Jdbc Control Developer's Guide

Table of contents

1. Jdbc Control Annotation's Reference

The Jdbc Control uses Java 1.5 annotations extensively. All annotations are defined in the org.apache.beehive.controls.system.jdbc.JdbcControl interface. Whenever possible annotations are checked for validity during compile time using an apt processor. The compile time checks include parsing the _statement_ member of the SQL annotation to make sure it conforms to the parameter substitution syntax expected by the Jdbc Control.

1.1. The ConnectionDataSource Annotation

The ConnectionDataSource annotation is a class-level annotation used to lookup a DataSource using the JNDI service.

Member Name	Value Type	Value Required	Description
jndiName	String	Yes	A data source name which can be used for a JNDI lookup
jndiContextFactory	Class <? extends JndiContextFactory>	No	A JNDI context factory

1.2. The ConnectionDriver Annotation

The ConnectionDriver annotation is a class-level annotation used to connect directly to a database instance using a connection URL.

Member Name	Value Type	Value Required	Description
databaseDriverClass	java.lang.Class	Yes	The database driver class
databaseURL	String	Yes	The database connection URL
userName	String	No	The username to connect to the database with
password	String	No	The password associated with userName
properties	String	No	A comma seperated list of properties for the connection

1.3. The ConnectionOptions Annotation

The ConnectionOptions annotation is a class-level annotation used to set options on a JDBC connection. It is used in conjunction with the ConnectionDataSource and ConnectionDriver annotations but is not required.

Member Name	Value Type	Value Required	Description
readOnly	|boolean	No	If set to true tells the database to optimize the connection for read-only access (still can do updates, etc), defaults to false
resultSetHoldability	HoldabilityType	No	Specifies ResultSet cursor holdability, defaults to close cursors after commit
typeMappers	TypeMapper[]	No	Type mappers implement the java.sql.SQLData interface and handle mappings between SQL UDTs and Java classes

1.4. The SQL Annotation

The SQL annotation is method annotation which specifies the SQL to send to the database as well as any other options for the query.

Member Name	Value Type	Value Required	Description
statement	String	Yes	The SQL statement to send to the database
arrayMaxLength	int	No	If the method return type is an array type, limit the size of the array to this value
batchUpdate	boolean	No	Defaults to false, JDBC 3.0 batch update
fetchSize	int	No	Performance hint for

			fetching ResultSet rows, defaults to zero, indicating db should determine fetch size.
fetchDirection	FetchDirection	No	Performance hint for fetching ResultSet rows, defaults to forward.
getGeneratedKeys	boolean	No	Defaults to false, JDBC 3.0 generated keys
generatedKeyColumnNa	String array	No	Defines column names of columns with generated keys to be returned
generatedKeyColumnInd	int array	No	Defines column indexes of columns with generated keys to be returned
iteratorElementType	Class	No	Defines type of class to iterate over when method return type is Iterator
maxRows	int	No	Limit the maximum number of rows returned by the database.
resultSetHoldabilityOver	HoldabilityType	No	Overrides value set by ConnectionOptions holdability element for the duration of the method call.
resultSetMapper	Class	No	Defines a custom ResultSetMapper for use with this method
scrollableResultSet	ScrollType enumeration	No	Enables the return of scrollable ResultSet's, default is non-scrollable. See JdbcControl.java for ScrollType values.

typeMappersOverride	TypeMapper[]	No	Overrides typemapper's set in the ConnectionOptions annotation.

2. Parameter Substitution in the SQL Annotation's Statement Member

You can use parameter substitution in the SQL annotation's _statement_ member to form a query dynamically. The client calls the method on the Jdbc control, passing in values for the method's parameters, and those parameter values are substituted into the SQL statement.

This topic describes substitution techniques and rules, including how to treat curly braces, how to substitute whole SQL statements, SQL phrases, simple parameters, and indirect parameters.

2.1. Substitution Criteria

Substitution is subject to the following criteria:

- **Substitution matching is case sensitive.** For example, the method parameter CustCity will not match the substitution pattern {custCity}.
- **The type of the method parameter must be compatible with the type of the associated database field in the statement.** If you attempt to substitute a Java String where the database expects a NUMBER, the statement will fail. For information on mapping between database types and Java types, see Mapping Database Field Types to Java Types in the Database Control.
- **Substitution will not occur if the substitution pattern contains spaces.** The Java Database Connectivity (JDBC) API allows access to built-in database functions via escapes of the form {fn user()}. If spaces occur in an item enclosed in curly braces ({}), the Database control treats the item as a JDBC escape and passes it on without substitution. For example, the custCity method parameter will not be substituted if the substitution is specified as {custCity } or { custCity}. For more information on JDBC escapes, please consult the documentation for your JDBC driver.
- **When substituting date or time values, use the classes in the java.sql package.** For example, attempting to substitute java.util.Date in a SQL Date field will not work. Use java.sql.Date instead.

2.2. Substituting Simple Parameters

If you are substituting individual values into a WHERE, LIKE, or AND clause, you may substitute them directly in the @SQL annotation's statement parameter without escaping the values with the {sql:} substitution syntax.

The following example illustrates simple parameter substitution:

```
@SQL(statement="SELECT name FROM customer WHERE city={custCity} AND
state={custState}")
public String [] getCustomersInCity( String custCity, String custState );
```

The value of the custCity method parameter is substituted in the query in place of the {custCity} item, and the value of the custState method parameter is substituted in the query in place of the {custState} item.

2.3. Treatment of Curly Braces Within Literals

Curly braces (\{\}) within literals (strings within quotes) are ignored. This means statements like the following will not work as you might expect. In the following example the curly braces have lost their substitution functionality, because they appear within single quotes.

```
 @SQL( statement="SELECT name FROM employees WHERE name LIKE
'%{partialName}%'")
public String[] partialNameSearch(String partialName);
```

Since the curly braces are ignored inside the literal string, the expected substitution of the partialName Java String into the SELECT statement does not occur. To avoid this problem, pre-format the match string before invoking the Jdbc control method, as shown below. Note that single quotes are not included in the pre-formatted string because single quotes are implicitly added to the substitution value when it is passed to the SQL query.

```
String partialNameToMatch = "%" + matchString + "%"
String [] names = myJdbcControl.partialNameSeach(partialNameToMatch);
```

Then pass the pre-formatted string to the Jdbc control:

```
@SQL ( statement="SELECT name FROM employees WHERE name LIKE
{partialNameToMatch}")
 public String[] partialNameSearch(String partialNameToMatch);
```

2.4. Substituting Indirect Parameters

Assume the following class is declared and is accessible to the Database control:

```
public static class Customer
{
    public String firstName;
    public String lastName;
```

```
    public String streetAddress;
    public String city;
    private String state;
    public String zipCode;
    public String getState() {return state};
}
```

You can then refer to the members of the Customer class in the SQL statement, as shown in the following example:

```
@SQL( statement="SELECT name FROM customer WHERE city={cust.city} AND
state={cust.state}")
public String [] getCustomersInCity( Customer cust );
```

Note: Class member variables and accessor (getXxx) methods must be public in order for the Database control to substitute them.

The dot notation is used to access the members of the parameter object.

The following list describes the precedence for resolving dot notations in substitutions given the substitution pattern {myClass.myMember}:

- If class myClass exposes public getMyMember() and setMyMember() methods, getMyMember() is called and the return value is substituted. For Boolean variables, substitute isMyMember() for getMyMemnber().
- Else if class myClass exposes a public field named myMember, myClass.myMember is substituted.
- Lastly, if class myClass implements java.util.Map, myClass.get("myMember") is called and the return value is substituted.
- Any combination of these may exist, as in {A.B.C} where B is a public member of A and B has a public getC() method.

If none of these conditions exist, the Jdbc control method will throw a com.bea.control.ControlException.

2.5. Generic Substitution

To pass a whole SQL statement to the database, use the substitution syntax shown in bold.

```
@SQL(statement="{sql: sqlStatement}")
public myRecordType myQuery( String sqlStatement );
```

The SQL statement placed within the bracket syntax {sql: } is escaped and passed directly to the database.

You can use same substitution syntax to pass in any part of a SQL statement, such as a WHERE or LIKE clause, or a column name. In the following example, filtering phrases can be substituted into the base SQL statement.

```
@SQL(statement="SELECT * FROM CUSTOMER {sql: whereClause}")
public myRecordType myQuery( String whereClause );
```

In the following example, a column name is dynamically written to the SQL statement by means of the {sql: } bracket syntax.

```
@SQL(statement="SELECT SUM( {sql: colName} ) FROM MYTABLE")
public int sumColumn(String colName);
```

2.6. Referring to Functions in Substitution Statements

If your database supports internal functions, you can refer to the internal function within the substitution syntax {sql: }. The following method refers to the function in(), by placing the function call within the brackets {sql: }.

```
    @SQL( statement="SELECT * FROM customer WHERE {sql:fn
in(custid,{customerIDs})}")
    Customer[] callInternalFunction(Integer[] customerIDs);
```

Not all databases and database drivers support internal functions within substitution brackets; for example, Oracle drivers do not support this scenario.

2.7. SQL Escapes Support

The SQL annotations statement member supports the use of the SQL Escape syntax within the SQL statement. SQL Escapes follow the standard escape syntax and may contain parameter substitutions. The set of supported escape keywords is:

- escape
- fn
- d
- t
- ts
- call
- ?=
- oj

The following examples illustrate some of the possible usages.

```
@SQL(statement="INSERT INTO USERS (creationDate, userName) VALUES({d
{creationDateFormat}},{userName})
public int addUser(String creationDateFormat, String userName) throws
SQLException;


@SQL(statement="INSERT INTO USERS (userId, userName) VALUES({?=
sp_userId()},{userName})")
public int addUser(String userName) throws SQLException;
```

3. Invoking Stored Procedures with the Jdbc Control

The following topics explain how to call and create stored procedures with the Jdbc Control.

3.1. Calling Stored Procedures with IN Parameters

If the stored procedure contains only IN parameters, you can call the procedure by passing method parameters to the procedure.

Assume the following procedure sp_updateData has been created on the database.

```
    CREATE OR REPLACE PROCEDURE sp_updateData
      (pkID   IN SMALLINT,
      intVal  IN INT)
    AS
    BEGIN
      UPDATE CUSTOMER
      SET NAME = intVal
      WHERE CUSTID = pkID;
    END sp_updateData;
```

The following database control method calls the procedure sp_updateData and passes two method parameters to the procedure.

```
   @SQL(statement="call sp_updateData({keyVal}, {intVal})"
   void call_sp_updateCust(short keyVal, int intVal);
```

The method parameters are substituted into the procedure call using the curly brace substitution syntax.

If you are calling this stored procedure against a Sybase database, you must include curly braces around the stored procedure call. For Sybase, the annotation value should look like this:

```
@SQL(statement="{call sp_updateData({keyVal}, {intVal})}")
```

3.2. Calling Stored Procedures with OUT Parameters

To call a procedure that contains OUT parameters:

1. Use a SQLParameter Array as the parameter of the Java method that calls the procedure.
2. Use question marks as placeholders for the parameters within the procedure call.

The SQLParameter class is an public inner class of JdbcControl.java, source follows:

```
public static class SQLParameter {
    public static final int IN = 1;
    public static final int OUT = 2;
    public static final int INOUT = IN | OUT;

    public Object value = null;
    public int type = Types.NULL;
    public int dir = IN;

    public SQLParameter(Object value) {
        this.value = value;
    }

    public SQLParameter(Object value, int type) {
        this.value = value;
        this.type = type;
    }

    public SQLParameter(Object value, int type, int dir) {
        this.value = value;
        this.type = type;
        this.dir = dir;
    }

    public Object clone() {
        return new SQLParameter(value, type, dir);
    }
}
```

For example, assume that the following procedure sp_squareInt exists on the database.

```
CREATE OR REPLACE PROCEDURE sp_squareInt
   (field1 IN INTEGER, field2 OUT INTEGER) IS
BEGIN
    field2 := field1 * field1;
END sp_squareInt;
```

The following Java method will call the procedure sp_squareInt.

```
@SQL(statement="{call sp_squareInt(?, ?)})"
```

```
    void call_sp_squareInt(SQLParameter[] params) throws SQLException;
```

Note that the method parameter params is not explicitly substituted into the procedure call {call sp_squareInt(?, ?)}. The substitution syntax {call ...} has special meaning within the @SQL statement annotation. When the substitution syntax {call myStoredProc(?,?,?...)} is encountered, it automatically distributes the elements of params into the procedure call.

The following shows how to construct an SQLParameter[] to call the procedure sp_squareInt.

```
        // Construct a SQLParameter[]
        // to hold two SQLParameter objects
        SQLParameter[] params = new SQLParameter[2];

        // Construct two objects corresponding to the initial values of the
        // stored procedure's two parameters.
        Object obj0 = new Integer\(x);
        Object obj1 = new Integer(0);

        // The stored procedure sp_squareInt has two parameters:
        // an IN parameter of data type INTEGER
        // and an OUT parameter of data type INTEGER.
        // params[0] is build to correspond to the IN parameter,
        // params[1] is build to correspond to the OUT parameter.
        params[0] = new SQLParameter(obj0, Types.INTEGER, SQLParameter.IN);
        params[1] = new SQLParameter(obj1, Types.INTEGER,
SQLParameter.OUT);

        // Call the stored procedure.
        // Note that the procedure does not return any value.
        // Instead the result of the procedure is loaded directly into the
OUT parameter,
        // and, in turn, into params[1].
        myJDBCControlFile.call_sp_squareInt(params);

        // Get the result loaded directly into params[1].
        return Integer.parseInt(params[1].value.toString());
```

Note that Jdbc control method call_sp_squareInt does not return the result of the procedure call. Instead the result of the procedure is loaded directly into the procedure's OUT parameter, and this in turn is loaded directly into the corresponding SQLParameter object. To get the result of the procedure, examine the .value property of the SQLParameter object.

```
    params[1].value
```

3.3. Wrapping Procedures in Functions

An alternative to calling stored procedures directly is to wrap them in stored functions, then call the wrapping function from your database control file.

For example the following Jdbc control method will create a function that wraps the procedure sp_squareInt.

```
    /**
     * Wraps a procedure in a function.
     * /
     @SQL(statement="CREATE OR REPLACE FUNCTION wrapProc (p1 INTEGER)
RETURN INTEGER IS p2 INTEGER; BEGIN sp_squareInt(p1, p2); RETURN p2; END;")
    public void create_wrapProc();
```

Once the procedure has been wrapped, you can call the function, instead of calling the procedure directly.

```
     @SQL(statement="SELECT wrapProc({x}) FROM DUAL")
     public int callWrapProc(int x, int y);
```

3.4. Creating Strored Procedures

You can also send any DDL statement to the database through a database control method.

```
    /**
     * A stored procedure that takes an integer, squares it, and loads
     * the result into an OUT parameter.
     * /
     @SQL(statement="CREATE OR REPLACE PROCEDURE sp_squareInt (field1 IN
INTEGER, field2 OUT INTEGER) IS BEGIN field2 := field1 * field1; END
sp_squareInt; ")
     void create_sp_squareInt() throws SQLException;
```

Some XA database drivers contain restrictions on code that rollsback or commits a transaction independently of the driver's transaction management. Since DDL statements are implicitly transactional (COMMIT is called whether or not it explicitly appears in the DDL statement), you may have to suspend the transaction with these XA drivers. For example if you send a DDL statement using the Oracle XA thin client without suspending the transaction, the driver throws the following exception:

ORA-02089: COMMIT is not allowed in a subordinate session

The following code suspends the transaction, executes the DDL statement, and then resumes the transaction.

```
       import javax.transaction.Transaction;
       import javax.transaction.TransactionManager;
       import javax.transaction.TxHelper;
```

```
            TransactionManager tm = TxHelper.getTransactionManager();
        Transaction saveTx = null;
        try
        {

            // Suspend the transaction
            saveTx = tm.forceSuspend();


            // Execute the DDL statement
            myDBControlFile.create_sp_squareInt();
        }
        finally
        {

            // Resume the transaction
            tm.forceResume(saveTx);
        }
```

4. Stored Functions

This topic explains how to call and create stored functions using Jdbc control.

4.1. Calling Stored Functions

To call a stored function, place the function call in an @SQL statement annotation. When the Java method callMyFunction is called, the SQL statement in the @SQLl statement annotation is passed to the database. Any data returned by the SQL statement is passed back to, and returned by, the Java method.

```
    @SQL(statement="SELECT my_function FROM DUAL")
    int callMyFunction() throws SQLException;
```

In most cases, the Jdbc control automatically converts between the appropriate database data types and Java data types. For example, if the database function my_function returns the database type INTEGER, the Java method callMyFunction() will automatically convert it into the Java type int.

You can substitute values dynamically into the database function call using curly braces. The following method passes the parameter int x to the function call.

4.2. Creating Stored Functions

You can also send any DDL statement to the database through a Jdbc control method.

```
    /**
```

```
     * A stored function that takes an integer, squares it, and returns the
     * result through the database control method.
     * /
     @SQL(statement="CREATE OR REPLACE FUNCTION fn_squareInt (field1 IN
INTEGER) RETURN INTEGER IS field2 INTEGER; BEGIN field2 := field1 * field1;
RETURN field2; END fn_squareInt;")
    void create_fn_squareInt() throws SQLException;
```

Some XA database drivers contain restrictions on code that rollsback or commits a transaction independently of the driver's transaction management. Since DDL statements are implicitly transactional (COMMIT is called whether or not it explicitly appears in the DDL statement), you may have to suspend the transaction with these XA drivers. For example if you send a DDL statement using the Oracle XA thin client without suspending the transaction, the driver throws the following exception:

ORA-02089: COMMIT is not allowed in a subordinate session

The following code suspends the transaction, executes the DDL statement, and then resumes the transaction.

```
        import javax.transaction.Transaction;
        import javax.transaction.TransactionManager;
        import javax.transaction.TxHelper;

                TransactionManager tm = TxHelper.getTransactionManager();
        Transaction saveTx = null;
        try
        {

            // Suspend the transaction
            saveTx = tm.forceSuspend();


            // Execute the DDL statement
            myDBControlFile.create_fn_squareInt();
        }
        finally
        {

            // Resume the transaction
            tm.forceResume(saveTx);
        }
```

5. Jdbc Control Return Type Mapping

When returning a value from a database, the Jdbc Control maps the JDBC ResultSet generated by the SQL to the calling method's return type. These mappings can be characterized as follows:

5.1. Mapping a Single Value

This topic describes how to write methods that return a single value from the database. The example provided represents a SELECT statement that requests only a single field of a single row. The return value of the method should be an object or primitive of the appropriate type for that field's data.

5.1.1. Returning a Single Column

The following example assumes a Customers table in which the field custid, representing the customer ID, is the primary key. Given the customer ID, the method looks up a single customer name.

```
@SQL(statement="SELECT name FROM customer WHERE custid={customerID}")
public String getCustomerName(int customerID);
```

In this example, the name field is of type VARCHAR, so the return value is declared as String. The method's customerID parameter is of type int. When the SQL statement executes, this parameter is mapped to an appropriate numeric type accepted by the database.

5.1.2. Returning an Update Count

Suppose that with the same database table a row is inserted; the following code could be used to get the update count from the insert statement:

```
@SQL(statement="INSERT INTO customer VALUES ({customerName},{customerID})")
public int insertCustomer(String customerName, int customerID);
```

5.2. Mapping a Single Row

This topic describes how to write methods on a Jdbc control that return a single row from the database. When you return a single row with multiple fields, your method must have a return type that can contain multiple values--either an object that is an instance of a class that you have built for that purpose, or a java.util.HashMap object.

If you know the names of the fields returned by the query, you will probably want to return a custom object. If the number of columns or the particular field names returned by the query are unknown or may change, you may choose to return a HashMap.

5.2.1. Returning an Object

You can specify that the return type of a Jdbc control method is a custom object, an instance of a class whose members correspond to fields in the database table. In most cases, a class whose members hold corresponding database field values is declared as an inner class (a class declared inside another class) in the Jdbc control's JCX file. However, it may be any Java class that meets the following criteria:

- The class must contain members with names that match the names of the columns that will be returned by the query. Because database column names are case-insensitive, the matching names are case-insensitive. The class may also contain other members, but members with matching names are required.
- The members must be of an appropriate type to hold a value from the corresponding column in the database.
- The class must be declared as public static if the class is an inner class.

The following example declares a Customer class with members corresponding to fields in the Customers table. The findCustomer method returns an object of type Customer:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}

@SQL(statement="SELECT custid,name FROM customer WHERE
custid={customerID})"
Customer findCustomer(int customerID)
```

Note: The Customer class above is simplified for the sake of clarity. For data modeling classes, it is generally good design practice to have private fields, with public setter and getter methods.

```
    public static class Customer
    {
        private int custid;
        private String name;

        public Customer() {};

        public int getCustid()
        {
            return this.custid;
        }

        public void setCustid(int custid)
        {
            this.custid = custid;
```

```
        }

        public String getName()
        {
            return this.name;
        }

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

5.2.2. Handling Empty Values When Returning Objects

If a database field being queried contains no value for a given row, the class member is set to null if it is an object and to 0 or false if it is a primitive. This may affect your decisions regarding the types you use in your class. If the database field contained no data, an Integer member would receive the value null, but an int member would receive the value 0. Zero may be a valid value, so using int instead of Integer makes it impossible for subsequent code to determine whether a value was present in the database.

If there is no column in the database corresponding to a member of the class, that member is also set to null or 0, depending on whether the member is an primitive or an object.

If the query returns columns that cannot be matched to the members of the class, an exception is thrown. If you don't know the columns that will be returned or if they may change, you should consider returning a HashMap instead of a specific class. For more information, see the Returning a HashMap section, below.

If no rows are returned by the query, the returned value of the Jdbc control method is null.

In the example given above, the method is declared as returning a single object of type Customer. So even if the database operation returns multiple rows, only the first row is returned to the method's caller. To learn how to return multiple rows to the caller, see Mapping Multiple Rows.

5.2.3. Returning a HashMap or Map

If the number of columns or the particular column names returned by the query are unknown or may change, you may choose to return a HashMap. To return a HashMap, declare the return value of the method as java.util.HashMap, as shown here:

```
@SQL(statement="SELECT * FROM customer WHERE custid={custID})"
public java.util.HashMap findCustomerHash(int custID);
```

The HashMap returned contains an entry for each column in the result. The key for each entry is the corresponding column name. The capitalization of the key names returned by HashMap.keySet() depends on the database driver in use, but all keys are case-insensitive when accessed via the HashMap's methods. The value is an object of the Java Database Connectivity (JDBC) default type for the database column.

In the example above, the method is declared as returning a single object of type java.util.HashMap. So even if the database operation returns multiple rows, only the first row is returned to the method's caller.

To learn how return multiple rows to the caller, see Mapping Multiple Rows.

The following code allows you to access the name field of the returned record:

```
@Control
private CustomerDBControl custDB;

public String getCustomerName(int custID)
{
    java.util.HashMap hash;
    String name;
    hash = custDB.findCustomerHash(custID);
    if( hash != null )
    {
        name = (String)hash.get("NAME");
    }
    else
    {
        name = new String("Customer not found");
    }
    return name;
}
```

If the query returns no rows, the returned value of the Jdbc control method is null.

5.3. Returning Multiple Rows from a Jdbc Control Method

This topic describes how to write a method on a Jdbc control that returns multiple rows from the database. It describes the ways in which you can perform this operation, including returning an array, returning an Iterator object, and returning a resultset.

5.3.1. Deciding How to Return Multiple Rows

A SELECT query may return one or more fields from multiple rows. A method on a Jdbc control that returns multiple rows should have a return type that can store these values. The Jdbc control method can return an array of objects, an Iterator, or a resultset.

Returning an array of objects is the easiest way to return multiple rows, so it is a good choice if you think your users will prefer simplicity when using your control. However, when an array is returned only one database operation is performed, and the entire resultset must be stored in memory. For large resultsets, this is problematic. You can limit the size of the returned array, but then you cannot provide a way for your user to get the remainder of the resultset. To learn how to return an array of objects, see the Returning an Array of Objects section, below.

While Iterators require more sophistication on the part of users of your control, they are more efficient at handling large resultsets. An Iterator is accessed one element (row) at a time via the Iterator's next() method, and it transparently makes repeated requests from the database until all records have been processed. An Iterator does not present the risk of running out of memory that an array presents. However, note that an Iterator returned from a database control cannot be used within a Page Flow controller class, because an Iterator wraps a ResultSet object, which is always closed by the time it is passed to the web-tier (where Page Flow files reside). For this reason, your Jdbc control should return an array of objects (see above) when it is called from a page flow controller. Also, an Iterator cannot be returned to a stateful process, because stateful processes cannot maintain an open database connection (which Iterators require). To learn about returning a java.util.Iterator, see the Returning an Iterator section, below.

Finally, you can choose to return a java.sql.ResultSet from a Jdbc control method. This grants complete access to the results of the database operation to clients of your control, but it requires knowledge of the java.sql package. Also, note that a ResultSet returned from a Jdbc control cannot be used within a Page Flow controller, because a ResultSet object is always closed by the time it is passed to the web-tier (where Page Flow files reside). For this reason, your Jdbc control should provide an array of objects when it is called from a Page Flow controller. To learn about returning a java.sql.ResultSet, see the Returning a Resultset section, below.

5.3.2. Returning an Array of Objects

To return an array of objects, declare the method's return type to be an array of the object you want to return. That type may be either a type you define, or it may be java.util.Hashmap.

Examples of both of these techniques are provided in the following sections.

5.3.3. Returning an Array of User-Defined Objects

The following example demonstrates how to return an array of objects whose type you have declared. In this case, an array of Customer objects is returned:

```
public static class Customer
{
    public int custid;
    public String name;
}

@SQL(statement="SELECT custid,name FROM customer WHERE custage<19",
arrayMaxLength=100)
Customer [] findAllMinorCustomers()
```

This example returns all rows in which the custage field contains a value less than 19.

When returning an array of objects, the class declared as the return type of the method must meet the criteria described in the Returning an Object section of the Returning a Single Row from a Jdbc Control topic. If no rows are returned by the query, the returned value of the Database control method is a zero-length array.

If you are returning an array from Jdbc control method, you can limit the size of the array returned by setting the arrayMaxLength attribute of the @SQL annotation. This attribute can protect you from very large resultsets that may be returned by very general queries. If arrayMaxLength is present, no more than that many rows are returned by the method.

The default value of arrayMaxLength is 1024. For very large ResultSets you can avoid excessive memory usage by returning an Iterator object as described below in the Returning an Iterator section, below.

5.3.4. Returning an Array of HashMaps

Returning an array of HashMaps is analogous to returning an array of user-defined objects, which is described in the preceding section.

The following example demonstrates returning an array of HashMaps:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}

@SQL(statement="SELECT custid,name FROM customer WHERE custage<19",
arrayMaxLength=100)
java.util.HashMap [] findAllMinorCustomersHash()
```

The array of HashMaps returned contains an element for each row returned, and each element of the array contains an entry for each column in the result. The key for each entry is

the corresponding column name. The capitalization of the key names returned by HashMap.keySet() depends on the database driver in use, but keys are case-insensitive when accessed via the HashMap's methods. The value returned is an object of the Java Database Connectivity (JDBC) default type for the database column.

If no rows are returned by the query, the returned value of the Jdbc control method is a zero-length array.

The following code shows how to access the name field of the returned records:

```
@Control
private CustomerDBControl custDB;

java.util.HashMap [] hashArr;
String name;

hashArr = custDB.findAllMinorCustomersHash();
for(i=0; i<hashArr.length; i++)
{
    name = (String)hashArr[i].get("NAME");
    // say hello to the all of the minors

   System.out.println("Hello, " + name + "!");
}
```

5.3.5. Returning an Iterator

When you want to return an Iterator object, you declare the method's return type to be java.util.Iterator. You then add the iteratorElementType attribute to the @SQL annotation to indicate the underlying type that the Iterator will contain. The specified type may be either a type you define, or it may be java.util.Hashmap. Examples of these techniques are given in the following sections. If your method returns an Iterator, a compile time error will be generated if the iteratorElementType annotation member has not been set.

The Iterator that is returned is only guaranteed to be valid for the life of the method call to which it is returned. You should not store an Iterator returned from a Jdbc control method as a static member of your web service's class, nor should you attempt to reuse the Iterator in subsequent method calls if it is persisted by other means.

5.3.6. Returning an Iterator with a User-Defined Object

To return an Iterator that encapsulates a user-defined type, provide the class name as the value of the iteratorElementType attribute of the @SQL annotation, as shown here:

```
public static class Customer
```

```
{
    public int custid;
    public String name;
    public Customer() {};
}

@SQL(statement="SELECT custid,name FROM customer"
iteratorElementType=Customer.class)
java.util.Iterator getAllCustomersIterator()
```

The class specified in the iterator-element-type attribute must meet the criteria described in Returning an Object.

The following example shows how to access the returned records:

```
CustomerJDBCControl.Customer cust;
java.util.Iterator iter = null;
iter = custDB.getAllCustomersIterator();
while (iter.hasNext())
{
    cust = (CustomerJDBCControl.Customer)iter.next();
    // say hello to every customer
    System.out.println("hello, " + cust.name + "!");
}
```

5.3.7. Returning an Iterator with HashMap

To return an Iterator that encapsulates a HashMap, provide java.util.HashMap as the value of the iterator-element-type attribute of the @SQL annotation, as shown here:

```
public static class Customer
{
    public int custid;
    public String name;
    public Customer() {};
}

@SQL(statement="SELECT custid,name FROM customer",
iteratorElementType=java.util.HashMap.class)
java.util.Iterator getAllCustomersIterator()
```

The following code shows how to access the returned records:

```
java.util.HashMap custHash;
java.util.Iterator iter = null;
int customerID;
String customerName;
iter = custDB.getAllCustomersIterator();
while (iter.hasNext())
```

```
{
    custHash = (java.util.HashMap)iter.next();
    customerID = (int)custHash.get("CUSTID");
    customerName = (String)custHash.get("NAME");
}
```

The HashMap contains an entry for each database column that is returned by the query. The key for each entry is the corresponding column name, in all uppercase. The value is an object of the JDBC default type for the database column.

5.3.8. Returning a ResultSet

The Jdbc control is designed to allow you to obtain data from a database in a variety of ways without having to understand the classes in the java.sql package. If you and your users do understand these classes, however, you can gain complete access to the java.sql.ResultSet object returned by a query.

If you want to return a resultset, you declare the method's return type to be java.sql.ResultSet. A client of your control then accesses the resultset directly to process the results of the database operation.

The following example demonstrates returning a resultset:

```
@SQL(statement="SELECT * FROM customer")
public java.sql.ResultSet findAllCustomersResultSet();
```

The following code shows how to access the returned resultset:

```
java.sql.ResultSet resultSet;
String thisCustomerName;
resultSet = custDB.findAllCustomersResultSet();
while (resultSet.next())
{
    thisCustomerName = new String(resultSet.getString("name"));
}
```

This example assumes the rows returned from the database operation include a column called name.

5.4. Returning Apache XMLBeans from a Jdbc Control

This topic assumes a strong understanding of Apache XML Beans. For additional information about XML Bean see the Apache XML Beans Site http://xmlbeans.apache.org/.

The following topic explains how to return XMLBean types from custom Jdbc controls.

An XMLBean is essentially an XML document with a Java API attached to it. The API is used for parsing and manipulating the data in the XML document. A typical XMLBean might represent database data in the following form.

```
<DOCTYPE XCustomer>
<XCustomer xmlns="java:///database/customer_db"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <XCustomerRow>
        <CUSTID>1<CUSTID>
        <NAME>Fred Williams<NAME>
        <ADDRESS>123 Slugger Circle<ADDRESS>
    <XCustomerRow>
    <XCustomerRow>
        <CUSTID>2<CUSTID>
        <NAME>Marnie Smithers<NAME>
        <ADDRESS>5 Hitchcock Lane<ADDRESS>
    <XCustomerRow>
    <XCustomerRow>
        <CUSTID>3<CUSTID>
        <NAME>Bill Walton<NAME>
        <ADDRESS>655 Tall Timbers Road<ADDRESS>
    <XCustomerRow>
<XCustomer>
```

The data can be accessed and manipulated using the XMLBean's API. For example, assume that custBean represents the XML document above. The following Java code extracts the Fred Williams from the document.

```
String name = custBean.getXCustomer().getXCustomerRowArray(1).getNAME();
```

Retrofitting database controls to return XMLBeans rather than RowSets, ResultSets, or Iterators, is a powerful technique because there are few restrictions on where XMLBeans can be imported. This is not the case with ResultSets and Iterators, which cannot be passed directly to web-tier classes (web services and page flows). Also, data in XMLBean form is very easy to manipulate because there is a rich API attached to the XMLBean.

5.4.1. Creating a Schema

The first step in using XMLBean classes is creating a schema from which the XMLBean classes can be generated. The schema you create for a database control must be capable of modeling the sorts of data returned from the database.

If you write your own schema, at a minimum, the schema's elements should have the same names as the fields in the database, which allows data returned from the database to be automatically mapped into the XMLBean.

When the XSD file is compiled, XMLBean types are generated that can be returned by the methods in the database control.

5.4.2. Editing Schemas to Create New "Document" Types

Note that only one of the generated types is a "Document" XMLBean type: XCustomerDocument. The other types, XCustomerDocument.XCustomer and XCustomerDocument.XCustomer.XCustomerRow, can only be used with reference to the "Document" type. This distinction is especially important because only "Document" types are eligible for direct participation in a business process, or to be passed to a web service. For this reason you may want to edit your schema to include "Document" types corresponding to other types in the Schema, especially if you have a very large schema with many nested types defined in terms of a single "Document" type.

To generate a new Document type for some element, move that element so that it becomes a top-level element in the schema. In the following example, the XCustomerRow element has been moved to the top-level of the schema: its original position has been replaced with a reference element: <xsd:element ref="XCustomerRow"/>.

```
<xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="java:///database/customer_db"
                    xmlns="java:///database/customer_db"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns:wld="http://www.bea.com/2002/10/weblogicdata"
                    elementFormDefault="qualified"
                    attributeFormDefault="unqualified">

    <xsd:element name="XCustomer"
wld:DefaultNamespace="java:///database/customer_db" wld:RowSet="true">
     <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
       <xsd:element ref="XCustomerRow"/>
      <xsd:choice>
     <xsd:complexType>
    <xsd:element>
    <xsd:element name="XCustomerRow">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="CUSTID" type="xsd:int" wld:JDBCType="INTEGER"
minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="NAME" type="xsd:string" wld:JDBCType="VARCHAR"
minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="ADDRESS" type="xsd:string"
wld:JDBCType="VARCHAR" minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="CITY" type="xsd:string" wld:JDBCType="VARCHAR"
```

```
minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="STATE" type="xsd:string" wld:JDBCType="CHAR"
minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="ZIP" type="xsd:string" wld:JDBCType="VARCHAR"
minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="AREA_CODE" type="xsd:string"
wld:JDBCType="CHAR" minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
          <xsd:element name="PHONE" type="xsd:string" wld:JDBCType="CHAR"
minOccurs="0" wld:TableName="MYSCHEMA.CUSTOMER"
nillable="true"><xsd:element>
        <xsd:sequence>
        <xsd:anyAttribute
namespace="http://www.bea.com/2002/10/weblogicdata"
processContents="skip">lt;xsd:anyAttribute>
      <xsd:complexType>
    <xsd:element>
<xsd:schema>
```

There are now two top-level elements, XCustomer and XCustomerRow, which compile into two corresponding "Document" types: XCustomerDocument and XCustomerRowDocument.

5.4.3. Returning a XMLBean Types from Control Methods

Once you have generated XMLBean types that model the database data, you can import these types into your Jdbc control.

```
import databaseCustomerDb.XCustomerDocument;
import databaseCustomerDb.XCustomerDocument.XCustomer;
import databaseCustomerDb.XCustomerDocument.Factory;
```

XMLBean types can be returned from the control's methods.

```
@SQL(statement="SELECT custid, name, address FROM customer")
public XCustomerDocument findAllCustomersDoc();
```

The data returned from the query is automatically mapped into the XMLBean because the names of the database fields match the fields of the XMLBean.

5.5. Mapping to a RowSet

This topic describes how to write methods on a Jdbc control that return a RowSet from the database. Since the RowSet implementations provided by the JDK are part of the javax.sql, package the JdbcControl does not support any of them by default. A sample

ResultSetMapper for RowSet's is included as part of the Jdbc Control's distribution but must be explicitly set in the @SQL annotation in order to be invoked.

The DefaultRowSetResultSetMapper will create a javax.sql.CachedRowSetImpl. The following example sets the resultSetMapper for the method getAllUsers() to the DefaultRowSetResultSetMapper which enables the Jdbc control to map the ResultSet to a RowSet.

```
@SQL(statement="SELECT * FROM USERS",
resultSetMapper=org.apache.beehive.controls.system.jdbc.DefaultRowSetResultSetMapper.cl
public RowSet getAllUsers() throws SQLException;
```

ResultSetMapper's can be created for other types of RowSets and almost any other type of mapping from a result set to any object. See the [Jdbc Control Custom ResultSetMappers] topic for more information.

5.6. Creating Customer Result Set Mappers

5.6.1. Overview

When the Jdbc Control maps a ResultSet to a return type, it first checks to see if a resultSetMapper has been set in the method's @SQL annotation. If a mapper has been set, it is always the one used for mapping the ResultSet to the method's return type. If resultSetMapper has not been set, the Jdbc control looks for a _resultSetMapper_ based on the method's return type.

Mapper Class Name	Method Return Type
DefaultIteratorResultSetMapper	Iterator
DefaultResultSetMapper	ResultSet
DefaultXmlObjectResultSetMapper	Classes derived from XmlObject
DefaultObjectresultMapper	Default to this mapper

5.6.2. Creating a custom ResultSet Mapper

To create your own ResultSet mapper, create a new class which extends the abstract class org.apache.beehive.controls.system.jdbc.ResultSetMapper. The mapToResultType() method does all the work of mapping the ResultSet to the method's return type -- it will be invoked by the JdbcControl when the control is ready to perform the mapping. Below is the code for the ResultSetMapper class.

```
/**
 * Extend this class to create new ResultSet mappers. The extended class
will be invoked by the JdbcController
 * when it is time to map a ResultSet to a method's return type.
 *
 * ResultSet mappers must be specified on a per method basis using the SQL
annotation's resultSetMapper field
 */
public abstract class ResultSetMapper {

    /**
     * Map a ResultSet to an object type
     *
     * @param context   A ControlBeanContext instance, see Beehive controls
javadoc for additional information
     * @param m         Method assoicated with this call.
     * @param resultSet Result set to map.
     * @param cal       A Calendar instance for time/date value resolution.
     * @return          The Object resulting from the ResultSet
     * @throws Exception On error.
     */
    public abstract Object mapToResultType(ControlBeanContext context,
Method m, ResultSet resultSet, Calendar cal)
            throws Exception;

    /**
     * Can the ResultSet which this mapper uses be closed by the Jdbc
control?
     * @return true if the ResultSet can be closed by the JdbcControl
     */
    public boolean canCloseResultSet() { return true; }
}
```

5.6.3. An Example

Suppose you have a return type class which needs to do some special processing of a ResultSet.

```
public final class CustomerX
{
     private String _customerName;
     private String _customerPhoneNumber;

     public void setCustomerName(String firstName, String lastName) {
          _customerName = firstName + " " + lastName;
     }

     public String getCustomerName() { return _customerName; }

     public void setCustomerPhoneNumber(int areaCode, String phoneNumber) {
          _customerPhoneNumber = "(" + areaCode + ")" + phoneNumber;
```

```
     }

    public String getCustomerPhoneNumber() { return _customerPhoneNumber; }
}
```

Let's assume the ResultSet contains the following columns:

Column Name	Type
FIRST_NAME	Varchar
LAST_NAME	Varchar
AREA_CODE	INT
PHONE_NUMBER	Varchar

Here's what the ResultSetMapper implementation might look like:

```
public final class CustomerXResultSetMapper extends ResultSetMapper {

    public Object mapToResultType(ControlBeanContext context, Method m,
ResultSet resultSet, Calendar cal)
            throws Exception
     {
          resultSet.next();
           CustomerX c = new CustomerX();
          final String fName = resultSet.getString("FIRST_NAME");
          final String lName = resultSet.getString("LAST_NAME");

          c.setCustomerName(fName, lName);

          final int aCode = resultSet.getInt("AREA_CODE");
          final int phone = resultSet.get("PHONE_NUMBER");

          c.setCustomerPhoneNumber(aCode, phone);

          return c;
     }
}
```

and finally the method and SQL annotation to invoke:

```
@SQL(statement="SELECT FIRST_NAME,LAST_NAME,AREA_CODE,PHONE_NUMBER FROM
customers WHERE userId={userId}",
         resultSetMapper=CustomerXResultSetMapper.class)
public CustomerX getCustomer(String userId);
```

5.6.4. Additional Examples

See the Jdbc Control Rowset Mapping topic for an example of using a ResultSet mapper to support the RowSet return type.

5.7. Database -> Java Type Mapping Tables

5.7.1. PointBase 4.4 Type Mappings

The following table lists the relationships between database types and Java types for the PointBase Version 4.4 database.

Java Data Types	JDBC Data Types	PointBase SQL Data Types (Version 4.4)
boolean	BIT	boolean
byte	TINYINT	smallint
short	SMALLINT	smallint
int	INTEGER	integer
long	BIGINT	numeric/decimal
double	FLOAT	real
double	DOUBLE	double
float	FLOAT	float
java.math.BigDecimal	NUMERIC	numeric
java.math.BigDecimal	DECIMAL	decimal
String	CHAR	char
String	VARCHAR	varchar
String	LONGVARCHAR	clob
java.sql.Date	DATE	date
java.sql.Time	TIME	time
java.sql.Timestamp	TIMESTAMP	timestamp
byte[]	BINARY	blob
byte[]	VARBINARY	blob
byte[]	LONGVARBINARY	blob

java.sql.Blob	BLOB	blob
java.sql.Clob	CLOB	clob

5.7.2. Oracle Type Mappings

Type Mappings for Oracle 8i

The following table lists the relationships between database types and Java types for the Oracle 8i database.

Java Data Types	JDBC Data Types	Oracle SQL Data Types (Version 8i)
boolean	BIT	NUMBER
byte	TINYINT	NUMBER
short	SMALLINT	NUMBER
int	INTEGER	NUMBER
long	BIGINT	NUMBER
double	FLOAT	NUMBER
float	REAL	NUMBER
double	DOUBLE	NUMBER
java.math.BigDecimal	NUMERIC	NUMBER
java.math.BigDecimal	DECIMAL	NUMBER
String	CHAR	CHAR
String	VARCHAR	VARCHAR2
String	LONGVARCHAR	LONG
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP	DATE
byte[]	BINARY	NUMBER
byte[]	VARBINARY	RAW
byte[]	LONGVARBINARY	LONGRAW

java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB

5.7.3. Derby Type Mappings

Type Mappings for Derby 10

Java Data Types	JDBC Data Types	Derby SQL Data Types (Version 4.4)
long	BIGINT	BIGINT
java.sql.Blob	BLOB	BLOB
String	CHAR	CHAR
java.sql.Clob	CLOB	CLOB
java.sql.Date	DATE	DATE
java.math.BigDecimal	DECIMAL	DECIMAL,NUMERIC
double	DOUBLE	DOUBLE [PRECISION]
float	FLOAT	float
int	INTEGER	integer
String	LONGVARCHAR	LONG VARCHAR
short	SMALLINT	SMALLINT
java.sql.Time	TIME	time
java.sql.Timestamp	TIMESTAMP	timestamp
String	VARCHAR	VARCHAR

5.8. New Features and Enhancements

JDBC 3.0 feature support as well as other new features are being added to the JdbcControl on a regular basis. Here some of the latest features which have been added:

- Support for custom mapping of SQL UDTs
- Support for ResultSet holdability (connection and statement level support)
- Support for fetchSize and direction
- Support for scrollable ResultSets
- Retrieval of auto-generated keys

- BOOLEAN and DATALINK data types
- Blob and Clob type support
- Batch Update support