Novell exteNd Composer

5.2

JDBC CONNECT USER'S GUIDE

Novell®

Legal Notices

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.

eDirectory is a trademark of Novell, Inc.

GroupWise is a registered trademark of Novell, Inc.

exteNd is a trademark of Novell, Inc.

exteNd Composer is a trademark of Novell, Inc.

exteNd Director is a trademark of Novell, Inc.

iChain is a registered trademark of Novell, Inc.

jBroker is a trademark of Novell, Inc.

NetWare is a registered trademark of Novell, Inc.

Novell is a registered trademark of Novell, Inc.

Novell eGuide is a trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

The Apache Software License, Version 1.1

Copyright (c) 2000 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

JDOM.JAR

Copyright (C) 2000-2002 Brett McLaughlin & Jason Hunter. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org. 4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun

Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun Logo Sun, the Sun logo, Sun Microsystems, JavaBeans, Enterprise JavaBeans, JavaServer

Contents

About This Book

Purpose

This guide describes how to use the exteNd Connect, referred to as the JDBC Component Editor. The JDBC Component Editor is a standard component editor in exteNd Composer.

Audience

This book is for developers and systems integrators who are planning to use exteNd Composer to develop database-aware services and components.

Prerequisites

This book assumes prior familiarity with exteNd Composer's work environment and deployment options. Some familiarity with Structured Query Language (SQL) is also assumed.

Additional documentation

For the complete set of Novell exteNd exteNd Director documentation, see the Novell Documentation Web Site (http://www.novell.com/documentation-index/index.jsp).

1 Welcome to exteNd Composer and JDBC

Before You Begin

Welcome to the *Novell exteNd JDBC Connect User's Guide*. This Guide is a companion to the *exteNd Composer User's Guide*, which details how to use all the features of Composer except for the Connect Component Editors. So, if you haven't looked at the *Composer User's Guide* yet, please familiarize yourself with it before using this Guide.

exteNd Composer provides separate Component Editors for each Connect, such as the JDBC connector. The special features of each component editor are described in separate Guides like this one.

If you have been using exteNd Composer, and are familiar with the core component editor (the XML Map Component Editor), then this Guide should get you started with the JDBC Component Editor.

NOTE: To be successful with this Component Editor, you must be familiar with writing and constructing SQL statements.

About exteNd Connects

Novell exteNd is built upon a simple hub and spoke architecture. The hub is a robust XML transformation engine that accepts XML documents, processes the documents, and returns an XML document. The spokes or Connects are plug-in modules that "XML enable" sources of data that are not XML-aware. These data sources can be anything from legacy COBOL / VSAM managed information to Message Queues to HTML pages. exteNd Connects can be categorized by the integration strategy each one employs to XML enable an information source. The integration strategies are a reflection of the major divisions used in modern systems designs for Internet based computing architectures. Depending on your B2Bi needs, exteNd can integrate your business systems at the User Interface, Program Logic, and/or Data levels.

What is JDBC?

JDBC is a Java-based API (Application Programming Interface) for executing SQL statements. While often mistaken as an acronym meaning "Java Database Connectivity," JDBC is in fact not an acronym at all, but a trademarked name. JDBC consists of a set of classes and interfaces written in the Java programming language that allows you to write one program to access different databases such as Oracle, Sybase, Informix, etc., rather than needing to write a separate program for each one.

You can write a single program using the JDBC API and the program is able to send SQL statements to the appropriate database. And since the application is written in the Java programming language, there is no need to write different applications to run on different platforms. The combination of Java and JDBC lets you write it once and run it anywhere, as the following illustration shows.
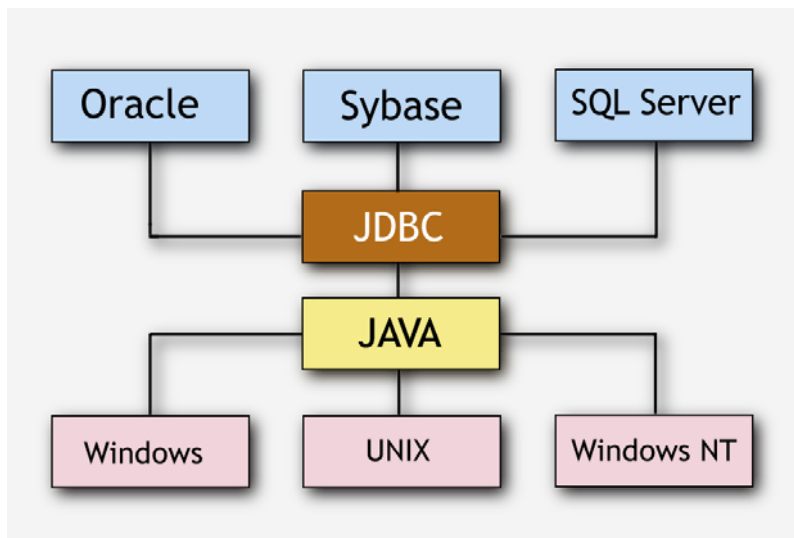
What Does JDBC Do?

JDBC makes it possible to do the following:

- Establish a connection with a database
- Send SQL statements (or queries) to be processed by the database
- Process the results of the database processing

JDBC is a low-level interface used to call SQL commands directly. It is integrated into Composer to interface between components and databases, allowing the program to establish connections with the databases, send the SQL statements, and process the results. Composer provides tools that enable visual construction of the necessary SQL commands.

About exteNd's JDBC Component

Much like the XML Map Component, the JDBC Component is designed to map, transform, and transfer data between two different XML templates (i.e., request and response XML documents). However, it is specialized to make a connection to a database, process SQL statements against the database using elements from a Message Part within the query, and then map the results of the query to a Part.

A JDBC Component can perform simple data manipulations, such as mapping and transferring data from one XML document to another, or from an XML document to a database table. It can also perform sophisticated manipulations, such as requesting data from disparate databases, transforming data from and to one or more documents, executing SQL transactions against the database, and even transforming the documents themselves. Like an XML Map Component, the JDBC Component can process XSL, send mail, and post and receive XML documents using the HTTP protocol.
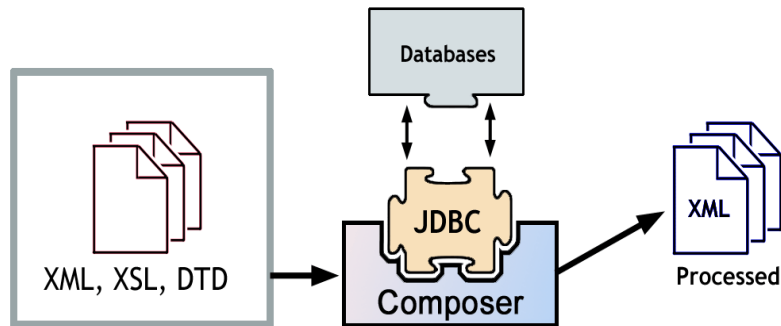
The JDBC Connect uses exteNd Composer as the backplane for XML-based data interactions, making it possible to reach into databases at runtime (and design time). Using exteNd Composer, you can assemble Action Models within a JDBC Component to carry out sophisticated data transformations, using HTTP (optionally) as a transport mechanism. Live database connections are available at design time, so that you can edit and debug SQL queries as part of the design process.

What Kinds of Applications Can You Build Using the JDBC Component Editor?

You can build any business-to-business application that needs to push data into or pull data from a JDBC-accessible data store and uses XML as the interchange format. For example, you can write an application that retrieves the description, picture and price of a product from a database and displays it in the user's browser. If the information resides in two or more databases, you can merge the information from separate databases before displaying it to the user.

2 Getting Started with the JDBC Component Editor

Creating a JDBC Connection Resource

Before you create a JDBC Component, you will find it necessary to create a Connection Resource to access the SQL database. Each Connect, including the JDBC connector, uses its own Connection type. Each Connection type is differentiated by the number and types of parameters used to connect to the specific external data source.

About Constant and Expression Driven Connection Parameters

You can specify Connection parameter values in one of two ways: as Constants or as Expressions. A constant based parameter uses the value you type in the Connection dialog every time the Connection is used. An expression based parameter allows you to set the value using a programmatic expression, which can result in a different value each time the connection is used at runtime. This allows the Connection's behavior to be flexible and vary based on runtime conditions each time it is used.

For instance, one very simple use of an expression driven parameter in a JDBC Connection would be to define the User ID and Password as PROJECT Variables (e.g. PROJECT.XPATH("USERCONFIG/MyDeployUser"). This way when you deploy the project, you can update the PROJECT Variables in the Deployment Wizard to values appropriate for the final deployment environment. At the other extreme, you could have a custom script that queries a Java business object in the Application Server to determine what User ID and Password to use.

➢ **To switch a parameter from Constant driven to Expression driven:**

  1 Click the right mouse button in the parameter field you are interested in changing.

  2 Select **Expression** from the context menu and the editor button will appear or become enabled.

  3 Click on the button and then create an expression that evaluates to a valid parameter value at runtime. (Strings should be wrapped in double-quotes.)
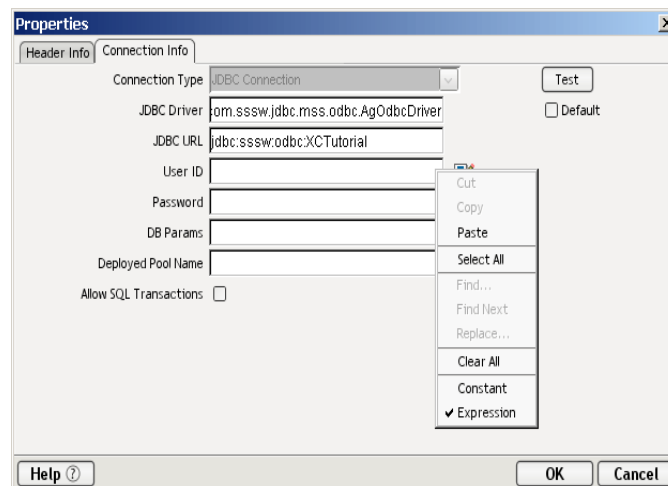
About JDBC Drivers and Connection Pools

When you create a Connection Resource, you are asked to provide a Driver Name and Connection Pool.

The JDBC Driver **sun.jdbc.odbc.JdbcOdbcDriver** is part of the JRE (Java Runtime Environment, which you can find under the Novell exteNd5 directory), and you can use this driver to establish your connection. But you can also obtain other JDBC drivers. For instance, the Novell exteNd Application Server has its own JDBC drivers. Also, you can visit the Web site of the vendor for the SQL database you're using and download their driver(s).

A *connection pool* is a set of database connections managed by the application server for the various applications it manages. It provides more efficient use of database and connection resources for multiple applications running in the same application server. This, in turn, can improve overall system performance. You can obtain the Pool Name for your application server from your Server Administrator.

For deployments within the Novell exteNd Application Server, the pool name will be JDBC/*DBName* where *DBName* is the name that was used when the connection pool was added to the server. For example, if you were connecting to the Samples50 database provided with the application server, the pool name would be JDBC/Samples50.

➢ **To create a JDBC connection resource:**

1   Select **File>New> xObject** and select the **Resource** tab. Click on **Connection**. The "Create a New Connection Resource" Wizard appears.
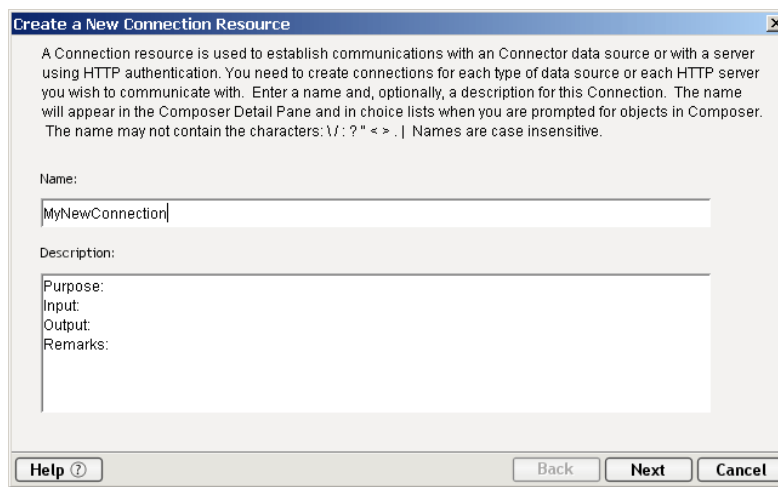
2   Type a **Name** for the connection object.

3   Optionally, type **Description** text.

4 Click **Next**.
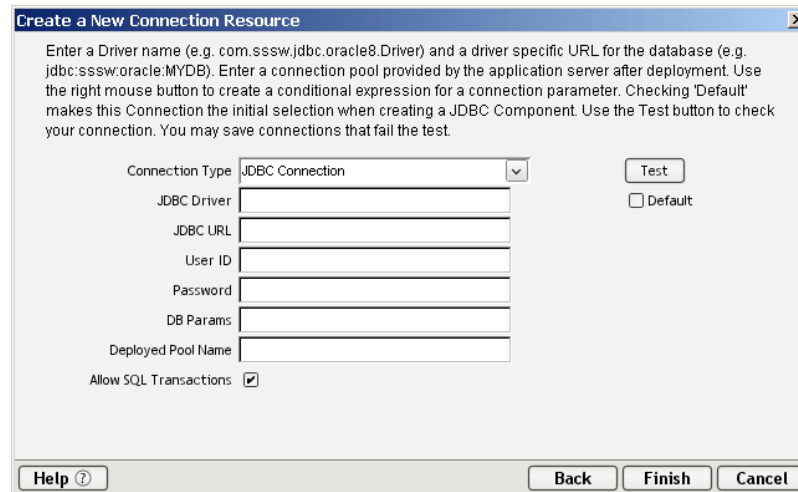
5 Select **JDBC Connection** from the **Connection Type** pull down menu.

6 In the **JDBC Driver** field, enter the name of the JDBC driver you want to use. For example, **com.mysql.jdbc.Driver** for the Novell exteNd driver. (For more information see "About JDBC Drivers and Connection Pools" on page 14.)

NOTE: This parameter, and all subsequent parameters in this dialog, can be dynamically set using Expressions. See "About Constant and Expression Driven Connection Parameters" earlier in this chapter.

7 In the **JDBC URI** field, enter the location of the database you want to reach. For example, **jdbc:mysql://localhost:63306/samples50** where **jdbc:mysql:** is required syntax by the driver and **samples50** is an ODBC Data Source Name defined on the specific computer where the component will run. In this example the database is installed on the localhost at port 63306. For deployment, you may maintain connections directly to the database, provided that the server allows for ODBC connectivity. The more likely scenario is that you will want to take advantage of the power of the application server in managing database access. In that case, you need to provide the connection pool name as described below.

NOTE: The **JDBC Driver** and **JDBC URI** fields are both case sensitive.

8 Enter a valid **User ID** to sign on to the selected database.

9 Enter a valid **Password** for the selected database.

10 In the **DB Params** field, enter any database-specific parameters that might apply to your connection. Note that parameters should be entered as **name=value** pairs. If more than one **name=value** param is specified, separate the pairs using semicolons, e.g., `param1=true;param2=true;param3=false`.

NOTE: If no database-specific parameters will be used, enter `false` in this field.

11 Enter a **Pool Name** if required. For more information, see "About JDBC Drivers and Connection Pools" on page 14.

NOTE: Connection pooling is only operational in the deployment environment. Setting the name here will not affect Composer connections. Only the deployed project will be affected.

12 Check the **Allow SQL Transactions** checkbox if you intend to exercise direct control over transactions (using SQL Begin, Commit, and Rollback verbs) in your component's Action Model.

NOTE: This checkbox is mainly for backwards compatiblity with pre 5.0 versions of Composer. If transactions are required, it is recommended that this checkbox remain unchecked. Explicit SQL "BEGIN", "COMMIT", or "ROLLBACK" actions should be substituted with the Composer Transaction action. This action uses JTA to manage the transaction. The Transaction action is available form the Composer Action>Advanced Actions Menu.

Checking the **Allow SQL Transactions** box has a number of effects:

◆ It turns auto-commit *off* for the JDBC driver. (The state of the auto-commit flag is restored, however, at the end of the transaction, before returning the connection back to the pool.)

◆ It causes all SQL commit and rollback commands to be translated to the corresponding JDBC connection calls.

◆ It causes Composer Enterprise Server to check the final Execute SQL Action in the component to see that the final action is a commit or a rollback. If the final action is *not* a commit or rollback, Composer Enterprise Server performs a rollback by default, so that a dirty connection (that is, a connection with uncommitted changes) is not inadvertently returned to the pool.

NOTE: For a further discussion of the **Allow SQL Transactions** checkbox, see the Transactions chapter of the *exteNd Composer Application Server Guide* for your application server.

13 Check the **Default** checkbox if you would like to use the current connection as the default connection for any new JDBC Components you create in your project.

14 Click **Test** to see if your connection is successful. A "success" or "failure" message appears for your connection. *You can continue creating the resource, even if your connection fails.*

NOTE: This does not test the connection pool (if defined).

15 Click **Finish**. The newly-created resource connection object appears in the Composer Connection Resource detail pane.
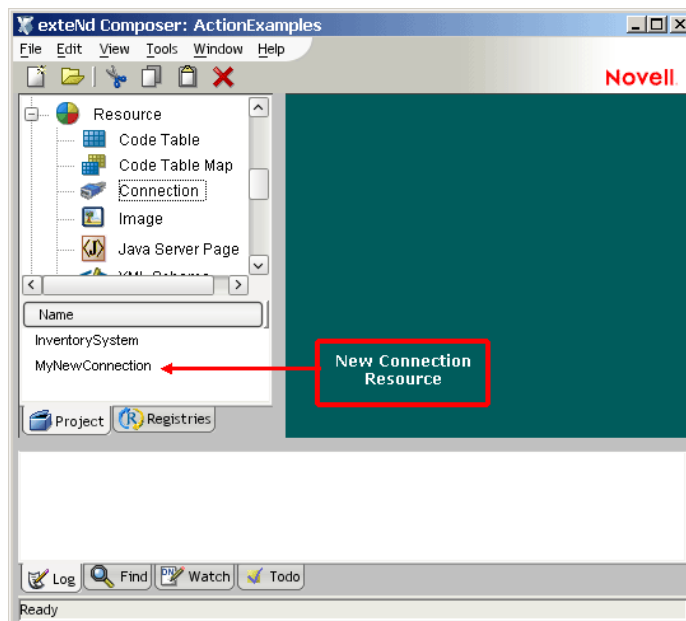
Creating XML Templates for Your Component

In addition to a connection resource, a JDBC component also requires that you have already created XML templates so that you have sample documents for designing your component. (See Chapter 5, *Creating XML Templates,* in the *Composer User's Guide* for more information.)

Also, if your component design calls for any other xObject resources such as custom scripts or code table maps, it is best to create these before creating the JDBC Component. For more information, see *Creating Custom Scripts* in the *Composer User's Guide*.

3 Creating a JDBC Component

Before Creating a JDBC Component

As with all exteNd components, the first step in creating a JDBC component is to specify the XML templates needed. (For more information, see *Creating a New XML Template* in the separate *Composer User's Guide.*) Once you've specified the XML templates, you can create a component, using the template's sample documents to represent the inputs and outputs processed by your component.

Also, as part of the process of creating a JDBC component, you can select a JDBC connection or you can create a new one. If you create the connection beforehand, then it is available to all new JDBC components. (See "Creating a JDBC Connection Resource" on page 13.)

➢ **To create a new JDBC component:**

1   Select **File>New>xObject**. Select the **Component** tab and then **JDBC**.

   NOTE: Alternatively, under **Component** in the Composer Navigator pane, you can highlight **JDBC**, click the right mouse button, then select **New**.

2   The "Create a New JDBC Component" Wizard appears.
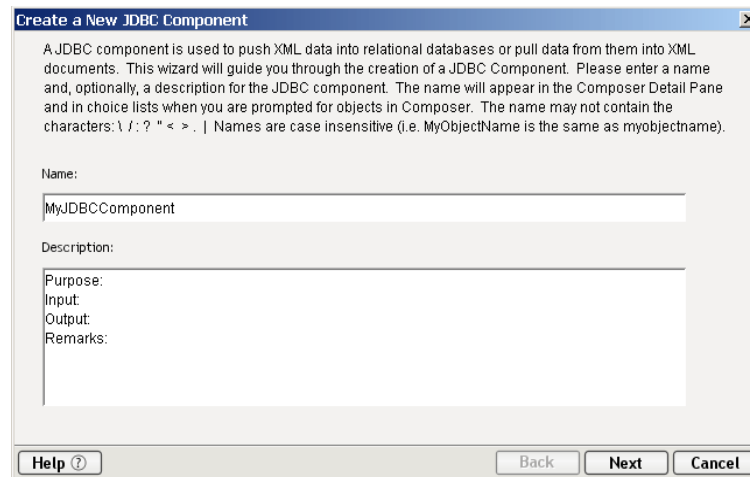
3   Enter a **Name** for the new JDBC Component.

4   Optionally, type **Description** text.

5   Click **Next**. The XML Input/Output Property Info panel of the New JDBC Component Wizard appears.
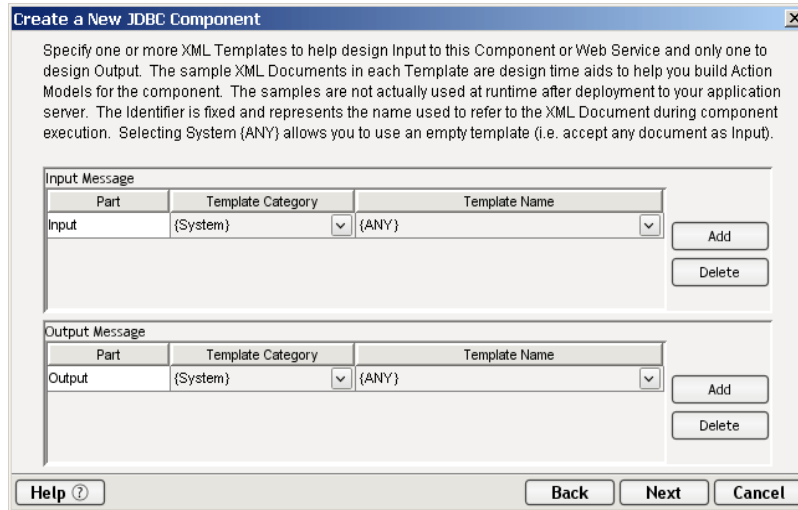
6   Specify the Input and Output templates as follows.

◆   Type in a name for the template under **Part** if you wish the name to appear in the DOM as something other than "Input".

◆   Select a **Template Category** if it is different than the default category.

◆   Select a **Template Name** from the list of XML templates in the selected **Template Category**.

◆   To add additional input XML templates, click **Add** and choose a **Template Category** and **Template Name** for each.

◆   To remove an input XML template, select an entry and click **Delete**.

7   Select an XML template for use as an Output DOM using the same steps outlined above.

NOTE: You can specify an input or output XML template that contains no structure by selecting {System}{ANY} as the Input or Output template. For more information, see "Creating an Output DOM without Using a Template" in the User's Guide.

8   Click **Next**. The Temp and Fault XML Template panel appears.
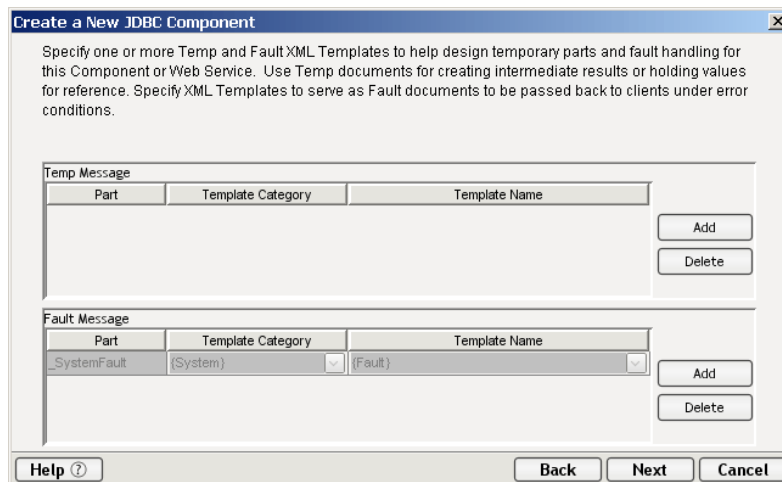
9   If desired, specify a template to be used as a scratchpad under the "Temp Message" pane of the dialog window. This can be useful if you need a place to hold values that will only be used temporarily during the execution of your component or are for reference only. Select a Template Category if it is different than the default category. Then select a Template Name from the list of XML templates in the selected Template Category.

10 Under the "Fault Message" pane, select an XML template to be used to pass back to clients when an error condition occurs.

11 As above, to add additional input XML templates, click **Add** and choose a Template Category and Template Name for each. Repeat as many times as desired. To *remove* an input XML template, select an entry and click **Delete**.

12 Click **Next.** The Connection Info panel of the "Create a New JDBC Component" Wizard appears.
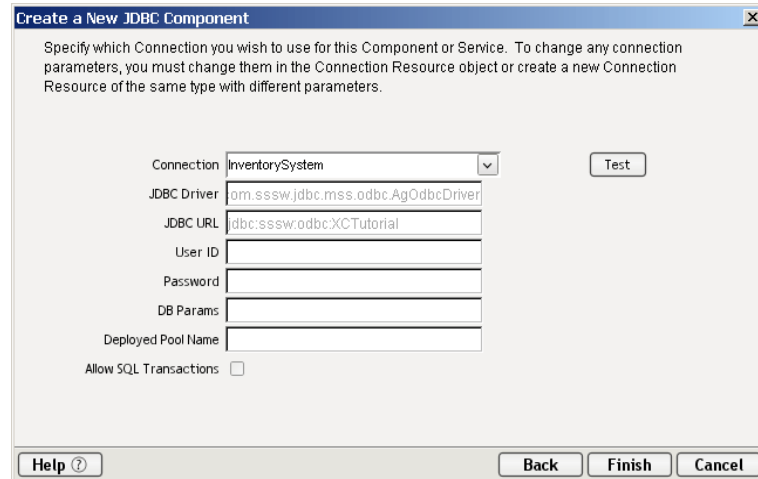
13 Select a **JDBC Connection** from the pull down list. For more information on the JDBC Connection, see "Creating a JDBC Connection Resource" on page 13.

14 Click **Finish**. The component is created and the JDBC Component Editor appears.
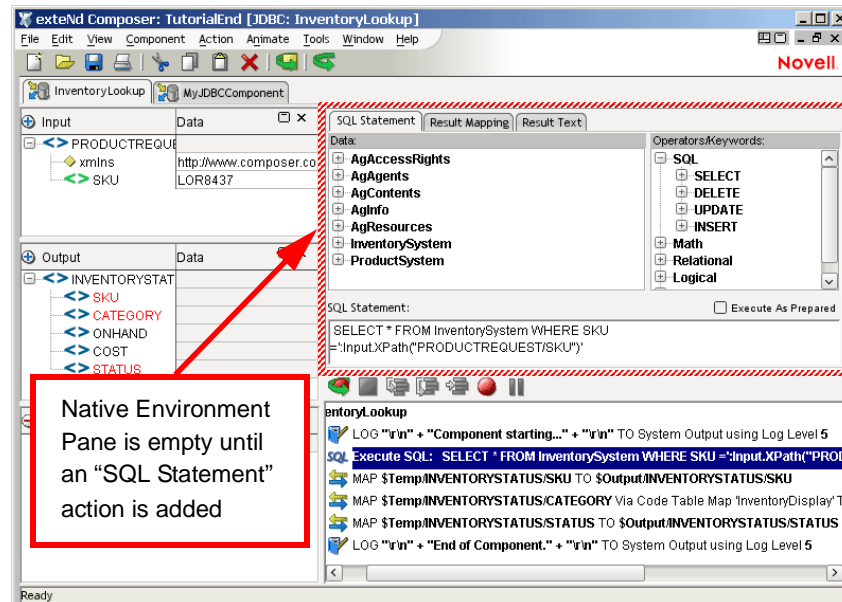
About the JDBC Component Editor Window

The JDBC Component Editor includes all the functionality of the XML Map Component Editor. It contains mapping panes for Input and Output XML documents as well as an Action pane.

The difference, however, is that the JDBC Component Editor also includes a Native Environment pane common to all Connects. It appears as a grey pane until you create an SQL Statement action, at which time it is populated with the Query pane, which is specific to the JDBC connector.

NOTE: To display the Query Pane, you must first select **SQL Statement** from the **Action** menu and create an SQL action. Otherwise, the pane remains greyed out.
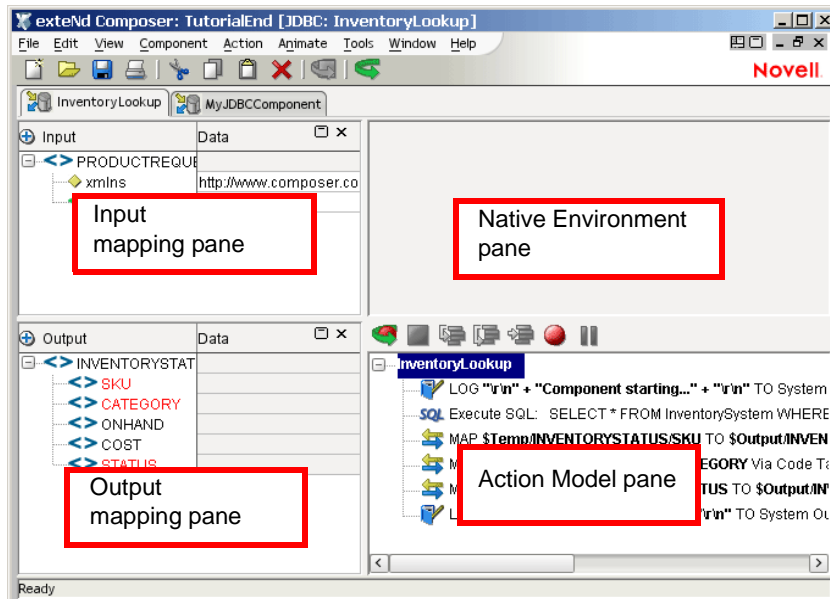
About the Query Pane

When the Query pane (i.e., the activated Native Environment pane) is showing—that is, when an SQL Statement action is selected—it becomes a fully functional SQL environment for creating and testing queries in real time. From this pane, you can perform the following:

◆ Take data from an Input Message (or other available Message Part) and use it to create or modify an SQL Query against a relational data source

◆ Take the results of that query and put it into a Message Part (e.g., Temp, Output, MyDom, etc.)

The Query pane includes three tabs: the SQL Statement tab, the Results Mapping tab, and the Results Text tab.

SQL Statement Tab

When the Query pane first opens, it displays the SQL Statement tab in a live SQL environment. The SQL Statement tab is where you'll write or build SQL commands. (See illustration below.) It may be necessary to resize the SQL Statement pane in order to see the SQL edit box. You can build whole or partial statements by doubleclicking nodes in the Data and/or SQL Operators trees, or by typing SQL straight into the bottom of the window.
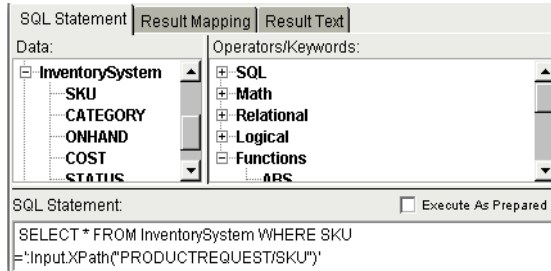
Result Mapping Tab

The **Result Mapping** tab allows you to map the result of your database query into an XML document. It also allows you to designate the exact XML branch element under which you'd like the query result to appear. The Result Mapping tab is shown below.
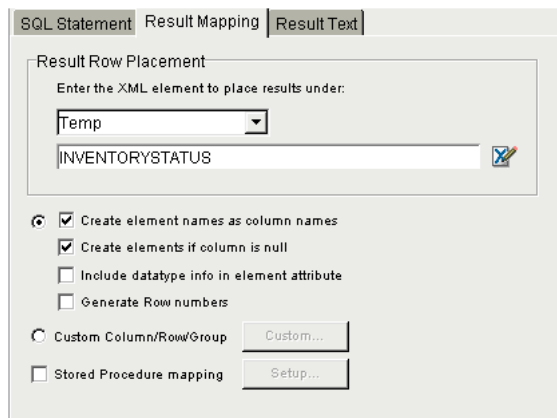
Result Text Tab

The **Result Text** tab (see below) displays the actual SQL statement sent and the data that was returned following the execution of the database query. This is helpful if errant data shows up in a Temporary or Output Part. You can compare the data from the Result Text tab with the data in the XML Message to see where the error occurred.
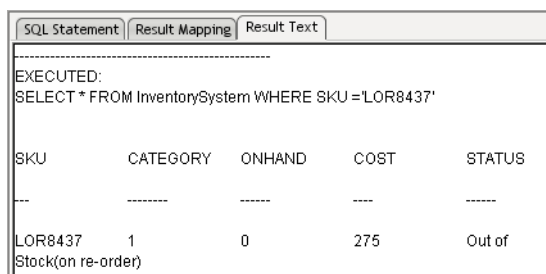
4 Performing JDBC Actions

About Actions

An *action* is similar to a programming statement in that it takes input in the form of parameters and performs specific tasks. Please see the chapters in the *Composer User's Guide* devoted to Actions.

Within the JDBC Component Editor, a set of instructions for processing XML documents or communicating with non-XML data sources is created as part of an *Action Model*. The Action Model performs all data mapping, data transformation, data transfer between SQL databases and XML documents, and data transfer within components and services.

An Action Model is made up of a list of actions. All actions within an Action Model work together. As an example, one Action Model might contain individual actions that read invoice data from a disk, retrieve data from an inventory database, map the result to a temporary XML document, make a conversion, and map the converted data to an output XML document.

The Action Model mentioned above would be composed of several discrete actions. These actions would:

- Open an invoice document and perform an SQL command to retrieve invoice data from a database
- Map the result to a temporary XML document
- Convert a numeric code using a Code Table and map the result to an Output XML document

Two of the actions available in Composer are specific to JDBC Components. These are the SQL Statement Action and the SQL Batch Action.
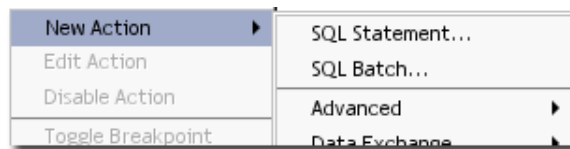
These actions are described below.

The SQL Statement Action

The SQL Statement action is most commonly used to query an existing database and then map the result to an XML document. However, the full set of SQL Data Manipulation Language (DML) statements can be utilized (including database inserts, deletes, and updates).

There are two ways to use the SQL Statement Action. The first is to create your SQL statement using the wizard. The second is to create a custom SQL statement either by typing it in directly or by selecting command statements from the ECMAScript Expression Builder. In either case, you should be familiar with SQL database commands and with the structure of the database(s) you are querying in order to create valid statements with the SQL Statement action.

Handling of Binary Data

When you obtain *binary data* from a database that supports binary types (such as MySQL, which supports CHAR BINARY, VARCHAR BINARY, TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB binaries), you are dealing with data that potentially contains characters and/or character combinations that are illegal in XML.

NOTE: Merely mapping such data into a CDATA section is not a satisfactory solution, because some characters (such as "angle brackets") are illegal in CDATA. Also, the character-combo "]]>" is not allowed *within* CDATA, since it signals the end of a CDATA section.

One satisfactory way to handle binary data is to use Base64 encoding, which essentially turns arbitrary byte streams into XML-safe ASCII streams. Composer's default behavior is to automatically Base64-encode binary data whenever possible, such as when binary data are returned from a database during a SELECT or other "read" operation. Conversely, Composer will automatically Base64-decode binary data before INSERTing or otherwise pushing it into a database. You do not have to take any special action to make this happen.

If you want to take direct control over encoding or decoding of data, you can do so with the Composer-defined ECMAScript extension methods **base64Encode()** and **base64Decode()**. The former takes a byte[ ] array argument and returns a String. The latter takes a String and returns a byte[ ] array.

Prepared Statements

The JDBC Connect has the ability to prepare (or precompile) SQL commands and cache them in memory so that when the same command executes over and over again (for example, in a loop), the cached statement can be reused, with new argument values inserted as need be. This can be a significant performance optimization in cases where statements execute many times.

You can designate any SQL statement as a "prepared statement," whether it was created manually or via the wizard, by using the "Execute as prepared" checkbox. This checkbox is located on the first dialog of the wizard, and also provided just above the SQL edit box for manually created SQL Statements:

☐ Execute As Prepared

By default, this checkbox is unchecked. For SQL Statement actions that are executed only once in the course of a service's lifetime, it is recommended that you leave the checkbox disabled. For statements inside loops, the checkbox can be checked.

NOTE: You may want to do some benchmarking to determine whether and to what degree using the **Execute as Prepared** checkbox is beneficial in a given application.

Creating an SQL Statement using the Wizard

The SQL Creation wizard leads you through the process of creating an SQL query. Composer offers you the ability to create SQL statements using the SELECT, DELETE, INSERT and UPDATE commands. Of course, the userid with which you access the database must have the privileges required to perform these actions for your JDBC component to work correctly. Most userids will be able to SELECT from tables by default, but often you must have special permission to perform DELETE, INSERT and UPDATE actions on tables. Check with your Database Administrator if you are in doubt.

The SQL SELECT Statement

The SQL Select Statement is used to select and return data from a table. For examples on how to use the SQL Select statement, refer to http://www.w3schools.com/sql/sql_select.asp. Depending on the size and structure of your table, a simple SELECT statement may return a lot of data. For this reason, SELECT statements are often filtered using a WHERE clause.

➢ **To create an SQL SELECT Statement action using the wizard:**

1   Create or open a JDBC Component.

2   Highlight a line in the Action Model where you want to place the SQL Statement action. The new action will be inserted below the line you highlight.

3   From the **Action** menu, select **New Action**, then **SQL Statement**.

4   Indicate that you wish to **Create a SQL statement using the wizard**.
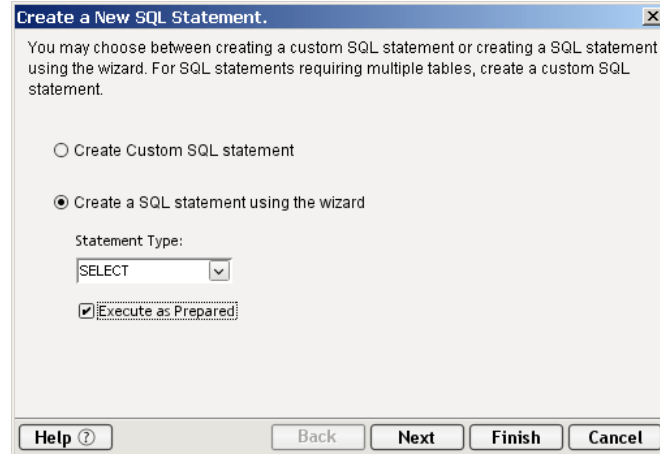
5   Choose **SELECT** as the Statement Type.

6   Click **Next** to display the dialog which allows you to choose a table from which to select your data.
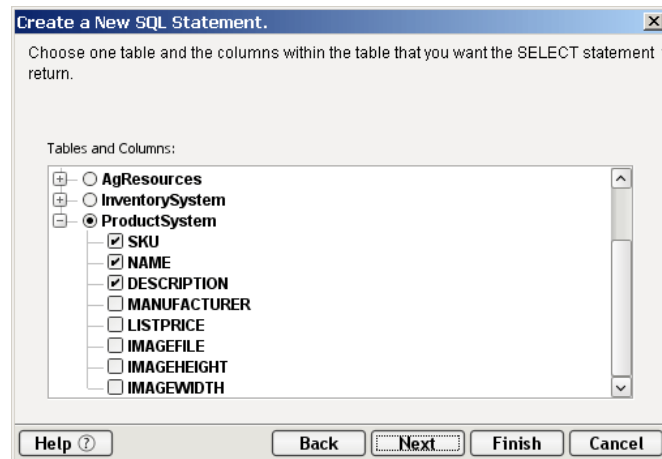
7   Select the table and columns used for the SELECT statement by checking the radio button check boxes associated with the required columns of the table you wish to use. You will notice that if you hover your cursor over a column, descriptive information about that column, such as its TYPE and whether or not it can be a NULL field appears.

NOTE: You can select or deselect all the columns in a table by checking or unchecking the box at the table level.

8   Click **Next** to bring up another dialog, which allows you to select columns to use in your WHERE statement to filter the results of the SELECT statement.
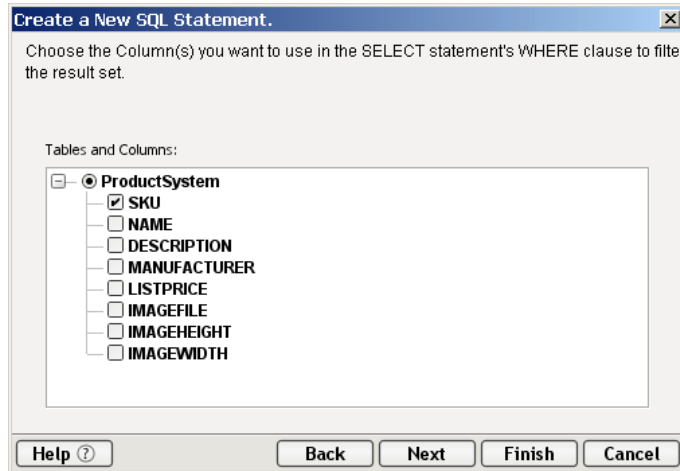
9   Click **Next** to move to the final dialog, in which you specify the Target Message Part and XPath placement for the results of your SQL Statement.
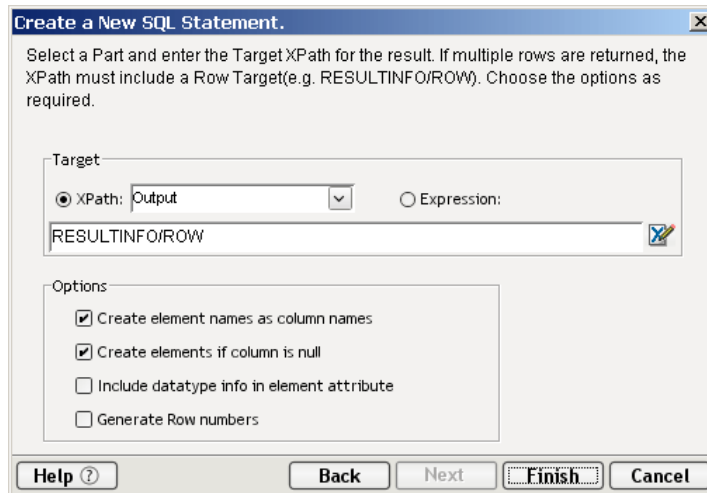
You can either specify an XPath, or select Expression to go to the ECMAScript Expression Builder and

Optionally, you may also choose to:

- **Create element names as column name**.
- **Create elements if column is null**. This creates XML elements with empty content if the column returned has no data.
- **Include data type info in element attribute**. This creates an attribute for each element indicating the data type of the result column.
- **Generate row numbers** (if applicable).

10   Click **Finish** to create the action and return to the JDBC Component Editor.

WHERE Clauses

The execute SQL SELECT statement is now displayed and highlighted in the Action Model. When focus is on this new action, the Native Environment Pane displays a two-tabbed dialog which includes a **WHERE** tab and a **Result Text** tab. WHERE will be visible by default. This tab will be used to filter the result set.
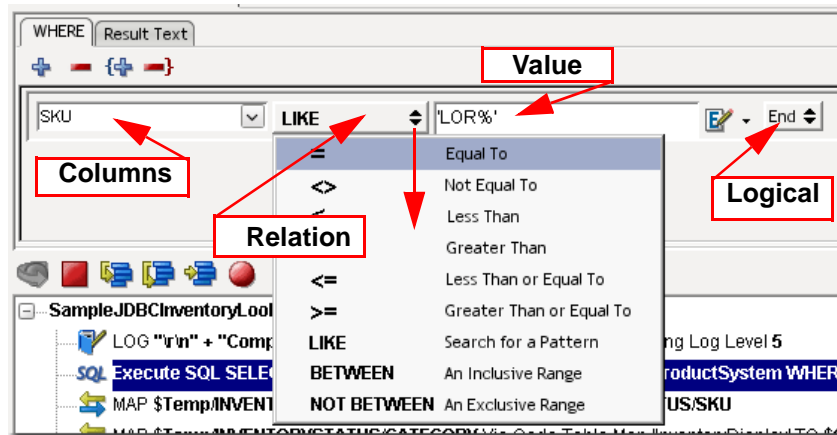
➢ **Filtering the resultset using the WHERE tab:**

1 Select the **Columns** you wish to filter using the dropdown menu. This list is populated according to the columns you chose in step three of the wizard. You may select one or more columns with which to filter the list. To add a column to the filter for the result set, click on the + icon. To delete a column, click the - icon. Columns can also be selected by group. To add a group, click the {+ icon. To delete a group, click the -} icon.

2 Select a **Relation** from the dropdown list. Examples for all these relational operators can be found at http://www.w3schools.com/sql/sql_where.asp.

When using the LIKE operator, the **%** symbol can be used as a wildcard character representing any number of missing characters at the beginning or ending of your matching pattern. Text values should be surrounded by single quotes, though most databases will also accept double quotes.

It is important to note that the BETWEEN...AND operator can be interpreted differently by different databases. With some, "between" is literal and only values in between your test cases will be selected. Some databases will include the test cases in your result set also. Some include the first case but not the last, and vice versa. In general, with SQL, you should follow the advice of that famous television lawyer and "Never ask a question you don't already know the answer to."

3 For **Value**, either a constant or an expression can be entered. You may also drag and drop fields from your XML Message Parts to create an expression.

4 The Logical dropdown menu allows you to create more complex WHERE clauses using **And/Or** logic. Or, you may complete the clause by selecting **End**.

Once you have adjusted your WHERE clause to filter your results appropriately, you will see the completed SQL statement in the Action Model.

If you open the Result Text tab, you will be able to see the text of the SQL and the results produced by running the query.
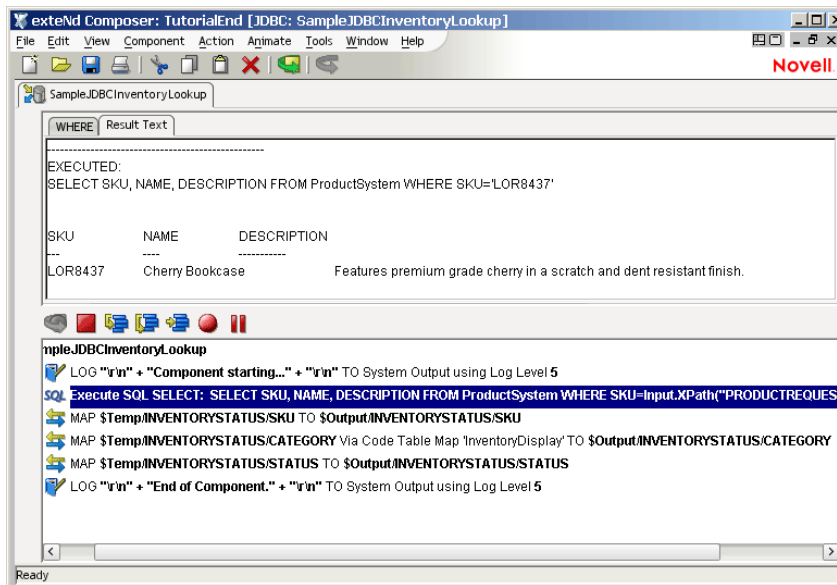
The SQL DELETE Statement

Delete statements in SQL are used to delete entire rows from tables. If you wish to delete, null out or otherwise modify individual column values within rows in a table, you should use the MODIFY command (described below). The steps to follow to create an SQL DELETE statement are fairly similar to those for creating an SQL SELECT statement.

➢ **To create an SQL DELETE Statement action using the wizard:**

1 Create or open a JDBC Component.

2 Highlight a line in the Action Model where you want to place the SQL Statement action. The new action will be inserted below the line you highlight.

3 From the **Action** menu, select **New Action**, then **SQL Statement**.

**4** Indicate that you wish to **Create a SQL statement using the wizard**.

5 Select **DELETE** for your Statement Type.

6 Click **Next** to select the table from which rows will be deleted using the DELETE statement.
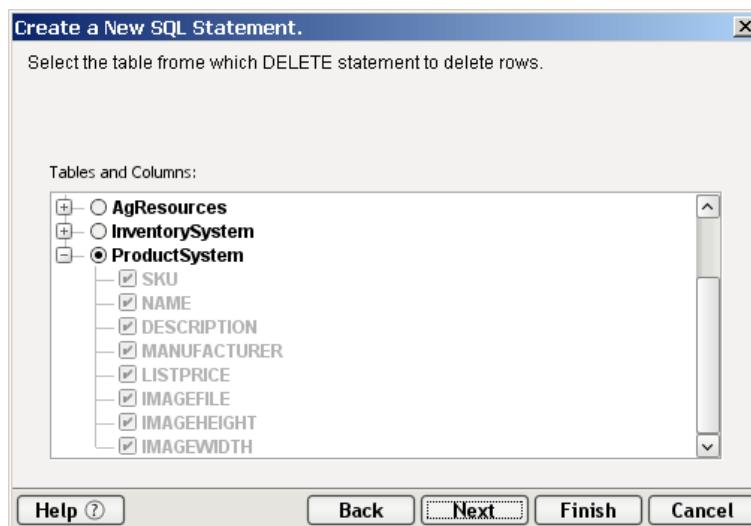
Only one table can be checked at a time. In the case of DELETE, you will not be able to select individual columns at this point in the wizard. This screen is for table selection only, and the columns are all selected and grayed out, indicating that they will all be available for selection in the next dialog of the wizard.

7 Click **Next** to open the next dialog, from which you will select the column(s) which will be used by the DELETE statement's WHERE clause to filter the records which will be deleted.
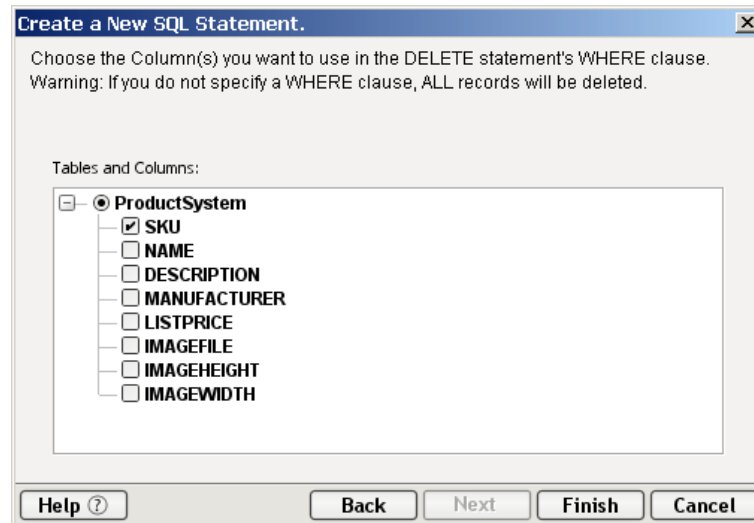
8 Click **Finish** to create the new action and display it in the Action Model. As described above in the SELECT statement, the WHERE tab will be displayed. Use the WHERE filtering (described in "WHERE Clauses" above) to complete your SQL Delete statement. The Result Text tab shows the text of the SQL and the results produced by running the statement.

The SQL INSERT Statement

Insert statements in SQL are used to insert entire rows into tables. If you wish to insert or otherwise modify individual column values within rows in a table, you should use the MODIFY command (described below). The steps to follow to create an SQL INSERT statement are also fairly similar to those for creating an SQL SELECT statement.

➢ **To create an SQL INSERT Statement action using the wizard:**

1 Create or open a JDBC Component.

2 Highlight a line in the Action Model where you want to place the SQL Statement action. The new action will be inserted below the line you highlight.

3 From the **Action** menu, select **New Action**, then **SQL Statement**.

4 Indicate that you wish to **Create a SQL statement using the wizard**.

5 Select **INSERT** for your Statement Type.

6   Click **Next** to select the table(s) into which rows will be inserted by the INSERT statement. At the same time, select the columns which will be provided with new data by the statement.
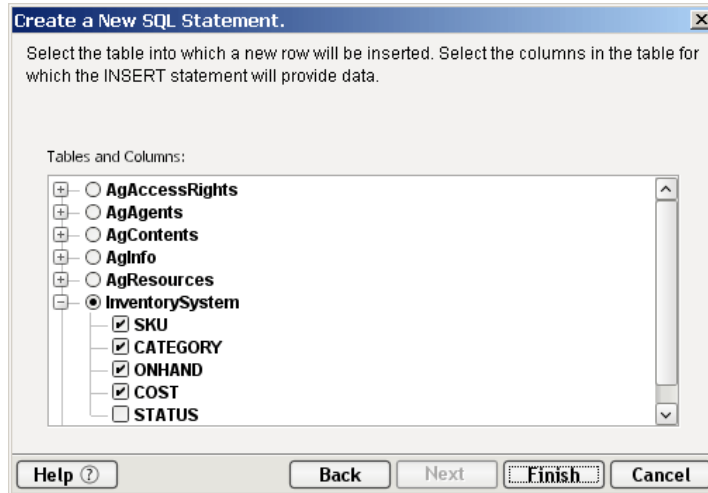
7   Click **Finish** to insert the new SQL Insert Statement into your Action Model and return to the Component Editor.

The Native Environment Pane displays two tabs: Column Values and Result Text. Column Values will be displayed by default.
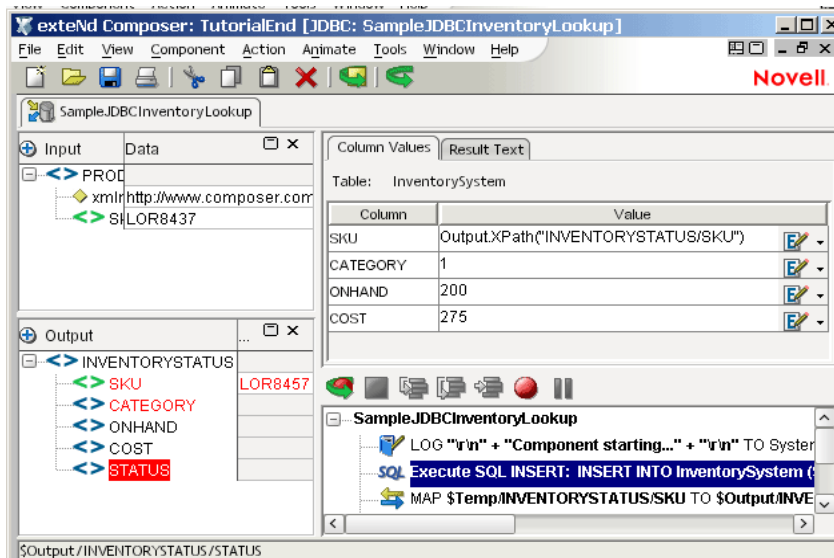
Specifying Column Values

The Column Values pane displays a table with two columns. The first presents a list of the columns selected during the final step of the SQL Insert wizard. In the second column, you will define the values for the columns of the row to be inserted. You also have the ability to drag and drop data from a Message Part to the Value column, as shown in the SKU example above.

As always, the Result Text tab shows the text of the SQL and the results produced by running the statement. You will notice that Composer automatically surrounds non-numeric data with single quotes.

```
------------------------------------------------
EXECUTED:
INSERT INTO InventorySystem (SKU, CATEGORY, ONHAND, COST) VALUES ('LOR8457', 1 , 200, 275)

SQLCODE = 0
UPDATECOUNT = 1
```

The SQL UPDATE Statement

Update statements in SQL are used to modify data within the rows and/or columns of a table. The steps to follow to create an SQL UPDATE statement are also fairly similar to those for creating an SQL SELECT statement.

➢ **To create an SQL UPDATE Statement action using the wizard:**

1 Create or open a JDBC Component.

2 Highlight a line in the Action Model where you want to place the SQL Statement action. The new action will be inserted below the line you highlight.

3 From the **Action** menu, select **New Action**, then **SQL Statement**.

4 Indicate that you wish to **Create a SQL statement using the wizard**.

5 Select **UPDATE** for your Statement Type.

6 Click **Next** to select the table(s) and columns to modify with the SQL statement.
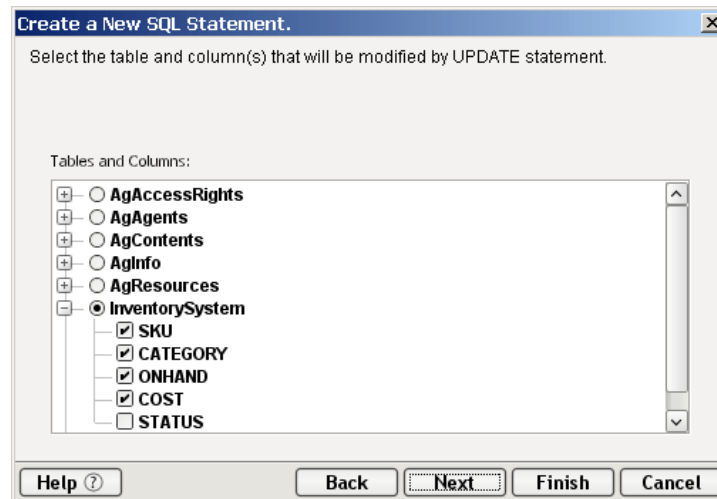
7 Click Next to bring up the final dialog, which allows you to select which columns will be used by the WHERE clause of the Update statement.
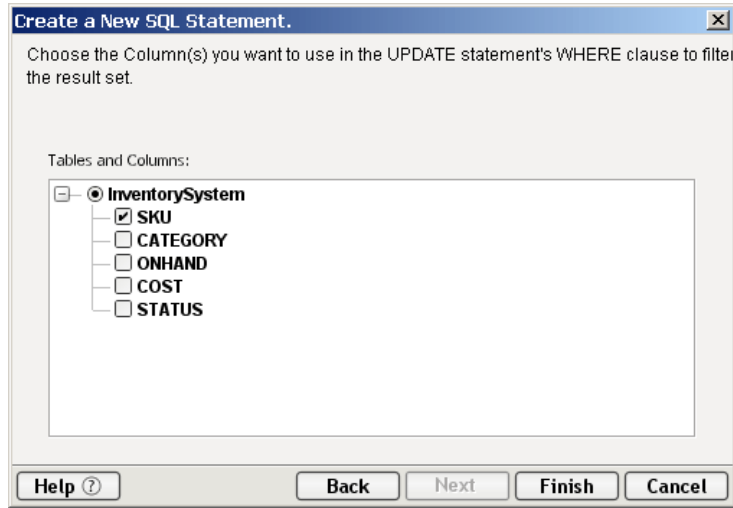
8 Select the appropriate columns and click Finish to complete the action and add it to the Action Model.

Just as with the SELECT and DELETE commands, the Native Environment Pane will display a Where tab and a Result Text Tab. In this case, though, it will also display a Column Values tab as seen with the Insert command.

Use the Where tab to filter the record set to be updated as demonstrated in "WHERE Clauses" on page -27 above. You may select the columns and define the criteria for those columns in order to update only the desired records.

Tab to Column Values to provide the values for each of the columns to be updated. Updating Column Values is demonstrated in "Specifying Column Values" above.

As always, the Result Text Tab shows the text of the SQL and the results produced by running the statement. You will notice that Composer automatically surrounds non-numeric data with single quotes.

```
| WHERE | Column Values | Result Text |
--------------------------------------------------------
EXECUTED:
UPDATE InventorySystem SET SKU = 'LOR8437', CATEGORY = '2', ONHAND =
366, COST = 12, STATUS = '1' WHERE SKU='LOR8437'

SQLCODE = 0
UPDATECOUNT = 0
```
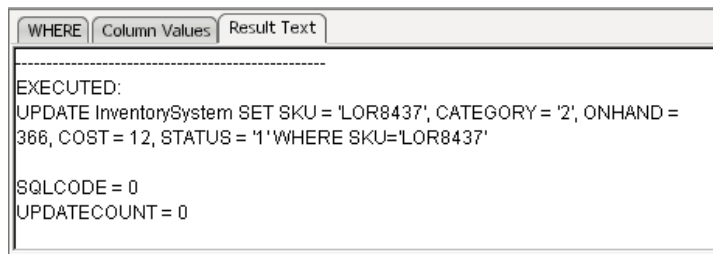
Editing a SQL Statement Created with the Wizard

Once you have created your SQL statement, you may find that you need to edit it. This is a two part process. Begin by double-clicking on the EXECUTE SQL action in the Action Model. This will bring up a tabbed dialog, as shown below.

```
Edit SQL Statement                                                    [X]
| Table and column selection panel | WHERE clause column selector | Result Map Properties |
Choose one table and the columns within the table that you want the SELECT statement to
return.

  Tables and Columns:
    [+]— ○ AgAgents                                              [^]
    [+]— ○ AgContents
    [+]— ○ AgInfo
    [+]— ○ AgResources
    [-]— ● InventorySystem
          — [✓] SKU
          — [✓] CATEGORY
          — [✓] ONHAND
          — [✓] COST
          — [✓] STATUS
    [+]— ○ ProductSystem                                         [v]

| Help ⑦ |                                    | OK | | Cancel |
```
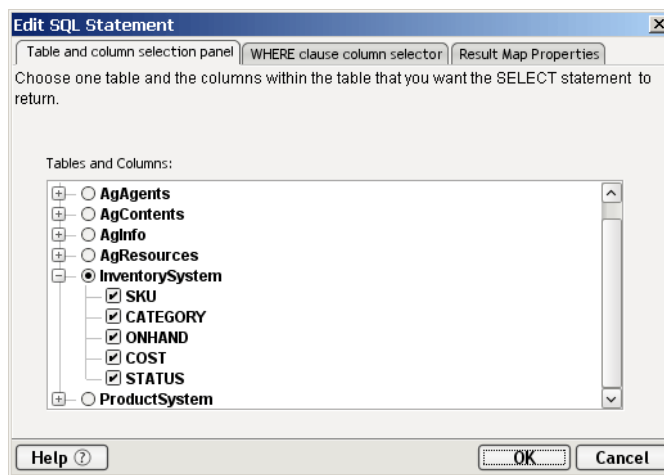
These tabs allow you modify the basic Table, Column and Target selections for the SQL Query. The number of tabs will vary according to the type of SQL Statement you are editing.

- The **Table and column selection panel** tab is available for all SQL statement types. It allows you to modify the tables and/or columns you had chosen to use in your SQL query.
- The **WHERE clause column selector** tab is available for the SELECT, DELETE and UPDATE statement types. Use this tab to modify the columns you had chosen to use for your Where clause.
- The **Result Map Properties** is available only for SQL SELECT actions created using the Wizard. Here you can modify the Target location for the results of your query.

Once you have edited the information in these tabs, you may need to further modify the SQL Statement using the additional tabs available when the item in the Action Model is clicked on a single time, or after you have clicked on **OK** in the Edit SQL Statement tabs, described above.

Back in the Native Environment Pane, you will see a screen that resembles the following.

```
| WHERE | Column Values | Result Text |
  ✚  ━  {✚ ━}

  | SKU          |v|   =   |≑|  'LOR8437'
```
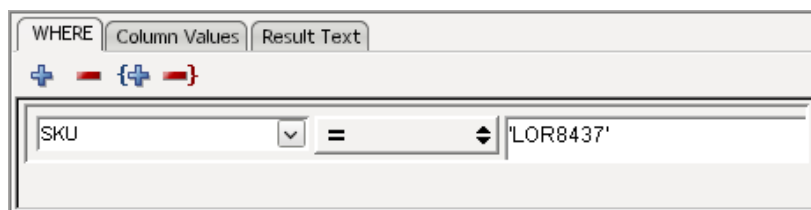
Again, the number of tabs shown will vary according to the type of SQL Statement.

- The **WHERE** tab is available for all SELECT, DELETE and Update SQL Statements prepared using the wizard. Here you can modify the filter chosen to limit your query.
- The **Column Values** tab is available for INSERT and UPDATE queries. Use this tab to modify the values you originally designated as being inserted or changed as a result of your SQL statement.
- The **Result Text** tab is available for all SQL Statements. It shows the query that was executed and the results it produced.

Creating an SQL Statement Manually

The manual creation of SQL statements for use in JDBC Components is done inside the Query/Result Mapping Pane.

Query/Result Mapping Pane

SQL Statement control box
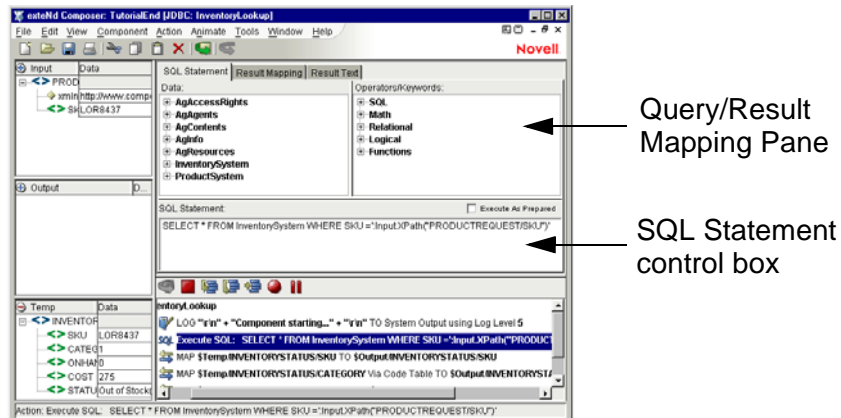
If you are editing a previously created action model that already contains SQL Statement actions, you can make the Query/Result Mapping Pane come into view simply by selecting (clicking on) any existing SQL Statement action. Otherwise, you will create an SQL Statement action.

➢ **To manually create an SQL Statement action:**

1. Create or open a JDBC Component.
2. Highlight a line in the Action Model where you want to place the SQL Statement action. The new action will be inserted below the line you highlight.
3. From the **Action** menu, select **New Action**, then **SQL Statement**.
4. Indicate that you wish to **Create a Custom SQL Statement**. The Query/Result Mapping pane appears in the Native Environment pane of the JDBC Component Editor window, as shown above.

Building an SQL Statement Manually

Building an SQL Statement manually involves bringing together data, operators, and keywords.

➢ **To build an SQL Statement:**

1. Place the cursor in the **SQL Statement** control box in the Query/Result Mapping pane.
2. Expand the **Data** columns and/or the **Operator/Keywords** by clicking the plus signs. The illustration below shows Data and Operator/Keywords trees look like with several parent nodes expanded.
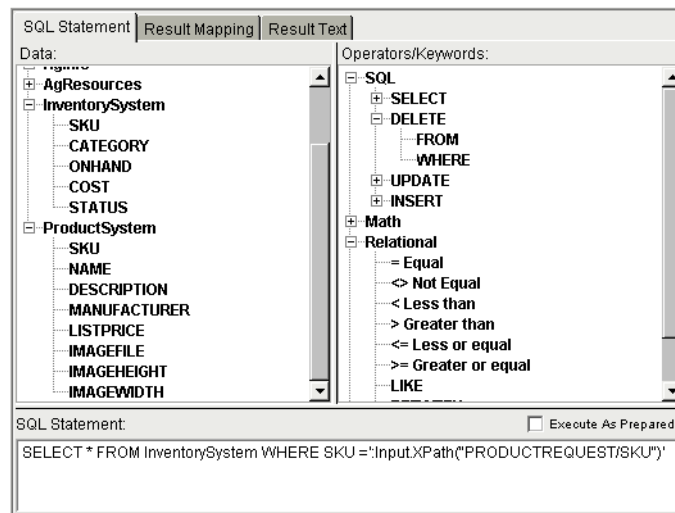
3 Double-click each **Data** column and/or **Operator/Keyword** that you would like to add to the **SQL Statement** box. When you double-click an item, it automatically appears in the **SQL Statement** box at the insertion point.

4 Optionally, you may drag elements from an open DOM tree (e.g., the Input DOM pane) into the **SQL Statement** box.

5 Optionally check the **Execute as Prepared** checkbox. (See discussion further above, under "Prepared Statements".)

Building an Example Query

Here is an example SQL statement:

```
SELECT * FROM ProductSystem WHERE SKU = ':Input.XPath("PRODUCTREQUEST/SKU")';
```

In order to build this statement, the component must satisfy the following:

♦ The component must be able to use a (previously defined) connection resource to connect to the database

♦ The database must have a table called ProductSystem that has a column called SKU

♦ The component must have a template containing a sample XML document with a root element, PRODUCTREQUEST, that has a child element named SKU

This example statement, in plain English, means:

"Select all columns from the database's ProductSystem table where a record's value in column SKU is equal to the content of the Input DOM's PRODUCTREQUEST/SKU element."

➢ To build the example statement:

1 Expand the **SQL** tree in the Expression builder and double-click **SELECT**.

2 Double-click * in the Expression Builder.

3 Double-click **FROM** in the Expression Builder.

4 Type **ProductSystem**.

5 Double-click **WHERE** in the Expression Builder.

6 Type **SKU =**.

7 Select **SKU** in the Input DOM and drag it into the SQL Statement control.

8 Optionally type a semicolon (**;**) at the end of the SQL Statement.

9   Select **File>Save**. The Query/Result Pane should look like this:
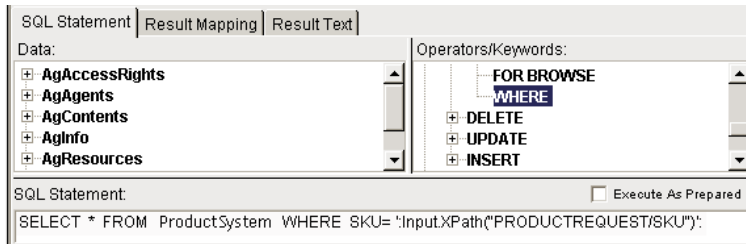
Mapping Results into the Output DOM

When you have created your SQL Statement manually, you must use the **Result Mapping** pane to select where to place the rows and columns of your results into the XML Document tree.

➢ **To use Result Mapping:**

1   Select the **Result Mapping** tab in the **Query/Results Mapping** pane. The Results Mapping pane appears.
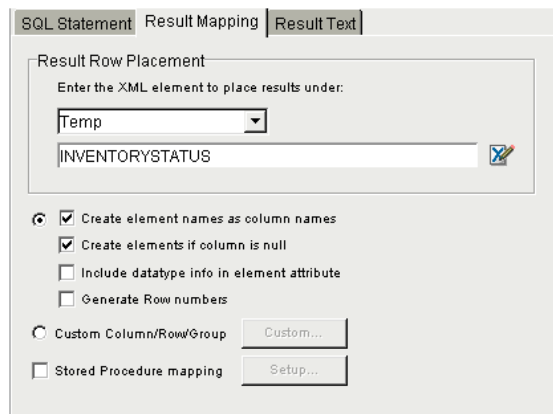
2   Under **Result Row Placement**, select the destination Part to which you would like the result of the SQL query mapped.

3   Next, select the Part element under which you'd like each result row to appear. If an appropriate Part is not listed, you may add another XML template using the **File>Properties>Messages** dialog from the menu. If a Part is not visible, go to **View>XML Documents>Show/Hide**.

4   Select options as follows:

Default Result Mapping: Choose the first radio button for standard Column/Row/Group mapping:

- ◆ **Create element name as column name**.
- ◆ **Create elements if column is null**. This creates XML elements with empty content if the column returned has no data.
- ◆ **Include data type info in element attribute**. This creates an attribute for each element indicating the data type of the result column.
- ◆ **Generate row numbers** (if applicable).

Custom Result Mapping: Choose the second radio button, Custom Column/Row/Group, to perform custom column, row, or group mapping (see Chapter 5).

Stored Procedure Mapping: Choose Stored Procedure mapping to map data returned from stored procedures. (see Chapter 6).

5   Select **File>Save**.

Editing a Manually Created SQL Statement

To edit a SQL statement once you have created it manually, simply click on the EXECUTE SQL action in the Action Model.
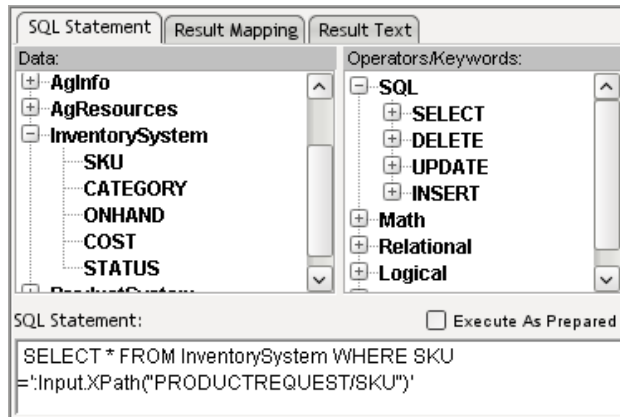
- Use the SQL Statement Tab to edit the Text of your SQL statement manually or use the methods above to change your selections of Data, Operators and Keywords.
- Use the Result Mapping Tab to modify the target placement for the returned data.
- Use the Result Text Tab to show the query that was executed and the results of the query.
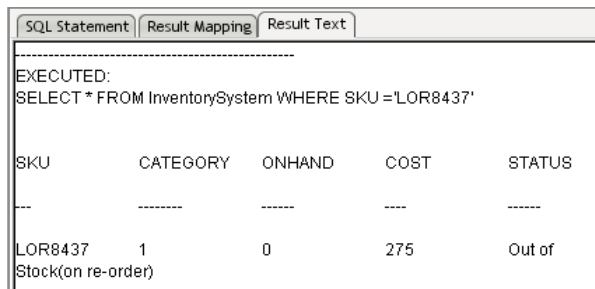
Executing the SQL Statement

After you have built the SQL Statement, either manually or using the wizard, click the **Execute** button to run it.
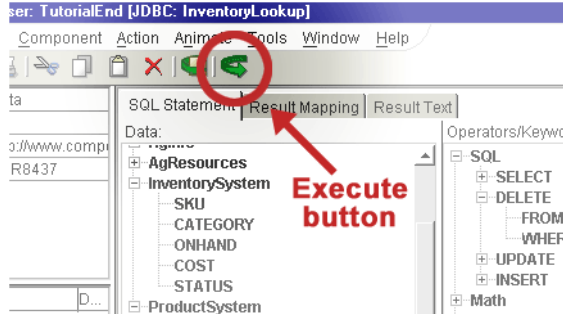
Checking the Results

You can check the results of your SQL statement by looking at the data retrieved in the familiar row and column format. To do so, click the **Result Text** tab. This tab is available for all SQL Statements, whether created manually or using the wizard.

```
SQL Statement | Result Mapping | Result Text

-------------------------------------------------
EXECUTED:
SELECT * FROM InventorySystem WHERE SKU ='LOR8437'


SKU           CATEGORY    ONHAND    COST      STATUS

---           --------    ------    ----      ------

LOR8437       1           0         275       Out of
Stock(on re-order)
```

If the query result returned by the SQL statement looks correct, you can continue designing your component's Action Model. Otherwise, you can return to the SQL Statement tab and debug your SQL as necessary.

Using Stored Procedures

Many RDBMS vendors provide the ability to execute procedural code stored in the RDBMS system. Using these stored procedures allows for high-performance interfaces that are independent of the underlying table implementations.

Using stored procedures can be helpful in controlling access to data. User access to data can be limited to the scope of the stored procedure. Limiting access to data with stored procedures preserves data integrity by insuring data is entered in a consistent manner. Stored procedures also improve efficiency. They're memory resident, which speeds execution. Their use decreases network traffic. Productivity is improved via their use since stored procedures only need to be written and debugged once but can be reused by many.

While often used interchangeably, for the sake of discussion we'll differentiate between the terms Procedures and Functions. A Procedure is a subroutine that doesn't necessarily return any data but may via the call's parameters or as external result sets. A Function, on the other hand, always returns *something*. Both Procedures and Functions can pass Parameters.

Novell exteNd Composer allows you to map parameters to stored procedures and functions, execute stored procedures and functions and map returned data to DOM/node combinations.

Syntax Requirements

In order to package the Procedure or Function call correctly, exteNd Composer requires certain formatting conventions be followed. For example:

{ – indicates that a call to a Function or Procedure follows

} – indicates the end of a call to a Function or Procedure

The syntax for procedures and functions support parameters which may be Expressions, Placeholders or Constants.

Expression: Expressions may be used to pass variable input data to a procedure or function. Expressions used as parameters in procedure and function calls are preceded with a colon (:) and enclosed in single quotes. (e.g. **':<variablename>'**).

Question Mark: Question Marks (?) may be used as parameters and serve as placeholders to which the procedure returns data. A question mark is also used for the result in a function.

Constant: Constants are used to pass input data in procedures and functions but, unlike expressions or placeholders cannot be used to accept returned data. Literal values are enclosed in single quotes.

Rules for Stored Procedure Parameters

Stored procedures may have Input Parameters, Input/Output parameters and Output Parameters.

Input Parameters: Input Parameters pass data to stored procedures. Input Parameters may be Constants or Expressions.

Input/Output Parameters: Input/Output Parameters pass data to stored procedures and accept data returned from stored procedures. Input/Output parameters must be Expressions.

Output Parameters: Output Parameters accept data returned from stored procedures. Output parameters may be either an Expression or a Question Mark as a placeholder.

Using Procedures and Functions in a JDBC Component

For all the examples below the following steps should be executed.

• Add a new SQL action

• **Execute as Prepared** is set to true (check the checkbox; see "Prepared Statements" on page 19).

NOTE: For mapping the results of stored procedures, see Chapter 6.

Syntax for running a Procedure from within exteNd Composer

Procedures that do not return a value:

```
{ call [<packagename>.]<procedurename>[([param1, param2…,paramn])]}
```

Example:

```
{ call composerDemoPackage.sp1_withParams('12345','George') }
```

Procedures that return a result set:

```
{ call [<packagename>.]<procedurename>[([param1, param2…?…..paramn])]}
```

where ? is a parameter to which the result set is returned. A result set may also be returned to other parameters which contain Expressions.

Example:

```
{ call composerDemoPackage.sp_withParams('93324', ':FirstName', ?)}
```

In this example '93324' is a constant, ':FirstName' is an Expression and ? is a placeholder.

NOTE: Only Oracle returns result sets as parameters. Non-Oracle RDBMSs may return result sets but, not as parameters.

Backward Compatibility for Oracle Procedures that return a result set:

Prior to version 4.0, exteNd Composer provided support for Oracle Procedures that return result sets as parameters. To do so, exteNd Composer (prior to version 4.0) required the user to specify the Oracle Cursor Position within the procedure call. The pre-Composer 4.0 syntax included **ocp:n** – where **ocp** stands for Oracle Cursor Position and **:n** indicates which parameter contains the cursor. This syntax was used in pre-4.0 versions of exteNd Composer and is maintained in version 4.0 and greater for backward compatibility.

```
{ call [<packagename>.]<procedurename>[([param1, param2…ocp:x…..paramn])]}
```

Example:

```
{ call composerDemoPackage.sp_withParams('93324', 'Melissa', ocp:3)}
```

NOTE: The contents of the result set will be returned in the same manner as a standard SELECT statement. The results will be automatically be mapped to the selected XML Document. The defaults are Output as the Document and RESULTINFO/ROW as the XPath location.

Syntax for Calling a Function from within Composer

Functions that return a result set:

```
{ ? = [<packagename>].<functionName>[([param1, param2…,paramn])]}
```

Example:

```
{ ? = call composerDemoPackage.fn_justOneReturn( ) }
```

Backward Compatibility for Oracle Functions that return a result set:

To provide backward compatibility with pre-4.0 versions of exteNd Composer, the following syntax will continue to be supported in exteNd Composer 4.0 and greater.

```
{ ocp:1 = [<packagename>].<functionName>[([param1, param2…,paramn])]}
```

Example:

```
{ ocp:1 = call composerDemoPackage.fn_justOneReturn( ) }
```

Other Methods of Calling Functions for Specific Tasks

You may call any function that does not update the database from within a select statement.

Example:

```
select fn_addMin(4,6) "Sum" from dual
```

To use a function that does not return a result set but updates the database, call it from within a function that *does* return a result set – see the example fn_callAddMin

Example:

```
{ ? = call composerDemoPackage.fn_callAddMin(22,44 ) }
```

Colons in SQL Statements

Colons are special characters in SQL Statements, because exteNd Composer treats colons as markers indicating the presence of ECMAScript immediately to the right. In the above action, the SQL Statement includes the string

```
':Input.XPath("PRODUCTREQUEST/SKU")'
```

which contains a colon followed by an ECMAScript expression involving the XPath() method. Without the colon, the string would be evaluated as a string-literal. With the colon, it is evaluated as an ECMAScript expression.

NOTE: If you need to use colons as literal values inside SQL Statements, escape every occurrence of a literal colon with a backslash. Otherwise, you may see errors.

The SQL Batch Action

Most database drivers allow batch execution of SQL statements in order to minimize demand on connection resources. For example, a user may want to insert data into a table in one database and delete data from a table in another database, all in one round trip. This is possible with the SQL Batch action.

SQL Batch actions allow you to specify that a particular group of SQL Statement actions should be accumulated into a single batch and transmitted to the database as a unit.

NOTE: SELECT operations may not be used in batches. Use only INSERT, DELETE, and UPDATE statements.

To access the SQL Batch action, right-click inside the action pane and choose **New Action > SQL Batch** as shown below.
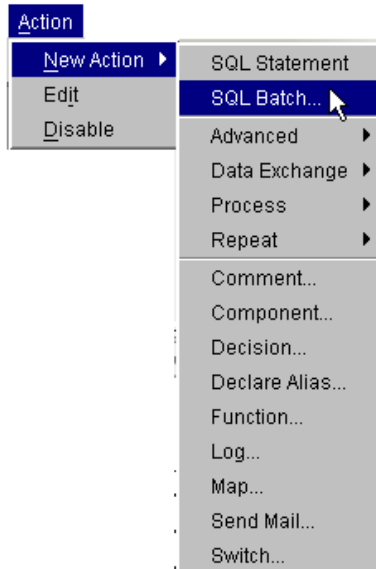
There are three SQL Batch commands, each of which places a new action in the action model: Start Batch, Execute Batch, and Discard Batch.

Start Batch

You must tell Composer where the beginning of a batch occurs, by placing a Start Batch statement before the first SQL Statement in a series of statements that you want to group. This command sets a checkpoint for rollback purposes (in case the batch does not finish normally).

From the first occurrence of this command until the next occurrence of an Execute Batch command (see below), SQL Statements are merely accumulated, rather than executed. *Execution of a batch does not occur until an Execute Batch command is reached.*

Regular (non-SQL) actions, such as Map and Function actions, are not affected by Batch operations. If you place Map actions, Function actions, or any other non-SQL actions within or after a group of batched SQL Statement actions, those actions will execute *before* the SQL Statements in your batch, because the batch cannot execute until an Execute Batch is reached.

Execute Batch

An Execute Batch command causes all SQL Statements in a batch to be sent, as a unit, to the database. (If no Execute Batch command is issued, none of the SQL Statements in the preceding batch will get executed.)

An Execute Batch statement can be placed immediately after a batch of SQL Statement actions, or it can be placed at some point downstream of the batched actions (possibly in one branch of a Decision action). In other words, you can create a batch in one location and execute it, conditionally, from another location in your action model.

Discard Batch

The Discard Batch command is a memory-de-allocating command that causes the previously held batch to go out of scope. It frees the memory held by the preceding batch.

Ordinarily, when an SQL batch executes without error, the batch is discarded automatically after it executes and there is no need to issue an explicit discard. You would use Discard Batch when you have an action model that contains two or more sequential SQL batches (each with its own Execute Batch command) wrapped in Try/On Fault statements. The need for the Discard Batch arises when one of the upstream batches executes abnormally (generating an exception). In order to continue to another batch, you need to purge the previous batch from memory (with a Discard Batch in the On Error branch of the "Try" action). Failure to use Discard Batch under these conditions would cause the next Start Batch to throw an exception. This scenario is shown in the illustration below.
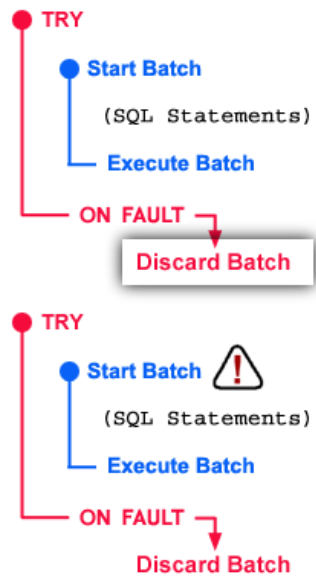
In the case depicted above, where there are two SQL batches (each enclosed in a Try/On Error action), failure to include a Discard Batch action in the error branch of the first Try will cause the next Start Batch to throw an exception (assuming the first batch fails).

In summary: When two or more batches will execute sequentially, wrap each in a Try/On Error action and *include a Discard Batch command in the On Error branch of each*.

For action models in which there is only a *single* SQL batch, Discard Batch is not necessary. After normal execution of a (single) batch, memory allocated to the batch is released automatically; and if the batch returns an error, the batch will go out of scope (and be garbage-collected) when the component itself goes out of scope.

Creating Batch actions

Batch actions are created using the SQL Batch menu command (available from **Action > New Action > SQL Batch** in the JDBC Component Editor main menu, or via **New Action > SQL Batch** in the contextual menu).

➢ **To create a SQL Batch action:**

1 Place the cursor in a line preceding the group of SQL Statements that you want to batch. Then press the right mouse button and select **New Action > SQL Batch**. The Batch setup dialog appears.
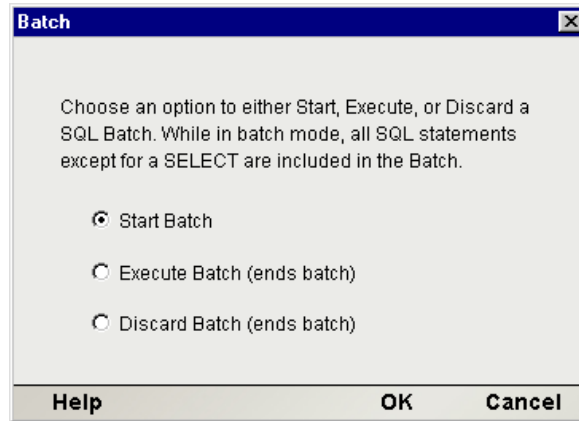
2 Choose the **Start Batch** radio button to insert a Start Batch command in your action model. Otherwise, choose **Execute Batch** or **Discard Batch**, as appropriate.

3 Click **OK** to dismiss the dialog. A new action appears in your action model.

JDBC-Specific Expression Builder Properties

SQL queries can result in certain status and/or error values being returned (for example, the number of records that were changed by an Update). Often, it is useful to be able to reference these values in ECMAScript expressions. The Expression Builder pick list (in the top portion of the Expression Editor window) contains properties that are specific to JDBC Actions involving SQL: namely, SQLSTATE, SQLCODE, and UPDATECOUNT. (See panel below.)
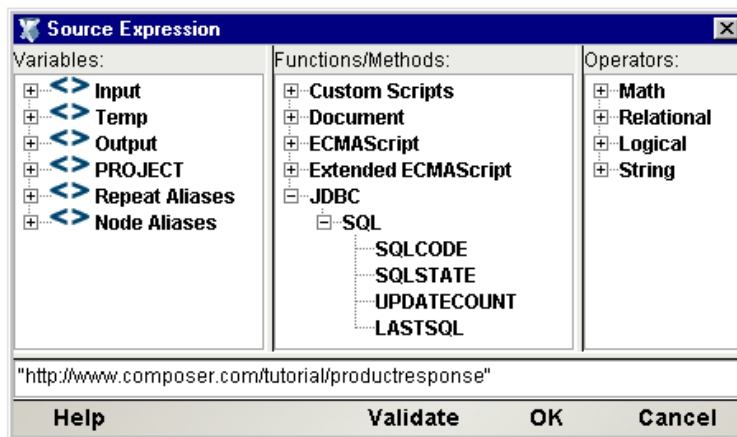
Using Other Actions in the JDBC Component Editor

In addition to the SQL Statement action, you have all the standard Basic and Advanced Composer actions at your disposal as well. The complete listing of Basic Composer Actions can be found in Chapter 7 of the *Composer User's Guide*. Chapter 8 contains a listing of the more Advanced Actions available to you.

Handling Errors and SQL Messages

SQL returns certain coded values when errors occur (i.e., no record was found in a Query) or as a report on the result of certain actions (i.e., how many records were changed by an Update). These results appear on the Result Text tab as three special variables labeled:

- SQLSTATE
- SQLCODE
- UPDATECOUNT
- LASTSQL

These variables are available to ECMAScript functions you may write and can be used for error handling within your JDBC component. For instance, you can create a Decision action to process after an SQL statement. Based on the value returned in the UPDATECOUNT variable, you can choose one or the other set of actions in the two branches of the Decision action. Likewise, error information contained in SQLSTATE or SQLCODE (which are standard SQL status variables) can be used to branch to appropriate recovery logic in case of error.

The LASTSQL variable is an exteNd-defined string variable which contains *the last SQL statement to actually execute* in the component in question. Logging the value of this variable can be useful for troubleshooting.

5 Using Custom Result Mapping

The following sections describe the similarities and differences between default and custom result mapping for the Execute SQL action. Custom mapping features are described in detail.

About Default Result Mapping

The mapping of data returned from an Execute SQL action is determined by specifications on the Result Mapping tab in the SQL Mapping pane. The two **Result Row Placement** controls allow you to determine where in the target document to place the result set data. The drop down list specifies the Message Part or Repeat alias context and the Expression edit box specifies the XPath location within the Context.

The Context is either the name of a Part in the component or the name of a Repeat alias already specified in the component (where the Repeat alias itself represents a Message Part context and XPath location). The Expression edit box specifies an XPath, the last element of which acts as the parent element for the returned results and will receive the data. The last element that receives the data is called the **Row Target**. If multiple rows are returned, then multiple Row Targets will be created. Each column returned in a row will appear as a child element of each Row Target.
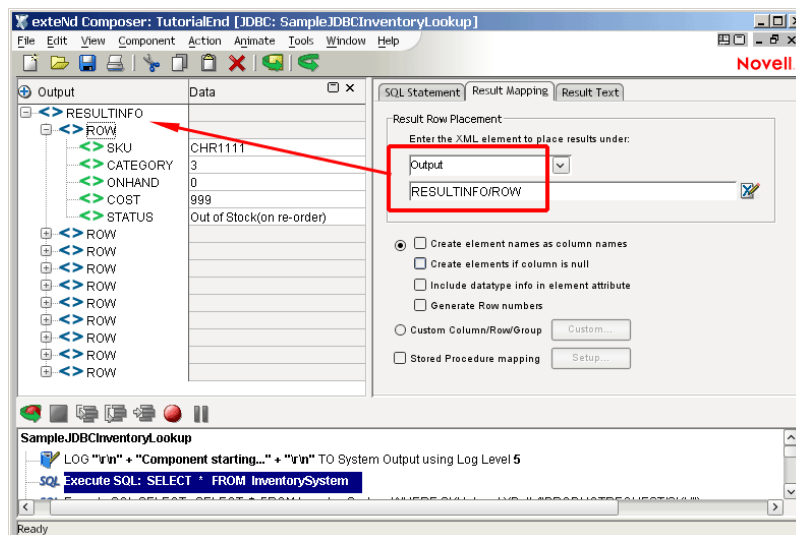
By default, the Row Target is named "ROW" and is a child of a root element named "RESULTINFO," and the results are written to Output, as shown above. Notice that no checkboxes are checked in the Result Mapping pane.

You can change the result mapping to use any target XPath of your choice. For example, you can use the Result Mapping tab to specify a Row Target such as **Temp/RESULTINFO/Result** as shown in the graphic below.
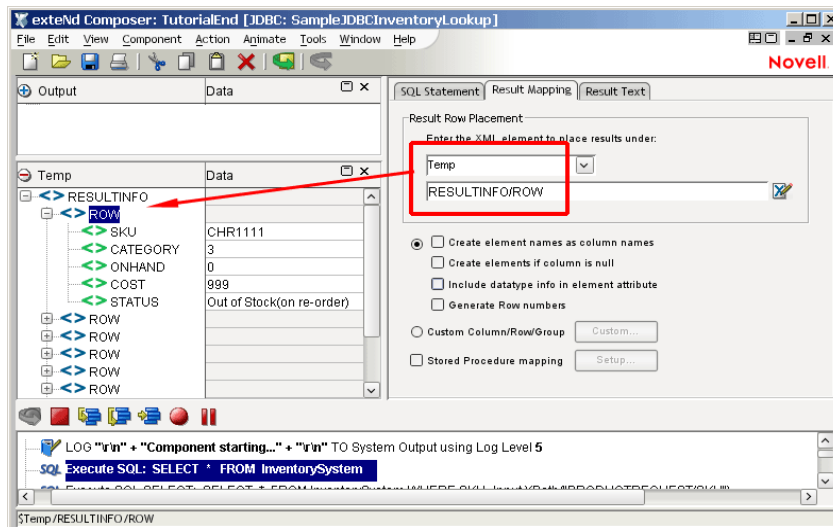
Result Mapping functionality includes the following default behaviors:

- Target element names created in the document are the same as column names returned in the result set
- All columns returned in the result set are mapped to the target document
- All columns are mapped to the same parent target element
- All rows are placed into a single document

NOTE: Any column names that contain spaces will have the spaces replaced with the underscore character since XML does not permit spaces in element names.

About Custom Result Mapping

Use custom result mapping to:

- Create target element names different than the column names returned
- Map columns to different row targets
- Group the result set data by one or more columns
- Map only group information
- Map group and detail information

Custom result mapping is accessed via the **Custom ...** button on the Result Mapping tab.
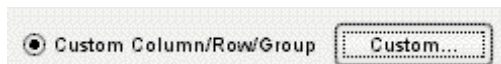
If you click this button, you will be presented with a dialog that has three tabs, labeled Map Target, Detail Rows, and Declare Group/Repeat.
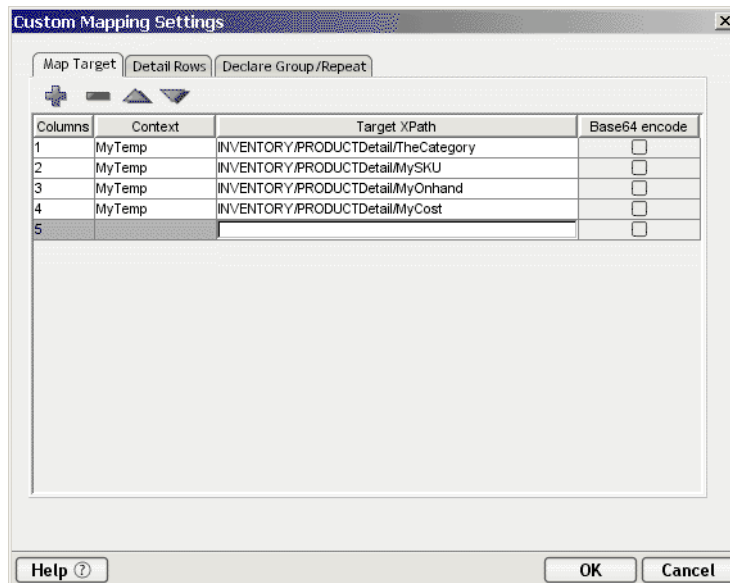
The use of this dialog is discussed in detail below.

About Custom Result Mapping and Aliases

Novell exteNd Composer's default mapping behavior is to iterate through a list of one or more nodes (i.e., elements specified by an XPath pattern) from a source document, and map them to a single target document XPath location. If the target location doesn't exist, Composer creates it. If you know the source list is greater than one, you must indicate to exteNd Composer whether you wish to map to the same physical target location for each member of the source list (i.e., overwrite the data in the specified physical target location), or create a new physical target location for each member of the source list (i.e., add new target locations as the repeated source is mapped). You indicate that you want to map each member of the source list to the same physical target location by specifying the Context as an actual DOM name. You indicate that you want to map each member of the source list to a *new* physical target location by specifying the Context using an alias.

NOTE: This is also true for the Repeat for Element and the Repeat for Group actions.

You can think of the multiple rows of data returned by a SELECT statement as a repeating set of elements in an XML document. In that case, you may choose to create a Declare Group action creating a list of Groups and Detail elements within the Groups. Then you would create a Repeat for Group action to process the Group list or detail of each Group. The Custom Map Target, Detail Rows, and Declare Group/Repeat tabs provide a similar alias ability for repeating rows in SQL result sets as the Declare Group and Repeat for Group actions do for repeating elements in a document.

Using the MapTarget Tab

The **Map Target** tab is used to:

◆ Create your own target element names for each result set column

◆ Specify a target Context for each result set column

The Map Target tab controls the mapping of *each* returned row's individual columns. For each column, you specify a Context – Target XPath combination. The Context – Target XPath combination is specified for each column in the order they are listed in the projection list for the SELECT statement in your Execute SQL action. You cannot use Custom Result Mapping without filling in the Map Target tab.

The Map Target table will initially appear without any rows. Use the + icon to add additional rows. Use the - icon to delete rows. Use the up and down arrows to arrange the rows of the Map Target table.

Column: This number refers to the columns in the order they are listed in your SELECT statement.

Context: This specifies the target document for the column. The Target XPath will be appended to the Context to produce the full XPath location for the column in the target document. The Context can be a:

◆ *Document* – You may use this choice if your result set contains only one row, otherwise each additional row will overwrite the previous row's data.

◆ *Detail Alias* – A Detail Alias is defined on the Detail Rows tab and consists of a Document name and partial Target XPath. Or the Detail Alias may consist of a Group Alias (defined on the Declare Group/Repeat tab) and partial Target XPath location. Using a Detail Alias tells exteNd Composer to create a new physical target location for each member of the source list (i.e., each row in a result set).

◆ *Group Alias* – A Group Alias is defined on the Declare Group/Repeat tab and consists of a Document name and partial XPath location. Using a Group Alias tells exteNd Composer to create a new physical target location once for each Group in the source list (i.e., where each group represents multiple rows in a result set).

◆ *Repeat Alias* – If the Execute SQL action is contained with a Repeat action in your Action Model you may choose its Target alias. In this case, the Context will resolve to a Document and partial XPath to which the Target XPath (see below) will be appended.

When grouping and mapping detail column data, the Declare Group/Repeat, Detail Rows, and Map Target tab work together to define the complete XPath location for the column. (See illustration.) For instance, a column on the Map Target tab will be represented by a Context and XPath. The Context may be a Detail Alias defined on the Detail Rows tab. The Detail Alias in turn will represent another Context and XPath. Its Context may be a Group Alias defined on the Declare Group/Repeat tab. Finally the Group Alias itself will represent another Context and XPath.

By defining the Group and Detail aliases separately, you are able to map rows with duplicate column data (the basis for your groups) just once into group header elements by using the Group alias as a context, and map columns with unique data (the detail of your groups) multiple times within the group header elements by using a Detail Alias whose Context is a Group Alias.

Target XPath: This is an XPath fragment that specifies the custom name to be given to the column and optionally pre-pended by any additional parent elements. The Target XPath will be pre-pended by the Context to produce the final location for the column in the target document.

Base64 encode: The checkbox in this column allows you to convert binary data to an XML-safe representation for use in a DOM element.

NOTE: Composer's default behavior is to automatically Base64-encode binary data returned from a database during a SELECT or other "read" operation. This is necessary to ensure that the target XML node contains no "illegal characters." See "Handling of Binary Data" on page 24 for additional info.
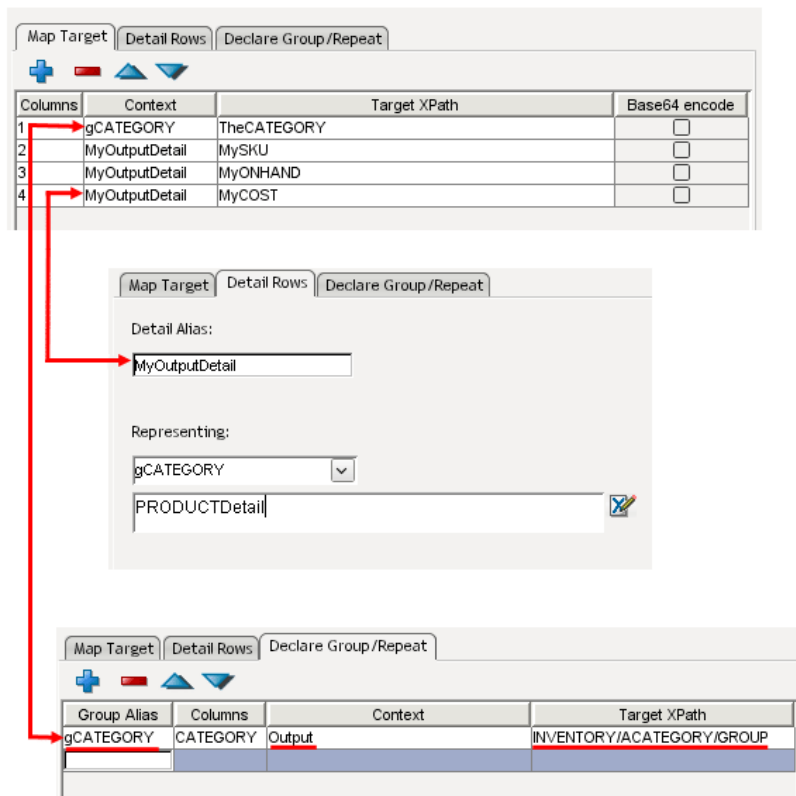
*The three tabs of the Custom Mapping Settings dialog can be used to define
sophisticated any-to-any mappings of result-set items to Part element.
Notice how user-defined aliases (representing, in each case,
a Part context and target XPath) can be substituted back
into earlier tab context slots.*

A processing summary for the Map Target tab is shown in the table below.

| SQL Results	Context = Document	Context = Alias
One Row Returned	One row target is found or created for the first (and only) result row.	One row target is found or created for the first (and only) result row.
Multiple Rows Returned	One row target is found or created for the first result row. Subsequent rows find and map to the *same* physical target location. (Without an alias, each row's data is overwritten by the next row until only the last row's data is left.)	One row target is created for *every* result row.

Looking at a MapTarget Example

Let's assume your Execute SQL action issued the following statement:

```
SELECT CATEGORY, SKU, ONHAND, COST FROM InventorySystem
```

Which returned the following row data:

Category	SKU	Onhand	Cost
3	CHR1111	0	999
2	DAD7777	89	245
4	GAR1234	17	100
1	LOR8437	0	275
1	LOR8438	21	375
4	MOM4666	233	300
4	RAC4567	156	230
4	ZAC9080	4	555

You could fill out the Map Target tab as shown below:
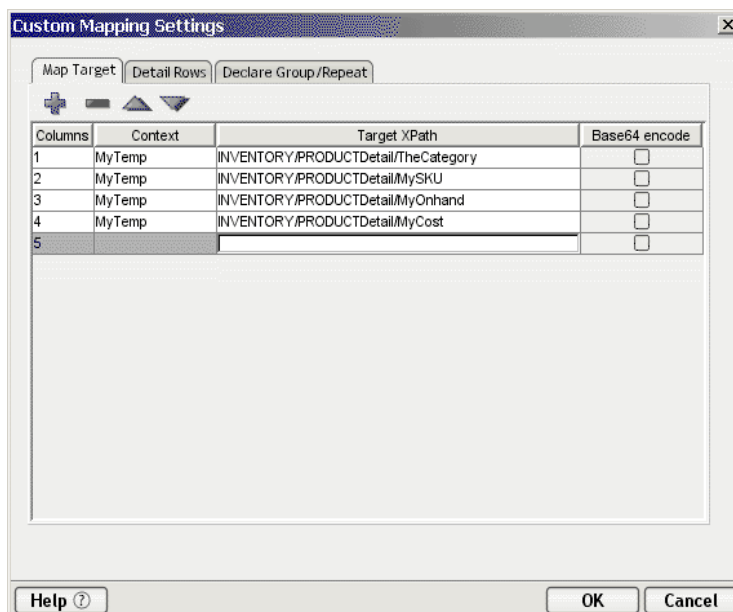
Column one according to the SELECT statement will be CATEGORY. The Context is a document named "MyTemp" and the target XPath location within the Context will be "INVENTORY/PRODUCTDetail/TheCATEGORY". Notice that CATEGORY is being renamed to TheCATEGORY and being pre-pended with parent elements of INVENTORY/PRODUCTDetail. This same logic applies to the remaining columns.

However, since we have yet to define or use any aliases, each row's column data will be written to the same four physical target locations specified on the tab. If only one row is returned, then its data will be mapped to the target document with no problems. If multiple rows are returned as in our example, then each successive row's data will overwrite the previous row's data until only the last row's data exists. (Only in rare cases will this situation be desirable.)

Normally, you use the Map Target tab by itself if only one row is returned and all you wish to do is change the names of the target elements to something different than the column names. (Or if you want to assign different parent elements to individual columns.)

To avoid overwriting data with multiple result-set rows, you need to use a *Detail Alias* from the Detail Rows tab telling exteNd Composer to create a new physical target location for each row mapped.

Using The Detail Rows Tab

The Detail Rows tab allows you to create a mapping alias tied to either a document Context or a Group/Repeat alias Context. Use of the Detail Rows tab is optional.

Detail Alias: This is a name you specify that will be referenced as a Context on the Map Target tab for mapping columns in a result set row.

Context: This is a document name or Group/Repeat alias you specify. The Target XPath will be appended to this Context to produce part of the final location for the column in the target document (the remaining part comes from the Target XPath on the Map Target tab). The Context can be a:

◆ *Document* – Using a Document name tells exteNd Composer to create a new physical target location once for each row in the result set.

◆ *Group Alias* – A Group Alias is defined on the Declare Group/Repeat tab and consists of a Document name and partial Target XPath location. Using a Group Alias tells exteNd Composer to create a new physical target location once for each detail row belonging to each Group (i.e., each group represents multiple rows in a result set).

Target XPath: This is an XPath fragment that you specify. It will be pre-pended by the Context on this tab and appended with the Target XPath on the Map Target tab to complete the final location for the column in the target document.

## Looking at a Detail Rows Example

Assuming your Execute SQL action issued the following statement:

```
SELECT CATEGORY, SKU, ONHAND, COST FROM InventorySystem
```

You could fill out the Detail Rows tab as shown below:
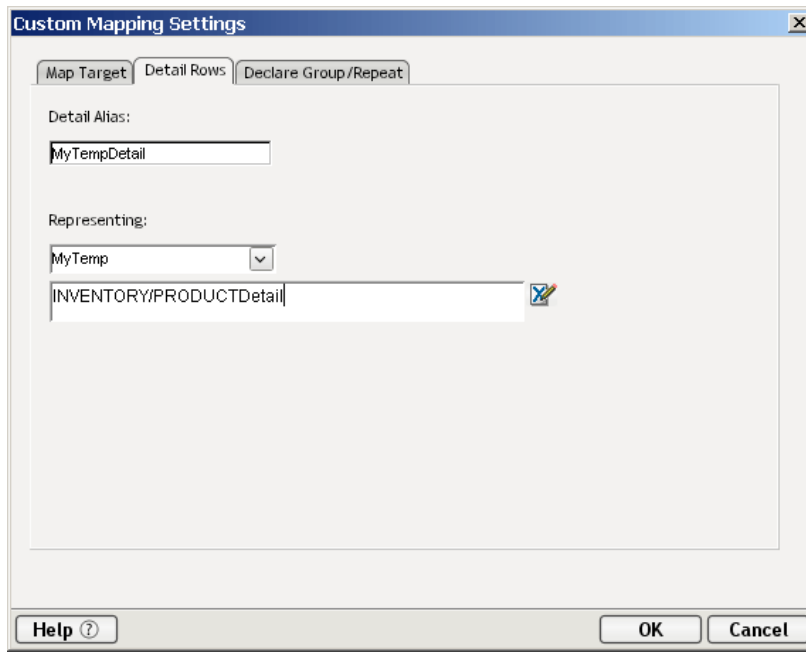
Since the Context MyTemp and Target XPath fragment INVENTORY/PRODUCTDetail are now specified on the Detail Rows tab (creating a new physical target location for each row), references to them must be replaced on the Map Target tab with the Detail Alias "MyTempDetail." Continuing the example used in the previous section, you would update the Map Target tab as follows:
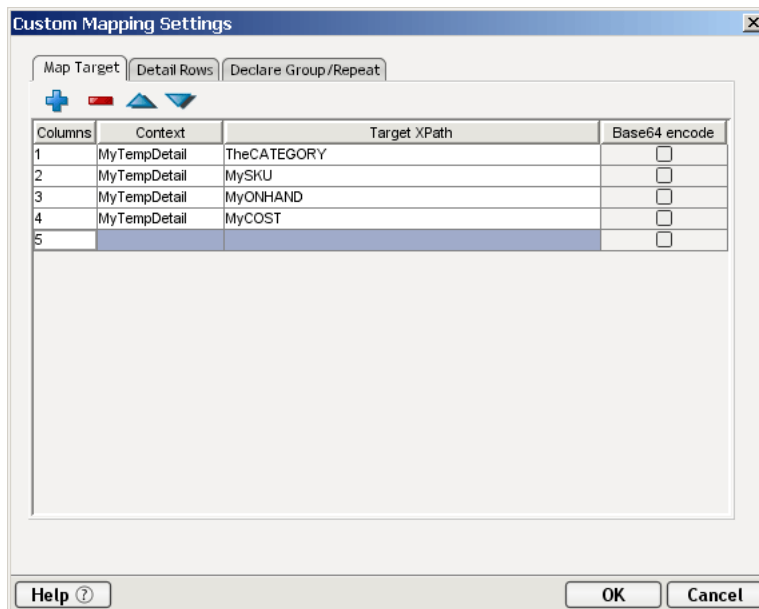
By using a Detail Alias specified on the Detail Rows tab, you will ensure that if multiple rows are returned in the result set, each row will create a new physical target location under INVENTORY/PRODUCTDetail.

When not used in conjunction with the Declare Group/Repeat tab, you can think of the Detail Rows tab as creating a "Repeat for Row" alias. If the Context for a Column on the Map Target tab is a Detail Alias (instead of a document), then exteNd creates a new Target XPath each time a row mapping occurs. In this way, *multiple rows in the result set create multiple Row Targets in the document without overwriting the previous row's data*. This is the same functionality provided by the Result Mapping tab's **Custom…** option, except that you get to rename the columns.

The result set data may not be arranged exactly the way we want, however. For example, the sub-trees under **PRODUCTDetail** (see illustration above) are listed without regard to product category information. If you look under **PRODUCTDetail/TheCATEGORY**, you can see that two rows belong to category 1, and one row each belong to categories 2 and 3. (This example is in the Action Examples project under the Sample directory in your Composer installation. You might want to step through the JDBC Component from which the above screen shot was taken, which is called "Custom Result Mapping in JDBC.")

Perhaps you'd rather see row data grouped according to category. To do this, you need to use a *Group Alias* from the Declare Group/Repeat tab.

Using the Declare Group/Repeat Tab

The Declare Group/Repeat tab is used to:

◆   Create groups of result set records based on one or more result set columns

◆   Create a Group Alias to use as a Context for Detail Rows

◆   Create a Group Alias to use as a Context for Map Targets (creating Group Headers)

By declaring a Group Alias you create a list comprised of the unique values found in a column across multiple rows. Any Map Target column that uses the Group Alias will map its column data only once for each unique Group essentially creating group header information.

In addition, each unique group value points to a list of the rows that belong to it. Any Detail Alias on the Detail Rows tab that uses the Group Alias will map its rows together for that group.

Group Alias: This is a name you specify that is referenced as a Context on the Map Target and/or Detail Rows tabs.

Columns: Specify one or more columns separated by a comma to create your groups. Using two columns means that only unique combinations of the concatenated values of the two columns will create a group.

NOTE: The columns you specify must form the basis of an ORDER BY clause in the SELECT statement for the Execute SQL action. If you omit the ORDER BY clause, your results will be unpredictable.

Context: This is a document name in the component or Repeat for Group or Repeat for Element alias in the Action Model that contains the Execute SQL action. The Target XPath is appended to this Context to produce part of the final location for the column in the target document. (The remaining part comes from the Target XPath on the Map Target tab and optionally from the Target XPath on the Detail Rows tab.) The Context can be a:

- Document – Using a Document name tells exteNd Composer to write to the same physical document for each Group.
- Repeat for Group Alias – If your Execute SQL action is inside a Repeat for Group action in your Action Model, then you may use its target alias as the Context for each Group. This tells exteNd Composer to create new Groups once for each Group processed in the enclosing Repeat for Group action.
- Repeat for Element Alias - If your Execute SQL action is inside a Repeat for Element action in your Action Model, then you may use its target alias as the Context for each Group. This tells exteNd Composer to create new Groups once for each repeating element processed in the enclosing Repeat for Element action.

Target XPath: This is an XPath fragment that you specify. It is pre-pended by the Context on this tab and appended with the Target XPath on the Map Target tab (and optionally with the Target XPath on the Detail Rows tab) to complete the final location for the column in the target document.

Looking at a Declare Group/Repeat Example

Assuming your Execute SQL action issued the following statement:

```
SELECT CATEGORY, SKU, ONHAND, COST FROM InventorySystem order by CATEGORY
```

You could fill out the Detail Rows tab as shown below:
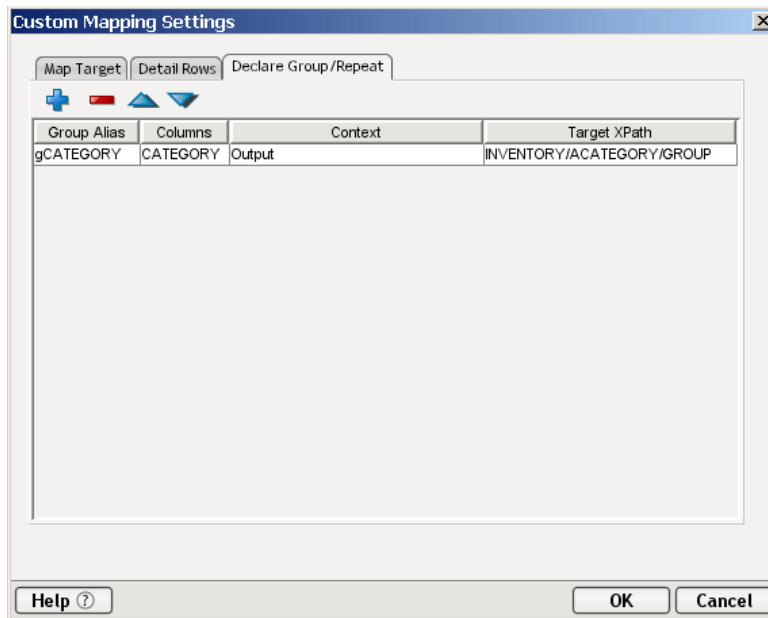
Similar to the example for Detail Rows, since the Context MyTemp (and Target XPath fragment INVENTORY/PRODUCT) is now specified on the Declare Group/Repeat tab, references to it must be replaced on the Detail Rows tab with the Group Alias "gCATEGORY." In addition, you are no longer listing just PRODUCTDetail under INVENTORY but rather groups of PRODUCTDetail so a new element is introduced into the Group's Target XPath called "ACATEGORYGroup." Thus for each Group mapped, a new ACATEGORYGroup element is created.

Continuing the example used in the previous two sections, you would update the Detail Rows tab as follows:
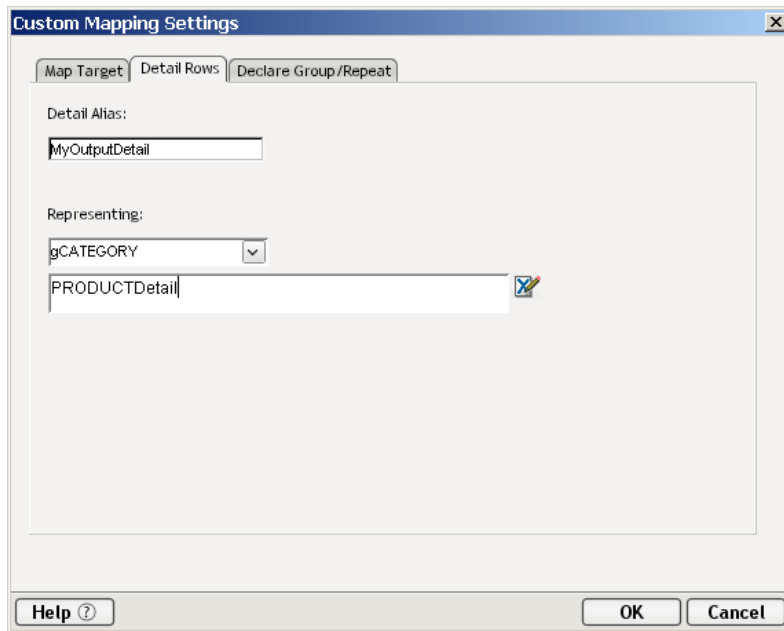
Notice that the Context of "MyTemp" has been replaced by the Group Alias gCATEGORY which represents MyTemp/INVENTORY/ACATEGORYGroup. This means that Detail Rows belonging to the Group are the only ones mapped, instead of all the Detail Rows.

Continuing the example used in the previous two sections, you would update the Map Target tab as follows:
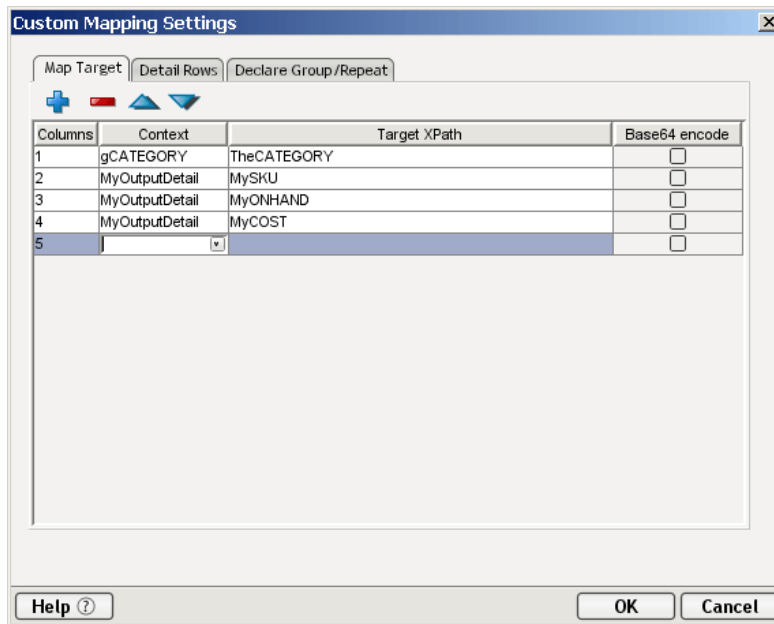
We have replaced the Context for the CATEGORY column with the Group Alias. This means that CATEGORY is only mapped once for each Group instead of once for each detail row.

When you declare a Group Alias, the result set rows are scanned and organized into groups establishing how many processing loops will occur during mapping. If eight rows are in the result set with only four different values (e.g., 3, 2, 4, 1, 1, 4, 4, 4) then there will be four group mapping loops (e.g., 1, 2, 3, 4) and eight detail loops tied to their appropriate group mapping loops (e.g., group one has its two detail rows, group two has its one detail row, group three has its one detail row, and group four has its four detail rows).

Using the prior graphics, you can trace how the final context for the Map Target columns is constructed for Column one and Column two. Column one is the CATEGORY from the result set. Its name in the DOM will be TheCATEGORY. Its ancestor elements are determined by the context "gCATEGORY" defined as MyTemp/INVENTORY/ACATEGORYGroup on the Declare Group/Repeat tab. So the final XPath for CATEGORY is:

```
Output/INVENTORY/ACATEGORYGroup/TheCATEGORY
```

Since the context for TheCATEGORY is a Group alias, it will be mapped once for each group or four times as determined earlier.

Column two is the SKU data from the result set. Its name in the DOM will be MySKU. Its ancestor elements are determined by the context "MyTempDetail" defined to be gCATEGORY (defined above) plus PRODUCTDetail. So the final context for the column will be MyTemp/INVENTORY/ACATEGORYGroup/PRODUCTDetail/MySKU. Since the context for MySKU is a Detail Alias, it is mapped once for each Detail Row. However, each Detail Row has a Context of a Group Alias limiting mapping to only those detail rows that belong to the Group.

6 Stored Procedures

Novell exteNd Composer supports the mapping of data returned by stored procedures. The following sections describes the stored procedure mapping features.

About Stored Procedure Mapping

Novell exteNd Composer allows for mapping the data returned by stored procedures to DOM/Node combinations. To do so, select the **Stored Procedure mapping** checkbox on the **Result Mapping** tab in the **Query/Results Mapping** Pane.

This will enable the **Setup...** button. Press the **Setup...** button to display the **Setup** dialog for **Stored Procedure Mapping.**

Binding Rules

It is important to understand exteNd Composer binds to all **Expressions** and placeholders represented by **Question Marks** placeholders (e.g. either ':<expression>' or ?) but not Constants (e.g. 'abc').

Using the Stored Procedure Mapping Setup Dialog

The Stored Procedure Mapping Setup dialog is used to map the data returned by a stored procedure. The Setup dialog allows you to specify Context - Target XPath combinations for the returned data.

Use the + and **-** controls to add and delete Context - Target XPath combinations.

Oracle RDBMSs return result sets as parameters. Non-Oracle RDBMSs return result sets but, not as parameters. Select the **Returns Result Set** check box when result sets are returned by non-Oracle RDBMSs. Selecting the **Returns Result Set** check box for non-Oracle RDBMSs enable exteNd Composer to find the returned result set.

NOTE: All Expressions and placeholders (e.g. ?) must be specified in the Stored Procedure Mapping Setup dialog in order to correctly map the returned data.

For each returned Input/Output parameter (which may be expressions) and each Output parameter (which may be either an expression or a ?) (see the **Rules for Stored Procedures** section in Chapter 4), complete the following:

Id: Based on the SQL parameters, Id is the number sequence of the return values you're expecting. Using **Id,** you will need to explicitly specify the sequence positions of each of the parameters containing either expressions (e.g. ':<ExpressionName>') or placeholders (e.g. ?). For example, the following procedure call has three parameters: a constant, 'Process', a placeholder, ? and an expression, ':Smith'. The value 'Process' does not need an Id in the **Stored Procedure Mapping** pane since exteNd Composer does not bind to values. The Id entries for the placeholder - ? and the variable ':Smith' are, respectively, 2 and 3. exteNd Composer binds to variables and placeholders, therefore, they must be specified in the **Stored Procedure Mapping** pane in order to properly map the data returned by a stored procedure.

Example:

```
{ call DemoPackage.sp_withParams('Process', ?, ':Smith') }
```

Qual: Qual qualifies the parameter as an Input parameter, an Output parameter or as an Input/Output parameter.

Data Type: Data Type is a drop down list which provides the following options: VARCHAR, DECIMAL, DATE, BINARY or Oracle Result Set. When **Oracle Result Set** is selected as an Input parameter (Qual=In), Context and Target XPath do not apply (N/A) and are, therefore, disabled.

Scale: The value of Scale specifies the decimal place precision.

Map: The Map checkbox is selected to map the parameter.

Context: this specifies the target document for the column. The Target XPath will be appended to the Context to produce the full XPath location for the column in the target document. The Context can be a:

◆ *Document* – You may use this choice if your result set contains only one row, otherwise each additional row will overwrite the previous row's data.

◆ *Detail Alias* – A Detail Alias is defined on the Detail Rows tab and consists of a Document name and partial Target XPath. Or the Detail Alias may consist of a Group Alias (defined on the Declare Group/Repeat tab) and partial Target XPath location. Using a Detail Alias tells exteNd Composer to create a new physical target location for each member of the source list (i.e., each row in a result set).

- *Group Alias* – A Group Alias is defined on the Declare Group/Repeat tab and consists of a Document name and partial XPath location. Using a Group Alias tells exteNd Composer to create a new physical target location once for each Group in the source list (i.e., where each group represents multiple rows in a result set).
- *Repeat Alias* – If the Execute SQL action is contained with a Repeat action in your Action Model you may choose its Target alias. In this case, the Context will resolve to a Document and partial XPath to which the Target XPath (see below) will be appended.
- *-- via standard --* will use the **Result Mapping** tab's Result Row Placement specification.
- *-- via custom --* will use the settings on the **Custom Mapping Settings** dialog.

When grouping and mapping detail column data, the Declare Group/Repeat, Detail Rows, and Map Target tab work together to define the complete XPath location for the column. (See illustration.) For instance, a column on the Map Target tab will be represented by a Context and XPath. The Context may be a Detail Alias defined on the Detail Rows tab. The Detail Alias in turn will represent another Context and XPath. Its Context may be a Group Alias defined on the Declare Group/Repeat tab. Finally the Group Alias itself will represent another Context and XPath.

By defining the Group and Detail aliases separately, you are able to map rows with duplicate column data (the basis for your groups) just once into group header elements by using the Group alias as a context, and map columns with unique data (the detail of your groups) multiple times within the group header elements by using a Detail Alias whose Context is a Group Alias.

Target XPath: This is an XPath fragment which will be appended to Context to specify the full XPath location int the target document.

Returned Result Set

A result set is mapped to a document with elements created from the result set's column names.

- Target element names created in the document are the same as column names returned in the result set
- All columns returned in the result set are mapped to the target document
- All columns are mapped to the same parent target element
- All rows are placed into a single document

NOTE: Any column names that contain spaces will have the spaces replaced with an underscore character since XML does not permit spaces in element names.

A JDBC Glossary

Connection Pool   A set of database connections managed by the application server for the various applications it manages.

Custom Result Mapping   The Custom Result Mapping dialog provides a similar alias ability for repeating rows in SQL result sets as the Declare Group and Repeat for Group actions do for repeating elements in a document.

Declare Group/Repeat Tab   This tab of the Custom Results Mapping dialog is used to create groups of result set records on one or more result set columns, create a Group Alias to use as a Context for Detail Rows, and create a Group Alias to use as a Context for Map Targets (creating Group Headers).

Detail Rows Tab   This tab of the Custom Results Mapping dialog allows you to create a mapping alias tied to either a document Context or a Group/Repeat alias Context. Use of the Detail Rows tab is optional.

DOM   A Document Object Model (DOM) is an XML document constructed as an object in a software program's memory. It provides standard methods for manipulating the object. In Composer, DOM is often synonymous with XML Document. DOMs are represented as hierarchical trees with a single root node.

DOM Context   The name of a DOM (Input, Output, Temp, etc.), or the name of a Repeat alias previously defined in the component. (The alias itself represents a DOM context, representing the nodepath hierarchy upstream of a given element.)

Execute SQL Action   Same as SQL Statement Action.

JDBC   A Sun trademark for the Java API for accessing relational database data. It is commonly assumed to mean Java Database Connectivity.

Map Target Tab   This tab of the Custom Results Mapping dialog is used to create target element names for each result set column and specify a target Context for each result set column.

Native Environment Pane   A pane in the JDBC Component Editor that simulates an actual SQL environment when you issue a query.

Query/Result Mapping Pane   (Same as the Native Environment Pane.) A pane in the JDBC Component Editor that includes three tabs: the SQL Statement tab, the Result Mapping tab, and the Results Text tab.

Result Mapping Tab   A tab in the Query/Result Mapping Pane that allows you to map the result of your database query to an XML document.

Result Text Tab   A tab in the Query/Result Mapping Pane that displays the actual data that was returned following the execution of the database query.

Row Target   The receiving element in a mapping operation is called the *row target*. It represents a specific position in the DOM tree of an XML file.

SQL Statement Action   Most commonly used to query an existing database and then map the result to an XML document.

SQL Statement Tab   A tab in the Query/Result Mapping Pane that allows you to write or build SQL commands.

SQLCODE   A global ECMAScript variable created by the execution of SQL statements. Contains a status code generated by the database engine.

SQLSTATE   A global ECMAScript variable created by the execution of SQL statements. Contains information generated by the database engine.

UPDATECOUNT   A global ECMAScript variable created by the execution of SQL statements. Contains a count of the number of rows changed by the database engine.

B Reserved Words

The following terms are reserved words in exteNd Composer for the JDBC Connect and should be avoided in any user created labels or objects.

- SQLCODE
- SQLSTATE
- UPDATECOUNT
- LASTSQL

Index

Symbols
% wildcard 27

A
action menu 44
action model 23
actions
    overview 23
    using basic and advanced 44
advanced actions 44
alias
    and custom result mapping 47
Allow SQL Transactions 15
And/Or logic in a WHERE clause 27
auto-commit 16

B
base64 encode 48
base64Decode() 24
base64Encode() 24
basic actions 44
batch actions (see SQL Batch) 40
BETWEEN...AND operator 27

C
code table map, creating 16
colons, special meaning in SQL action 40
commit 16
component
    creating new 17
component editor window 20
connection
    creating 13
    dirty 16
connection pool 14
    definition of 61
Constant and Expression Driven Connections 13
context 48, 53
creating SQL using the Wizard 24
Custom Mapping Settings 59
custom result mapping 46, 47
    definition of 61

D
custom script
    creating 16

Data Type 58
database-specific parameters 15
DB Params 15
declare group/repeat example 54
Declare Group/Repeat tab 53
    definition of 61
default result mapping 45
detail alias
    used as a context 48, 58
detail rows example 51
Detail Rows tab
    definition of 61
Discard Batch 42
document, used as a context 48, 58

E
ECMAScript
    in SQL Statements 40
ECMAScript functions, using 44
errors and SQL messages 44
example query 35
Execute as Prepared 24
Execute Batch 41
Execute SQL action
    definition of 61
executing the SQL statement 37
Expressions 58

G
group alias
    creating 53
    used as a context 48, 59

I
Id 58

W

WHERE Clauses
    filtering within the wizard 27
WHERE clauses
    And/Or logic 27
wildcards 27

X

XML template
    creating 16