EclipseLink

Developing Persistence Architectures
Using EclipseLink Database Web Services

Release 2.4

July 2012

Java Persistence API (JPA) Extensions Reference for EclipseLink, Release 2.4

Copyright © 2012, by The Eclipse Foundation under the Eclipse Public License (EPL).

http://www.eclipse.org/org/documents/epl-v10.php

The initial contribution of this content was based on work copyrighted by Oracle and was submitted with permission.

Print date: January 28, 2013

Contents

Preface v

Audience v
Related Documents v
Conventions v

1 Overview of EclipseLink's DBWS Implementation

Understanding XML-to-Relational Mapping (XRM) 1-2
Configuration 1-2
XML Schema Definition 1-4
Creating EclipseLink DBWS Services 1-5
Creating EclipseLink DBWS Services Using the DBWSBuilder Utility 1-6
Creating an EclipseLink DBWS Service from a Database Table 1-7
Creating an EclipseLink DBWS Service from a SQL Statement 1-7
Using Parameter Binding 1-8
Creating an EclipseLink DBWS Service from a Stored Procedure 1-8
Customizing an EclipseLink DBWS Service 1-9
Performing Simple Customization 1-9
Performing Intermediate Customization 1-10
Performing Advanced Customization 1-10
Using DBWSBuilder API 1-11
Using the DBWS Design Time Component 1-13

2 Creating DBWS Services

Creating EclipseLink DBWS Service from a Database Table 2-1
Generated EclipseLink DBWS service descriptor 2-3
SOAP Messaging 2-4
Using an EclipseLink SessionCustomizer 2-5
Example 2-5
Advanced Customization 2-6
Using Existing EclipseLink ORM and OXM Mappings 2-7
Creating a DBWS Service from SQL Statements 2-7
Creating from Results Sets from Custom SQL SELECT Statements 2-7
Generated EclipseLink DBWS Service Descriptor 2-9
SOAP Messaging 2-9
Custom SQL as Sub-operation of Table-based Web Service 2-10

Creating based on Schema-formatted Results from Custom SQL SELECT Statements 2-11
Additional information at Design Time 2-12
Understanding Issues and Limitations 2-13
Repeated labels 2-13
Compatible column label sets 2-13
Creating from a Stored Procedure 2-13
Example 2-13
Creating from a Stored Function 2-14
Example 2-14
Creating from a Stored Procedure with complex PL/SQL arguments 2-15
Example 2-15
Creating from a Stored Function with complex PL/SQL arguments 2-17
Example 2-17
Creating from an Overloaded PL/SQL Stored Procedure 2-19
Example 2-19

3 Advanced Methods of Accessing DBWS Designtime API

Using DBWSBuilder with Ant 3-1
Example 3-1

Preface

EclipseLink provides specific annotations (*EclipseLink extensions*) in addition to supporting the standard Java Persistence Architecture (JPA) annotations. You can use these EclipseLink extensions to take advantage of EclipseLink's extended functionality and features within your JPA entities.

Audience

This document is intended for application developers who want to develop applications using EclipseLink with Java Persistence Architecture (JPA). This document does not include details about related common tasks, but focuses on EclipseLink functionality.

Developers should be familiar with the concepts and programming practices of

- Java SE and Java EE.
- Java Persistence Architecture 2.0 specification (`http://jcp.org/en/jsr/detail?id=317`)
- Eclipse IDE (`http://www.eclipse.org`)

Related Documents

For more information, see the following documents:

- EclipseLink Documentation Center at `http://www.eclipse.org/eclipselink/documentation/`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Overview of EclipseLink's DBWS Implementation

EclipseLink DBWS provides Java EE-compliant, client-neutral access to relational database artifacts via a Web service. EclipseLink DBWS extends EclipseLink's core capabilities while leveraging its existing ORM and OXM components.

EclipseLink DBWS includes two parts

- A **design-time** component, the `DBWSBuilder` command-line utility, that generates the necessary deployment artifacts.
- A **runtime provider** component that takes a service descriptor (along with related deployment artifacts) and realizes it as a JAX-WS 2.0 Web service. The runtime provider uses EclipseLink to bridge between the database and the XML SOAP Messages used by Web service clients.

An EclipseLink DBWS service may include any number of the following **operations**:

1. `insert` – inserts into the database persistent entities described by an XML document.
2. `update` – updates database persistent entities described by an XML document.
3. `delete` – removes from the database persistent entities described by an XML document.
4. `query` – retrieves from the database persistent entities described by an XML document.

   Selection criteria for Query operations can be specified by:

   - custom `SQL SELECT` statement
   - Stored Procedure invocation
   - EclipseLink Named Query (that can use the complete range of EclipseLink ORM Expression Framework APIs)
   - JP-QL

The XML documents used by an **operation** conform to an XML Schema Definition (`.xsd` file).

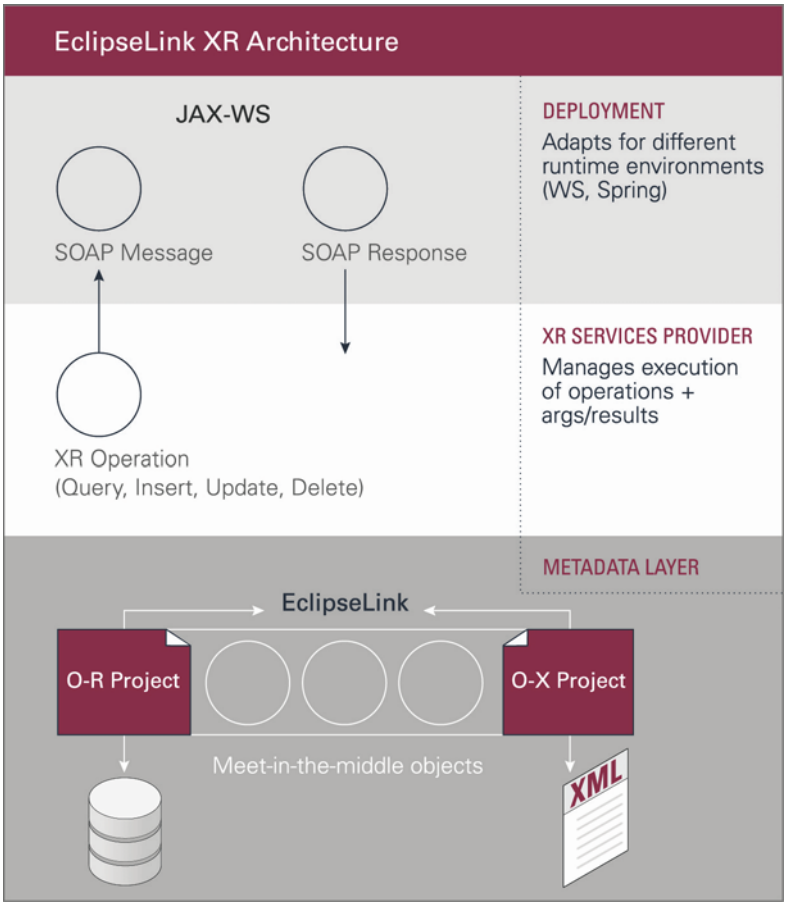
This chapter includes the following sections:

- Understanding XML-to-Relational Mapping (XRM)
- Creating EclipseLink DBWS Services
- Using the DBWS Design Time Component

Understanding XML-to-Relational Mapping (XRM)

EclipseLink's ORM and OXM features provides the basis for a powerful bridge between a database's relational structure(s) and XML's hierarchical structure.

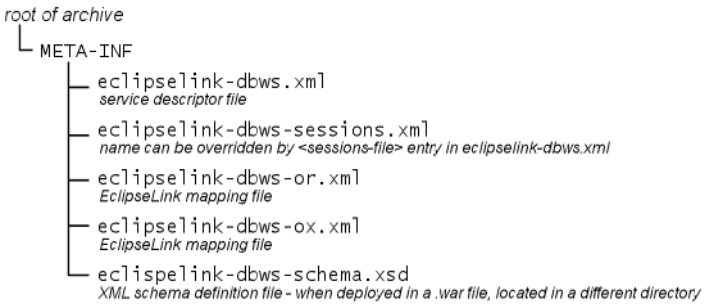
Figure 1–1 EclipseLink XR Architecture



Configuration

A typical EclipseLink DBWS service is packaged in an archive (`.jar` or `.war` file) with a service descriptor file `eclipselink-dbws.xml` in the `META-INF` directory (or `WEB-INF/classes/META-INF` when packaged in a `.war` file). To bridge the relational database and XML worlds, an EclipseLink `sessions.xml` (`eclipselink-dbws-sessions.xml`) points to two Eclipse projects: one for the ORM side, the other for the OXM side. The service also requires an XML Schema Definition file `eclipselink-dbws-schema.xsd` which in conjunction with the OXM project, specifies how information from the database is to be "shaped" into XML documents.

Figure 1–2 Typical EclipseLink DBWS Service Files



Note: Not all files are displayed.

The EclipseLink DBWS service descriptor file, `eclipselink-dbws.xml`, is easy to read, with minimal required information and simple defaults for omitted fields. This allows for auto-generation by a utility or manual editing. Example 1–1 illustrates a sample DBWS service descriptor file.

Example 1–1 Example DBWS Service descriptor file

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  >
  <name>example</name>
  <sessions-file>example-dbws-sessions.xml</sessions-file>
  <query>
    <name>countEmployees
    <result>
      <type>xsd:int</type>
    </result>
    <sql><!--[CDATA[select count(*) from EMP]]--></sql>
  </query>
</dbws>
```

Table 1–1 describes the elements of the EclipseLink DBWS service descriptor file.

Table 1–1 EclipseLink DBWS Service Descriptor File Elements

Element	Description	Required?	Default
`name`	Name of the EclipseLink DBWS service.	Yes, set by the `projectName` property in the `DBWSBuilder`.	None
`sessions-file`	Name of the EclipseLink `sessions.xml` file.	No	`eclipselink-dbws-sessions.xml`
Any of the following: ■ `insert` ■ `update` ■ `delete` ■ `query`	Service operations	At least one operation	None

XML Schema Definition

The EclipseLink DBWS service schema file `eclipselink-dbws-schema.xsd` can be created by hand, or auto-generated by the design-time `DBWSBuilder` utility that derives XML element-tag names from Database metadata (column names, types, nullable, and so on).

The DBWSBuilder utility will not generate an XML Schema Definition when the information returned by a query operation has no pre-determined structure, such as:

- a `resultSet` from a custom SQL **query operation**
- the results from a Stored Procedure **query operation**
- the row-count from an **update operation**

In these cases, the EclipseLink DBWS runtime provider uses information only available at the time of query execution to build the XML document:

Example 1–2 Example Simple XML Format (SXF) document

Element tag names are direct copies of table's column names.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<simple-xml-format>
  <simple-xml>
    <EMPNO>7788</EMPNO>
    <ENAME>SCOTT</ENAME>
    <JOB>ANALYST</JOB>
    <MGR>7566</MGR>
    <HIREDATE>1987-04-19T00:00:00.000-0400</HIREDATE>
    <SAL>3000</SAL>
    <DEPTNO>20</DEPTNO>
  </simple-xml>
  <simple-xml>
    <EMPNO>7369</EMPNO>
    <ENAME>SMITH</ENAME>
    <JOB>CLERK</JOB>
    <MGR>7902</MGR>
    <HIREDATE>1980-12-17T00:00:00.000-0400</HIREDATE>
    <SAL>800</SAL>
    <DEPTNO>20</DEPTNO>
  </simple-xml>
</simple-xml-format>
```

These XML documents are "dumb," as they cannot be validated against any pre-determined schema - or more accurately, only the following very *permissive* "sequence-of-any" schema can validate such documents:

Example 1–3 Simple XML Format Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <xsd:complexType name="simple-xml-format">
    <xsd:sequence>
      <xsd:any minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The element tags `simple-xml-format` and `simple-xml` can be customized by setting the appropriate properties on an **operation**.

Creating EclipseLink DBWS Services

You can generate a WAR file containing the EclipseLink DBWS service descriptor along with all required deployment artifacts for a JAX-WS 2.0 Web service (WSDL, XML schema, `web.xml`, EclipseLink object-relational mapping (ORM) and object-XML mapping (OXM) native project XML files, and so on).

Figure 1–3 Contents of WAR File

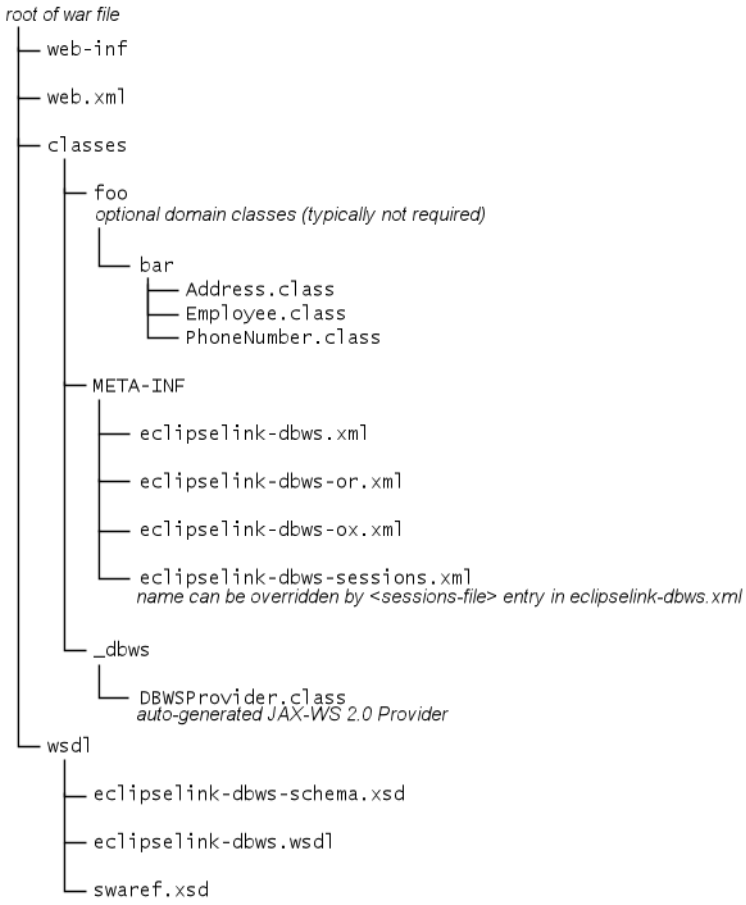


Table 1–2 EclipseLink DBWS Service .war File Contents

File	Description
`web.xml`	The Web application deployment file, required for deployment as a JAX-WS Web service. See JSR-109 for details.
`eclipselink-dbws.xml`	The EclipseLink DBWS service descriptor file, described in Table 1–1.

Table 1–2 (Cont.) EclipseLink DBWS Service .war File Contents

File	Description
`eclipselink-dbws-or.xml`	The EclipseLink ORM project XML file. For more information, see "Introduction to Relational Projects http://wiki.eclipse.org/Introduction_to_Relational_Projects_%28ELUG%29.
`eclipselink-dbws-ox.xml`	The EclipseLink OXM project XML file. For more information, see "Introduction to XML Projects" http://wiki.eclipse.org/Introduction_to_XML_Projects_%28ELUG%29
`eclipselink-dbws-sessions.xml`	The EclipseLink `sessions.xml` file for the EclipseLink DBWS service. This file contains references to the EclipseLink ORM and OXM project XML files. For more information, see "Introduction to EclipseLink Sessions" http://wiki.eclipse.org/Introduction_to_EclipseLink_Sessions_%28ELUG%29.
`eclipselink-dbws-schema.xsd`	Contains XML type definitions for operation arguments and return types. The `DBWSBuilder` utility automatically generates this file from database metadata to derive `element-tag` names and types.
`eclipselink-dbws.wsdl`	Contains entries for all operations in the EclipseLink DBWS service, required for deployment as a JAX-WS Web service. See JSR-109 for details (http://jcp.org/en/jsr/detail?id=109).
`swaref.xsd`	Contains XML type definitions for SOAP attachments, optional

Note that the files `swaref.xsd` and `web.xml` have names and content determined by their roles in Web deployment and cannot be changed.

The deployable `.war` file has been verified to work with the Oracle WebLogic Server 10.3 JavaEE container. See http://www.oracle.com/technology/software/products/ias/htdocs/wls_main.html?rssid=rss_otn_soft for more information.

An alternate deployable JAR file has been verified to work as a JavaSE 6 "containerless" EndPoint. See http://java.sun.com/javase/6/docs/api/javax/xml/ws/Endpoint.html and http://wiki.eclipse.org/EclipseLink/Examples/DBWS/AdvancedJavase6Containerless for more information.

Creating EclipseLink DBWS Services Using the DBWSBuilder Utility

This section describes how to create EclipseLink DBWS services using the `DBWSBuilder` utility.

You can use the EclipseLink DBWS design-time utility `DBWSBuilder` to create deployment files. `DBWSBuilder` is a Java application that processes the operations

described in an EclipseLink DBWS builder XML file to produce all the required deployment artifacts.

Be sure to set the following environment variables in the <ECLIPSELINK_HOME>\bin\setenv.cmd (or setenv.sh file) before invoking DBWSBuilder:

- JAVA_HOME
- DRIVER_CLASSPATH

There are script files provided for invoking DBWSBuilder. They are located in the <ECLIPSELINK_HOME>\utils\dbws directory. The scripts are dbwsbuilder.cmd for Windows usage, and dbwsbuilder.sh for other operating systems. Run the dbwsbuilder.cmd (or dbwsbuilder.sh) script without any arguments to display the help information

Using DBWSBuilder, you can generate an EclipseLink DBWS service from the following sources:

- an existing relational database table;
- one or more SQL SELECT statements;
- a stored procedure.

Creating an EclipseLink DBWS Service from a Database Table

You can create an EclipseLink DBWSBuilder XML file with a <table> query operation, as follows:

Example 1–4 Sample DBWSBuilder XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">table_test</property>
    ... database properties ...
  </properties>
  <table
    schemaPattern="%"
    tableNamePattern="dbws_crud"
  />
</dbws-builder>
```

For more information, see "Creating EclipseLink DBWS Service from a Database Table" on page 2-1.

Creating an EclipseLink DBWS Service from a SQL Statement

You can create an EclipseLink DBWS builder XML file with a <sql> query operation, as follows:

Example 1–5 Sample DBWS Builder XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">sql_test</property>
    ... database properties ...
  </properties>
  <sql name="employeeInfo" simpleXMLFormatTag="employee-info"
xmlTag="aggregate-counts">
    <text>
```

```
      <![CDATA[select count(*) as "COUNT", max(SAL) as "MAX-Salary" from EMP]]>
    </text>
  </sql>
</dbws-builder>
```

Using Parameter Binding The SQL `SELECT` statement for a `<sql>` operation may have parameters that must be bound to a datatype from the `eclipselink-dbws-schema.xsd`, or to any of the basic XSD datatypes. The SQL `SELECT` string uses JDBC-style `?` markers to indicate the position of the argument. The `<sql>` operation uses nested `<binding>` elements to match the datatype to the parameters. The order in which `<binding>` elements are defined must match the order of `?` markers in the SQL string:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">sql_binding_test</property>
    ... database properties ...
  </properties>
  <sql name="findEmpByName" isCollection="true" isSimpleXMLFormat="true">
    <text>
      <![CDATA[select * from EMP where EMPNO = ? and LAST_NAME = ?]]>
    </text>
    <binding name="EMPNO" type="xsd:int"/>
    <binding name="LAST_NAME" type="xsd:string"/>
  </sql>
</dbws-builder>
```

The argument named **EMPNO** is bound to an `integer` type, while the argument named **LAST_NAME** is bound to a `string` type.

For more information, see "Creating a DBWS Service from SQL Statements" on page 2-7.

Creating an EclipseLink DBWS Service from a Stored Procedure

You can create an EclipseLink DBWS builder XML File with a `<procedure>` query operation, as shown in Example 1–6.

Example 1–6 Using a <procedure> Query

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">procedure_test</property>
    ... database properties ...
    </properties>
  <procedure
    returnType="empType"
    catalogPattern="SOME_PKG"
    schemaPattern="SCOTT"
    procedurePattern="GetEmployeeByEMPNO_DEPTNO"/>
  </procedure>
</dbws-builder>
```

For more information, see "Creating from a Stored Procedure" on page 2-13.

Customizing an EclipseLink DBWS Service

There are a number use-cases that require an EclipseLink DBWS Service to be customized. The use-cases can be subdivided into the following categories:

- Simple – changing an `<element-tag>` to an "attribute";
- Intermediate – customizing the EclipseLink ORM or OXM projects;
- Advanced – manually generating all required deployment artifacts.

Performing Simple Customization

By default, `DBWSBuilder`-generated `eclipselink-dbws-schema.xsd` file derives `<element-tag>` names from the database table metadata, as shown in Example 1–7.

Example 1–7 DBWSBuilder-generated eclipselink-dbws-schema.xsd File

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <xsd:complexType name="empType">
    <xsd:sequence>
      <xsd:element name="empno" type="xsd:int" xsi:nil="false"/>
      <xsd:element name="ename" type="xsd:string" xsi:nil="true"/>
      <xsd:element name="job" type="xsd:string" xsi:nil="true"/>
      <xsd:element name="mgr" type="xsd:int" minOccurs="0" xsi:nil="true"/>
      <xsd:element name="hiredate" type="xsd:dateTime" xsi:nil="true"/>
      <xsd:element name="sal" type="xsd:decimal" xsi:nil="true"/>
      <xsd:element name="comm" type="xsd:int" minOccurs="0" xsi:nil="true"/>
      <xsd:element name="deptno" type="xsd:int" xsi:nil="true"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Use the `NamingConventionTransformer` to change an <element> tag to an attribute, as shown in Example 1–8.

Example 1–8 Converting to an Attribute

```
public interface NamingConventionTransformer {

    public enum ElementStyle {
        ELEMENT, ATTRIBUTE, NONE
    };

    public String generateSchemaName(String tableName);

    public String generateElementAlias(String originalElementName);

    public ElementStyle styleForElement(String originalElementName);
}
```

For more information, see "Naming Convention for schema elements" in the EclipseLink documentation: http://wiki.eclipse.org/EclipseLink/Examples/DBWS/DBWSIntermediateAttribute.

Performing Intermediate Customization

The primary reason to use an EclipseLink SessionCustomizer is to enable programmatic access to the EclipseLink API. Using this API, you can retrieve the object-relational or object-XML mapping descriptors from the session, and then use these descriptors to add, change, or delete mappings. You could also consider turning off the session cache, or changing the transaction isolation level of the database connection.

The following example shows how to implement a `org.eclipse.persistence.config.SessionCustomizer`:

```
package some.java.package;

import org.eclipse.persistence.config.SessionCustomizer;
import org.eclipse.persistence.sessions.Session;
import org.eclipse.persistence.sessions.DatabaseLogin;

public class MySessionCustomizer implements SessionCustomizer {

  public MySessionCustomizer() {
  }

  public void customize(Sesssion session) {
    DatabaseLogin login = (DatabaseLogin)session.getDatasourceLogin();
    login.setTransactionIsolation(DatabaseLogin.TRANSACTION_READ_UNCOMMITTED);
  }
}
```

In the `DBWSBuilder` builder XML file, specify if the customization applies to the ORM project or the OXM project, as the following example shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">customize_test</property>
     ...
    <property
name="orSessionCustomizerClassName">some.java.package.MyORSessionCustomizer</prope
rty>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">customize_test</property>
     ...
    <property
name="oxSessionCustomizerClassName">some.java.package.MyOXSessionCustomizer</prope
rty>
```

For more information, see "Session Customization" in the EclipseLink documentation: http://wiki.eclipse.org/Introduction_to_EclipseLink_Sessions_%28ELUG%29#Session_Customization.

Performing Advanced Customization

You can customize an EclipseLink DBWS service by creating your own `project.xml` and `sessions.xml` files. Using your preferred utility, you can do the following:

- map your objects to your relational database in an EclipseLink relational project;
- map your objects to your XML schema in an EclipseLink XML project:
- create an EclipseLink `sessions.xml` file that references both projects.

In this way, you can control all aspects of the relational and XML mapping. This approach is best when you want to customize most or all details. See "Using Existing EclipseLink ORM and OXM Mappings" on page 2-7 for more information.

Using DBWSBuilder API

The EclipseLink DBWS design-time utility, `DBWSBuilder`, is a Java application that generates EclipseLink DBWS files and assembles them into deployable archives.

It is normally invoked from the command-line via its main method:

```
prompt > dbwsbuilder.cmd -builderFile {path_to_builder.xml} -stageDir {path_to_
stageDir} -packageAs {packager}
```

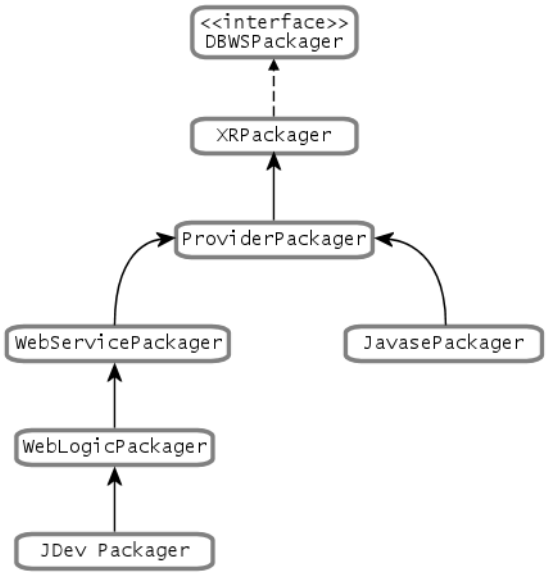
The given builder XML file (Example 1–9) is parsed by the OXM Project `org.eclipse.persistence.tools.dbws.DBWSBuilderModelProject` producing model objects that represent properties and `<table>` operations. Thus the public class `org.eclipse.persistence.tools.dbws.DBWSBuilder` can be populated programmatically through property setters (i.e. `setDriver()`, `setUrl()`) - table; SQL operations via `addSqlOperation()`.

Example 1–9 Sample Builder XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    <properties>
        <property name="projectName">test</property>
        <property name="driver">oracle.jdbc.OracleDriver</property>
        <property name="password">tiger</property>
        <property name="url">jdbc:oracle:thin:@localhost:1521:ORCL</property>
        <property name="username">scott</property>
    </properties>
    <table
      catalogPattern="%"
      schemaPattern="SCOTT"
      tableNamePattern="EMP"
    />
</dbws-builder>
```

The packager specified on the command-line is represented by a class that implements the `org.eclipse.persistence.tools.dbws.DBWSPackager` interface. There is a hierarchy of concrete implementations of this interface, shown in Figure 1–4:

Figure 1–4 Hierarchy of Concrete Implementation



The primary responsibility of a `DBWSPackager` is to provide `java.io.OutputStream`'s for the output generated by `DBWSBuilder`:

Example 1–10 Sample DBWSPackager

```
// call-backs for stream management
public OutputStream getSchemaStream() throws FileNotFoundException;
public void closeSchemaStream(OutputStream schemaStream);

public OutputStream getSessionsStream(String sessionsFileName) throws
FileNotFoundException;
public void closeSessionsStream(OutputStream sessionsStream);

public OutputStream getServiceStream() throws FileNotFoundException;
public void closeServiceStream(OutputStream serviceStream);

public OutputStream getOrStream() throws FileNotFoundException;
public void closeOrStream(OutputStream orStream);

public OutputStream getOxStream() throws FileNotFoundException;
public void closeOxStream(OutputStream oxStream);

public OutputStream getWSDLStream() throws FileNotFoundException;
public void closeWSDLStream(OutputStream wsdlStream);

public OutputStream getSWARefStream() throws FileNotFoundException;
public void closeSWARefStream(OutputStream swarefStream);

public OutputStream getWebXmlStream() throws FileNotFoundException;
public void closeWebXmlStream(OutputStream webXmlStream);

public OutputStream getProviderClassStream() throws FileNotFoundException;
public void closeProviderClassStream(OutputStream codeGenProviderStream);

public OutputStream getProviderSourceStream() throws FileNotFoundException;
public void closeProviderSourceStream(OutputStream sourceProviderStream);
```

Once all the model objects have been built, the builder is invoked either through the `start()` method, or alternatively via the `build(...)` method, which overrides the streams from the `DBWSPackager`, allowing the streams to be managed externally to the packager:

```
public void start() ...

public void build(OutputStream dbwsSchemaStream, OutputStream dbwsSessionsStream,
        OutputStream dbwsServiceStream, OutputStream dbwsOrStream, OutputStream
dbwsOxStream,
        OutputStream swarefStream, OutputStream webXmlStream, OutputStream
wsdlStream,
        OutputStream codeGenProviderStream, OutputStream sourceProviderStream,
Logger logger) ...
```

Using the DBWS Design Time Component

You can use the EclipseLink DBWS design-time utility `DBWSBuilder` to create deployment files. `DBWSBuilder` is a Java application that processes the operations described in an EclipseLink DBWS builder XML file to produce all the required deployment artifacts.

Be sure to set the following environment variables in the `<ECLIPSELINK_HOME>\utils\dbws\setenv.cmd` (or `setenv.sh file`) before invoking DBWSBuilder:

- `JAVA_HOME`
- `DRIVER_CLASSPATH`

There are script files provided for invoking `DBWSBuilder`. They are located in the `<ECLIPSELINK_HOME>\utils\dbws` directory. The scripts are `dbwsbuilder.cmd` for Windows usage, and `dbwsbuilder.sh` for other operating systems.

Example 1–11 DBWSBuilder usage

```
prompt > dbwsbuilder.cmd -builderFile {path_to_dbws_builder.xml} -stageDir  {path_
to_stageDir}
-packageAs[:archive_flag - archive, noArchive, ignore] {packager} [additional
args]
Available packagers:
    -packageAs:[default=not supported] jdev
    -packageAs:[default=archive] javase [jarFilename]
    -packageAs:[default=archive] wls [warFilename]
    -packageAs:[default=archive] glassfish [warFilename]
    -packageAs:[default=archive] jboss [warFilename]
    -packageAs:[default=archive] war [warFilename]
    -packageAs:[default=archive] was [warFilename]
    -packageAs:[default=not supported] eclipse
```

Using `DBWSBuilder`, you can generate an EclipseLink DBWS service from the following sources:

- an existing relational database table. See "Creating an EclipseLink DBWS Service from a Database Table" on page 1-7.
- one or more SQL SELECT statements. See "Creating an EclipseLink DBWS Service from a SQL Statement" on page 1-7.
- a stored procedure. See "Creating an EclipseLink DBWS Service from a Stored Procedure" on page 1-8.

- a stored function . See "Creating from a Stored Function" on page 2-14.

2

Creating DBWS Services

This chapter contains the following sections:

- Creating EclipseLink DBWS Service from a Database Table
- Using an EclipseLink SessionCustomizer
- Creating a DBWS Service from SQL Statements
- Creating from a Stored Procedure
- Creating from a Stored Function
- Creating from a Stored Function with complex PL/SQL arguments
- Creating from an Overloaded PL/SQL Stored Procedure

Creating EclipseLink DBWS Service from a Database Table

You can create a web service that exposes a database table's CRUD (**C**reate/**R**ead[findByPK and findAll]/**U**pdate/**D**elete) operations. EclipseLink supports this for any table or multiple tables (use patterns supporting % for catalog, schema or table names) on any database on which the JDBC driver reliably and accurately delivers the table's metadata via the JDBC metadata APIs (`java.sql.DatabaseMetaData`).

EclipseLink uses the `DBWSBuilder` utility to generate a DBWS XML schema, using the following rules:

- table name ==> translate any characters not supported by XML[1] ==> translate to_lowercase ==> add suffix 'Type' ==> top-level complex type in `.xsd` file
- column name ==> translate any characters not supported by XML[1] ==> translate to_lowercase ==> becomes `<element-tag>` name
  - All columns are expressed as elements
  - `BLOB` columns are automatically mapped to `xsd:base64Binary`
  - `xsd:base64Binary` elements can be included in-line to the XML document, or handled as binary attachments (`SwaRef` or `MTOM` style).

Example 2–1 uses the **EMP** table (Table 2–1) from the Oracle **scott** database schema:

[1] Same algorithm documented as part of the SQL/X (a.k.a. SQL/XML:2003) specification.

Table 2–1 Sample EMP Table

OWNER	TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NULLABLE?
SCOTT	EMP	EMPNO	NUMBER	22	4	0	N
SCOTT	EMP	ENAME	VARCHAR2	10	(null)	(null)	Y
SCOTT	EMP	JOB	VARCHAR2	9	(null)	(null)	Y
SCOTT	EMP	MGR	NUMBER	22	4	0	Y
SCOTT	EMP	HIREDATE	DATE	7	(null)	(null)	Y
SCOTT	EMP	SAL	NUMBER	22	7	2	Y
SCOTT	EMP	COMM	NUMBER	22	7	2	Y
SCOTT	EMP	DEPTNO	NUMBER	22	2	0	Y

Example 2–1 Example

The DBWSBuilder utility requires a DBWS configuration file as input, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">emp</property>
    ... database properties
  </properties>
  <table
    catalogPattern="%"
    tableNamePattern="EMP"
  />
</dbws-builder>
```

Use this command to execute the DBWSBuilder:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls emp.war
```

where

- `dbws-builder.xml` is the DBWS configuration file (as shown previously)
- `output_directory` is the output directory for the generated files
- `-packageAs` is the platform on which the web service will be deployed

The `DBWSBuilder`-generated `eclipselink-dbws-schema.xsd` file derives `<element-tag>` names from the Database table metadata in Table 2–1:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <xsd:complexType name="empType">
    <xsd:sequence>
      <xsd:element name="empno" type="xsd:int" xsi:nil="false"/>
```

```
      <xsd:element name="ename" type="xsd:string" xsi:nil="true"/>
      <xsd:element name="job" type="xsd:string" xsi:nil="true"/>
      <xsd:element name="mgr" type="xsd:int" minOccurs="0" xsi:nil="true"/>
      <xsd:element name="hiredate" type="xsd:dateTime" xsi:nil="true"/>
      <xsd:element name="sal" type="xsd:decimal" xsi:nil="true"/>
      <xsd:element name="comm" type="xsd:int" minOccurs="0" xsi:nil="true"/>
      <xsd:element name="deptno" type="xsd:int" xsi:nil="true"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Generated EclipseLink DBWS service descriptor

The CRUD operations are illustrated in the generated EclipseLink DBWS service descriptor (`eclipselink-dbws.xml`) file, as shown here:

Example 2–2 Sample CRUD Operations

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="urn:emp"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <name>emp</name>
   <sessions-file>eclipselink-dbws-sessions.xml</sessions-file>
   <update>
      <name>update_empType</name>
      <parameter>
         <name>theInstance</name>
         <type>ns1:empType</type>
      </parameter>
   </update>
   <insert>
      <name>create_empType</name>
      <parameter>
         <name>theInstance</name>
         <type>ns1:empType</type>
      </parameter>
   </insert>
   <query>
      <name>findByPrimaryKey_empType</name>
      <parameter>
         <name>id</name>
         <type>xsd:decimal</type>
      </parameter>
      <result>
         <type>ns1:empType</type>
      </result>
      <named-query>
         <name>findByPrimaryKey</name>
         <descriptor>empType</descriptor>
      </named-query>
   </query>
   <delete>
      <name>delete_empType</name>
      <parameter>
         <name>theInstance</name>
         <type>ns1:empType</type>
      </parameter>
   </delete>
   <query>
```

```
      <name>findAll_empType</name>
      <result isCollection="true">
         <type>ns1:empType</type>
      </result>
      <named-query>
         <name>findAll</name>
         <descriptor>empType</descriptor>
      </named-query>
   </query>
</dbws>
```

SOAP Messaging

The following SOAP Message invokes the `<findAll_empType>` operation for the **emp** DBWS service:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <findAll_empType xmlns="urn:empService" xmlns:urn="urn:emp"/>
  </env:Body>
</env:Envelope>
```

returning:

```
<?xml version="1.0" encoding="utf-16"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header />
  <SOAP-ENV:Body>
    <srvc:findAll_empTypeResponse xmlns="urn:emp" xmlns:srvc="urn:empService">
      <srvc:result>
        <empType>
          <empno>7369</empno>
          <ename>SMITH</ename>
          <job>CLERK</job>
          <mgr>7902</mgr>
          <hiredate>1980-12-17T00:00:00.0-05:00</hiredate>
          <sal>800</sal>
          <deptno>20</deptno>
        </empType>
        <empType>
          <empno>7499</empno>
          <ename>ALLEN</ename>
          <job>SALESMAN</job>
          <mgr>7698</mgr>
          <hiredate>1981-02-20T00:00:00.0-05:00</hiredate>
          <sal>1600</sal>
          <comm>300</comm>
          <deptno>30</deptno>
        </empType>
        ....
      </srvc:result>
    </srvc:findAll_empTypeResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Using an EclipseLink SessionCustomizer

When using an EclipseLink `SessionCustomizer` with DBWS, you can access to the EclipseLink API to retrieve the OR (`object-relational`) or OX (`object-XML`) mapping descriptors from the session. You can then use the descriptors to add, change, or delete mappings.

For more information, see "Session Customization in the EclipseLink documentation: http://wiki.eclipse.org/Introduction_to_EclipseLink_Sessions_%28ELUG%29#Session_Customization

Example

This example illustrates how to implement an EclipseLink `SessionCustomizer`:

```
package some.java.package;

import org.eclipse.persistence.config.SessionCustomizer;
import org.eclipse.persistence.sessions.Session;
import org.eclipse.persistence.sessions.DatabaseLogin;

public class MySessionCustomizer implements SessionCustomizer {

  public MySessionCustomizer() {
  }

  public void customize(Sesssion session) {
    DatabaseLogin login = (DatabaseLogin)session.getDatasourceLogin();
    // enable 'dirty' reads
    login.setTransactionIsolation(DatabaseLogin.TRANSACTION_READ_UNCOMMITTED);
  }
}
```

In the `DBWSBuilder` configuration file, you must use the `orSessionCustomizerClassName` or `oxSessionCustomizerClassName` to specify if the customization applies to the ORM or ORX project (respectively), as shown here:

Example 2–3 ORM Project

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">customize_test</property>
     ...
    <property
name="orSessionCustomizerClassName">some.java.package.MyORSessionCustomizer</prope
rty>
```

Example 2–4 ORX Project

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <properties>
    <property name="projectName">customize_test</property>
     ...
    <property
```

```
name="oxSessionCustomizerClassName">some.java.package.MyOXSessionCustomizer</prope
rty>
```

Advanced Customization

You can further customize an EclipseLink DBWS service by creating your own EclipseLink `project.xml` and `sessions.xml` files. Using your preferred utility, you can:

- map your objects to your relational database in an EclipseLink relational project
- map your objects to your XML schema in an EclipseLink XML project
- create an EclipseLink `sessions.xml` file that references both projects.

In this way, you can control all aspects of the relational and XML mapping. This approach is best when you want to customize most or all details.

In Example 2–5, a DBWS service is constructed from existing EclipseLink project maps with identical case-sensitive aliases (for Descriptors that are common between the projects).

Example 2–5 Sample DBWS Service

```
<?xml version="1.0" encoding="UTF-8"?>
<object-persistence version="Eclipse Persistence Services - some version (some
build date)" xmlns="http://www.eclipse.org/eclipselink/xsds/persistence"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:eclipselink="http://www.eclipse.org/eclipselink/xsds/persistence">
   <name>SomeORProject</name>
   <class-mapping-descriptors>
      <class-mapping-descriptor xsi:type="relational-class-mapping-descriptor">
         <class>some.package.SomeClass</class>
         <alias>SomeAlias</alias>
...


<?xml version="1.0" encoding="UTF-8"?>
<object-persistence version="Eclipse Persistence Services - some version (some
build date)" xmlns="http://www.eclipse.org/eclipselink/xsds/persistence"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:eclipselink="http://www.eclipse.org/eclipselink/xsds/persistence">
   <name>SomeOXProject</name>
   <class-mapping-descriptor xsi:type="xml-class-mapping-descriptor">
      <class>some.package.SomeClass</class>
      <alias>SomeAlias</alias>
...
```

Note: When building a DBWS web service in this way (that is, without the `DBWSBuilder` Utility) be sure to create all the necessary deployment artifacts.

Using Existing EclipseLink ORM and OXM Mappings

A DBWS service may be constructed using pre-existing EclipseLink ORM and OXM maps (both Project classes and Project deployment XML are supported) with identical case-sensitive aliases for Descriptors that are common between the projects.

Example 2–6 Using an ORM Map

```
<?xml version="1.0" encoding="UTF-8"?>
<object-persistence version="Eclipse Persistence Services - some version (some
build date)" xmlns="http://www.eclipse.org/eclipselink/xsds/persistence"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:eclipselink="http://www.eclipse.org/eclipselink/xsds/persistence">
   <name>SomeORProject</name>
   <class-mapping-descriptors>
      <class-mapping-descriptor xsi:type="relational-class-mapping-descriptor">
         <class>some.package.SomeClass</class>
         <alias>SomeAlias</alias>
...
```

Example 2–7 Using an OXM Map

```
<?xml version="1.0" encoding="UTF-8"?>
<object-persistence version="Eclipse Persistence Services - some version (some
build date)" xmlns="http://www.eclipse.org/eclipselink/xsds/persistence"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:eclipselink="http://www.eclipse.org/eclipselink/xsds/persistence">
   <name>SomeOXProject</name>
   <class-mapping-descriptor xsi:type="xml-class-mapping-descriptor">
      <class>some.package.SomeClass</class>
      <alias>SomeAlias</alias>
...
```

When building a DBWS web service in this way (that is, without the `DBWSBuilder` Utility) be sure to create all the necessary deployment artifacts

Creating a DBWS Service from SQL Statements

This section includes information on:

- Creating from Results Sets from Custom SQL SELECT Statements
- Creating based on Schema-formatted Results from Custom SQL SELECT Statements

Creating from Results Sets from Custom SQL SELECT Statements

EclipseLink DBWS can create a Web service that exposes the results of executing custom SQL `SELECT` statements, without exposing the actual SQL. There is no metadata to determine the structure of the returned data -- the Simple XML Format schema is used.

The SQL `SELECT` statements targeted for this service are in the `DBWSBuilder` builder XML file, as shown here:

Example 2–8 Sample DBWSBuilder XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">testSql</property>
    ... database properties
  </properties>
  <sql
    name="count"
    simpleXMLFormatTag="aggregate-info"
    xmlTag="count"
    >
    <text><![CDATA[select count(*) from EMP]]></text>
  </sql>
  <sql
    name="countAndMaxSalary"
    simpleXMLFormatTag="aggregate-info"
    xmlTag="count-and-max-salary"
    >
    <text><![CDATA[select count(*) as "COUNT", max(SAL) as "MAX-Salary" from
EMP]]></text>
  </sql>
</dbws-builder>
```

Use this command to create the web service:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls testSql.war
```

where

- `dbws-builder.xml` is the DBWS builder XML configuration file, as shown previously
- `output_directory` is the output directory for the generated files
- `-packageAs` the platform on which the web service will be deployed

The generated `eclipselink-dbws-schema.xsd` file is the schema for the Simple XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <xsd:complexType name="simple-xml-format">
    <xsd:sequence>
      <xsd:any minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The element tags `simple-xml-format` and `simple-xml` are customized in the SQL operations. For example, `<simple-xml-format>` = `<aggregate-info>`, `<simple-xml>` = `<count-and-max-salary>`.

Generated EclipseLink DBWS Service Descriptor

The SQL operations are included in the DBWS service descriptor file (`eclipselink-dbws.xml`) created by EclipseLink, as well as the settings to alter the default Simple XML Format `<element-tag>` name.

Example 2–9 Sample XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="urn:testSql"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <name>testSql</name>
   <sessions-file>eclipselink-dbws-sessions.xml</sessions-file>
   <query>
      <name>count</name>
      <result>
         <type>simple-xml-format</type>
         <simple-xml-format>
            <simple-xml-format-tag>aggregate-info</simple-xml-format-tag>
            <simple-xml-tag>count</simple-xml-tag>
         </simple-xml-format>
      </result>
      <sql>
         <![CDATA[select count(*) from EMP]]>
      </sql>
   </query>
   <query>
      <name>countAndMaxSalary</name>
      <result>
         <type>simple-xml-format</type>
         <simple-xml-format>
            <simple-xml-format-tag>aggregate-info</simple-xml-format-tag>
            <simple-xml-tag>count-and-max-salary</simple-xml-tag>
         </simple-xml-format>
      </result>
      <sql>
         <![CDATA[select count(*) as "COUNT", max(SAL) as "MAX-Salary" from EMP]]>
      </sql>
   </query>
</dbws>
```

SOAP Messaging The following SOAP Message invokes the `<count>` operation for the `testSql` DBWS service:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <count xmlns="urn:testSqlService" xmlns:urn="urn:testSql"/>
  </env:Body>
</env:Envelope>
```

returning:

```
<?xml version="1.0" encoding="utf-16"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header />
  <SOAP-ENV:Body>
    <srvc:countResponse xmlns:srvc="urn:testSqlService">
      <srvc:result>
        <aggregate-info xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:type="simple-xml-format">
          <count>
            <COUNT_x0028__x002A__x0029_>14</COUNT_x0028__x002A__x0029_>
          </count>
        </aggregate-info>
      </srvc:result>
    </srvc:countResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

> **Note:** You should be aware of the `resultSet` for `select count(*);` the characters '(', '*' and ')' are not valid for XML element tags and are replaced by the well-known transformation, which documented as part of the SQL/X specification (SQL/XML:2003).

Custom SQL as Sub-operation of Table-based Web Service The "SOAP Messaging" on page 2-9 operation returns unstructured data. However, it is possible to nest such operations within the context of a Table-based operation; then, the nested operations can be configured to re-use the schema element type of the parent table and return structured data:

```
<dbws-builder>
  <properties>
    <property name="projectName">empSql</property>
    ... database properties
  </properties>
  <table
    catalogPattern="%"
    tableNamePattern="EMP"
    >
    <sql
      name="findEmpByName"
      isCollection="true"
      returnType="empType"
      >
      <text><![CDATA[select * from EMP where ENAME like ?]]></text>
      <binding name="ENAME" type="xsd:string"/>
    </sql>
  </table>
 </dbws-builder>
```

The generated EclipseLink DBWS service descriptor `eclipselink-dbws.xml` file:

Example 2–10 Sample eclipselink-dbws.xml File

```
<dbws xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="urn:testSql"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <name>empSql</name>
   <sessions-file>eclipselink-dbws-sessions.xml</sessions-file>
   <update>
      <name>update_empType</name>
      <parameter>
         <name>theInstance</name>
         <type>ns1:empType</type>
      </parameter>
   </update>
   ...
```

```
   <query>
      <name>findEmpByName</name>
      <parameter>
         <name>ENAME</name>
         <type>xsd:string</type>
      </parameter>
      <result isCollection="true">
         <type>ns1:empType</type>
      </result>
      <sql>
         <![CDATA[select * from EMP where ENAME like #ENAME]]>
      </sql>
   </query>
</dbws>
```

Creating based on Schema-formatted Results from Custom SQL SELECT Statements

EclipseLink can also create a web service in which the "shape" of the returned result is determined at design-time, not runtime. Normally, the custom SQL `SELECT` statement returns `java.sql.ResultSets` and the `java.sql.ResultSetMetaData` APIs (`getColumnCount`, `getColumnLabel`, `getColumnType`, etc.) can be used to determine the name and datatype of the returned information.

EclipseLink DBWS uses the Simplified XML Format (SXF) to create an XML document to describe the ResultSet's information. However, because this document can change arbitrarily, the SXF schema is extremely "loose" – the use of `xsd:any` places virtually no restriction on the document.

Example 2–11 Sample Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <xsd:complexType name="simple-xml-format">
    <xsd:sequence>
      <xsd:any minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 2–12 Instance document:

```
<source lang="xml">
<?xml version = '1.0' encoding = 'UTF-8'?>
<simple-xml-format>
  <simple-xml>
    <EMPNO>7788</EMPNO>
    <ENAME>SCOTT</ENAME>
    <JOB>ANALYST</JOB>
    <MGR>7566</MGR>
    <HIREDATE>1987-04-19</HIREDATE>
    <SAL>3000</SAL>
    <DEPTNO>20</DEPTNO>
  </simple-xml>
  <simple-xml>
    <EMPNO>7369</EMPNO>
    <ENAME>SMITH</ENAME>
    <JOB>CLERK</JOB>
```

```
    <MGR>7902</MGR>
    <HIREDATE>1980-12-17</HIREDATE>
    <SAL>800</SAL>
    <DEPTNO>20</DEPTNO>
  </simple-xml>
</simple-xml-format>
```

Additional information at Design Time

As indicated previously, the `java.sql.ResultSetMetaData` APIs provide enough information, if available at design-time, from which you could generate a schema, as shown here:

Example 2–13 Sample Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:emp"
xmlns="urn:emp" elementFormDefault="qualified">
   <xsd:complexType name="empType">
      <xsd:sequence>
         <xsd:element name="empno" type="xsd:decimal"/>
         <xsd:element name="ename" type="xsd:string" minOccurs="0"
nillable="true"/>
         <xsd:element name="job" type="xsd:string" minOccurs="0" nillable="true"/>
         <xsd:element name="mgr" type="xsd:decimal" minOccurs="0"
nillable="true"/>
         <xsd:element name="hiredate" type="xsd:date" minOccurs="0"
nillable="true"/>
         <xsd:element name="sal" type="xsd:decimal" minOccurs="0"
nillable="true"/>
         <xsd:element name="comm" type="xsd:decimal" minOccurs="0"
nillable="true"/>
         <xsd:element name="deptno" type="xsd:decimal" minOccurs="0"
nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:element name="empType" type="empType"/>
</xsd:schema>
```

The DBWS sql operation is enhanced with an additional SQL statement that is executed at design-time -- the statement will not return any rows (such as when the `WHERE` clause evaluates to **false** in Example 2–14):

Example 2–14 Executing Additional SQL Statements

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    <properties>
        <property name="projectName">emp</property>
        ...
    </properties>
    <sql
        name="Semployees"
        isCollection="false"
        returnType="empType"
        >
        <statement><![CDATA[select * from EMP where ENAME like 'S%']]></statement>
        <build-statement><![CDATA[select * from EMP where 0=1]]></build-statement>
    </sql>
```

```
</dbws-builder>
```

Understanding Issues and Limitations

Be aware of the following limitation:

- Repeated labels
- Compatible column label sets

Repeated labels Valid SQL allows multiple identical columns. For example, consider the following SQL:

```
SELECT ENAME, ENAME FROM EMP WHERE LIKE 'S%'
```

ENAME	ENAME
SMITH	SMITH
...	...
SCOTT	SCOTT

In this example, a `SELECT` statement that uses `UNION` could return a set of column labels where a label is repeated.

`DBWSBuilder` maintains a list of "already processed columns" and will throw an exception when it detects a duplicate.

Compatible column label sets The runtime and design-time SQL statements **must** return compatible column label sets. EclipseLink performs no pre-processing to ensure that the column sets are the same; the error will be detected at runtime when the service is invoked.

Creating from a Stored Procedure

EclipseLink DBWS can create a Web service that exposes a Stored Procedure (or multiple procedures). Because it is not possible to determine the structure of the returned data from the Stored Procedure's metadata, EclipseLink uses the Simple XML Format schema. The EclipseLink DBWS runtime produces an XML document that is simple and "human-readable."

EclipseLink DBWS supports any combination of `IN`, `OUT` and `IN OUT` arguments. Additionally, EclipseLink also supports procedures in packages that are overloaded (that is, the same name but different parameters).

Example

This example uses the following Stored Procedure:

```
DROP PROCEDURE TESTECHO;
CREATE OR REPLACE PROCEDURE TESTECHO(T IN VARCHAR2, U OUT VARCHAR2) AS
BEGIN
    U := CONCAT(T, '-test');
END;
```

The DBWSBuilder utility requires a DBWS configuration XML file as input, as shown here:

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

```
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">testEcho</property>
    ... database properties
  </properties>
  <procedure
    name="testEcho"
    procedurePattern="TESTECHO"
    isSimpleXMLFormat="true"
  />
</dbws-builder>
```

Execute the `DBWSBuilder`, as shown here:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls testEcho.war
```

where

- `dbws-builder.xml` is the DBWS builder configuration XML file above
- `output_directory` is the output directory for the generated files
- `-packageAs` specifies the platform on which the web service will be deployed

The generated `eclipselink-dbws-schema.xsd` file is the schema for the Simple XML format, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  >
  <xsd:complexType name="simple-xml-format">
    <xsd:sequence>
      <xsd:any minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

You can customize the `simple-xml-format` and `simple-xml` tags by setting the appropriate properties on an SQL operation.

Creating from a Stored Function

EclipseLink DBWS can create a Web service that exposes a simple Stored Function.

Example

In this example, the following stored function will be used:

```
DROP FUNCTION TESTECHO;
CREATE OR REPLACE FUNCTION TESTECHO(T IN VARCHAR2) RETURN VARCHAR2 IS retVal
VARCHAR2
BEGIN
      retVal := CONCAT('test-' , T);
      RETURN retVal;
END TESTECHO;
```

The DBWSBuilder utility requires a DBWS configuration XML file as input, as shown here:

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

```
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">testEcho</property>
    ... database properties
  </properties>
  <procedure
    name="testEcho"
    procedurePattern="TESTECHO"
    returnType="xsd:string"
  />
</dbws-builder>
```

Execute the **DBWSBuilder**, as shown here:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls testEcho.war
```

where

- `dbws-builder.xml` is the DBWS builder configuration XML file above
- `output_directory` is the output directory for the generated files
- `-packageAs` the platform on which the web service will be deployed

Creating from a Stored Procedure with complex PL/SQL arguments

With EclipseLink, you can create a DBWS web service from a stored procedure that uses complex PL/SQL types as either an `IN`, `OUT`, or `IN OUT` argument.

Example

In this example, the following stored procedure is used:

```
PROCEDURE P1(OLDREC IN ARECORD, FOO IN VARCHAR2, AREC OUT ARECORD) IS
  BEGIN
    AREC.T1 := ... some processing based upon OLDREC
    AREC.T2 := ... AND FOO
    AREC.T3 := ...
  END P1;
```

Type `ARECORD` is defined in the PL/SQL package `SOMEPACKAGE` as follows:

```
CREATE OR REPLACE PACKAGE SOMEPACKAGE AS
  TYPE TBL1 IS TABLE OF VARCHAR2(111) INDEX BY BINARY_INTEGER;
  TYPE TBL2 IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  TYPE ARECORD IS RECORD (
    T1 TBL1,
    T2 TBL2,
    T3 BOOLEAN
  );
  PROCEDURE P1(OLDREC IN ARECORD, FOO IN VARCHAR2, AREC OUT ARECORD);
END SOMEPACKAGE;
```

Because PL/SQL record and collection types cannot be transported via JDBC, EclipseLink will generate an anonymous block of PL/SQL code that contains the functions to convert to and from JDBC and PL/SQl types. To be successful, each PL/SQL type or collection type that will appear in an `IN`, `IN OUT`, or `OUT OF RETURN` argument (or any PL/SQL record or collection type that is nested within these arguments) *must* have an equivalent JDBC type. The name of this type must be in the form `<package name>_<type name>`.

For this example, the following JDBC types are required:

```
CREATE OR REPLACE TYPE SOMEPACKAGE_TBL1 AS TABLE OF VARCHAR2(111)

CREATE OR REPLACE TYPE SOMEPACKAGE_TBL2 AS TABLE OF NUMBER

CREATE OR REPLACE TYPE SOMEPACKAGE_ARECORD AS OBJECT (
  T1 SOMEPACKAGE_TBL1,
  T2 SOMEPACKAGE_TBL2,
  T3 BOOLEAN
)
```

The DBWSBuilder utility requires a DBWS configuration file as input.

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">testPLSQLProcedure</property>
    ... database properties
  </properties>
  <plsql-procedure
    name="plsqlprocedure"
    catalogPattern="SOMEPACKAGE"
    procedurePattern="P1"
  />
</dbws-builder>
```

Notice that `returnType` is set to `SOMEPACKAGE_ARECORD`. This value indicates a complex type in the generated EclipseLink DBWS schema (as shown below). In this case, it is constructed based on the contents of the `SOMEPACKAGE` package.

Execute the `DBWSBuilder`, as shown here:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls testPLSQLProcedure.war
```

where

- `dbws-builder.xml` is the DBWS builder configuration file (as shown previously).
- `output_directory` is the output directory for the generated files.
- `-packageAs` is the platform on which the web service will be deployed.

The generated `eclipselink-dbws-schema.xsd` file follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:plsqlprocedure" xmlns="urn:plsqlprocedure"
elementFormDefault="qualified">
   <xsd:complexType name="SOMEPACKAGE_TBL1">
      <xsd:sequence>
         <xsd:element name="item" type="xsd:string" maxOccurs="unbounded"
nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="SOMEPACKAGE_TBL2">
      <xsd:sequence>
         <xsd:element name="item" type="xsd:decimal" maxOccurs="unbounded"
nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="SOMEPACKAGE_ARECORD">
```

```
      <xsd:sequence>
         <xsd:element name="t1">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="item" type="xsd:string" maxOccurs="unbounded"
nillable="true"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="t2">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="item" type="xsd:decimal"
maxOccurs="unbounded" nillable="true"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="t3" type="xsd:boolean" nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="simple-xml-format">
      <xsd:sequence>
         <xsd:any minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:element name="SOMEPACKAGE_TBL1" type="SOMEPACKAGE_TBL1"/>
   <xsd:element name="SOMEPACKAGE_TBL2" type="SOMEPACKAGE_TBL2"/>
   <xsd:element name="SOMEPACKAGE_ARECORD" type="SOMEPACKAGE_ARECORD"/>
</xsd:schema>
```

Creating from a Stored Function with complex PL/SQL arguments

Starting with EclipseLink 2.3, you can create a DBWS web service from a stored function that uses complex PL/SQL types as either an `IN` or return argument.

Example

In this example, the following stored function is used:

```
FUNCTION F1(OLDREC IN ARECORD, FOO IN VARCHAR2) RETURN ARECORD IS
  arec ARECORD; -- temp var
  BEGIN
    arec.T1 := ... some processing based upon OLDREC
    arec.T2 := ... AND FOO
    arec.T3 := ...
    RETURN arec;
  END F1;
```

Type `ARECORD` is defined in the PL/SQL package `SOMEPACKAGE` as follows:

```
CREATE OR REPLACE PACKAGE SOMEPACKAGE AS
  TYPE TBL1 IS TABLE OF VARCHAR2(111) INDEX BY BINARY_INTEGER;
  TYPE TBL2 IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  TYPE ARECORD IS RECORD (
    T1 TBL1,
    T2 TBL2,
    T3 BOOLEAN
  );
```

```
  FUNCTION F1(OLDREC IN ARECORD, FOO IN VARCHAR2) RETURN ARECORD;
END SOMEPACKAGE;
```

Because PL/SQL types and collection types cannot be transported via JDBC, EclipseLink will generate an anonymous block of PL/SQL code that contains the functions to convert to and from JDBC and PL/SQl types. To be successful, each PL/SQL type or collection type that will appear in an `IN`, `IN OUT`, or `OUT OF RETURN` argument (or any PL/SQL record or collection type that is nested within these arguments) *must* have an equivalent JDBC type. The name of this type must be in the form `<package name>_<type name>`.

For this example, the following JDBC types are required:

```
CREATE OR REPLACE TYPE SOMEPACKAGE_TBL1 AS TABLE OF VARCHAR2(111)

CREATE OR REPLACE TYPE SOMEPACKAGE_TBL2 AS TABLE OF NUMBER

CREATE OR REPLACE TYPE SOMEPACKAGE_ARECORD AS OBJECT (
  T1 SOMEPACKAGE_TBL1,
  T2 SOMEPACKAGE_TBL2,
  T3 BOOLEAN
)
```

The DBWSBuilder utility requires a DBWS configuration file as input.

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">testPLSQLFunction</property>
    ... database properties
  </properties>
  <plsql-procedure
    name="plsqlfunction"
    catalogPattern="SOMEPACKAGE"
    procedurePattern="F1"
    returnType="SOMEPACKAGE_ARECORD"
  />
</dbws-builder>
```

Notice that `returnType` is set to `SOMEPACKAGE_ARECORD`. This value indicates a complex type in the generated EclipseLink DBWS schema (as shown below). In this case, it is constructed based on the contents of the `SOMEPACKAGE` package.

Execute the `DBWSBuilder`, as shown here:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls testPLSQLFunction.war
```

where

- `dbws-builder.xml` is the DBWS builder configuration file (as shown previously).
- `output_directory` is the output directory for the generated files.
- `-packageAs` is the platform on which the web service will be deployed.

The generated `eclipselink-dbws-schema.xsd` file follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:plsqlfunction" xmlns="urn:plsqlfunction"
elementFormDefault="qualified">
   <xsd:complexType name="SOMEPACKAGE_TBL1">
```

```
      <xsd:sequence>
         <xsd:element name="item" type="xsd:string" maxOccurs="unbounded"
nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="SOMEPACKAGE_TBL2">
      <xsd:sequence>
         <xsd:element name="item" type="xsd:decimal" maxOccurs="unbounded"
nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="SOMEPACKAGE_ARECORD">
      <xsd:sequence>
         <xsd:element name="t1">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="item" type="xsd:string" maxOccurs="unbounded"
nillable="true"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="t2">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="item" type="xsd:decimal"
maxOccurs="unbounded" nillable="true"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="t3" type="xsd:boolean" nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="simple-xml-format">
      <xsd:sequence>
         <xsd:any minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:element name="SOMEPACKAGE_TBL1" type="SOMEPACKAGE_TBL1"/>
   <xsd:element name="SOMEPACKAGE_TBL2" type="SOMEPACKAGE_TBL2"/>
   <xsd:element name="SOMEPACKAGE_ARECORD" type="SOMEPACKAGE_ARECORD"/>
</xsd:schema>
```

Creating from an Overloaded PL/SQL Stored Procedure

Starting in release 2.3, EclipseLInk DBWS can create a web service that exposes multiple PL/SQL stored procedures. Instead of specifying all the stored procedures within the DBWS builder file, you can specify a single procedure name and "overload" it with different parameters.

> **Note:** This feature requires a database, such as Oracle, that supports overloading.

Example

In this example, the stored procedure contains different parameters:

```
CREATE PROCEDURE P(SIMPLARRAY IN TBL1, FOO IN VARCHAR2) AS
```

```
BEGIN
  -- 2 arguments SIMPLARRAY and FOO
END P;
CREATE PROCEDURE P(SIMPLARRAY IN TBL1, FOO IN VARCHAR2, BAR IN VARCHAR2) AS
BEGIN
  -- (same name 'P') 3 arguments SIMPLARRAY, FOO and BAR
END P;
```

EclipseLink DBWS supports any combination of the `IN`, `OUT` and `IN OUT` arguments.

Type **TBL1** is defined in PL/SQL Package `SOMEPACKAGE` as follows:

```
CREATE OR REPLACE PACKAGE SOMEPACKAGE AS
  TYPE TBL1 IS TABLE OF VARCHAR2(111) INDEX BY BINARY_INTEGER;
  PROCEDURE P(SIMPLARRAY IN TBL1, FOO IN VARCHAR2);
  PROCEDURE P(SIMPLARRAY IN TBL1, FOO IN VARCHAR2, BAR IN VARCHAR2);
END SOMEPACKAGE;
```

The `DBWSBuilder` utility requires a DBWS configuration file as input, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">testOverloadedProcedure</property>
    ... database properties
  </properties>
  <plsql-procedure
    name="overloadedProcedure"
    catalogPattern="SOMEPACKAGE"
    procedurePattern="P"
  />
</dbws-builder>
```

Use this command to execute the `DBWSBuilder`:

```
prompt > dbwsbuilder.cmd -builderFile dbws-builder.xml -stageDir output_directory
-packageAs wls testEcho.war
```

where

- `dbws-builder.xml` – The DBWS configuration file (as shown previously)
- `output_directory` – The output directory for the generated files
- `-packageAs` – Specifies the platform on which the web service will be deployed

When generating queries and the WSDL in which overloaded procedures are used, a unique index identifies each procedure. The index starts at **1** and increments by one, for each overloaded procedure.

In this example, EclipseLink generates the following `eclipselink-dbws.wsdl` (Web Services Description Language):

```
<wsdl:definitions
     name="plsqloverloadService"
     targetNamespace="urn:plsqloverloadService"
     xmlns:ns1="urn:plsqloverload"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="urn:plsqloverloadService"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    >
    <wsdl:types>
        <xsd:schema elementFormDefault="qualified"
```

```
targetNamespace="urn:plsqloverloadService" xmlns:tns="urn:plsqloverloadService"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xsd:import namespace="urn:plsqloverload"
schemaLocation="eclipselink-dbws-schema.xsd"/>
            <xsd:complexType name="p1ResponseType">
                <xsd:sequence>
                    <xsd:element name="result">
                        <xsd:complexType><xsd:sequence><xsd:any/></xsd:sequence></
xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="p1RequestType">
                <xsd:sequence>
                    <xsd:element name="SIMPLARRAY" type="ns1:SOMEPACKAGE_TBL1"/>
                    <xsd:element name="FOO" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="p2ResponseType">
                <xsd:sequence>
                    <xsd:element name="result">
                        <xsd:complexType><xsd:sequence><xsd:any/></xsd:sequence></
xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="p2RequestType">
                <xsd:sequence>
                    <xsd:element name="SIMPLARRAY" type="ns1:SOMEPACKAGE_TBL1"/>
                    <xsd:element name="FOO" type="xsd:string"/>
                    <xsd:element name="BAR" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:element name="p2" type="tns:p2RequestType"/>
            <xsd:element name="p1" type="tns:p1RequestType"/>
            <xsd:element name="p1Response" type="tns:p1ResponseType"/>
            <xsd:element name="p2Response" type="tns:p2ResponseType"/>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="p2Request"><wsdl:part name="p2Request"
element="tns:p2"/></wsdl:message>
    <wsdl:message name="p2Response"><wsdl:part name="p2Response"
element="tns:p2Response"/></wsdl:message>
    <wsdl:message name="p1Request"><wsdl:part name="p1Request"
element="tns:p1"/></wsdl:message>
    <wsdl:message name="p1Response"><wsdl:part name="p1Response"
element="tns:p1Response"/></wsdl:message>
    <wsdl:portType name="plsqloverloadService_Interface">
        <wsdl:operation name="p2">
            <wsdl:input message="tns:p2Request"/>
            <wsdl:output message="tns:p2Response"/>
        </wsdl:operation>
        <wsdl:operation name="p1">
            <wsdl:input message="tns:p1Request"/>
            <wsdl:output message="tns:p1Response"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="plsqloverloadService_SOAP_HTTP"
type="tns:plsqloverloadService_Interface">
        <soap:binding style="document"
```

```
transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="p2">
            <soap:operation soapAction="urn:plsqloverloadService:p2"/>
            <wsdl:input><soap:body use="literal"/></wsdl:input>
            <wsdl:output><soap:body use="literal"/></wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="p1">
            <soap:operation soapAction="urn:plsqloverloadService:p1"/>
            <wsdl:input><soap:body use="literal"/></wsdl:input>
            <wsdl:output><soap:body use="literal"/></wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="plsqloverloadService">
        <wsdl:port name="plsqloverloadServicePort"
binding="tns:plsqloverloadService_SOAP_HTTP">
            <soap:address location="REPLACE_WITH_ENDPOINT_ADDRESS"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

3

Advanced Methods of Accessing DBWS Designtime API

This chapter contains the following sections:

- Using DBWSBuilder with Ant
- Using with javac

Using DBWSBuilder with Ant

With EclipseLink DBWS, you can invoke the `DBWSBuilder` from Apache Ant (`http://ant.apache.org/`) to generate the necessary files, compile, and package the application with additional Ant targets.

Example

This example illustrates how to use Ant to generate a deployable web archive. For this example, consider the following file layout:

`<example-root>`

`dbws-builder.xml` (see Example 3–1)
`build.xml` (see Example 3–2)
`build.properties` (see Example 3–3)

`jlib`
`eclipselink.jar`
`eclipselink-dbwsutils.jar`
`javax.servlet.jar`
`javax.wsdl.jar`
`ojdbc6.jar`
`org.eclipse.persistence.oracleddlparser.jar`

`stage`
All generated artifacts will saved here, most importantly `simpletable.war`.

To run the DBWS builder in this example, simply type `ant` in the *`<example-root>`* directory. The builder packages the generated artifacts into the web archive (`simpletable.war`) in the `stage` directory. This `.war` file can then be deployed to WebLogic.

Example 3–1 Sample DBWS Builder File (dbws-builder.xml)

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

```
<dbws-builder xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <properties>
    <property name="projectName">simpletable</property>
    ... database properties
  </properties>

  <table
    schemaPattern="SCOTT"
    tableNamePattern="SIMPLETABLE"
  />
</dbws-builder>
```

Example 3–2 Sample Build XML File (build.xml)

```
<?xml version="1.0"?>
<project name="simpletable" default="build">
  <property file="${basedir}/build.properties"/>

  <path id="build.path">
    <fileset
      dir="${jlib.dir}"
      includes="eclipselink.jar
                eclipselink-dbwsutils.jar
                org.eclipse.persistence.oracleddlparser.jar
                javax.wsdl.jar
                javax.servlet.jar
                ojdbc6.jar"
      >
    </fileset>
  </path>

  <target name="build">
    <java
      classname="org.eclipse.persistence.tools.dbws.DBWSBuilder"
      fork="true"
      classpathRef="build.path"
      >
      <arg line="-builderFile ${dbws.builder.file} -stageDir ${stage.dir}
-packageAs ${server.platform} ${ant.project.name}.war"/>
    </java>
  </target>
</project>
```

Example 3–3 Sample Build Properties File (build.properties)

```
custom = true
build.sysclasspath=ignore

stage.dir=${basedir}/stage
jlib.dir=${basedir}/jlib
server.platform=wls
dbws.builder.file=dbws-builder.xml
```