BEA Liquid Data for WebLogic™

Invoking Liquid Data Queries Programmatically

Release: 1.0
Document Date: October 2002
Revised:

Invoking Liquid Data Queries Programmatically

Part Number	Date	Software Version
N/A	October 2002	1.0

Contents

About This Document

What You Need to Know viii
e-docs Web Site viii
How to Print the Document viii
Related Information ix
Contact Us! ix
Documentation Conventions x

1. About the Liquid Data Query API

About Liquid Data Queries 1-1
- Stored Queries 1-2
- Ad Hoc Queries 1-2
- Parameterized Queries 1-2

Components of the Liquid Data Query API 1-3
- Packages 1-3
- Query Execution EJB 1-3
- Query Parameters 1-4
- Query Attributes 1-5
- Query Results 1-5

Types of Java Clients 1-5
- EJB Clients 1-6
- JSP Clients 1-6

2. Invoking Queries in EJB Clients

- Step 1: Connect to the Liquid Data Server 2-1
- Step 2: Specify Query Parameters 2-4
- Step 3: Execute the Query 2-6

Step 4: Process the Results of the Query 2-8

3. Invoking Queries in JSP Clients

About the Liquid Data Tag Library 3-1
Scope of the Liquid Data Tag Library 3-2
Location of the Liquid Data Tag Library 3-2
Tags in the Liquid Data Tag Library 3-2
query Tag 3-2
param Tag 3-3
Processing Steps 3-3
Step 1: Reference the Liquid Data Tag Library 3-4
Step 2: Connect to the Liquid Data Server 3-4
Step 3: Specify Query Parameters 3-5
Step 4: Execute the Query 3-5
Executing Stored Queries 3-6
Executing Ad Hoc Queries 3-6
Handling Exceptions 3-7
Step 5: Process the Query Results 3-7

4. Invoking Queries in Web Service Clients

Finding the WSDL URL for Generated Web Services 4-1
Invoking Web Services Programmatically 4-2

5. Invoking Queries in Business Process Manager Applications

Liquid Data and the BPM Component 5-1
Setting Up a Query Invocation in BPM Client Applications 5-2

6. Invoking Queries in BEA WebLogic Portal Applications

Invoking Liquid Data Queries as EJB Clients 6-1
Invoking Liquid Data Queries as JSP Clients 6-2

7. Using Custom Functions

About Custom Functions 7-1
Defining Custom Functions 7-2
Step 1: Write the Custom Function Implementation in Java 7-3
Rules for Writing Custom Function Implementations 7-3

Correspondence Between XML and Java Data Types........................ 7-3
Step 2: Create the Custom Functions Library Definition File 7-4
Contents of a CFLD File... 7-4
Structure of a CFLD File 7-5
Elements and Attributes in a CFLD File.. 7-6
Step 3: Register the Custom Function in the Administration Console 7-7
Examples of Custom Functions.. 7-7
Example That Uses Simple Types 7-7
Implementation of Custom Functions for Simple Types.................... 7-8
CFLD File That Declares Custom Functions for Simple Types......... 7-9
Query That Uses the Custom Functions for Simple Types............... 7-11
Example That Uses Complex Types 7-13
Implementation of a Custom Function for a Complex Type 7-13
CFLD File That Declares the Custom Function for a Complex Type...... 7-13
Query That Uses the Custom Function for a Complex Type............ 7-15

Index

About This Document

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

This following topics are covered:

- Chapter 1, "About the Liquid Data Query API," describes concepts that you'll need to understand in order to invoke Liquid Data queries programmatically.
- Chapter 2, "Invoking Queries in EJB Clients," describes how to invoke Liquid Data queries from EJB clients.
- Chapter 3, "Invoking Queries in JSP Clients," describes how to invoke Liquid Data queries from JSP clients.
- Chapter 4, "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 5, "Invoking Queries in Business Process Manager Applications," describes how to invoke Liquid Data queries as Business Operations in Business Process Manager workflows.
- Chapter 6, "Invoking Queries in BEA WebLogic Portal Applications," describes how to invoke Liquid Data queries in BEA WebLogic Portal applications.
- Chapter 7, "Using Custom Functions," describes how to write Java code for custom functions that extend the power and functionality of Liquid Data.

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 WebLogic1.0 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`*

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	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 About the Liquid Data Query API

This topic describes concepts that you need to understand in order to invoke queries programmatically using the BEA Liquid Data for WebLogic™ Query API. It contains the following sections:

- 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 *Product Overview*.

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 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 topic 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:

- Standalone Java applications
- Java servlets
- Java Server Pages (JSPs)
- Java Beans
- Other EJBs
- Business operations in workflows that execute in the Business Process Management (BPM) component of WebLogic Integration
- WebLogic Portal

- Web Services

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

EJB Clients

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 2, "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 Clients

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 3, "Invoking Queries in JSP Clients."

CHAPTER

2 Invoking Queries in EJB Clients

This topic 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 Clients" on page 1-6.

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 2-1 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 1-2.

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

Listing 2-2 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 2-3 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. 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 2-4 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

3 Invoking Queries in JSP Clients

This topic 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 Clients" on page 1-6.

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 topic 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
- 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.

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 3-1 Attributes of the query tag

Attribute	Description
`name`	Specifies the name of a stored query from which to retrieve results.

Table 3-1 Attributes of the query tag (Continued)

Attribute	Description
`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.

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

```
<lds:query name="MyStoredQuery" server="t3://222.222.22:7001">
```

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 3-2 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: Reference the Liquid Data Tag Library
- Step 2: Connect to the Liquid Data Server

- Step 3: Specify Query Parameters
- Step 4: Execute the Query
- Step 5: Process the Query Results

Step 1: 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 3-2, including the following code near the top of each JSP page:

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

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

Step 2: 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 3-1 Non-Local JSP Client Connecting to Liquid Data Server

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

...

<lds:query ... server="t3://222.222.22:7001">

...

</lds:query>
```

Step 3: 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 3-3.

Listing 3-2 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 4: Execute the Query

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

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 3-3 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 3-4 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 5: 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.

A typical post-processing step followed by JSP clients at this point would be to apply an XSL transform to the query result XML content in order to convert it to a presentable format. This can normally be conveniently accomplished 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 3-5 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>
```

CHAPTER

4 Invoking Queries in Web Service Clients

This topic 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

5 Invoking Queries in Business Process Manager Applications

This topic describes how to invoke BEA Liquid Data for WebLogic™ queries from workflows in the Business Process Manager (BPM) component of BEA WebLogic Integration. It contains the following sections:

- Liquid Data and the BPM Component
- Setting Up a Query Invocation in BPM Client Applications

Liquid Data and the BPM Component

Workflows in the Business Process Management (BPM) component of WebLogic Integration can invoke Liquid Data queries in BPM *business operations* using the Liquid Data EJB API. For comprehensive information about BPM, see Business Process Management in the WebLogic Integration documentation.

A business operation represents a method call on an EJB, including any variables that are passed to it as parameters, and result values that are returned to the workflow.

You define business operations using the business operations facility in the WebLogic Integration Studio, as described in "Configuring Business Operations" in Configuring Workflow Resources in *Using the WebLogic Integration Studio*. Once defined, individual workflows can use the Perform Business Operation action to invoke the business operation and, optionally, assign the results of the query to a workflow variable.

Setting Up a Query Invocation in BPM Client Applications

To invoke a query using the Liquid Data EJB API, you must first define the business operation using the WebLogic Integration Studio. For each business operation, you define the name of the business operation, the JNDI name of the Query EJB to be invoked (`com.bea.ldi.server.QueryHome`), and the method to invoke. Stateless session EJB references persist for the duration of a workflow instance.

For more information, see "Configuring Business Operations" in Configuring Workflow Resources in *Using the WebLogic Integration Studio*.

6 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 topic 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 Development Guide" in 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 2, "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 3, "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 6-1 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

7 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 7-3.

  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 7-4.

  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 7-7.

  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 7-8
- "Implementation of a Custom Function for a Complex Type" on page 7-13

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 7-1, "Relationship Between XML and Java Data Types," on page 7-3.

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 7-1 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`

Table 7-1 Relationship Between XML and Java Data Types

XML Data Type	Corresponding Java Data Type
`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 7-9
- "CFLD File That Declares the Custom Function for a Complex Type" on page 7-13

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 7-1 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 7-2 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.
`<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 ACLs 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 topic 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 7-2 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("9223372036854775807")),
   $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 7-3 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 7-4 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 7-5 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>
```

Index

A

ad hoc queries
  defined 1-2
  executing in JSP clients 3-6

B

`bea.ldi.server.QueryBean` stateless session bean 1-3
Business Process Manager
  invoking queries from 5-1
  setting up query invocation 5-2

C

clients
  EJB clients 1-6
  JSP clients 1-6
  types of clients 1-5
  Web service clients 4-1
`com.bea.ldi.server` package 1-3
`com.bea.ldi.server.common` package 1-3
connecting to the Liquid Data Server
  EJB clients 2-1
  JSP clients 3-4
custom function library definition (CFLD) files
  attributes, described 7-6
  contents of 7-4
  creating 7-4
  elements, described 7-6
  structure of 7-5
custom functions
  about custom functions 7-1
  custom function library definition (CFLD) files, creating 7-4
  defining 7-2
  implementing in Java 7-3
  registering in the Administration Console 7-7
  samples
    complex types 7-13
    simple types 7-7
customer support contact information 1-ix

D

documentation, where to find it 1-viii

E

EJB clients
  connecting to Liquid Data server 2-1
  defined 1-6
  executing queries 2-6
  invoking queries 2-1
  processing results 2-8
  query parameters, specifying 2-4
executing queries
  EJB clients 2-6
  JSP clients 3-5

I

invoking queries
- Business Process Manager component of WebLogic Integration 5-1
- EJB clients 2-1
- JSP clients 3-3
- Web service clients 4-1
- WebLogic Portal
  - EJB client 6-1
  - JSP client 6-2

J

JSP clients
- connecting to the server 3-4
- defined 1-6
- executing queries 3-5
- handling exceptions 3-7
- invoking queries 3-3
- processing results 3-7
- tag library 3-1

L

Liquid Data Query API
- `bea.ldi.server.QueryBean` 1-3
- packages 1-3
- query attributes 1-5
- query parameters 1-4

P

packages in the Liquid Data Query API 1-3
`param` tag 3-3
parameterized queries
- described 1-2
- EJB clients 2-4

printing product documentation 1-viii
processing query results
- EJB clients 2-8
- JSP clients 3-7

Q

queries
- about queries in Liquid Data 1-1
- ad hoc queries 1-2, 3-6
- attributes 1-5
- execution in EJB clients 2-6
- parameterized queries 1-2
- parameters 1-4, 2-4, 3-5
- result processing in EJB clients 2-8
- stored queries 1-2, 3-6

query parameters
- EJB clients 2-4
- JSP clients 3-5

query results 1-5
`query` tag 3-2

R

related information 1-ix

S

server, EJB client connecting to 2-1
stored queries
- defined 1-2
- executing in JSP clients 3-6

support, technical 1-ix

T

tag library 3-1
tags
- connecting to the server 3-4
- `param` tag 3-3
- query parameters 3-5
- `query` tag 3-2

W

Web service clients
- invoking queries 4-1

invoking Web services
programmatically 4-2
WSDL URL, finding 4-1
WebLogic Portal
EJB clients 6-1
JSP clients 6-2
WSDL URL, in Web service clients 4-1

X

XSL transform 3-8