyRowMapper`.

```
        new SqlOutParameter("out_actors_cur", OracleTypes.CURSOR,
                ParameterizedBeanPropertyRowMapper.newInstance(Actor.class)) ❶
```

❶ Here we specify the parameter name and the SQL type which is `OracleTypes.CURSOR` and instantiate a `ParameterizedBeanPropertyRowMapper` to be used to map row data to the `Actor` class.

If you prefer to use the classic StoredProcedure class then the equivalent configuration would look like this:

```
    private class ReadActorsProc extends StoredProcedure {

        public ReadActorsProc(DataSource dataSource) {
            super(dataSource, "read_actors");
            declareParameter(
                new SqlOutParameter("out_actors_cur", OracleTypes.CURSOR,
                        ParameterizedBeanPropertyRowMapper.newInstance(Actor.class))
            );

        }

        public List execute() {
            Map in = Collections.emptyMap();
            Map out = this.execute(in);
            return (List) out.get("out_actors_cur");
        }

    }
```

8. Custom DataSource Connection Configurations

8.1 Configuration of a Proxy Authentication

The Oracle JDBC implementation provides access to Oracle's Proxy Authentication feature. The Proxy Authentication lets you configure a connection pool using a proxy user account with very limited rights. Then, during the connection process, you would specify the actual user name for the end user. This user name must be configured to allow a proxy connection through the user proxy ("grant connect through proxyuser").

This is valuable for web applications where you typically set up a data source with a shared database user. If this shared user is a proxy user account and you supply the actual end user name then the proxy authentication feature will make any database access this user performs to be performed with the end users actual database user account.

To use this feature you must provide an implementation of the `ConnectionUsernameProvider` interface. This interface has a single method named `getUserName` that should return the user name for the current end user to be connected via the proxy user. It's up to the application developer to provide the appropriate implementation. One type of implementation would be to retrieve the current principal or user name from the `SecurityContextHolder` provided when you use Spring Security.

An example of what this implementation could look like is:

```
public class CurrentUsernameProvider implements ConnectionUsernameProvider {

    public String getUserName() {
        Object principal =
            SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        if (principal instanceof UserDetails) {
            return ((UserDetails)principal).getUsername();
        } else {
            return principal.toString();
        }
    }

}
```

See the Spring Security reference manual for more detail regarding the use of the `SecurityContextHolder`.

Connection proxy authentication is configured using the `username-connection-proxy` element. You also need to provide a reference to the user name provider that implements the `ConnectionUsernameProvider` interface mentioned above.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:orcl="http://www.springframework.org/schema/data/orcl"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/data/orcl
       http://www.springframework.org/schema/data/orcl/spring-data-orcl-1.0.xsd">

    <orcl:pooling-datasource id="dataSource"
        properties-location="classpath:orcl.properties">
        <orcl:username-connection-proxy connection-context-provider="usernameProvider"/> ❶
    </orcl:pooling-datasource>

    <bean id="usernameProvider"
      class="org.springframework.data.jdbc.test.CurrentUsernameProvider"/>

</beans>
```

❶ The connection proxy user name provider is specified here.

To set up the database proxy user and to grant the user accounts to participate in the proxy authentication you could use this SQL:

```
-- create the new proxy user account
create user proxyuser identified by proxypasswd;
grant create session to proxyuser;
-- grant existing user to connect  through the proxy
alter user spring grant connect through proxyuser;
```

In your connection properties file (orcl.properties) you would need to provide the proxy user credentials:

```
url=jdbc:oracle:thin:@//localhost:1521/xe
username=proxyuser
password=proxypasswd
```

Note

We are currently only supporting proxy authentication using user name with no password authentication for the user connecting through the proxy. Support for other types of proxy connections will be provided in future releases.

8.2 Configuration of a Custom DataSource Connection Preparer

There are times when you want to prepare the database connection in certain ways that aren't easily supported using standard connection properties. One example would be to set certain session properties in the SYS_CONTEXT like MODULE or CLIENT_IDENTIFIER. This chapter explains how to use a `ConnectionPreparer` to accomplish this. The example will set the CLIENT_IDENTIFIER.

We will need to add a `ConnectionInterceptor` using AOP and then configure the `ConnectionInterceptor` with a `ConnectionPreparer` implementation that performs the necessary preparations. Lets first look at our custom `ClientIdentifierConnectionPreparer` that implements the `ConnectionPreparer` interface. There is only a single method named `prepare` that

needs to be implemented. The prepared connection is the return value which gives you an opportunity to wrap the connection with a proxy class if needed.

```
package org.springframework.data.jdbc.samples;

import org.springframework.data.jdbc.support.ConnectionPreparer;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;

public class ClientIdentifierConnectionPreparer implements ConnectionPreparer {

    String prepSql = "{ call DBMS_SESSION.SET_IDENTIFIER('SPRING') }"; ❶

    public Connection prepare(Connection conn) throws SQLException {
        CallableStatement cs = conn.prepareCall(prepSql); ❷
        cs.execute();
        cs.close();
        return conn;
    }

}
```

❶ We define the SQL needed to set the CLIENT_IDENTIFIER attribute.

❷ We prepare a `CallableStatement` and execute it.

This example sets the CLIENT_IDENTIFIER to a fixed value, but you could implement a ConnectionPreparer that would use the current users login id. That way you can capture user login information even if your data source is configured with a shared user name.

The following application context entries show how this could be configured for your data source.

```
    <orcl:pooling-datasource id="dataSource" ❶
        connection-properties-prefix="conn"
        properties-location="classpath:orcl.properties"/>

    <aop:config> ❷
        <aop:advisor
            pointcut="execution(java.sql.Connection
 javax.sql.DataSource.getConnection(..))"
            advice-ref="testInterceptor"/>
    </aop:config>

    <bean id="testInterceptor"
          class="org.springframework.data.jdbc.aop.ConnectionInterceptor">
        <property name="connectionPreparer" ref="connectionPreparer"/> ❸
    </bean>

    <bean id="connectionPreparer"
        class="org.springframework.data.jdbc.samples.ClientIdentifierConnectionPreparer"/
> ❹
```

❶ The regular dataSource definition, no extra configuration needed here.

❷ The AOP configuration defining the pointcut as the `getConnection` method.

❸ The interceptor that has its `connectionPreparer` property set to our custom `ClientIdentifierConnectionPreparer`.

❶ A bean defining the custom `ClientIdentifierConnectionPreparer`.

Every time a new connection is obtained the connection preparer will set the CLIENT_IDENTIFIER. During database processing the value it was set to can be accessed using a call to a standard Oracle function - "`sys_context('USERENV', 'CLIENT_IDENTIFIER')`"