BEA Liquid Data for WebLogic™

Application Developer's Guide

Version 8.1
Document Date: July 2003
Revised: December 2003

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

1. Application Development with the Liquid Data API

2. Using Liquid Data Controls to Develop Workshop Applications

3. Invoking Queries in EJB Clients

4. Invoking Queries in JSP Clients

9. Setting Complex Parameter Types

10.Using the Cache Purging APIs

Index

About This Document

This document describes how to use the BEA Liquid Data for WebLogic Workshop Control, EJB API, and JSP tag library.

This following topics are covered:

- Chapter 1, "Application Development with the Liquid Data API," describes concepts that you'll need to understand in order to invoke Liquid Data queries programmatically.
- Chapter 2, "Using Liquid Data Controls to Develop Workshop Applications," describes how to use the Liquid Data control to develop applications that run Liquid Data queries.
- Chapter 3, "Invoking Queries in EJB Clients," describes how to invoke Liquid Data queries from EJB clients.
- Chapter 4, "Invoking Queries in JSP Clients," describes how to invoke Liquid Data queries from JSP clients.
- Chapter 5, "Invoking Queries in Web Service Clients," describes how to invoke Liquid Data queries as Web service clients that access Web services that were generated using the Liquid Data node in the Administration Console.
- Chapter 6, "Invoking Queries in WebLogic Integration Business Processes," describes how to invoke Liquid Data queries in WebLogic Integration Business Processes.
- Chapter 7, "Invoking Queries in BEA WebLogic Portal Applications," describes how to invoke Liquid Data queries in BEA WebLogic Portal applications.

- Chapter 8, "Using Custom Functions," describes how to write Java code for custom functions that extend the power and functionality of Liquid Data.
- Chapter 9, "Setting Complex Parameter Types," describes how to write Java code which accesses streaming XML data in a Liquid Data query.
- Chapter 10, "Using the Cache Purging APIs," describes how to write Java code to purge the entries in the Liquid Data query cache.

What You Need to Know

This document is intended mainly for EJB and JSP developers responsible for developing the client-server deployment strategy for data integration applications.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the Liquid Data documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the Liquid Data documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

For more information in general about Java and XQuery, refer to the following sources.

- The Sun Microsystems, Inc. Java site at:

  `http://java.sun.com/`

- The World Wide Web Consortium XML Query section at:

  `http://www.w3.org/XML/Query`

For more information about BEA products, refer to the BEA documentation site at:

`http://edocs.bea.com/`

Contact Us!

Your feedback on the BEA Liquid Data documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the Liquid Data documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA Liquid Data for WebLogic 8.1 release.

If you have any questions about this version of Liquid Data, or if you have problems installing and running Liquid Data, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

Convention	Item
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[ ]	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
. . .	Indicates one of the following in a command line: • That an argument can be repeated several times in a command line • That the statement omits additional optional arguments • That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER 1

Application Development with the Liquid Data API

This chapter describes types of applications you can build with Liquid Data and the tools available to application developers. It also describes concepts relevant to the using the using the BEA Liquid Data for WebLogic Query API to invoke queries. It contains the following sections:

- Types of Application Development
- About Liquid Data Queries
- Components of the Liquid Data Query API
- Types of Java Clients

For reference information about the Liquid Data Query API, see the Liquid Data Javadoc. For an introduction to the XQuery standard, see "Liquid Data Implements the XQuery Standard" in "Liquid Data Concepts" in the *Concepts Guide*.

Types of Application Development

Developers can display data from Liquid Data queries in a wide variety of applications. Liquid Data includes an application programming interface (API) to access Liquid Data queries. You can develop Liquid Data applications using the following application development models:

- WebLogic Workshop Development
- EJB Development
- JSP Tag Library Development

WebLogic Workshop Development

WebLogic Workshop includes a rich integrated development environment (IDE) designed for rapidly building J2EE applications. The workshop IDE includes a wide array of application development tools such as Java controls for accessing enterprise resources, NetUI tag libraries to display enterprise data in the applications, and drag-and-drop functionality to make it easy to assemble rich and scalable application.

Liquid Data includes a Java Control Extension for WebLogic Workshop. The Liquid Data Control allows application developers using Workshop to easily access data from Liquid Data. The Liquid Data Control also generates an XMLBean interface to the data. Developers can then use the XMLBean interface to rapidly create rich user interfaces to the data returned from a Liquid Data query.

For details on using the Liquid Data Control, see Chapter 2, "Using Liquid Data Controls to Develop Workshop Applications."

EJB Development

EJB clients are any applications that invoke queries on the Liquid Data Server using the Liquid Data EJB API. All Java clients can leverage the flexibility and the powerful data integration properties offered by XQuery in order to meet their data access needs. All these types of clients access the EJB remote interfaces directly, therefore they can be collectively characterized as *EJB clients*. For more information about EJB clients, see Chapter 3, "Invoking Queries in EJB Clients."

Note: A special kind of EJB client is the Data View Builder itself, which may be used by data architects and developers to build and execute queries.

JSP Tag Library Development

In addition to the procedural Liquid Data API, JSP clients, in particular, may use the Liquid Data Server tag library, which provides a declarative way to extend their querying and data access capabilities. The Liquid Data Server tag library is typically deployed within the web application that contains the JSP clients. The declarative nature of the tag library makes it simpler for JSP clients to issue stored or ad hoc, fixed or parameterized, queries. These JSP clients form a second family of API clients, collectively characterized as *tag library clients.* For more information about JSP clients, see Chapter 4, "Invoking Queries in JSP Clients."

About Liquid Data Queries

This section describes the following Liquid Data query concepts:

- Stored Queries
- Ad Hoc Queries
- Parameterized Queries

For more information about Liquid Data queries, see "Key Concepts of Query Building" in "Overview and Key Concepts" in *Building Queries and Data Views*.

Stored Queries

Stored queries have been predefined by the personnel (typically data architects) of the organization that operates the Liquid Data Server. Stored queries are assigned a unique name starting with an alphabetic character (A-Z a-z) and reside in the Liquid Data server repository. Clients may execute stored queries by merely specifying their name and parameters, if any. For more information about the server repository, see "Managing the Liquid Data Repository" in the Liquid Data *Administration Guide*.

Ad Hoc Queries

An *ad hoc query* is a query that has not been stored in the Liquid Data repository as a stored query but rather is passed to the Liquid Data server on the fly. Ad hoc queries are defined by the client. In effect, clients need to provide the actual content of ad hoc queries to the server at run time.

Parameterized Queries

Although queries may return results that are of general interest, it is often the case that the content of query results, and therefore also the content of the query, needs to be customized in order to better fit the client's needs. This requirement is commonly addressed through the use of *parameterized queries*, which are queries that allow for substitution of parts of the query with parameters whose value can be provided (and changed) per query execution.

The Liquid Data Server API provides support for parameterized queries using named parameters. When parameterized queries are used, clients need to provide the value and the type of each named parameter in the query.

Components of the Liquid Data Query API

This section describes the components of the Liquid Data Query API. It contains the following sections:

- Packages
- Query Execution EJB
- Query Parameters
- Query Attributes
- Query Results

For reference information about the Liquid Data Query API, see the Liquid Data Javadoc.

Packages

The Liquid Data API includes the following packages:

Table 1-1 Packages in the Liquid Data Query API

Package Name	Description
`com.bea.ldi.server`	Defines the Liquid Data query execution EJBs, including their home and remote interfaces.
`com.bea.ldi.server.common`	Defines interfaces and classes for query parameters, query results, query result exceptions, and attributes for query evaluation.

Query Execution EJB

The `com.bea.ldi.server` package defines the following stateless session bean:

`bea.ldi.server.QueryBean`

The `com.bea.ldi.server` package also defines the home and remote interfaces for this EJB. The query execution EJB, along with the Liquid Data Server, can be deployed in a cluster.

Query Parameters

The `com.bea.ldi.server.common.QueryParameters` class represents parameters that are specified for parameterized queries prior to query execution. In addition to Java primitive types

(byte, float, int, long, short, and double) that you can specify using setxxxx() methods, query parameters can be any of the following types:

- java.lang.Boolean
- java.lang.Byte
- java.lang.Double
- java.lang.Float
- java.lang.Integer
- java.lang.Long
- java.lang.Short
- java.lang.String
- java.math.BigDecimal
- java.math.BigInteger
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

The QueryParameters class provides methods for setting parameters based on these types as well as a getParameters() method that collects defined query parameters in a java.util.Map object.

Query Attributes

The com.bea.ldi.server.common.QueryAttributes interface provides a variable (LARGE_DATA) that specifies whether the query is expected to produce a large final result set or large intermediate result sets.

Query Results

The com.bea.ldi.server.common.QueryResult interface represents the results of a query. The QueryResult interface provides methods for retrieving the query results, expressed in XML, as a DOM document (org.w3c.dom.Document), determining whether the query result is empty, printing the query results as XML to a specified device, and deallocating local and server resources.

Types of Java Clients

Any authorized Java client can use Java Naming and Directory Interface (JNDI) to obtain references to the EJBs and use them to issue queries against the Liquid Data Server.

Different types of Java clients include:

- Java applications created using WebLogic Workshop and Liquid Data controls
- Standalone Java applications
- Java servlets
- Java Server Pages (JSPs)
- Java Beans
- Other EJBs
- Business operations in business processes that execute in a WebLogic Integration Business Process
- WebLogic Portal
- Web Services

Both local and remote clients can access the Liquid Data Query API.

CHAPTER 2

Using Liquid Data Controls to Develop Workshop Applications

This chapter describes how to use the Liquid Data control in WebLogic Workshop to develop applications that use data from Liquid Data queries. Applications can use the data to display results in a web application, to use in a Web Service, to use as an input to a WebLogic Integration business process, or in many other ways. The following topics are included:

WebLogic Workshop and Liquid Data

WebLogic Workshop allows you to create Liquid Data Controls. The Liquid Data Control is a Java Control, and it allows you to very rapidly generate robust applications that use results from Liquid Data queries (for example, to display in a web application or to use in a WebLogic Integration business process). This section describes the Liquid Data control and the applications you can create with it.

Liquid Data Control

The Liquid Data Control is available in WebLogic Workshop. The Liquid Data Control is an extensible Java Control that accesses the Liquid Data server to execute queries from applications developed in WebLogic Workshop. The Liquid Data Control is available with all of the other Java Controls in WebLogic Workshop (for example, the database control). When you use the Liquid Data Control in WebLogic Workshop, Workshop displays a query wizard which connects to a Liquid Data server to get the query metadata needed for configuring the control. After you select the queries to use in your Liquid Data Control, Workshop generates `XMLBean` classes for the target schemas associated with the queries and then generates a Liquid Data Control (`.jcx`) file.

XMLBean Generation

When you create a Liquid Data control in WebLogic Workshop, the Liquid Data Control wizard generates `XMLBean` classes for each query in the control. The Liquid Data Control wizard uses the schema associated with the stored query in the Liquid Data repository to generate the structure for the `XMLBean` classes. The `XMLBean` classes provide Java methods to traverse the XML result set returned from Liquid Data.

The `XMLBean` classes are automatically generated in a schema project in the workshop application. There is one schema project per Liquid Data Control (`.jcx`) file.

Use With Page Flow, Web Services, Business Processes

You can use the Liquid Data Control like other controls in WebLogic Workshop, and you can take advantage of Workshop features to use Liquid Data Controls in Web Services, Page Flows and WebLogic Integration business processs. For example, you can generate a page flow from your Liquid Data control and then use the `XMLBeans` to bind the data returned from Liquid Data to the JSPs in your application.

Liquid Data Control JCX File

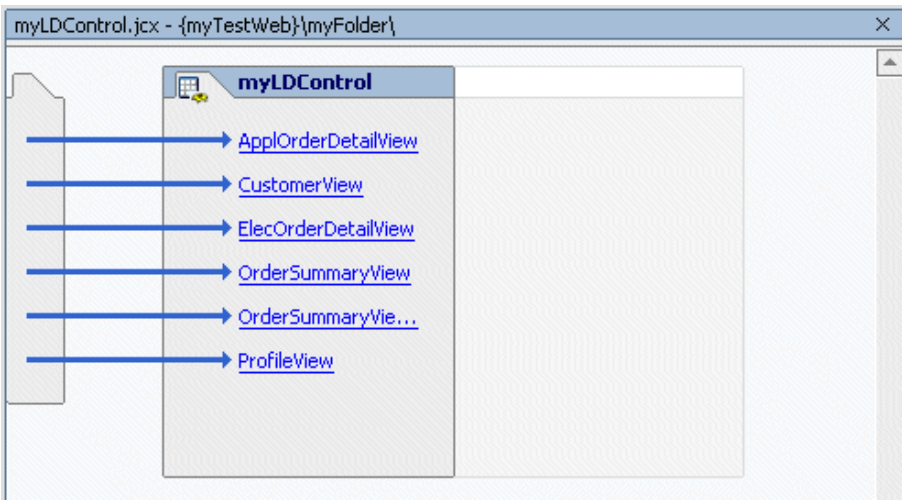
When you create a Liquid Data Control, WebLogic Workshop generates a Java Control Extension (`.jcx`) file. The file contains methods corresponding to the queries in which the control accesses, shows the schema files of each query as a comment, and contains a commented method which, when uncommented, allows you to pass any XQuery statement to execute an ad-hoc query. This section describes the Liquid Data Control (`.jcx`) file and includes the following sections:

- Design View
- Source View
- Schema Project Location
- Running Ad-Hoc Queries through the Liquid Data Control

Design View

The design view of the Liquid Data Control (`.jcx`) file shows the available methods in a graphical view.

Figure 2-1 Design View of a Control File



With the right-click menu, you can add, modify (for example, change the query accessed by a method), rename, and delete methods. The right-click menu is context-sensitive; it displays different items if the mouse cursor is over a method, or in the control portion of the design pane.

Source View

The source view shows the source code of the Java Control Extension (`.jcx`) file. It includes as comments the schema used to generate the `XMLBean` classes for each query. The signature for each method shows the return type of the method. The return type is the `XMLBean` class which was generated for the schemas.

This file is a generated file and the only time you should need to edit the source code is if you want to add a method to run an ad-hoc query, as described in "Running Ad-Hoc Queries through the Liquid Data Control" on page 2-7.

The following shows the source code for a generated Liquid Data Control (`.jcx`) file. It shows the package declaration, import statements, connection properties, the schema project and filename used with the `ApplOrderDetailView` query, and the method that executes the `ApplOrderDetailView` query.

```
package myFolder;

 import weblogic.jws.control.*;
 import com.bea.ld.control.LDControl;

 /**
 *   @jc:LiquidData urlKey="myApp.myAppWeb.myFolder.anotherLDControl"
 */
public interface anotherLDControl extends LDControl,
com.bea.control.ControlExtension
 {

    /* Generated methods corresponding to stored queries.
     */

    /**
     * @return Schema Project: myAppWeb-myFolder-anotherLDControl-Schemas
           Schema File: rtl\OrderDetailView.xsd
           TargetNamespace: urn:retailer
           Element Name: OrderDetailView
     *
     * @param orderid java.lang.String
     *
     * @param custid java.lang.String
```

```
     *
     * @jc:Stored-Query Name="rtl.ElecOrderDetailView"
     */
    retailer.OrderDetailViewDocument ElecOrderDetailView
                    (java.lang.String orderid, java.lang.String custid);

    /**
     *  Default method to execute an ad hoc query.
     *  This method can be customized to have a differnt method name
     *  (e.g. runMyQuery), return a XML Bean class (e.g. Customer),
     *  or to have one or both of the following two extra parameters:
     *  com.bea.ldi.server.common.QueryParameters and
     *  com.bea.ldi.server.common.QueryAttributes
     *  e.g. exec(String query, QueryParameters params);
     *  e.g. exec(String query, QueryAttributes attrs);
     *  e.g. exec(String query, QueryParameters params,
     *  QueryAttributes attrs);
     *
    com.bea.xml.XmlObject executeXQuery(String query);
     */

}
```

Schema Project Location

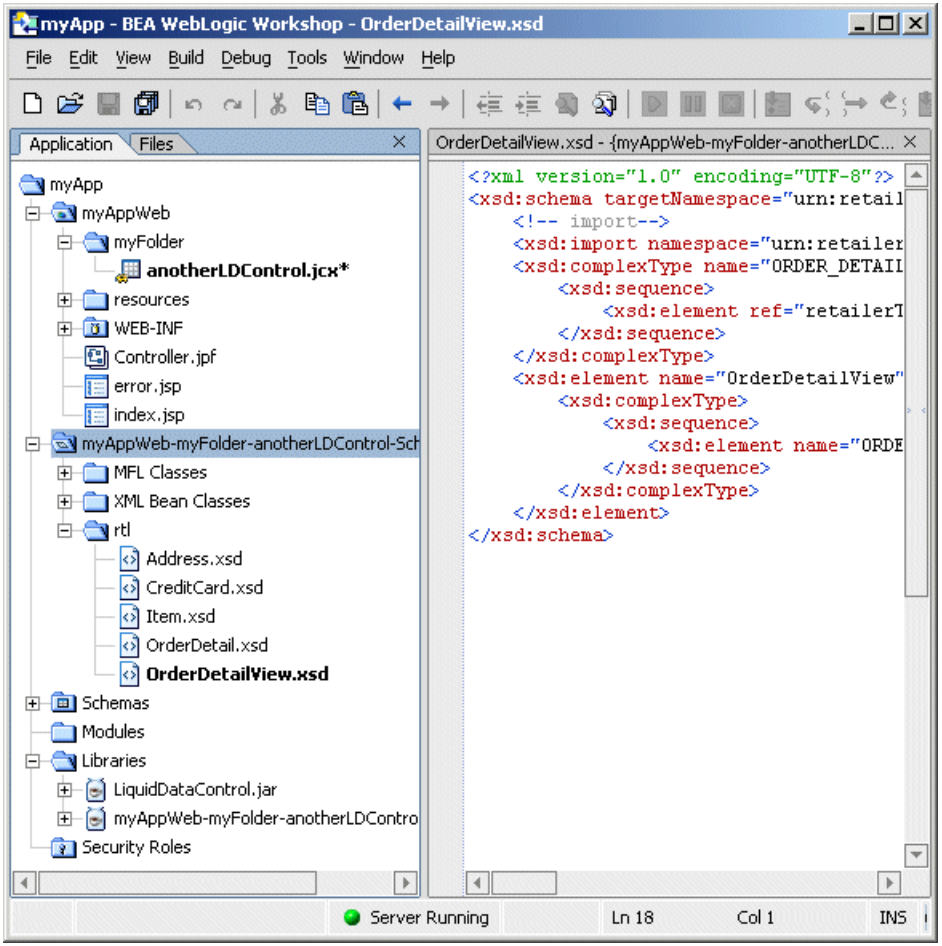
The `XMLBean` classes corresponding to the queries in the Liquid Data Control (`.jcx`) file are generated in a schema project. There is one schema project for each control. The schema project(s) also contain a copy of the schema files associated with the queries in the Liquid Data Control (`.jcx`) file. The JAR file for the `XMLBean` classes is generated in the `Libraries` directory of your WebLogic Workshop application.

The `@return Schema Project` section of the generated Liquid Data Control (`.jcx`) file displays the location of the schemas corresponding to the query method below this section in the control file. For example, the following code snippet from a generated Liquid Data Control (`.jcx`) file shows the name of the schema project and the name of the target schema.

```
/**
     * @return Schema Project: myAppWeb-myFolder-anotherLDControl-Schemas
           Schema File: rtl\OrderDetailView.xsd
           TargetNamespace: urn:retailer
           Element Name: OrderDetailView
     *
     * @param orderid java.lang.String
     *
     * @param custid java.lang.String
     *
     * @jc:Stored-Query Name="rtl.ElecOrderDetailView"
     */
    retailer.OrderDetailViewDocument ElecOrderDetailView
                    (java.lang.String orderid, java.lang.String custid);
```

The name of the schema project is `myAppWeb-myFolder-anotherLDControl-Schemas`, and the schema file is in the `rtl` subdirectory and is named `OrderDetailView.xsd`. Figure 2-2 shows the generated schema project in WebLogic Workshop.

Figure 2-2 Generated Schema Project in WebLogic Workshop



Running Ad-Hoc Queries through the Liquid Data Control

At the bottom of the generated Liquid Data Control (`.jcx`) file is a comment showing methods you can add which allow you to run an ad-hoc query through the control. To add one of these methods, uncomment the appropriate method and add a return type to the signature.

```
/**
*  Default method to execute an ad hoc query. This method can be customized
* to have a differnt method name (e.g. runMyQuery), return a XML Bean class
```

```
* (e.g. Customer), *  or to have one or both of the following two extra
* parameters: com.bea.ldi.server.common.QueryParameters and
* com.bea.ldi.server.common.QueryAttributes
*  e.g. exec(String query, QueryParameters params);
*  e.g. exec(String query, QueryAttributes attrs);
*  e.g. exec(String query, QueryParameters params, QueryAttributes attrs);
*             com.bea.xml.XmlObject executeXQuery(String query); */
```

Creating Liquid Data Controls

You can create Liquid Data controls in a variety of WebLogic Workshop projects. This section includes the following procedures to create Liquid Data controls:

- General Steps to Create a Liquid Data Control
- To Create a Liquid Data Control in a Web Project
- To Create a Liquid Data Control in a Web Service Project
- To Add a Liquid Data Control to an Existing Web Service File
- To Create a Test Web Service From a Liquid Data Control

The steps are similar for creating Liquid Data controls in other types of WebLogic Workshop projects.

General Steps to Create a Liquid Data Control

This section describes the general steps for creating a Liquid Data control. For detailed steps for creating a Liquid Data control in a Web Project or in a Web Service project, see "To Create a Liquid Data Control in a Web Project" on page 2-14 or "To Create a Liquid Data Control in a Web Service Project" on page 2-15.

Step 1: Create a Project in an Application

Before you can create a Liquid Data control in WebLogic Workshop, you must create an application and create a project in the application. You can create a Liquid Data control in most types of Workshop projects, but the most common projects in which you create Liquid Data controls are Web Projects, Web Service Projects, Portal Web Projects, or a Process Web Projects.

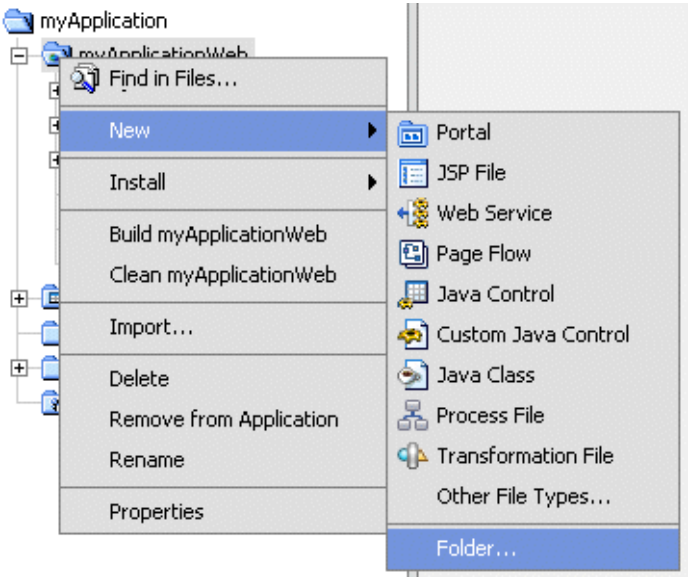
Step 2: Start the Liquid Data Server, If It is Not Already Running

Make sure the Liquid Data server is running. The Liquid Data server can be running locally (on the same domain as WebLogic Workshop) or remote (on a different domain from workshop). If the Liquid Data server is not running, start up the domain in which it runs.

Step 3: Create a Folder in a Project

Create a folder in the project to hold the Liquid Data control(s). You can also create other controls (database controls, for example) in the same folder, if needed. Workshop controls cannot be created at the top-level of a project directory structure; they must be created in a folder. When you create the folder, enter a name that makes sense for your application.

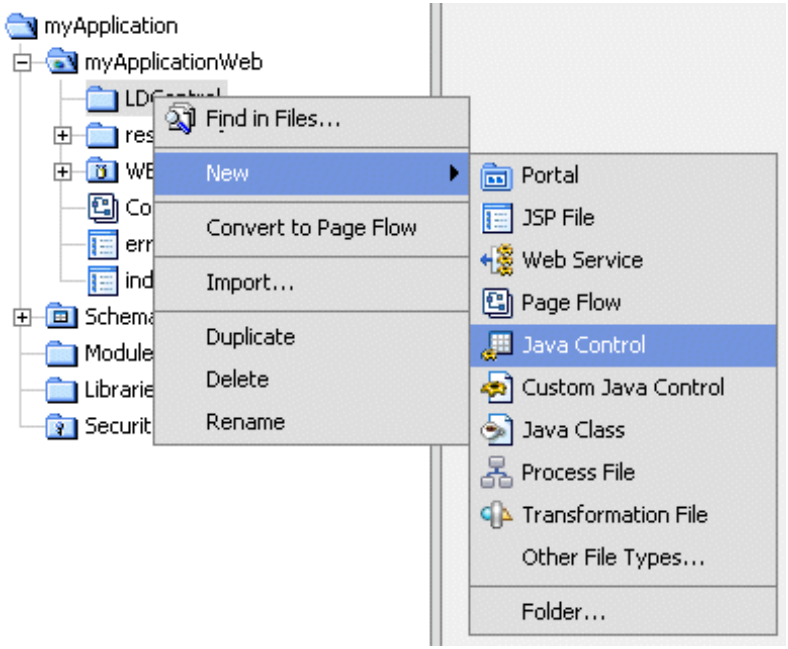
Figure 2-3 Create a New Folder in WorkshopLiquid Data



Step 4: Create the Liquid Data Control

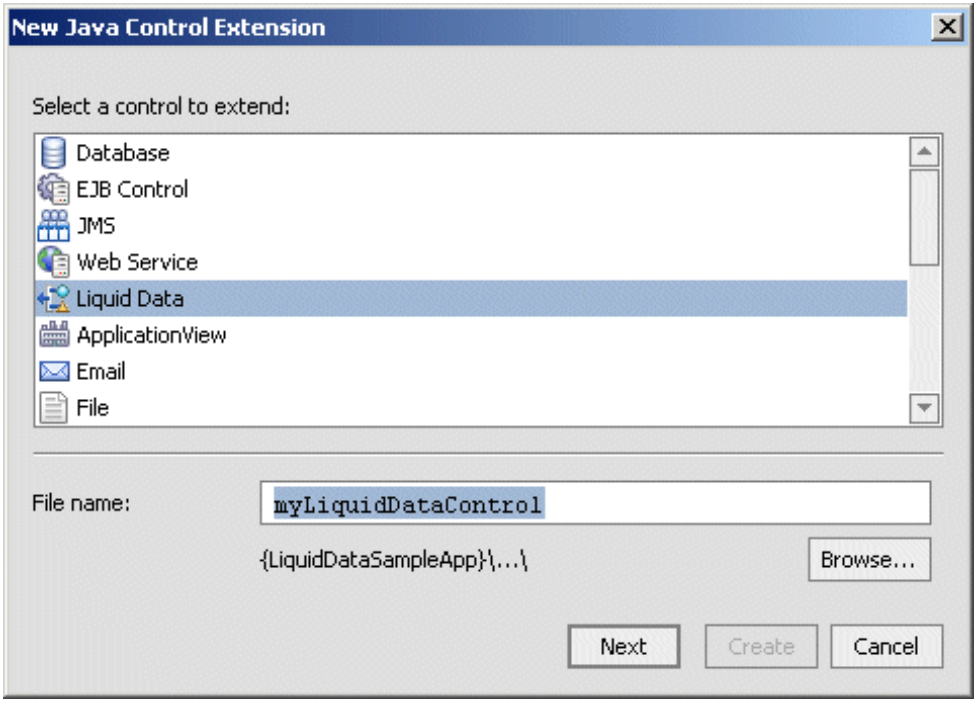
The Liquid Data Control is a Java Control Extension. To create a Liquid Data Control, start the Java Control wizard by selecting your folder within a project, right-clicking, and selecting New —> Java Control, as shown in Figure 2-4. You can also create a control using the File —> New—> Java Control menu item.

Figure 2-4 Create a New Liquid Data Control



Then select Liquid Data from the New Java Control Extension dialog, as shown in Figure 2-5. Enter a filename for the control (`.jcx`) file and click Next.

Figure 2-5 Liquid Data Control in WebLogic Workshop



Note: The `LiquidDataControl.jar` file is copied into the `Libraries` directory (if it does not already exist) of your application when you create a Liquid Data Control.

Step 5: Enter Connection Information to the Liquid Data Server

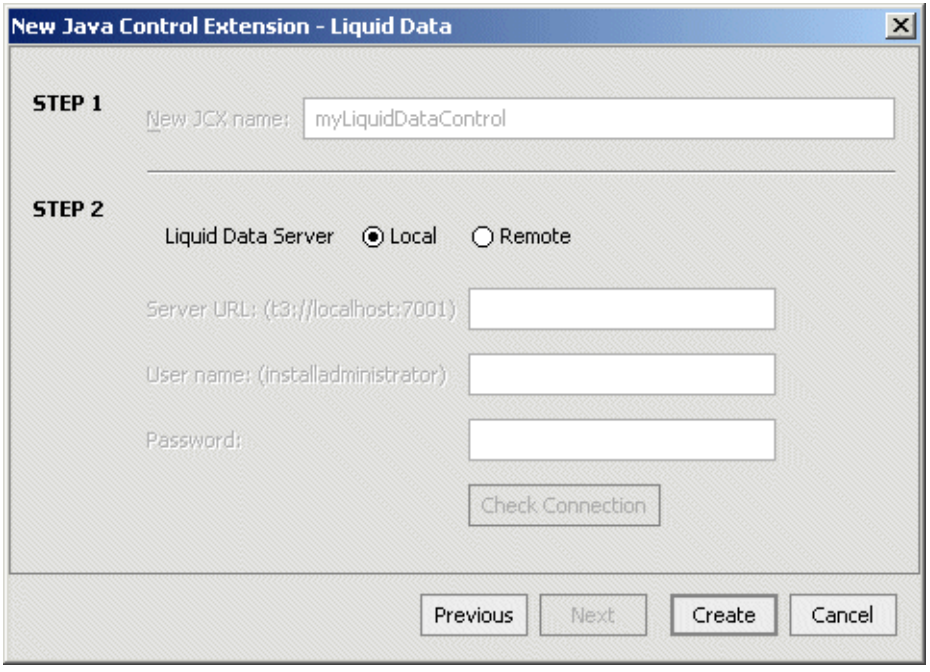
A screen similar to the one in Figure 2-6 allows you to enter connection information to your Liquid Data server. If the server is local, the Liquid Data control uses the connection information stored in the application properties (to view these settings, access the Tools —> Application Properties menu item in the IDE).

If the Liquid Data server is remote, click the Remote button and fill in the appropriate server URL, user name, and password.

Note: You can specify a different username and password with which you connect to a local machine on the Liquid Data Control Wizard Connection Information dialog, too. To do this, click the Remote button and enter the connection information (with a different username and password) for your local machine. The security credentials specified through the Application Properties or through the Liquid Data Control Wizard are only used for creating the Liquid Data Control (`.jcx`) file, not for testing queries through the control. For more details, see "Security Considerations With Liquid Data Controls" on page 2-34.

When the information is correct, click Create to go to the next step.

Figure 2-6 Liquid Data Control Wizard—Connection Information



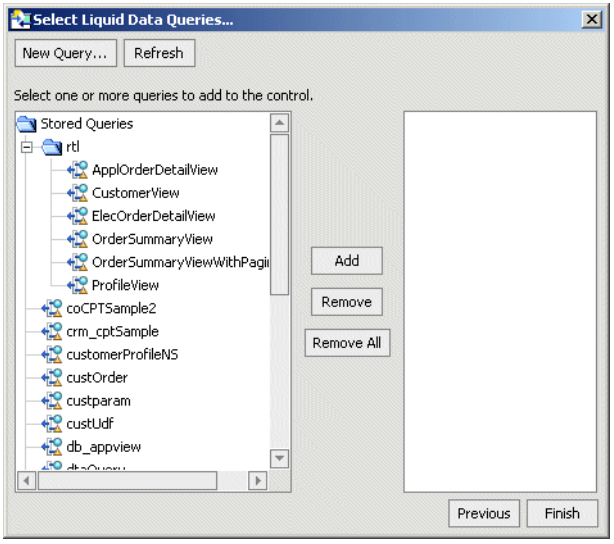
Step 6: Select Queries to Add to the Control

In the Select Liquid Data Queries screen, select queries from the left pane and click Add to add those queries to the control. If you mouse over a query, the signature of the control method for the query appears in a tooltip popup. A "fetching metadata" message appears if the signature has not yet been retrieved from the Liquid Data server.

Note: Only stored queries with a schema configured appear in the Stored Queries list. For details on configuring stored queries, see "Configuring Stored Queries" in the *Administration Guide*. You can also deploy stored queries from directly from the Data View Builder, as described on "Deploying a Query" in *Building Queries and Data Views*.

Select one or more queries, add them to the right pane, and click Finish. When you click Finish, the Liquid Data Control (`.jcx`) file is generated and `XMLBean` classes corresponding to the schema for each stored query in the control are generated. The `XMLBeans` are stored in the `Libraries` directory of the Workshop Application. In the `Libraries` directory, there is one JAR file for each Liquid Data control, with the `XMLBeans` included in the JAR file. The JAR files are named according to the project and directory hierarchy for the control (`.jcx`) file.

Figure 2-7 Liquid Data Control Wizard—Select Queries



Note: The stored queries should be named according to the "Naming Conventions for Stored Queries" described in *Building Queries and Data Views*. If a stored query contains illegal characters (for example, a hyphen), the method generated in the Liquid Data Control (`.jcx`) file might be an invalid Java name, causing compilation errors. If a method name is invalid, you can change the name to make it valid.

New Query

Clicking the New Query button launches the Data View Builder. You can then use the Data View Builder to create, modify, test, and deploy new queries.

Refresh

The Refresh button updates the stored query list from the Liquid Data server. If you create and deploy a new query with the Data View Builder, click the Refresh button to display the new query in the wizard.

To Create a Liquid Data Control in a Web Project

This section describes the basic steps for creating a Liquid Data control in a new Web Project. If you are adding the control to an existing project, you might not need to perform each step (for example, creating a new project). Perform the following steps to create a Liquid Data control in a new WebLogic Workshop Web Project.

1. Make sure your Liquid Data domain is running.
2. Start WebLogic Workshop.
3. Either open an existing Workshop application or create a new application (File —> New —> Application).
4. Select the top-level folder of your Workshop application, right-click, and select New —> Project.
5. Select Web Project as the type of project, enter a name, and click Create.
6. Create a folder in your Web Project. To create the new folder, select the project folder you just created, right-click, and select New —> Folder (see Figure 2-3). Enter a name for the folder and click OK.
7. Select the new folder, right-click, and select New —> Java Control (see Figure 2-4).
8. In the New Java Control Extension wizard, select Liquid Data as the control type, enter a name, and click Next (see Figure 2-5).

9. If your Liquid Data server is local to your machine, accept the default and click Create (see Figure 2-6). If Liquid Data is running on a remote server, click the Remote button, enter your connection information, test your connection, and click Create.
10. In the Edit Liquid Data Control - Select Queries screen, select any queries you want accessible to your control from the left pane and click Add to add them to the right pane (see Figure 2-7).
11. If you want to create any new queries, click the Create New Query button to launch the Data View Builder, where you can create, test, and deploy queries.
12. If you have added any Liquid Data queries, click Refresh to display the new queries.
13. After you have added all the queries you need in the wizard, click Finish.

    Workshop generates the `.jcx` Java Control Extension file for your Liquid Data control. Each method in the `.jcx` file returns an `XMLBean` type corresponding to the stored query schema. The `XMLBean` classes for each query are automatically generated when you create the Liquid Data control. The `XMLBean` classes are stored in the `Libraries` directory of the Workshop Application.

To Create a Liquid Data Control in a Web Service Project

This section describes the basic steps for creating a Liquid Data control in a new Web Service. If you are adding the control to an existing Web Service, you might not need to perform each step (for example, creating a new project). Perform the following steps to create a Liquid Data control in a new WebLogic Workshop Web Service Project.

1. Make sure your Liquid Data domain is running.
2. Start WebLogic Workshop.
3. Either open an existing Workshop application or create a new application (File —> New —> Application).
4. Select the top-level folder of your Workshop application, right-click, and select New —> Project.
5. Select Web Service Project as the type of project, enter a name, and click Create.
6. Create a folder in your Web Service Project. To create the new folder, select the project folder you just created, right-click, and select New —> Folder (see Figure 2-3). Enter a name for the folder and click OK.
7. Select the new folder, right-click, and select New —> Java Control (see Figure 2-4).

8. In the New Java Control Extension wizard, select Liquid Data as the control type, enter a name, and click Next (see Figure 2-5).
9. If your Liquid Data server is local to your machine, accept the default and click Create (see Figure 2-6). If Liquid Data is running on a remote server, click the Remote button, enter your connection information, test your connection, and click Create.
10. In the Edit Liquid Data Control - Select Queries screen, select any queries you want accessible to your control from the left pane and click Add to add them to the right pane (see Figure 2-7).
11. If you want to create any new queries, click the Create New Query button to launch the Data View Builder, where you can create, test, and deploy queries.
12. If you have added any Liquid Data queries, click Refresh to display the new queries.
13. After you have added all the queries you need in the wizard, click Finish.

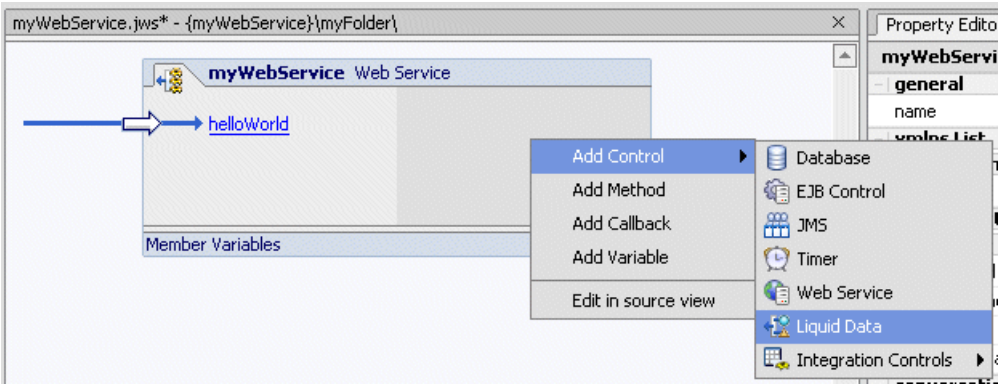
    Workshop generates the `.jcx` Java Control Extension file for your Liquid Data control. Each method in the `.jcx` file returns an `XMLBean` of the type corresponding to the schema from the stored query. The `XMLBean` for each query is automatically generated when you create the Liquid Data control. The `XMLBeans` are stored in the `Libraries` directory of the Workshop Application.

To Add a Liquid Data Control to an Existing Web Service File

Perform the following steps to add a Liquid Data Control to an existing Web Service `.jws` file.

1. Make sure your Liquid Data domain is running.
2. In WebLogic Workshop, open an existing Web Service `.jws` file.
3. Click the Design View tab on the Web Service.
4. In the graphical representation of the Web Service, right-click and select Add Control —> Liquid Data.

Figure 2-8 Add a Liquid Data Control to Web Service

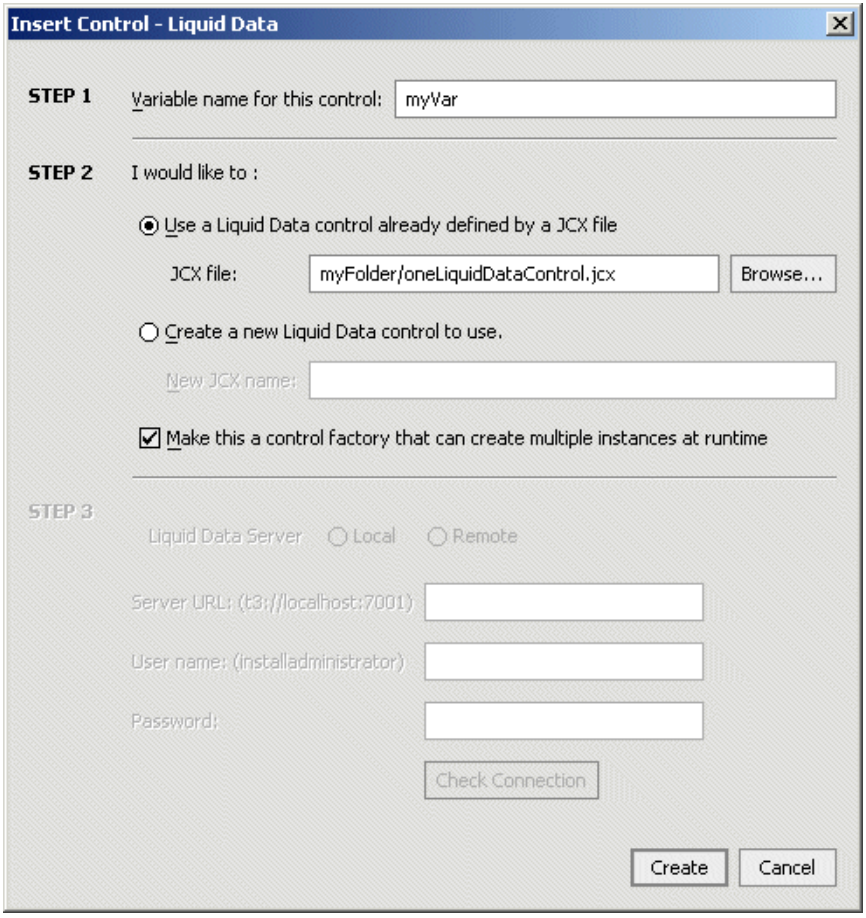


5. In the Insert Control Wizard, enter a variable name for the control (STEP 1 in the dialog in Figure 2-9). The variable name can be any valid variable name that is unique in the Web Service.

6. In the Insert Control Wizard, either browse to an existing Liquid Data Control (it must be in the same project as the Web Service) or click the Create a New Liquid Data Control button.

7. If you want the control to be a factory, check the Make This a Control Factory button. If the control is a factory, it will create multiple instances at runtime if a query is called multiple times. Otherwise, requests to the control are serialized and each request for a given query must complete before another can begin.

Figure 2-9 Insert Control Wizard



8. If your Liquid Data server is running on a separate domain from Workshop, click remote (in STEP 3 of the Insert Control Wizard dialog). For details about specifying local or remote Liquid Data server, see "Step 5: Enter Connection Information to the Liquid Data Server" on page 2-12.
9. Click the Create button on the Insert Control Wizard.
10. If you created a new control, choose the queries for your control, as described in "Step 6: Select Queries to Add to the Control" on page 2-13.

To Create a Test Web Service From a Liquid Data Control

Perform the following steps to generate and test a web service from a Liquid Data Control.

1. Select a Liquid Data Control (`.jcx`) file, right-click, and select Generate Test JWS File.

   Workshop generates the `.jws` Java Web Service file for your Liquid Data control.

2. Select your Web Service project, right-click, and select Build Project.

   Workshop builds an asynchronous Web Service from the `.jws` file.

3. When the build is complete, double-click the `.jws` file to open it.

4. On the Design View of the Web Service, notice the `startTestDrive` and `finishTestDrive` methods, as well as a method for each of the queries you specified in the Liquid Data Control wizard.

5. Click the test button (or select Debug —> Start from the Workshop menu) to test the web service.

6. Click the `startTestDrive` button to start the conversation for the Web Service.

7. Click the Continue this Conversation link (in the left corner of the test page).

8. Enter values for any query parameters (if the query has parameters) and click the button with the name corresponding to the query you want to execute.

   The Web Service executes the query and the results are returned to the test browser.

9. If you want to run the query again or run other queries in the Web Service, click Continue this Conversation, enter any needed parameters and click the button with the name corresponding to the query you want to execute.

10. To end the Web Service conversation, click the Continue this Conversation link and then click the `finishTestDrive` button.

Modifying Existing Liquid Data Controls

This section describes the ways you can modify an existing Liquid Data control. It contains the following procedures:

- To Change the Query Associated With a Single Control Method
- To Add a New Method to a Control

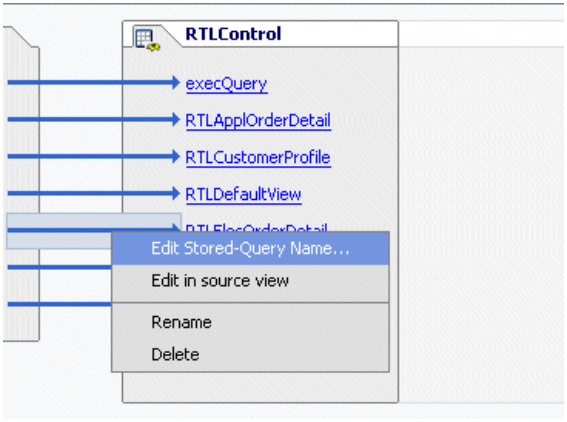
- To Invoke the Query Wizard to Modify an Existing Control
- Updating an Existing Control if Schemas Change

To Change the Query Associated With a Single Control Method

Perform the following steps to change the query that a method in a Liquid Data Control accesses.

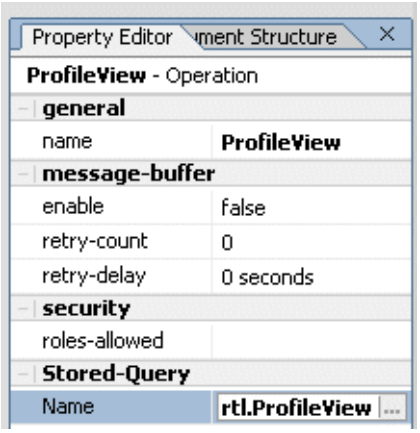
1. In WebLogic Workshop, open the Design View for a Liquid Data Control (`.jcx`) file.
2. Select the method you want to change, right-click, and select Edit Stored-Query Name to bring up the Liquid Data Control Wizard.

Figure 2-10 Changing the Query a Method Accesses



Note: You can also access the Liquid Data Control Wizard from the property editor

Figure 2-11 Opening the Property Editor from the Stored-Query Name Property



3. If you are accessing a remote Liquid Data server, enter a password on the connection information screen. If you are using a local Liquid Data server, the connection information screen does not appear.

4. In the Property editor, navigate to the query you want the method to access, select it, and click OK.

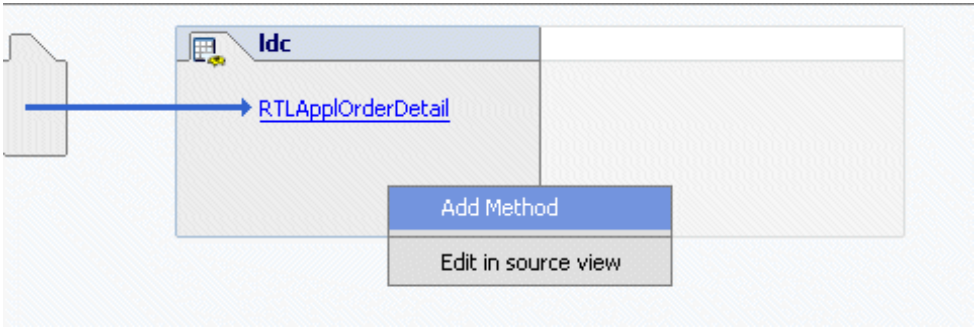
To Add a New Method to a Control

Perform the following steps to add a new method to an existing Liquid Data control.

1. In Workshop, open an existing control in Design View.

2. In the control Design View, move your mouse inside the box showing the control methods, right-click, and select Add Method, as shown in Figure 2-12.

Figure 2-12 Add a Method to a Control



3. Enter a name for the new method.
4. Move your mouse over the new method, right-click, and select Edit Stored-Query Name to launch the query wizard (see Figure 2-10).

   Alternately, you can launch the query wizard from the Stored-Query name property, as shown in Figure 2-11.
5. If you are accessing a remote Liquid Data server, enter a password on the connection information screen. If you are using a local Liquid Data server, the connection information screen does not appear.
6. In the Property editor, navigate to the query you want the method to access, select it, and click OK.

To Invoke the Query Wizard to Modify an Existing Control

You can use the query wizard to modify one or more queries accessed in an existing Liquid Data Control. A query corresponds to a method in the Liquid Data Control (`.jcx`) file. Perform the following to invoke the Liquid Data query wizard and modify the query selection for an existing Liquid Data Control.

1. In WebLogic Workshop, open the Design View for a Liquid Data Control (`.jcx`) file.
2. If you are not already viewing the Property Editor, select View —> Property Editor from the Workshop menu.
3. If you want to change any of the connection information, enter a value for the `url` or `username` attributes in the property editor.

4. In the Liquid Data section of the property editor, click the three dots (see Figure 2-13) to launch the query wizard.

Figure 2-13 Invoking the Query Wizard from the Workshop Property Editor

5. If you entered new values for the `url` or `username` attributes, or if the Liquid Data Server is in a remote domain, the Liquid Data Connection Information screen appears. Enter a password and click OK. If you did not change the value of these attributes, then the wizard opens to the Property Editor where you select queries.

6. Add or remove queries as you need in the Property Editor screen and click OK. For details, see "Step 6: Select Queries to Add to the Control" on page 2-13.

Updating an Existing Control if Schemas Change

If any of the schemas corresponding to any methods in a Liquid Data Control change, then you must update the Liquid Data Control to regenerate the `XMLBeans` for the changed schemas. Perform the following steps to update a Liquid Data Control

1. In WebLogic Workshop, open the Liquid Data Control (`.jcx`) file.

2. In the Liquid Data section of the property editor, click the three dots (see Figure 2-13) to invoke the Liquid Data query wizard.

3. If you are accessing a remote Liquid Data server, enter a password on the connection information screen. If you are using a local Liquid Data server, the connection information screen does not appear.

4. In the query wizard Property Editor, click OK. Workshop regenerates the Liquid Data Control and the `XMLBeans` for all the schemas used by queries in the control.

Using NetUI to Display Liquid Data Results

WebLogic Workshop includes NetUI, which allows you to rapidly assemble applications that display data returned from Liquid Data queries.

When you create a Liquid Data control, `XMLBean` classes are generated for the target schema of each stored query included as a method in the control. The following sections represent the basic steps for using NetUI to display results from a Liquid Data Control:

- Generating a Page Flow From a Control
- Adding a Liquid Data Control to an Existing Page Flow
- Adding XMLBean Variables to the Page Flow
- Displaying Query Results in a Table or List

Generating a Page Flow From a Control

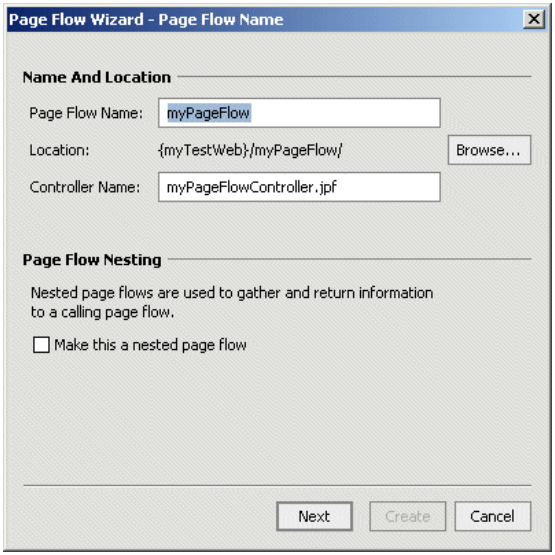
You can generate a page flow from a Liquid Data Control (`.jcx`) file. When you generate the page flow, Workshop creates the page flow, a start page (`index.jsp`), and a JSP file for each method you specify in the Page Flow wizard.

To Generate a Page Flow From a Control

Perform the following steps to generate a page flow from a Liquid Data control.

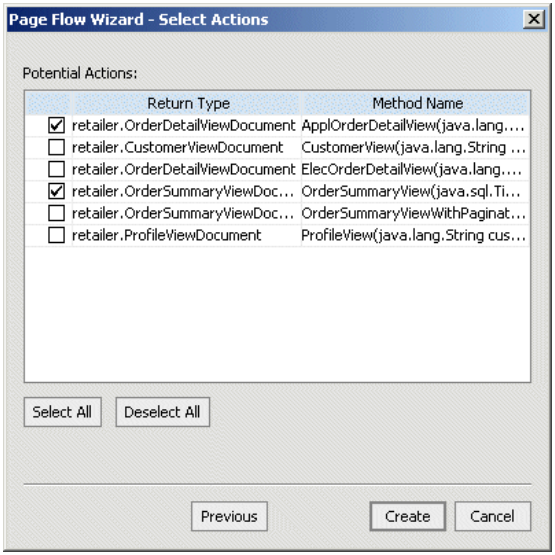
1. Select a Liquid Data Control (`.jcx`) file from the application file browser, right-click, and select Generate Page Flow.
2. In the Page Flow Wizard, enter a name for your Page Flow and click Next.

Figure 2-14 Enter a Name for the Page Flow



3. On the Page Flow Wizard - Select Actions screen, check the methods for which you want a new page created. The wizard has a check box for each method in the control.

Figure 2-15 Choose Liquid Data Methods for the Page Flow



4. Click Create.

   Workshop generates the `.jpf` Java Page Flow file, a start page (`index.jsp`), and a JSP file for each method you specify in the Page Flow wizard.

5. Add and initialize the variables to the `.jpf` file for the XMLBeans. For details, see "Adding XMLBean Variables to the Page Flow" on page 2-27.

6. Drag and drop the XMLBean variables to your JSPs to bind the data from Liquid Data to your page layout. For details, see "Displaying Query Results in a Table or List" on page 2-30.

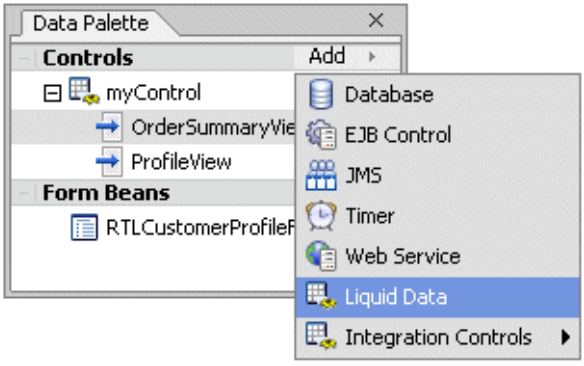
7. Build and test the application in WebLogic Workshop.

Adding a Liquid Data Control to an Existing Page Flow

You can add a Liquid Data Control to an existing Page Flow `.jpf` file. The procedure is the same as adding a Liquid Data Control to a Web Service, described in "To Add a Liquid Data Control to an Existing Web Service File" on page 2-16, except instead of opening the Web Service in Design View, you open the Page Flow `.jpf` file in Action View.

You can also add a control to an existing page flow from the Page Flow Data Palette (available in Flow View and Action View of a Page Flow), as shown in Figure 2-16.

Figure 2-16 Adding a Control to a Page Flow from the Data Palette



Adding XMLBean Variables to the Page Flow

In order to use the NetUI features to drag and drop data from an `XMLBean` into a JSP, you must first create one or more variables in the page flow `.jpf` file. The variables must be of the `XMLBean` type corresponding to the schema associated with the query. If you create a single variable at the top level of the `XMLBean` class (the same as the return type of the method in the Liquid Data Control (`.jcx`) file), the `NetUI` repeater wizard can then access all the data from the query.

Defining a single variable in the page flow `.jpf` file for the top-level class of the `XMLBean` (the same as the return type of the method in the Liquid Data Control (`.jcx`) file) provides you access to all the data from the query (through the NetUI repeater wizard). When you create the Liquid Data control and the `XMLBeans` are generated, the `XMLBean` generation defines an array for each element in the schema that is repeatable. You might want to add other variables corresponding to other arrays in the `XMLBean` classes to make it more convenient to drag and drop data onto a JSP, but it is not required. Define each variable with a type corresponding to the `XMLBean` object of the parent node.

Define the variables in the class that extend the `PageFlowController` class. For example, consider the case where you are trying to display XML data of the following form:

```
<CUSTOMER>data</CUSTOMER>
......<PROMOTION>promotion data</PROMOTION>
......
```

You can add the following code snippet, which shows two variables (shown in bold type) added to the page flow:

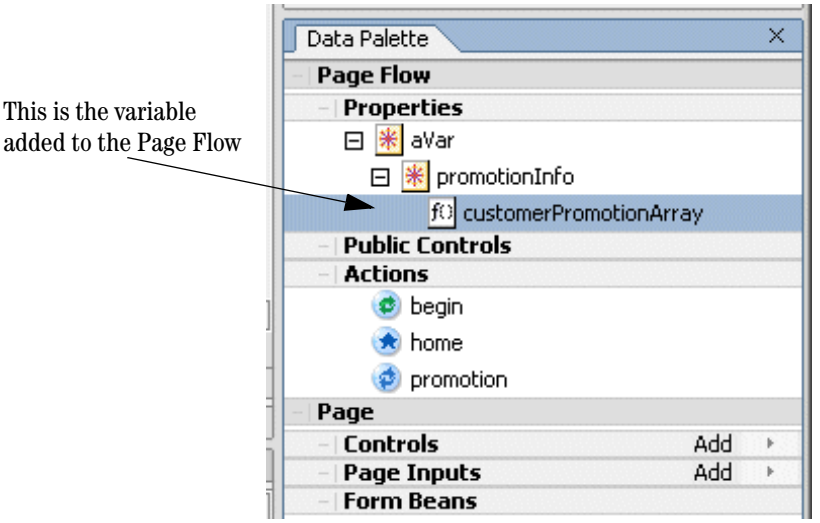
```
public class myPageFlowController extends PageFlowController
{
/**
     * This is the control used to generate this pageflow
     *  @common:control
     */
    private aLDControl myControl;

   // Add public Variables with XMLBeam types from the generated XMLBeans.
   // The type matches the return type of the method corresponding to the
   // query in the Liquid Data Control (.jcx) file.
   public com.mycorp.crm.CUSTDocument aVar;
```

This code snippet declares one variable in the page flow, `aVar`, and the variable will display in the IDE to allow for drag-and-drop operations onto JSP files.

Note: The variables of `XMLBean` type in the page flow must be declared `public`.

You must also initialize the variable in the page flow method corresponding to the page flow action that calls the query. For details, see "To Initialize the Variable in the Page Flow" on page 2-29.

Figure 2-17 Page Flow Variables for XMLBean Objects



When you drag-and-drop an array onto a JSP file, the NetUI Repeater Wizard appears and guides you through selecting the data you want to display.

To Add a Variable to a Page Flow

Perform the following steps to add a variable of `XMLBean` type for your query.

1. Open your Page Flow (`.jpf`) file in Workshop.
2. In the variable declarations section of your Page Flow class, enter a variable with the `XMLBean` type corresponding to the schema elements you want to display. Depending on your schema, what you want to display, and how many queries you are using, you might need to add several variables.
3. To determine the `XMLBean` type for the variables, perform the following:
4. In your Liquid Data control, examine the method signature for each method that corresponds to a query. The return type is the root level of the `XMLBean`. Create a variable of that type. For example, if the signature for a control method is as follows:

```
mySchema.CUSTOMERPROFILEDocument myQuery(java.lang.String custid);
```

create a variable as follows:

```
public mySchema.CUSTOMERPROFILEDocument myCustomerVar;
```

5. After you create your variables, initialize them as described in To Initialize the Variable in the Page Flow.

To Initialize the Variable in the Page Flow

You must initialize your `XMLBean` variables in the Page Flow. Initializing the variables ensures that the data bindings to the variables work correctly and that there are no tag exceptions when the JSP displays the results the first time.

Perform the following steps to initialize the `XMLBean` variables in the Page Flow:

1. Open your Page Flow (`.jpf`) file in Workshop.
2. In the page flow action which corresponds to the Liquid Data query for which you are going to display the data, add some code to initialize the variables used in the query displayed in that action.

The following sample code shows an example of initializing a variable on the Page Flow. The code (and comments) in bold is what was added. The rest of the code was generated when the Page Flow was generated from the Liquid Data control (see "Generating a Page Flow From a Control" on page 2-24).

```
/**
     * Action encapsulating the control method :RTLCustomerProfile
     * @jpf:action
     * @jpf:forward name="success" path="index.jsp"
     * @jpf:catch method="exceptionHandler" type="Exception"
     */
    public Forward RTLCustomerProfile( RTLCustomerProfileForm aForm )
        throws Exception
    {
        schemasBeaComLdCustview.PROFILEVIEWDocument var =
                    myControl.RTLCustomerProfile( aForm.custid );
        getRequest().setAttribute( "results", var );

        //initialize the profile variable to var from the above statement

        profile=var;

        return new Forward( "success" );
    }
```

Displaying Query Results in a Table or List

Once you create and initialize your variables in the Page Flow, you can drag and drop the variables onto a JSP file. When you drag and drop an `XMLBean` variable onto a JSP File, Workshop displays the repeater wizard to guide you through the process of selecting the data you want to display. The repeater wizard provides choices for displaying the results in an HTML table or in a list.

To Add a Repeater to a JSP File

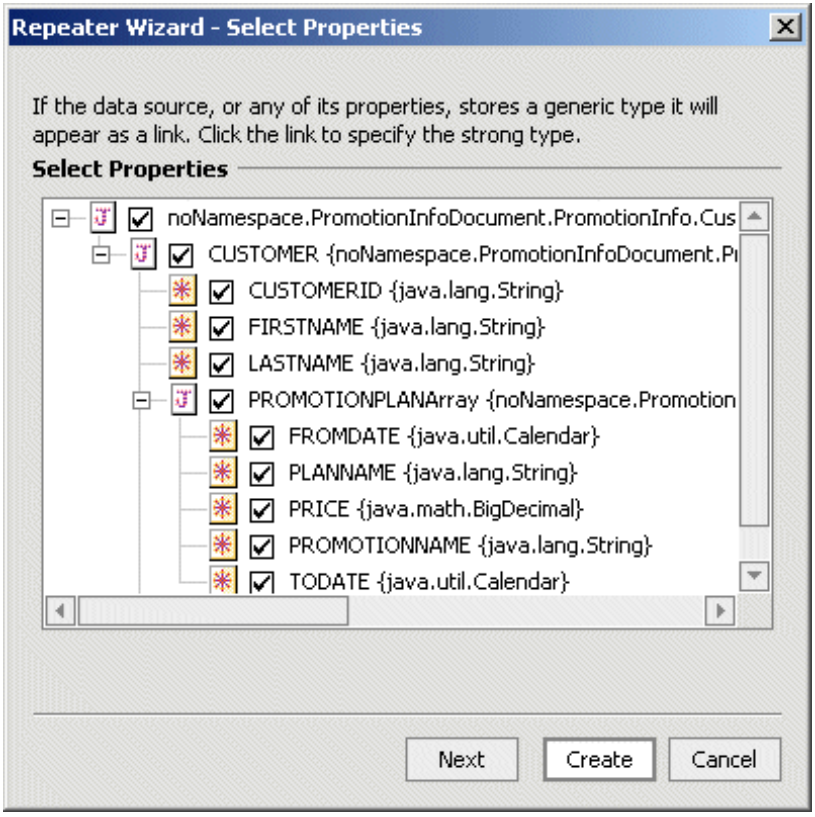
Perform the following to add a NetUI repeater tag (used to display the data from a Liquid Data query) to a JSP file.

1. Open a JSP file in your Page Flow project where you want to display data.
2. In the Data Palette —> Page Flow Properties, locate the variable containing the data you want to display.
3. Expand the nodes of the variable to expose the node that contains the data you want to display. If the variable does not traverse deep enough into your schema, you will have to create another variable to expose the part of your schema you require. For details, see "To Initialize the Variable in the Page Flow" on page 2-29.
4. Select the node you want and drag and drop it onto the location of your JSP file in which you want to display the data. You can do this either in Design View or Source View.

Note: You can only drag and drop leaf nodes from the Page Flow Properties.

5. Workshop displays the repeater wizard.

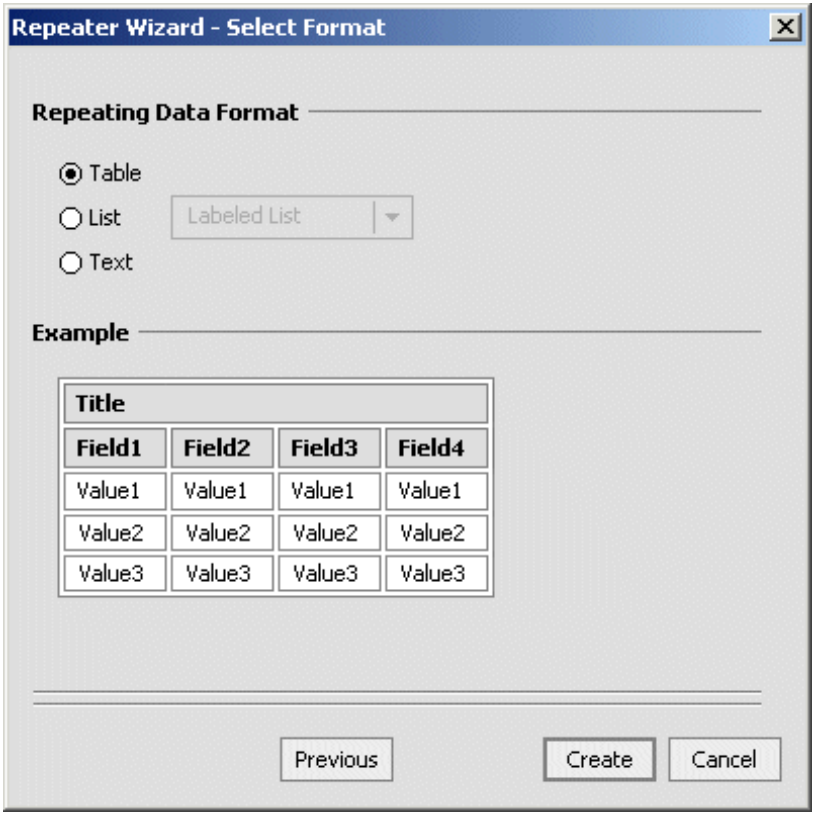
Figure 2-18 Repeater Wizard



6. In the repeater wizard, navigate to the data you want to display and uncheck any fields that you do not want to display. There might be multiple levels in the repeater tag, depending on your schema.

7. Click Next. The Select Format screen appears.

Figure 2-19 Repeater Wizard Select Format Screen



8. Choose the format to display your data in and click Create.

   Workshop generates the layout for your data.

9. Click the test button to see your data display.

To Add a Nested Level to an Existing Repeater

You can create repeater tags inside of other repeater tags. You can display nested repeaters on the same page (in nested tables, for example) or you can set up Page Flow actions to display the nested level on another page (with a link, for example).

Perform the following steps to create a nested repeater tag.

1. Add a repeater tag as described in "To Add a Repeater to a JSP File" on page 2-30.

2. Add a column to the table where you want to add the nested level.
3. Drag and drop the array from your variable corresponding to your nested level into the data cell you created in the table.
4. In the repeater wizard, select the items you want to display.
5. Click the Create button in the repeater wizard to create the repeater tags.
6. Click the test button to test the application.

To Add Code to Handle Null Values

Perform the following steps to add code in your JSP file to handle null values for your data. It is a common JSP design pattern to add conditional code to handle null values. If you do not handle null values, your page will display tag errors if it is rendered before the queries on it are executed.

1. Add a repeater tag as described in "To Add a Repeater to a JSP File" on page 2-30.
2. Open the JSP file in source view.
3. Find the `netui-data:repeater` tag in the JSP file.
4. If the `dataSource` attribute of the `netui-data:repeater` tag directly accesses an array variable from the page flow, then you can set the `defaultText` attribute of the `netui-data:repeater` tag. For example:

```
<netui-data:repeater dataSource="{pageFlow.promo}" defaultText="no data">
```

   If the `dataSource` attribute of the `netui-data:repeater` tag accesses a child of the variable from the page flow, you must add `if/else` logic in the JSP file as described below.

5. If the `defaultText` attribute does not work for your `netui-data:repeater` tag, add code before and after the tag to test for null values. The following is sample code. The code in bold is added, the rest is generated by the repeater wizard. This code uses the profile variable initialized in "To Initialize the Variable in the Page Flow" on page 2-29.

```
<%
PageFlowController pageFlow = PageFlowUtils.getCurrentPageFlow(request);
if ( ((pF2Controller)pageFlow).profile == null
        ||
((pF2Controller)pageFlow).profile.getPROFILEVIEW().getCUSTOMERPROFILEArray
() == null
        ||
((pF2Controller)pageFlow).profile.getPROFILEVIEW().getCUSTOMERPROFILEArray
```

```
().length == 0){
      %>
         <p>No data</p>
         <% } else {%>
<netui-data:repeater dataSource=
        "{pageFlow.profile.PROFILEVIEW.CUSTOMERPROFILEArray}">
        <netui-data:repeaterHeader>
            <table cellpadding="2" border="1" class="tablebody" >
            <tr>
<!- the rest of the table and NetUI code goes here -->
<td><netui:label value
      ="{container.item.PROFILE.DEFAULTSHIPMETHOD}"></netui:label></td>
            </tr>
        </netui-data:repeaterItem>
        <netui-data:repeaterFooter></table></netui-data:repeaterFooter>
    </netui-data:repeater>
         <% }%>
```

6. Test the application.

Security Considerations With Liquid Data Controls

This section describes security considerations to be aware of when developing applications using Liquid Data controls. The following sections are included:

- Security Credentials Used to Create Liquid Data Control
- Testing Controls With the Run-As Property in the JWS File
- Trusted Domains

Security Credentials Used to Create Liquid Data Control

The WebLogic Workshop Application Properties (Tools —> Application Properties) allow you to set the connection information to connect to the domain in which you are running. You can either use the connection information specified in the domain `boot.properties` file or override that information with a specified username and password.

When you create a Liquid Data Control (`.jcx`) file and are connecting to a local Liquid Data server (Liquid Data on the same domain as Workshop), the user specified in the Application Properties is used to connect to the Liquid Data server. When you create a Liquid Data Control and are connecting to a remote Liquid Data server (Liquid Data on a different domain from Workshop), you specify the connection information in the Liquid Data Control Wizard Connection information dialog (see Figure 2-6).

When you create a Liquid Data Control, the Control Wizard displays all queries to which the specified user has access privileges. The access privileges are defined by any security policies set on the queries, either directly or indirectly.

Note: The security credentials specified through the Application Properties or through the Liquid Data Control Wizard are only used for creating the Liquid Data Control (`.jcx`) file, not for testing queries through the control. To test a query through the control, you must get the user credentials either through the application (from a login page, for example) or by using the run-as property in the Web Service file.

Testing Controls With the Run-As Property in the JWS File

For testing, you can use the run-as property to test a control running as a specified user. To set the run-as property in a Web Service, open the Web Service and enter a user for the run-as property in the WebLogic Workshop property editor. Queries run through a Liquid Data Control used by the Web Service

When a query is run from an application, the application must have a mechanism for getting the security credential. The credential can come from a login screen, it can be hard-coded in the application, or it can be imbedded in a J2EE component (for example, using the run-as property in a `.jws` Web Service file).

Note: The Liquid Data Control property editor shows a run-as property, but the run-as property in the Liquid Data Control does not cause the Liquid Data Control to run as the specified user. If you want to use this feature, you must specify the run-as property in the `.jws` file, not in the `.jcx` file.

Trusted Domains

If the Liquid Data server is on different domain from WebLogic Workshop, then both domains must be set up as trusted domains.

Domains are considered trusted domains if they share the same security credentials. With trusted domains, a user that exists on one domain need not be authenticated on the other domain (as long as the user exists on both domains).

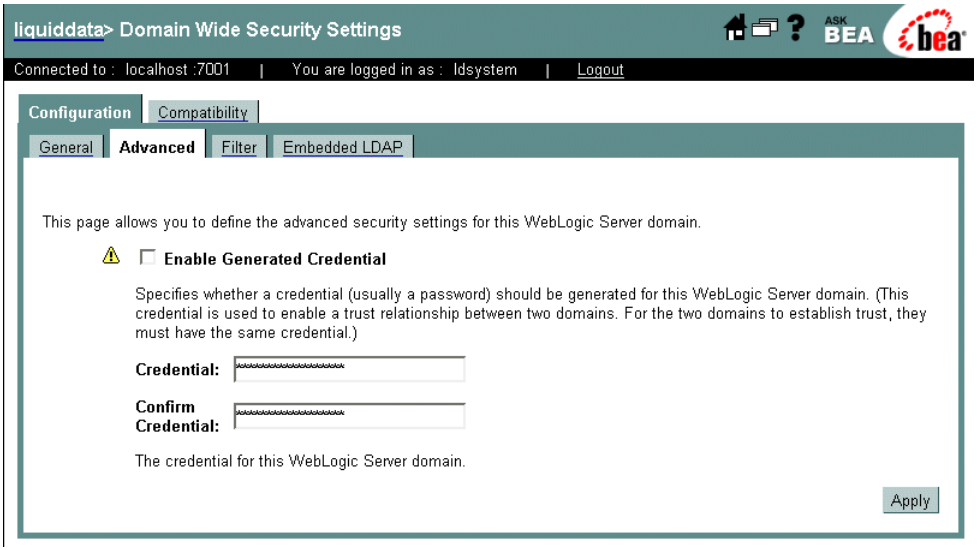
Note: After configuring domains as trusted, you must restart the domains before the trusted configuration takes effect.

To Configure Trusted Domains

Perform the following steps to configure domains as a trusted:

1. Log into the WebLogic Administration Console as an administrator.
2. Click the node corresponding to your domain.
3. At the bottom of the General tab for the domain configuration, click the link labeled "View Domain-wide Security Links."
4. Click the Advanced tab.

Figure 2-20 Setting up Trusted Domains



5. Uncheck the Enable Generated Credential box, enter and confirm a credential (usually a password), and click Apply.
6. Repeat this procedure for all of the domains you want to set up as trusted. The credential must be the same on each domain.

For more details on WebLogic security, see "Configuring Security for a WebLogic Domain" in the WebLogic Server documentation.

Moving Your Liquid Data Control Applications to Production

When you move any Liquid Data deployment from development to production, you must move Liquid Data and WebLogic Server resources (JDBC Connection Pools, Liquid Data Data Sources, the Liquid Data repository, and so on) from the development environment to the production environment. For details about deploying Liquid Data, see the Liquid Data *Deployment Guide*.

For applications that use Liquid Data controls, you must also deploy and update the `ldcontrol.properties` file, which contains connection information for Liquid Data controls. This section describes the development to production lifecycle and provides the basic steps for moving an application containing Liquid Data controls from development to production. The following sections are included:

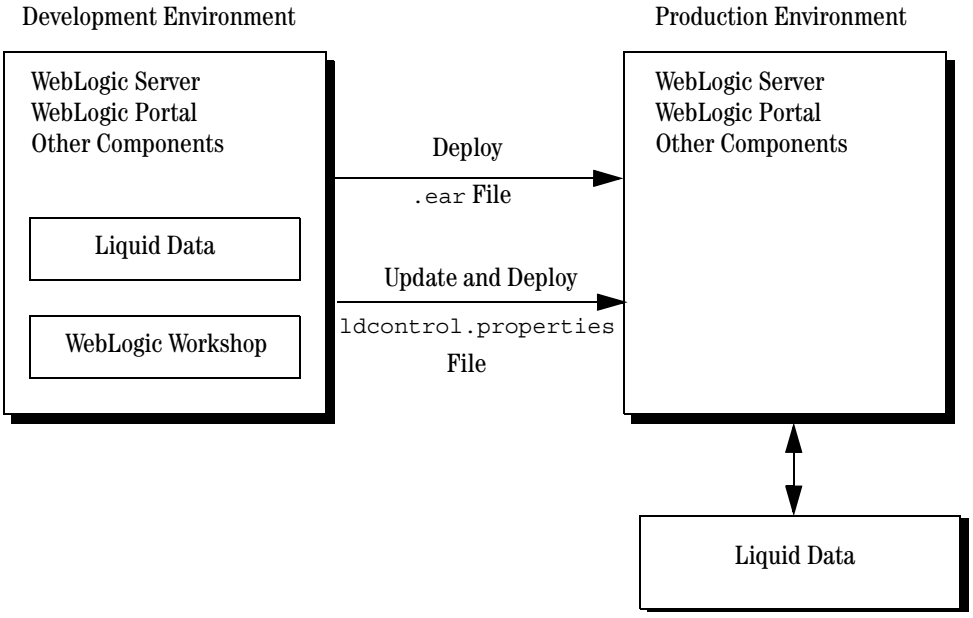
- Development to Production Lifecycle Architecture
- Steps For Deploying to Production

Development to Production Lifecycle Architecture

In a typical development scenario, you will develop your applications in one environment and then deploy them in another. There are two main artifacts that you need to deploy on the production environment:

- Packaging Liquid Data JAR Files in Application .ear Files
- Liquid Data ldcontrol.properties File

Figure 2-21 Development to Production Lifecycle



Packaging Liquid Data JAR Files in Application .ear Files

After you have developed and tested your application using WebLogic Workshop in your development environment, you must create a `.ear` file for deployment to your production server(s). All the resources the application needs are already included in the application `Libraries` directory, so the only thing you need to do is create the `.ear` file for the application.

To Generate the .ear File in Workshop

Perform the following steps to generate an enterprise archive file (.ear) in WebLogic Workshop:

1. Open your application in WebLogic Workshop, if it is not already opened.
2. Select Build —>Build EAR from the Workshop menu.

When the build is complete, WebLogic Workshop lists the `.ear` file location in the Build window.

Liquid Data ldcontrol.properties File

Each domain that runs Liquid Data Control applications has a *single* `ldcontrol.properties` file, which stores the connection information for *all* Liquid Data Control applications running in the domain. The `ldcontrol.properties` file is located at the root directory of your domain where the Liquid Data Control application `.ear` file is deployed. There is an entry in the `ldcontrol.properties` file for each control you have created in each application.

The entries in the `ldcontrol.properties` file are of the following form:

```
AppName.ProjectName.FolderName.jcxName=t3\://hostname\:port
```

where:

Name	Description
`AppName`	The name of the WebLogic Workshop application.
`ProjectName`	The name of the WebLogic Workshop Project which contains the Liquid Data Control.
`FolderName`	The name of the folder which contains the Liquid Data Control.
`jcxName`	The name of the Liquid Data Control file (without the `.jcx` extension). For example, if the control file is named `myLDControl.jcx`, the entry in this file is `myLDControl`.
`hostname`	The hostname or IP address of the Liquid Data Server for this control.
`port`	The port number for the Liquid Data Server for this control.

Note: The colons (`:`) in the URL must be escaped with a backslash (`\`) character.

If the URL value is missing, the Liquid Data Control uses the connection information from the domain `config.xml` file.

The following is a sample `ldcontrol.properties` file.

```
#Fri Oct 31 15:30:36 PST 2003
myTest.myTestWeb.myFolder.Untitled=t3\:myLDServer\:7001
myTest.myTestWeb.myFolder.myControl=
SampleApp.LiquidDataSampleApp.Controls.RTLControl=t3\:myLDServer\:7001
SampleApp.Untitled.NewFolder.Untitled=t3\:yourLDServer\:7001
testnew.Untitled.NewFolder.ldc=
test.testWeb.NewFolder.Untitled=
```

Steps For Deploying to Production

This section describes the following basic steps for moving an application from development to production:

- Step 1: Generate Enterprise Application Archive (.ear) in Workshop
- Step 2: Merge ldcontrol.properties File entries to Production Server
- Step 3: Deploy Enterprise Application Archive (.ear) on Production Server

Step 1: Generate Enterprise Application Archive (.ear) in Workshop

Use WebLogic Workshop to generate the .ear file for your application as described in "To Generate the .ear File in Workshop" on page 2-38.

Step 2: Merge ldcontrol.properties File entries to Production Server

Merge the entries in the `ldcontrol.properties` file from the root level of your development domain with the `ldcontrol.properties` file in the root level of the production domain. There must be one entry for each Liquid Data Control. If the `ldcontrol.properties` file does not exist in the production domain, copy it from your development domain.

You must also update the URLs in each entry of the file to reference the production Liquid Data servers. For details on and for the syntax of the `ldcontrol.properties` file, see "Liquid Data ldcontrol.properties File" on page 2-39.

Step 3: Deploy Enterprise Application Archive (.ear) on Production Server

Deploy your enterprise archive (`.ear`) file on the production WebLogic Server. You deploy the `.ear` file from the *domain* —> Deployments —> Applications node of the WebLogic Server Administration Console. The `.ear` file must be accessible from the filesystem in which the WebLogic administration server is running. For details on deploying `.ear` files, see "Deploying WebLogic Server Applications" from the WebLogic Server documentation.

CHAPTER 3

Invoking Queries in EJB Clients

This chapter describes how to execute BEA Liquid Data for WebLogic queries in EJB clients. It contains the following steps:

- Step 1: Connect to the Liquid Data Server
- Step 2: Specify Query Parameters
- Step 3: Execute the Query
- Step 4: Process the Results of the Query

For more information about EJB clients, see "EJB Development" on page 1-2.

Step 1: Connect to the Liquid Data Server

An EJB client may use standard JNDI and EJB calls in order to obtain a reference to the remote interfaces of the query execution session beans.

To do so, a remote client first needs to set up the JNDI initial context by specifying the `INITIAL_CONTEXT_FACTORY` and `PROVIDER_URL` environment properties.

- The value of `INITIAL_CONTEXT_FACTORY` should be set to `weblogic.jndi.WLInitialContextFactory`.
- The value of `PROVIDER_URL` should reflect the location (URI) of the application server hosting the Liquid Data Server (for example, `t3://localhost:7001`).

A local client, i.e. a client that resides on the application server that hosts the Liquid Data Server, may bypass these steps by using the settings in the default context obtained by invoking the empty initial context constructor (i.e. by calling `new InitialContext()`).

At this stage, the client may also optionally authenticate itself by passing its security context to the corresponding JNDI environment properties `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIALS`. Alternatively, the client may use the query execution API as an anonymous (default) user.

Once the JNDI context is set up, the client may use the JNDI names of the remote home interfaces of the stateless query execution session bean in order to perform a lookup and obtain remote references to the `EJBHome` objects.

The JNDI name for the home interface of the query execution SSB is `bea.ldi.server.QueryHome`. The home interface may finally be used to obtain references to the `EJBObject` objects of the session bean.

The JNDI name for the remote interface of the query execution SSB is `com.bea.ldi.server.QueryHome`. The home interface may finally be used to obtain references to the `EJBObject` object (`com.bea.ldi.server.Query`) of the session bean.

The code excerpt below is an example of a remote client that obtains a reference to the `EJBObject` of the stateless query execution session bean and it illustrates the concepts discussed above:

Listing 3-1 Obtaining a Reference to EJB Object Query

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import com.bea.ldi.server.*;
...// more code
private static final String QUERY_HOME_JNDI_NAME = "bea.ldi.server.QueryHome";
   ...// more code
QueryHome queryHome = null;
```

```
  Query query = null;
  // obtain a remote Query reference
  try {
queryHome = lookupQueryHome();
Object obj = queryHome.create();
query = (com.bea.ldi.server.Query) narrow(obj, com.bea.ldi.server.Query.class);
  }
  catch (Exception e) {
// code to handle the exception
  }
    ...// more code
  /**
   * Lookup the EJB home in the JNDI tree of the specified Liquid Data Server.
   */
  private QueryHome lookupQueryHome()
    throws NamingException {
Context ctx = getInitialContext();
// Lookup the bean's home using JNDI
Object home = ctx.lookup(QUERY_HOME_JNDI_NAME);
return (QueryHome) narrow(home, QueryHome.class);
  }
  /**
   * Obtains the JNDI context.
   */
  private Context getInitialContext() throws NamingException {
// Set up the environment properties
 Hashtable h = new Hashtable();
```

```
      h.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
h.put(Context.PROVIDER_URL, "t3://localhost:7001");
h.put(Context.SECURITY_PRINCIPAL, "username");
h.put(Context.SECURITY_CREDENTIALS, "password");
// Get an InitialContext
return new InitialContext(h);
  }
  /**
   * RMI/IIOP clients should use this narrow function
   */
  private Object narrow(Object ref, Class c) {
   return PortableRemoteObject.narrow(ref, c);
  }
```

Step 2: Specify Query Parameters

Parameterized queries need to be configured before they are executed. Parameterized queries require the client to specify the values of the query parameters. Query parameters allow for dynamic binding of parts of an XQuery query to values specified at runtime. The presence of a parameter in an XQuery query is manifested through the use of the following notation:

`$#`*`pname`*

where *`pname`* is a unique name across the query assigned to the parameter. In general, parameters may be used in those places inside a query where a constant could be used. For a list of valid parameter types, see "Parameterized Queries" on page 1-3.

The following sample query illustrates the use of a parameter inside a query.

Listing 3-2 Parameterized XQuery Query

```
<root>
```

```
{
for $b in document("bib")//book,
   $pub in $b/publisher
where $pub = $#publisher
return
   <result>
      {$b/title}
      {$b/author}
   </result>
}
</root>
```

The following code excerpt demonstrates the sequence of calls required to set the parameter for such a query using the Liquid Data Server API.

Listing 3-3 Setting Query Parameters

```
import com.bea.ldi.server.common.QueryParameters;
... // more code
QueryParameters qp = new QueryParameters();
qp.setString("publisher", "Morgan Kaufmann Publishers");
```

The value of a parameter can be overwritten and reused in a new query execution by setting it to a new value. Using anything other than a String for a parameter name, or setting a parameter value of an invalid type, results in a `RuntimeException`.

Step 3: Execute the Query

Once the reference to the `EJBObject` of the query execution session bean has been obtained and the query has been configured by setting any query parameters or attributes, the query is ready to be executed.

When executing a query, its fully qualified name must be used. When the physical location of the query is:

repository/stored_queries/*dir1*/*dir2*/SQ1.xq

It would be qualified as:

dir1.dir1.SQ1

For example, if the query is located at:

LDrepository/stored_queries/inventory/widgetsales.xq

the query is qualified as:

inventory.widgetsales

The `Query` remote interface offers a variety of execution calls based on whether the query is parameterized or fixed and whether it is stored or ad hoc.

As an example, assuming that the client has obtained a reference to a `Query` object, as shown in the following code listing, and the String variable `queryString` has been loaded with the contents of an ad hoc XQuery query, the following excerpt shows how to obtain the query result.

Listing 3-4 Execution of an Ad Hoc Non-Parameterized Query

```
import com.bea.ldi.server.Query;

import com.bea.ldi.server.common.QueryResult;

... // more code

Query query = null;

... // obtain reference to Query

QueryResult result = null;

try {

   result = query.execute(queryString);

}
```

```
if !(result.isEmpty()){
... // process result
}
else {
... // query returned no data
}
catch(RemoteException e) {
   // code to handle the exception
}
finally {
   try {
   query.remove();
}
catch(Exception e) {
    // code to handle the exception
   }
}
```

If a stored query is to be executed, then call `executeStored(`*`queryName`*`)`, where the `String` variable *`queryName`* is assumed to contain the name of the stored query to be executed.

If the query is parameterized, once the query parameters have been set, they should be passed in the execution call, that is, the `execute(queryString)` call should be replaced with the calls `execute(queryString, qp)` and `executeStored(queryName, qp)` in the case of ad hoc and stored queries respectively. The `QueryParameters` variable `qp` in the previous calls is assumed to be loaded with the query parameters.

Note that all execution calls are remote and therefore they may throw a `RemoteException`, which should be handled by the client. Note also, that once the query result has been retrieved, the client may release resources by removing the `EJBObject`. If the query is parameterized, the client may use the `Query` reference to execute the same query multiple times, possibly setting different values for

the query parameters each time, before removing the `EJBObject`. In any case, other server-side resources related to query execution (for example, database cursors) are automatically released once a query has been executed.

Step 4: Process the Results of the Query

Further processing of the query result at the client side may take various forms ranging from merely extracting, or printing out the XML string to using the DOM representation of the result in order to drill into specific subsets of it.

The result is fully materialized on the server in the form of an unformatted XML string, which is transmitted to the client. The client may then extract the XML content of the query result as a String using `toXML()` method. Alternatively, the client may use the `getDocument()` call in order to obtain the DOM representation of the result, provided that a JAXP-compliant parser is available in the client environment. In either case, the client is free to process the result using any XML processor (for example, using an XSLT processor to convert the result to a presentable format like HTML).

CHAPTER 4

Invoking Queries in JSP Clients

This chapter describes how to invoke BEA Liquid Data for WebLogic queries in JSP client applications using the Liquid Data tag library. It contains the following sections:

- About the Liquid Data Tag Library
- Processing Steps

For more information about JSP clients, see "JSP Tag Library Development" on page 1-2.

Note: The following discussion assumes that you are familiar with the use of custom tag libraries. For more information, see *Programming WebLogic JSP Tag Extensions* in the WebLogic Server documentation.

About the Liquid Data Tag Library

This section introduces the Liquid Data tag library. It contains the following sections:

- Scope of the Liquid Data Tag Library
- Location of the Liquid Data Tag Library
- Making the Tag Library Accessible to a Web Application
- Tags in the Liquid Data Tag Library

Scope of the Liquid Data Tag Library

The goal of the Liquid Data tag library is to provide simple declarative means for JSP clients to obtain access to the XML results of XQuery queries. Tag library clients need only be concerned with the configuration of parameterized queries. The following section provides detailed information on how to set up query parameters in this case.

Location of the Liquid Data Tag Library

The Java classes and other file resources required by tag library clients are packaged inside `LDS-client.jar`, `LDS-em-client.jar`, and `LDS-taglib.jar`. The tag library descriptor file (`taglib.tld`) defines the elements and attributes in the Liquid Data tag library. The `taglib.tld` is stored under `META-INF` inside the `LDS-taglib.jar` file.

Making the Tag Library Accessible to a Web Application

In order to use the Liquid Data tag library in a web application, you must do the following:

- Copy the LDS-taglib.jar File to the WEB-INF/lib Directory
- Add the <taglib> Entry to the web.xml File

Copy the LDS-taglib.jar File to the WEB-INF/lib Directory

Each web application that uses the Liquid Data tag library must have a copy of the `LDS-taglib.jar` file in the *`application_context`*`/WEB-INF/lib` directory. The `LDS-taglib.jar` file is installed with Liquid Data as the following file:

`bea_home``/liquiddata/server/lib/LDS-taglib.jar`

Add the <taglib> Entry to the web.xml File

You must add the following entry to the `web.xml` file for each web application that uses the Liquid Data tag library:

```
<taglib>
    <taglib-uri>LDSTLD</taglib-uri>
    <taglib-location>/WEB-INF/lib/LDS-taglib.jar</taglib-location>
</taglib>
```

Tags in the Liquid Data Tag Library

The Liquid Data tag library contains the following tags:

- query Tag
- param Tag

query Tag

The `query` tag specifies the query to execute and the host machine on which to run the query. The `query` tag has the following attributes.

Table 4-1 Attributes of the query tag

Attribute	Description
`name`	Specifies the name of a stored query from which to retrieve results.
`server`	Specifies the host machine on which the Liquid Data Server is running. Use only when JSP clients are deployed on different machine from the one hosting the Liquid Data Server.
`username`	Specifies the WebLogic user name for the query request. If you do not specify a `username`, the query is sent as the `guest` user.
`password`	Specifies the password for the `username` attribute.

The following example specifies the stored query on the specified host machine.

```
<lds:query name="MyStoredQuery" server="t3://222.222.22:7001"
username="ldsystem" password="ldsecurity">
```

param Tag

The `param` tag specifies a query parameter as a name-value pair. For each parameter, you specify a separate `param` tag. The `param` tag has the following attributes.

Table 4-2 Attributes of the param tag

Attribute	Description
`name`	Name of the query parameter.
`value`	Value of the query parameter.

The following example specifies the name of a publisher in the `param` tag.

```
<lds:param name="publisher" value="<%=\"Morgan Kaufmann Publishers\"%>"/>
```

Processing Steps

This section describes the process of executing queries from JSP clients. It contains the following steps:

- Step 1: Add the Tag Library to your Web Application
- Step 2: Reference the Liquid Data Tag Library
- Step 3: Connect to the Liquid Data Server
- Step 4: Specify Query Parameters
- Step 5: Execute the Query
- Step 6: Process the Query Results

Step 1: Add the Tag Library to your Web Application

You must add the `LDS-taglib.jar` file to the `WEB-INF/lib` directory of your web application and must update the `web.xml` file for the web application. For details, see "Making the Tag Library Accessible to a Web Application" on page 4-2.

Step 2: Reference the Liquid Data Tag Library

To use the tags in the Liquid Data tag library, you must reference them in each JSP page. To reference the JSP tags described in "Tags in the Liquid Data Tag Library" on page 4-3, including the following code near the top of each JSP page:

```
<%@ taglib uri="LDSTLD" prefix="lds" %>
```

Note: The default prefix (`lds:`) is configurable.

Step 3: Connect to the Liquid Data Server

Tag library clients are JSP clients. JSP clients that are deployed on the same application server that hosts Liquid Data Server do not need to take any steps in order to connect to Liquid Data Server, as this case is supported by default.

JSP clients deployed on a server other than the one hosting Liquid Data Server need to specify the location (URL) of the server hosting Liquid Data Server using the `server` attribute of the `query` tag, as shown in the following example.

Listing 4-1 Non-Local JSP Client Connecting to Liquid Data Server

```
<%@ taglib uri="LDSTLD" prefix="lds" %>

...

<lds:query ... server="t3://222.222.22:7001" username="ldsystem"
password="ldsecurity">

...

</lds:query>
```

Step 4: Specify Query Parameters

In the Liquid Data tag library, the `query` tag accepts a nested `param` tag, which may be used to specify the name and the value of a parameter applied to the XQuery query represented by the query tag. The following excerpt illustrates how to set the parameter for the query shown in Listing 4-3.

Listing 4-2 Setting the Query Parameters

```
<%@ taglib uri="LDSTLD" prefix="lds" %>
...
<lds:query ... server="t3://222.222.22:7001">
...
   <lds:param name="publisher"
      value="<%=\"Morgan Kaufmann Publishers\"%>"/>
</lds:query>
```

The value of the parameter is a JSP expression that is evaluated at run time. Quotes are escaped out. The supported parameter types are the same as those supported for EJB clients. The actual type of the parameter is implied by the Java type of the value specified as the content of the `value` attribute. So, for example, a value `Date.valueOf("2002-03-01")` would correspond to a parameter of type `java.sql.Date`. A query that uses multiple parameters would require the use of as many `param` elements.

Step 5: Execute the Query

The Liquid Data Server Tag Library supports both ad hoc and stored queries.

When executing a query, its fully qualified name must be used. When the physical location of the query is:

repository/stored_queries/*dir1*/*dir2*/SQ1.xq

The query is qualified as:

dir1.dir1.SQ1

For example, if the query is located at:

LDrepository/stored_queries/inventory/widgetsales.xq

the query would be qualified as:

inventory.widgetsales

Executing Stored Queries

Stored queries are specified by having their name being passed as the value of the `name` attribute of the `query` tag, as shown in the following example of a parameterized, stored query.

Listing 4-3 Sample Stored Query

```
<%@ taglib uri="LDSTLD" prefix="lds" %>
...
<lds:query name="MyStoredQuery" server="t3://222.222.22:7001">
   <lds:param name="publisher"
      value="<%=\"Morgan Kaufmann Publishers\"%>"/>
</lds:query>
```

Executing Ad Hoc Queries

Ad hoc queries should have their content directly embedded inside the `query` element, as shown in the following example.

Listing 4-4 Sample Ad Hoc Query

```
<%@ taglib uri="LDSTLD" prefix="lds" %>
...
<lds:query server="t3://222.222.22:7001">
   <lds:param name="publisher"
      value="<%=\"Morgan Kaufmann Publishers\"%>"/>
   <root>
   {
   for $b in document("bib")//book,
      $pub in $b/publisher
      where $pub = $#publisher
```

```
        return
           <result>
              {$b/title}
              {$b/author}
           </result>
        }
    </root>
</lds:query>
```

Handling Exceptions

Any exception that is thrown during query execution should be handled using standard JSP error handling techniques.

Step 6: Process the Query Results

Query execution results in the unformatted XML content of the query result becoming available to the JSP client for further processing.

Typically, you perform some kind of post-processing to the query results for display purposes. JSP clients can apply an XSL transform to the XML query result in order to convert it to a presentable format. You can perform the transformation by enclosing the `query` tag with another custom tag that performs the XSL transformation.

For example, the following listing uses the `x:transform` tag described in the JavaServer Pages *Standard Tag Library 1.0 Specification*, which is published by the Sun Microsystems, Inc. at the following URL:

http://java.sun.com/products/jsp/jstl/index.html

Listing 4-5 Applying an XSL Transform to the Query Result

```
<%@ taglib uri="LDSTLD" prefix="lds" %>
<%@ taglib uri="X" prefix="x" %>
...
```

```
<x:transform  xsltUrl="url-to-xsl-script">
   <lds:query server="t3://222.222.22:7001">
      <lds:param name="publisher"
         value="<%=\"Morgan Kaufmann Publishers\"%>"/>
      <root>
         for $b in document("bib")//book,
         $pub in $b/publisher
         where $pub = $#publisher
         return
            <result>
               {$b/title}
               {$b/author}
            </result>
         }
      </root>
   </lds:query>
</x:transform>
```

You can also use the JSP tag library provided with WebLogic server to perform the XSLT transformation. For details on this tag library, see:

http://e-docs.bea.com/wls/docs81/xml/xml_apps.html

The following sample code uses the WebLogic JSP tag library to perform the XSLT transformation.

Listing 4-6 Using the WebLogic Server XSLT JSP Tag Library for an XSL Transform

```
<%@ taglib uri="LDSTLD" prefix="lds" %>
<%@ taglib uri="xmlx" prefix="x" %>

<x:xslt media="http">
  <x:xml>
    <lds:query name="viewSample" server="t3://localhost:7001"
          username="ldsystem" password="ldsystem" >
    </lds:query>
  </x:xml>
  <x:stylesheet media="http"  uri="http.xsl"/>
</x:xslt>
```

CHAPTER 5

Invoking Queries in Web Service Clients

This chapter introduces how to invoke BEA Liquid Data for WebLogic™ queries in Web service client applications. It contains the following sections:

- Finding the WSDL URL for Generated Web Services
- Invoking Web Services Programmatically

For more information about Liquid Data-generated Web services, see "Generating and Publishing Web Services" in the Liquid Data *Administration Guide*.

Finding the WSDL URL for Generated Web Services

After generating a Web service for a selected stored query, the Administration Console displays a confirmation message that shows the URL of the generated Web service. The URL of the WSDL of a generated Web service has the following pattern:

`http://HOSTNAME:PORT/liquiddata/query_name/webservice?WSDL`

For example, if the stored query is named `order.xq`, then the URL of its WSDL is:

`http://localhost:7001/liquiddata/order/webservice?WSDL`.

Invoking Web Services Programmatically

You invoke Liquid Data Web services that were generated in the Administration Console using the same approach that you would use for invoking any WebLogic Web Service. For more information, see "Invoking Web Services" in *Programming WebLogic Web Services* in the WebLogic Server documentation.

CHAPTER 6

Invoking Queries in WebLogic Integration Business Processes

This chapter describes how to invoke BEA Liquid Data for WebLogic queries from business processs in BEA WebLogic Integration. It contains the following sections:

Liquid Data and WebLogic Integration Business Processes

Business processes in WebLogic Integration can invoke Liquid Data queries using a Liquid Data Control. For comprehensive information about WebLogic Integration, see the WebLogic Integration documentation.

A business process can use a Liquid Data query to easily gather data from disparate data sources. For an example of a business process that uses a Liquid Data query, open the Liquid Data sample application in WebLogic Workshop. The sample application is installed in the following location:

```
<BEA_HOME>/weblogic81/samples/liquiddata/SampleApp/SampleApp.work
```

The sample application uses a WebLogic Integration business process to submit updated user profile data entered by the user (for example, address, credit card, and so on), and then use a Liquid Data query to return the updated profile.

Setting Up a Liquid Data Query in a Business Process

You can easily add Liquid Data queries to WebLogic Integration business processes by creating a Liquid Data Control that accesses the query or queries you want to use in the business process. The procedure for invoking a Liquid Data query is similar to that of invoking a database query with a database control.

There are three basic steps to adding Liquid Data Queries to a WebLogic Integration business processes:

- Create the Liquid Data Control
- Adding a Liquid Data Control to a JPD File
- Setting Up the Control in the Business Process

Create the Liquid Data Control

Before you can run a Liquid Data query in a WebLogic Integration business process, you must create a Liquid Data Control that accesses the query or queries you want to run in your business process. For details, see “Using Liquid Data Controls to Develop Workshop Applications” on page 2-1.

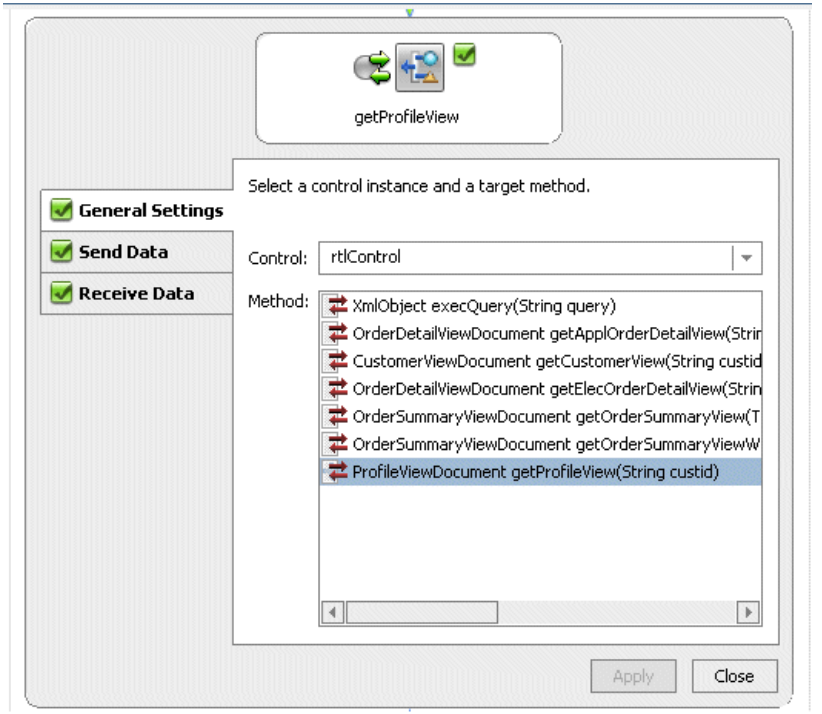
Adding a Liquid Data Control to a JPD File

Once you have created a Liquid Data Control, you can add it to a business process the same way you add any other control to a business process. For example, you can drag and drop the control into the WebLogic Integration business process in the place where you want to run your Liquid Data query or you can add the Liquid Data Control to the Data Palette. For comprehensive information about using WebLogic Integration, see the WebLogic Integration documentation.

Setting Up the Control in the Business Process

Once you have added the Liquid Data Control to the business process, you must configure it in your business process. As shown in Figure 6-1, you must select the query in the General Settings section of the Liquid Data Control portion of the business process, specify input parameters for the query in the Send Data section, and specify the output of the query in the Receive Data section.

Figure 6-1 WebLogic Integration Business Process Accessing a Liquid Data Control



For an example of using a Liquid Data Control in a WebLogic Integration business process, see the Liquid Data sample application.

CHAPTER 7

Invoking Queries in BEA WebLogic Portal Applications

BEA WebLogic Portal™ users can invoke the BEA Liquid Data for WebLogic Query API from WebLogic Portal. Calls to the Liquid Data query API are transparent to Portal users. This chapter includes the following sections:

- Invoking Liquid Data Queries as EJB Clients
- Invoking Liquid Data Queries as JSP Clients

For general information about developing portals, see the WebLogic Portal documentation.

Invoking Liquid Data Queries as EJB Clients

WebLogic Portal needs to be configured to find the Liquid Data query EJB (`com.bea.ldi.server.QueryHome`), a stateless session bean. For more information, see Chapter 3, "Invoking Queries in EJB Clients."

Invoking Liquid Data Queries as JSP Clients

WebLogic Portal can invoke Liquid Data queries using the Liquid Data Query API and the Liquid Data tag library. Invocations of Liquid Data queries are transparent to Portal users. For more information, see Chapter 4, "Invoking Queries in JSP Clients."

To invoke Liquid Data queries, you first need to deploy Liquid Data and WebLogic Portal according to the instructions in "Deploying with WebLogic Portal" in "Deployment Tasks" in *Deploying Liquid Data*. Once deployed, you can access the Liquid Data query API from a `portlet.jsp` file using the JSP tag library. For example, the following JSP code invokes a query named `isq` on port 7001 of a server named `myserver`:

Listing 7-1 Sample JSP Code Invoking the Liquid Data Query API

```
<!-- Declare the LD taglib library -->
<%@ taglib uri="LDSTLD" prefix="lds" %>
<!-- Execute the stored procedure "isq" at server "myserver" -->
<lds:query name="isq" server="t3://myserver:7001">
</lds:query>
```

CHAPTER 8

Using Custom Functions

This section describes how to create custom functions in BEA Liquid Data for WebLogic. It contains the following sections:

- About Custom Functions
- Defining Custom Functions
- Examples of Custom Functions

About Custom Functions

Liquid Data provides a set of standard functions to use when creating data views and queries. You can also define *custom functions* in the Liquid Data server repository to use in the Data View Builder or in hand-coded queries. Custom functions, which are implemented as Java methods, allow you to extend the power and functionality of Liquid Data. Queries can invoke custom functions during query execution just as they can standard functions.

A custom function is:

- Implemented in Java code, as described in "Step 1: Write the Custom Function Implementation in Java" on page 8-2.

  You can package Java implementations in a JAR file that is stored in the `custom_lib` folder of the Liquid Data repository. If any custom functions refer to addition Java libraries that are *not* stored in the `custom_lib` folder of the repository, then you *must* specify those folders in the Liquid Data `CLASSPATH` that you configure on the General tab in the Liquid Data node of the Administration Console. For more information, see "Configuring Liquid Data Server Settings" in the Liquid Data *Administration Guide*.

- Declared as a method in a *custom functions library definition* (`.CFLD`) file, as described in "Step 2: Create the Custom Functions Library Definition File" on page 8-4.

  A *function library* is a collection of one or more declared custom functions that Liquid Data manages as a single unit. Each function library usually corresponds to a Java class file that contains the function implementations. However, the function library can also reference functions that are implemented in several Java class files. You store custom functions library definition files in the `custom_functions` folder of the Liquid Data repository.

- Registered on the Repository tab in the in the Administration Console, as described in "Step 3: Register the Custom Function in the Administration Console" on page 8-6.

  Once configured as custom functions, descriptions in the Liquid Data server repository will show up as functions available for use in any Data View Builder client or hand-coded XQuery that connects to this server.

- Invoked in a query in the same way that you would invoke a standard function.

Defining Custom Functions

This section describes the sequence of tasks for defining custom functions for use in the Data View Builder. The process of defining custom functions involves the following steps:

- Step 1: Write the Custom Function Implementation in Java
- Step 2: Create the Custom Functions Library Definition File
- Step 3: Register the Custom Function in the Administration Console

Once a custom function is created, declared, and registered, you can invoke them in queries created using the Data View Builder.

Step 1: Write the Custom Function Implementation in Java

To define a custom function, you first write its implementation in Java and then compile it. The custom function implementation can exist in a single or multiple Java class files. A single Java class file can contain implementations of multiple custom functions. You package Java implementation in a JAR file that is stored in the `custom_lib` folder of the Liquid Data repository.

For examples of custom function implementations, see:

- "Implementation of Custom Functions for Simple Types" on page 8-7
- "Implementation of a Custom Function for a Complex Type" on page 8-12

Rules for Writing Custom Function Implementations

When writing a custom function, you must comply with the following rules:

- Declare the custom function as a static method.
- For parameters and returned values, you must use the data types described in Table 8-1, "Relationship Between XML and Java Data Types," on page 8-3.
- Use an alphabetic character (A-Z a-z) as the first letter of your custom function name.

Correspondence Between XML and Java Data Types

The following table describes the correspondence between XML and Java data types.

Note: For XML data types, the `xs` prefix corresponds to the XML schema namespace described at the following URL: `http://www.w3.org/2001/XMLSchema`.

Table 8-1 Relationship Between XML and Java Data Types

XML Data Type	Corresponding Java Data Type
`xs:boolean`	`java.lang.Boolean`
`xs:byte`	`java.lang.Byte`
`xs:short`	`java.lang.Short`
`xs:integer`	`java.lang.Integer`
`xs:long`	`java.lang.Long`
`xs:float`	`java.lang.float`
`xs:double`	`java.lang.double`
`xs:decimal`	`java.math.BigDecimal`
`xs:string`	`java.lang.String`
`xs:dateTime`	`java.util.Calendar`
Complex Element Type	`org.w3c.dom.Element`

Step 2: Create the Custom Functions Library Definition File

After implementing a custom function in Java, you must declare the custom function in a custom functions library definition (CFLD) file. A CFLD file describes each custom function in a structured XML format. You store custom functions library definition files in the `custom_functions` folder of the Liquid Data repository.

For examples of custom function implementations, see:

- "CFLD File That Declares Custom Functions for Simple Types" on page 8-9
- "CFLD File That Declares the Custom Function for a Complex Type" on page 8-12

Contents of a CFLD File

A CFLD file contains the following information:

- Complex element definitions (for custom functions that operate on complex types)
- Custom function signatures
- Custom function implementation bindings—function name, return type, class, method, and any arguments
- Run-time attributes—running the custom function synchronously or asynchronously

Structure of a CFLD File

A CFLD file has the following structure:

Listing 8-1 Structure of a CFLD File

```
<?xml version = "1.0" encoding = "UTF-8"?>
<definitions>
   <types>
      <xs:schema> complex types </xs:schema>
   </types>
   <functions>
      <function name="Name of the function" return_type="Return Type"
```

```
        class="Implementation class" method="Implementation method"
           asynchronous="boolean value"? > *
        <argument type="Argument Type" label="Argument label"/> *
        <presentation group="Data View Builder Presentation Group" />
        <description>Function Description</description>
      </function>
   </functions>
</definitions>
```

Elements and Attributes in a CFLD File

The following table describes the elements in a CFLD file.

Table 8-2 Elements in a CFLD File

Element	Attribute	Description
`<types>`		Declares any complex data types that a custom function can accept as parameters or return as results, if applicable.
`<functions>`		Function definitions for all functions.
`<function>`		Function definition for a single function.
	`name`	Name of the function in the form of `prefix:localname`. The prefix must be declared in the `<types>` section.
	`return type`	Return type of the function, which can be either a supported XML simple data type or a complex data type declared in the `<types>` section.
	`class`	Implementation class.
	`method`	Implementation method.
	`asynchronous`	Optional. Determines whether the method should be executed asynchronously (`true`) in a separate thread or not (`false`). Specify `true` for functions that execute more slowly than other functions.

Table 8-2 Elements in a CFLD File (Continued)

Element	Attribute	Description
`<argument>`		Argument declarations.
	`type`	Type of the argument (`simple` or `complex`).
	`label`	Optional. Label for the function that the Data View Builder displays in the list.
`<presentation group>`		For a group of related custom functions, if specified, defines the label of a custom tab that appears in the Data View Builder.
`<description>`		Text that describes the function in some detail.

Step 3: Register the Custom Function in the Administration Console

After implementing a custom function and creating the CFLD file, you must register the custom function using the Administration Console. Registration involves the following tasks:

- Adding the JAR and CFLD files for the custom function to the `custom_lib` folder and `custom_functions` folder, respectively, in the Liquid Data Server repository.
- Adding the path to the JAR file in the Custom Functions Classpath field on the General tab in the Liquid Data node, if any other JAR is referenced.
- Creating a custom function description for each set of custom functions.
- If security is enabled, assign roles to the custom function description and to the JAR and CFLD files in the Liquid Data Server repository.

For detailed instructions, see "Configuring Access to Custom Functions" in the Liquid Data *Administration Guide*.

Examples of Custom Functions

This section provides examples of custom functions that use simple and complex types. It includes the following sections:

- Example That Uses Simple Types
- Example That Uses Complex Types

Example That Uses Simple Types

This example shows how to create, declare and use custom functions that operate on simple types.

Implementation of Custom Functions for Simple Types

The following Java code implements custom functions. These functions implement a simple echo operation that returns its argument back to the caller.

Listing 8-2 Java Code for Custom Functions That Use Simple Types

```
package cf;
import java.math.*;
import java.util.Date;
public class CustomFunctions
{
   public static BigDecimal echoDecimal(BigDecimal v)
   {
      return v;
   }
   public static Integer echoInteger(Integer v)
   {
      return v;
   }
   public static Float echoFloat(Float v)
   {
      return v;
   }
   public static String echoString(String v)
   {
      return v;
```

```
    }
    public static Boolean echoBoolean(Boolean v)
    {
       return v;
    }
    public static Calendar echoDateTime(Calendar v)
    {
       return v;
    }
    public static Long echoLong(Long v)
    {
       return v;
    }
    public static Short echoShort(Short v)
    {
       return v;
    }
    public static Byte echoByte(Byte v)
    {
       return v;
    }
    public static Double echoDouble(Double v)
    {
       return v;
    }
}
```

CFLD File That Declares Custom Functions for Simple Types

The following sample CFLD file declares the custom functions for simple types.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<definitions>
   <types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      </xs:schema>
   </types>
   <functions>
      <function name="echoString" return_type="xs:string"
         class="cf.CustomFunctions" method="echoString" >
         <argument type="xs:string" />
      </function>
      <function name="echoBoolean" return_type="xs:boolean"
         class="cf.CustomFunctions" method="echoBoolean" >
         <argument type="xs:boolean" />
      </function>
      <function name="echoByte" return_type="xs:byte"
         class="cf.CustomFunctions" method="echoByte" >
         <argument type="xs:byte" />
      </function>
      <function name="echoShort" return_type="xs:short"
         class="cf.CustomFunctions" method="echoShort" >
         <argument type="xs:short" />
      </function>
      <function name="echoInteger" return_type="xs:integer"
         class="cf.CustomFunctions" method="echoInteger" >
```

```
            <argument type="xs:integer" />
        </function>
        <function name="echoLong" return_type="xs:long"
            class="cf.CustomFunctions" method="echoLong" >
            <argument type="xs:long" />
        </function>
        <function name="echoFloat" return_type="xs:float"
            class="cf.CustomFunctions" method="echoFloat" >
            <argument type="xs:float" />
        </function>
        <function name="echoDouble" return_type="xs:double"
            class="cf.CustomFunctions" method="echoDouble" >
            <argument type="xs:double" />
        </function>
        <function name="echoDecimal" return_type="xs:decimal"
            class="cf.CustomFunctions" method="echoDecimal" >
            <argument type="xs:decimal" />
        </function>
        <function name="echoDateTime" return_type="xs:dateTime"
            class="cf.CustomFunctions" method="echoDateTime" >
            <argument type="xs:dateTime" />
        </function>
    </functions>
</definitions>
```

Query That Uses the Custom Functions for Simple Types

After the function library is registered in Liquid Data, it can be called from the following query (`mycf` is the logical name specified in the CFLD file):

```
let
   $es:=mycf:echoString("hello"),
   $ebool:=mycf:echoBoolean(xf:true()),
   $eb:=mycf:echoByte(cast as xs:byte("127")),
   $eh:=mycf:echoShort(cast as xs:short("32767")),
   $ei:=mycf:echoInteger(cast as xs:integer("2147483647")),
   $el:=mycf:echoLong(cast as xs:long("92233720368547758O7")),
   $ef:=mycf:echoFloat(cast as xs:float("1.0")),
   $ed:=mycf:echoDouble(cast as xs:double("2.0")),
   $edec:=mycf:echoDecimal(cast as xs:decimal("1.5")),
   $edateTime:=mycf:echoDateTime(cast as xs:dateTime("1999-05-31
13:20:00.0")),
return
   <echo>
      <string>{$es}</string>
      <boolean>{$ebool}</boolean>
      <byte>{$eb}</byte>
      <short>{$eh}</short>
      <integer>{$ei}</integer>
      <long>{$el}</long>
      <float>{$ef}</float>
      <double>{$ed}</double>
      <decimal>{$edec}</decimal>
      <dateTime>{$edateTime}</dateTime>
   </echo>
```

Example That Uses Complex Types

This example shows how to create, declare and use a custom function that takes a complex type as a parameter and returns a complex type.

Implementation of a Custom Function for a Complex Type

The following Java code implements a custom function for a complex type. This function simply returns its parameter.

Listing 8-3 Custom Function for a Complex Type

```
package mycf;
import org.w3c.dom.Element;
public static Element echoElement(Element v)
{
   return v;
}
```

CFLD File That Declares the Custom Function for a Complex Type

The following sample CFLD file declares the custom function for a complex type.

Listing 8-4 CFLD File That Declares the Custom Function for a Complex Type

```
<?xml version = "1.0" encoding = "UTF-8"?>
<definitions>
   <types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
         <xs:element name = "book">
            <xs:complexType>
```

```
            <xs:sequence>
               <xs:element ref = "title"/>
               <xs:element ref = "author" maxOccurs = "unbounded"/>
               <xs:element ref = "publisher"/>
               <xs:element ref = "price"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name = "title" type = "xs:string"/>
      <xs:element name = "author">
         <xs:complexType>
            <xs:sequence>
               <xs:element ref = "last"/>
               <xs:element ref = "first"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name = "publisher" type = "xs:string"/>
      <xs:element name = "price" type = "xs:string"/>
      <xs:element name = "last" type = "xs:string"/>
      <xs:element name = "first" type = "xs:string"/>
   </xs:schema>
</types>
<functions>
   <function name="echoBook" return_type="book"
      class="mycf.CustomFunctions2" method="echoElement" >
      <argument type="book" />
```

```
      </function>
   </functions>
</definitions>
```

Query That Uses the Custom Function for a Complex Type

After the function is registered in Liquid Data, it can be called from the following query:

Listing 8-5 Sample Query That Uses the Custom Function for a Complex Type

```
for $b in document("bib")//book
let $c:=echoBook($b)
return
<ans>
{
   for $t in $c/title
   return $t
}
</ans>
```

Setting Complex Parameter Types

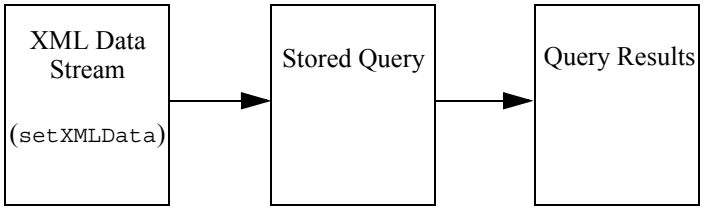
This section describes how to set an XML data stream to input to a complex parameter type. It contains the following sections:

Architecture of Complex Parameter Types

Complex parameter types provide a facility to use streaming XML data as an input to Liquid Data. You can define an XML data stream of an arbitrary type, and you can use that XML data as input to a query.

Figure 9-1 shows the overall architecture of sending XML data as an input to a query.

Figure 9-1 Setting XML Data as an input to a stored query



To evaluate a Complex Parameter Type query from the EJB API, you use the `setXMLData` method on the `QueryParameters` object.

The following Java code sets the XML data for the input to a stored query, then executes the stored query.

```
query = (Query) home.create();
QueryParameters qp = new QueryParameters();
qp.setXMLData("CPTSAMPLE", queryParam);
qr= query.executeStored(queryName, qp);
```

where `CPTSAMPLE` is a stored query and `queryParam` is some XML `String` value.

Sample Complex Parameter Type Code

This section provides sample Java code, using the `QueryParameters.setXMLData` method, to input XML data into a complex parameter type. For information on defining complex parameter types to Liquid data, see Using Complex Parameter Types in the *Administration Guide*. For information on using complex parameter types in the Data View Builder, see Using Complex Parameter Types In Queries in *Building Queries and Data Views*.

The Data View Builder project for the example shown here is installed in the following directory:

```
BEA_HOME/weblogic81/samples/liquiddata/buildQuery/db-cpt
```

The code shown in this sample is installed as the following file:

```
BEA_HOME/weblogic81/samples/liquiddata/ejbAPI/src/ejbSample/QueryWithCp
       tParamClient.java
```

The section is divided into the following parts:

- Sample Query
- Sample Code
- Compiling and Running the Sample Code

Sample Query

Assume the following sample query is saved in the Liquid Data repository as the stored query named `crm_cptSample.xq`. This query uses a complex parameter type (`CPTSAMPLE`) which contains promotion plan names, and then combines those promotion plan names with the details from the CRM database (`PB-CR`).

```
{--Generated by Data View Builder 8.1--}
namespace crm1 = "urn:schemas-bea-com:ld-crmp"
namespace crm = "urn:schemas-bea-com:ld-cptSample"

<crm1:db>
  {
  for $CPTSAMPLE.PROMOTION_2 in ($#CPTSAMPLE of type element
        crm:db)/crm:PROMOTION
  let $PROMOTION_PLAN_3 :=
    for $PB_CR.PROMOTION_PLAN_4 in
          document("PB-CR")/db/PROMOTION_PLAN
    where ($CPTSAMPLE.PROMOTION_2/crm:PROMOTION_NAME  eq
          $PB_CR.PROMOTION_PLAN_4/PROMOTION_NAME)
    return
    <PROMOTION_PLAN>
    <PROMOTION_NAME>{ xf:data($PB_CR.PROMOTION_PLAN_4/PROMOTION_NAME)
}</PROMOTION_NAME>
    <PLAN_NAME>{ xf:data($PB_CR.PROMOTION_PLAN_4/PLAN_NAME)}</PLAN_NAME>
    <FROM_DATE>{ cast as
       xs:string(xf:data($PB_CR.PROMOTION_PLAN_4/FROM_DATE)) }</FROM_DATE>
    <TO_DATE>{ cast as xs:string(xf:data($PB_CR.PROMOTION_PLAN_4/TO_DATE))
            }</TO_DATE>
    <PRICE>{ cast as xs:string(xf:data($PB_CR.PROMOTION_PLAN_4/PRICE)) }
    </PRICE>
    </PROMOTION_PLAN>
  where xf:not(xf:empty($PROMOTION_PLAN_3))
    return
    <PROMOTION>
    <STATE>{ xf:data($CPTSAMPLE.PROMOTION_2/crm:STATE) }</STATE>
    <PROMOTION_NAME>{ xf:data($CPTSAMPLE.PROMOTION_2/crm:PROMOTION_NAME)
}</PROMOTION_NAME>
    { $PROMOTION_PLAN_3 }
```

```
  </PROMOTION>
  }
</crm1:db>
```

Sample Code

The following code sample shows the `setXMLData` method used to input data into a query that uses a complex parameter type source. To simplify the sample code, this sample creates a `String` variable named `queryParam` to represent the XML data stream; you typically will use the `setXMLData` method to reference an object which contains XML data.

```
package ejbSample;

import java.rmi.RemoteException;
import java.util.Properties;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.bea.ldi.server.*;
import com.bea.ldi.server.common.*;

import java.io.*;
import java.rmi.*;

public class QueryWithCptParamClient {
   private String url=null;
   private String JNDI_NAME="bea.ldi.server.QueryHome";
   private String queryName=null;
   private static QueryHome home=null;
   private  Query  query =null;

 //  public static boolean stop=false;

   private String queryParam =
 "<?xml version=\"1.0\" encoding=\"UTF-8\"?> " +
 "     <db xmlns=\"urn:schemas-bea-com:ld-cptSample\"
```

```
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"

xsi:schemaLocation=\"urn:schemas-bea-com:ld-cptSample
crm-p-cptSample.xsd\">" +
   "<PROMOTION>" +
     "<STATE>CA</STATE>" +
     "<PROMOTION_NAME>BROADBAND UPSELL</PROMOTION_NAME>" +
 " </PROMOTION>" +
   "<PROMOTION>" +
     "<STATE>TX</STATE>" +
     "<PROMOTION_NAME>WIRELESS UPSELL</PROMOTION_NAME>" +
   "</PROMOTION>" +
   "<PROMOTION>" +
     "<STATE>WA</STATE>" +
     "<PROMOTION_NAME>NEW PRODUCTS</PROMOTION_NAME>" +
   "</PROMOTION>" +
   "<PROMOTION>" +
     "<STATE>AZ</STATE>" +
     "<PROMOTION_NAME>HOLIDAY PROMOTION</PROMOTION_NAME>" +
   "</PROMOTION>" +
   "<PROMOTION>" +
     "<STATE>NV</STATE>" +
 "      <PROMOTION_NAME>SALES PROMOTION</PROMOTION_NAME>" +
 "       </PROMOTION>" +
 "       </db>";

   /* normally you would pass the argument for the parameter. But
    * in this example we are hardcoding the XML data stream
   */
   public QueryWithCptParamClient(String url, String queryName){
          this.url= url;
          this.queryName=queryName;
   }
   public static void main(String[] args) throws Exception
   {
        QueryWithCptParamClient qpc = new QueryWithCptParamClient(args[0],
args[1]);
          qpc.runQuery();
```

```
 }

 public void runQuery() throws Exception{
    QueryResult qr=null;
    try{
         if(home==null)
            home = lookupHome();
       // log("Creating a query client");
        query = (Query) home.create();
        QueryParameters qp = new QueryParameters();
        qp.setXMLData("CPTSAMPLE", queryParam);
        qr= query.executeStored(queryName, qp);
        System.out.println("Query Result: >>>>>>>\n");
        if(!qr.isEmpty())
          qr.printWithFormat(new OutputStreamWriter(System.out), true);
    }catch(Exception e){
        throw e;
    }finally{
        qr.close();
    }
 }


/**
 * Lookup the EJBs home in the JNDI tree
 */
private QueryHome lookupHome()
  throws NamingException
{
  // Lookup the beans home using JNDI
  Context ctx = getInitialContext();

  try {
    return (QueryHome)ctx.lookup(JNDI_NAME);
   } catch (NamingException ne) {
     ne.printStackTrace();
   log("The client was unable to lookup the EJBHome.  Please make sure ");
    log("that you have deployed the ejb with the JNDI name "+JNDI_NAME+"
```

```
on the WebLogic server at "+url);
      throw ne;
    }
  }

   private Context getInitialContext() throws NamingException {

    try {
      // Get an InitialContext
      Properties h = new Properties();
      h.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
      h.put(Context.PROVIDER_URL, url);
      return new InitialContext(h);
    } catch (NamingException ne) {
      ne.printStackTrace();
      log("We were unable to get a connection to the WebLogic server at
"+url);
      log("Please make sure that the server is running.");
      throw ne;
    }
  }

  private static void log(String s) {
    System.out.println(s);
  }

}
```

Compiling and Running the Sample Code

Perform the following steps to build and run the `crm-cpt` complex parameter type example.

1. Open a command window.
2. Navigate to the *BEA_HOME*`/weblogic81/samples/domains/liquiddata` directory as in the following example:

   ```
   cd /bea/weblogic81/samples/domains/liquiddata
   ```

3. Run the `setLDExamplesEnv.cmd` script (`setLDExamplesEnv.sh` on UNIX systems) to set up the environment for the samples, as follows:

   ```
   setLDExamplesEnv
   ```

4. Change to the *BEA_HOME*`/weblogic81/samples/liquiddata/ejbAPI/build` directory as in the following example:

   ```
   cd /bea/weblogic81/samples/liquiddata/ejbAPI/build
   ```

5. Run `ant` to build the samples, as follows:

   ```
   ant
   ```

6. Change to the *BEA_HOME*`/weblogic81/samples/liquiddata/ejbAPI/obj` directory as in the following example:

   ```
   cd /bea/weblogic81/samples/liquiddata/ejbAPI/obj
   ```

7. Run the following command to add the local directory to your `CLASSPATH` environment directory:

   ```
   set classpath=%CLASSPATH%;./
   ```

8. Run the sample with the following command, which specifies the Java class with the URL of WebLogic Server as the first argument and the name of the query to run as the second argument:

   ```
   java ejbSample.QueryWithCptParamClient t3://localhost:7001 crm_cptSample
   ```

Note: Make sure your Liquid Data Samples domain is running or this command will fail.

When you run this sample successfully, results similar to the following appear in your command window:

```
D:\bea\weblogic81\samples\liquiddata\ejbAPI\obj>java
ejbSample.QueryWithCptParamClient t3://localhost:7001 crm_cptSample
Result: >>>>>>>

<crm1:db xmlns:crm1="urn:schemas-bea-com:ld-crmp">
  <PROMOTION>
    <STATE>CA</STATE>
    <PROMOTION_NAME>BROADBAND UPSELL</PROMOTION_NAME>
    <PROMOTION_PLAN>
      <PROMOTION_NAME>BROADBAND UPSELL</PROMOTION_NAME>
      <PLAN_NAME>High Speed Holidays</PLAN_NAME>
      <FROM_DATE>2001-11-22</FROM_DATE>
      <TO_DATE>2002-12-31</TO_DATE>
      <PRICE>100</PRICE>
    </PROMOTION_PLAN>
  </PROMOTION>
  <PROMOTION>
    <STATE>TX</STATE>
    <PROMOTION_NAME>WIRELESS UPSELL</PROMOTION_NAME>
    <PROMOTION_PLAN>
      <PROMOTION_NAME>WIRELESS UPSELL</PROMOTION_NAME>
      <PLAN_NAME>Family Holiday Connect</PLAN_NAME>
      <FROM_DATE>2001-11-22</FROM_DATE>
      <TO_DATE>2002-12-31</TO_DATE>
      <PRICE>49.99</PRICE>
    </PROMOTION_PLAN>
  </PROMOTION>
  <PROMOTION>
    <STATE>WA</STATE>
    <PROMOTION_NAME>NEW PRODUCTS</PROMOTION_NAME>
    <PROMOTION_PLAN>
      <PROMOTION_NAME>NEW PRODUCTS</PROMOTION_NAME>
      <PLAN_NAME>New Phone for the Holidays</PLAN_NAME>
      <FROM_DATE>2001-11-22</FROM_DATE>
      <TO_DATE>2002-12-31</TO_DATE>
```

```
        <PRICE>149.99</PRICE>
      </PROMOTION_PLAN>
    </PROMOTION>
    <PROMOTION>
      <STATE>AZ</STATE>
      <PROMOTION_NAME>HOLIDAY PROMOTION</PROMOTION_NAME>
      <PROMOTION_PLAN>
        <PROMOTION_NAME>HOLIDAY PROMOTION</PROMOTION_NAME>
        <PLAN_NAME>New Year New Connections</PLAN_NAME>
        <FROM_DATE>2001-11-22</FROM_DATE>
        <TO_DATE>2002-12-31</TO_DATE>
        <PRICE>39.99</PRICE>
      </PROMOTION_PLAN>
    </PROMOTION>
    <PROMOTION>
      <STATE>NV</STATE>
      <PROMOTION_NAME>SALES PROMOTION</PROMOTION_NAME>
      <PROMOTION_PLAN>
        <PROMOTION_NAME>SALES PROMOTION</PROMOTION_NAME>
        <PLAN_NAME>Family Plan</PLAN_NAME>
        <FROM_DATE>2001-11-22</FROM_DATE>
        <TO_DATE>2002-12-31</TO_DATE>
        <PRICE>39.99</PRICE>
      </PROMOTION_PLAN>
    </PROMOTION>
  </crm1:db>
```

Using the Cache Purging APIs

This section describes how to purge the cache for a query using the cache EJB API. It contains the following sections:

- The com.bea.ldi.cache.ejb Package
- Security Issues When Using the Cache APIs
- Writing Java Code to Purge Cache Entries

For details on configuring the Liquid Data results cache, see "Configuring the Results Cache" in the *Administration Guide*.

The com.bea.ldi.cache.ejb Package

The `com.bea.ldi.cache.ejb` package has a `CacheRemote` interface with methods you can use to purge cache entries for queries. There is a `purgeCache()` method to purge the entire cache, and there are methods to purge entries for a specified query. For details on the APIs, see the Javadoc.

Security Issues When Using the Cache APIs

If security is enabled in your Liquid Data domain, then the user who makes the API call to purge the cache must have the necessary permissions to purge the cache. If the user does not have the needed permissions, the purge will fail. For details about setting a security policy for purging the cache, see "Configuring a Security Policy for Purging the Cache Results" in the Liquid Data *Administration Guide*.

Writing Java Code to Purge Cache Entries

You can write Java code to purge the cache entries for a query. There are methods to purge the entire cache, purge all the entries for a specified query, and to purge the instances of a query with specified parameters. This section describes the basic steps required to write Java code to purge cache entries in the results cache, including a sample program, and includes the following subsections:

- Enable Caching in Liquid Data
- Import the Liquid Data Packages
- Lookup the EJB Home in the JNDI Tree
- Sample Cache Purging Code

Enable Caching in Liquid Data

In order for the Liquid Data cache APIs to purge any cache entries, you must configure and enable the results cache in Liquid Data. Setting up and enabling the cache involves the following general steps:

- Creating the cache database
- Creating the cache `LDCacacheDS` data source
- Enabling Results Caching in Liquid Data
- Setting cache policies for individual queries

For details on setting up the Liquid Data results cache, see "Configuring the Results Cache" in the *Administration Guide*.

Import the Liquid Data Packages

Any program that uses the Liquid Data cache APIs must import the Liquid Data packages. Depending on which APIs the program uses, you should have import statements similar to the following:

```
//import the Liquid Data cache and QueryParameters Packages
import com.bea.ldi.cache.ejb.CacheRemote;
import com.bea.ldi.cache.ejb.CacheRemoteHome;
import com.bea.ldi.server.common.QueryParameters;
```

Lookup the EJB Home in the JNDI Tree

A program that purges a result cache entry must find the JNDI entry for the Cache EJB. The Cache EJP has the following JNDI name:

```
bea.ldi.cache.CacheHome
```

If the JNDI name does not exist in the JNDI tree, then the EJB is not properly deployed and the cache APIs will not be available.

The following code shows a sample JNDI lookup.

```
//Declare JNDI_NAME variable
private String JNDI_NAME="bea.ldi.cache.CacheHome";

// Lookup the EJBs home in the JNDI tree
   private CacheRemoteHome lookupHome()
    throws NamingException
  {
    // Lookup the beans home using JNDI
    Context ctx = getInitialContext();

    try {
      return (CacheRemoteHome)ctx.lookup(JNDI_NAME);
     } catch (NamingException ne) {
       ne.printStackTrace();
     log("The client was unable to lookup the EJBHome.  Please make sure ");
      log("that you have deployed the ejb with the JNDI name "+JNDI_NAME+
          " on the WebLogic server at "+url);
      throw ne;
    }
  }
```

Sample Cache Purging Code

The following sample program uses the `purgeCache` method to purge the Liquid Data query cache.

```
package myFolder;

//import the Liquid Data cache and QueryParameters Packages
import com.bea.ldi.cache.ejb.CacheRemote;
import com.bea.ldi.cache.ejb.CacheRemoteHome;
import com.bea.ldi.server.common.QueryParameters;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;


public class QueryPurgeClient {
   private String url=null;
   private String JNDI_NAME="bea.ldi.cache.CacheHome";
   private String cacheQueryName=null;
   private QueryParameters qparams = null;
   private static CacheRemoteHome home=null;

 //  public static boolean stop=false;

   /* normally you would pass the argument for the parameter. But in this
    * example we are hardcoding the stream
   */
   public QueryPurgeClient(String url, String queryName,
                           QueryParameters qparams){
    this.url= url;
    this.cacheQueryName=queryName;
    this.qparams = qparams;

   }

   public static void main(String[] args) throws Exception  {
    // ignore query parameters for now
    QueryPurgeClient qpc = new QueryPurgeClient(args[0], args[1],null);
    qpc.runPurgeCache(args[1]);
   }
```

```
  // Create a method to purge cache entries for a specified query
  // This method uses com.bea.ldi.cache.ejb.CacheRemote.purgeCache(queryName)
  //
   public void runPurgeCache(String queryName) throws Exception{

      try{
           if(home==null)
              home = lookupHome();
         // log("Creating a query client");
          CacheRemote cacheQuery = (CacheRemote) home.create();
          cacheQuery.purgeCache(queryName);
          log("Purged all cached instances of: >>>>>>>\n" + cacheQueryName);
      } catch(Exception e){
          e.printStackTrace();
          throw e;
      }
   }

// Lookup the EJBs home in the JNDI tree
  private CacheRemoteHome lookupHome()
    throws NamingException
  {
    // Lookup the beans home using JNDI
    Context ctx = getInitialContext();

    try {
      return (CacheRemoteHome)ctx.lookup(JNDI_NAME);
     } catch (NamingException ne) {
       ne.printStackTrace();
      log("The client was unable to lookup the EJBHome.  Please make sure ");
      log("that you have deployed the ejb with the JNDI name "+JNDI_NAME+
          " on the WebLogic server at "+url);
      throw ne;
    }
  }
```

```
// Get the context for WebLogic Server, to make sure it is running
  private Context getInitialContext() throws NamingException {

    try {
      // Get an InitialContext
      Properties h = new Properties();
      h.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
      h.put(Context.PROVIDER_URL, url);
      return new InitialContext(h);
    } catch (NamingException ne) {
      ne.printStackTrace();
      log("We were unable to get a connection to the WebLogic server at "+url);
      log("Please make sure that the server is running.");
      throw ne;
    }
  }

  private static void log(String s) {
    System.out.println(s);
  }
}
```

Index

J

L

P

Q

R

S

T

W

X