Building database applications with WebSphere and DB2

Presented by DB2 Developer Domain

`http://www7b.software.ibm.com/dmdd/`

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

Should I take this tutorial?

This tutorial is for developers who want to create a Java-based Web service to access or update a DB2 database. It demonstrates the following concepts through the use of a simple commodity trading system:

- How to use WebSphere Studio to work with the DB2 database
- How to create and use DB2 user-defined functions and Java stored procedures
- How to create a Java application that interacts with the DB2 database
- How to turn that Java application into a Web service

WebSphere Studio includes WebSphere Studio Application Developer and WebSphere Studio Site Developer configurations, which ease the process of creating Java-based Web services and other Java applications. This tutorial uses the term WebSphere Studio to encompass both products. The screenshots and examples were created with Application Developer.

Readers should have an basic understanding of database concepts and SQL. They should also be familiar with Java. An understanding of Web services is also helpful, but not required.

What is this tutorial about?

One common use for Web services is the enabling of remote access to DB2 database information. Using parts of a simple commodity trading system, you will learn how to select from, insert into, and update a DB2 database from WebSphere Studio and from a Java application. You will then convert this application to a Web service with the help of WebSphere Studio.

The commodity trading system tracks available prices and quantities of the fictional commodity Zip-Air (when you really need fresh air in a hurry) in a DB2 database. The Web service adjusts available quantities when the buyer uses the Web service to make a purchase. Sellers can use the Web service (through a stored procedure) to set the minimum and maximum quantities they are willing or able to sell and their price. The available price at any given time is directly related to the quantity the buyer is willing to purchase. In general, the more a buyer is willing to purchase at one time, the better the price, as determined by a user-defined function.

The tutorial covers the following:

- Connecting to a database using WebSphere Studio
- Designing and creating database objects such as tables within WebSphere Studio
- Using WebSphere Studio tools to create and execute SQL statement objects
- Creating, registering, and using user-defined functions
- Accessing SQL statements and stored procedures from a Java application
- Using DB2s Procedure Builder to create a stored procedure out of a Java class
- Creating and testing a Web service from the newly built Java application

Tools

The following tools should be installed and tested prior to beginning the tutorial:

- **DB2 Universal Database**

  A *free trial version* of the single-user version of the database, the DB2 Universal Database Personal Edition, is available for download. This tutorial uses version 7.2, but the principles should be the same for version 8 when the public release is available.
- **WebSphere Studio Application Developer or WebSphere Studio Site Developer**

  These integrated development environments enable developers to connect to a database to examine or modify it, and also to create Java applications and automate the creation of Web services. Free trial versions of both are available for download. Download *Application Developer for Windows* or *Linux*, or *Site Developer for Windows*.

You can also download the source code for this tutorial. It is available in the file *source.zip*.

About the author

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Florida, and is the author of three books on Web development, including *Java and XML From Scratch* (Que) and the upcoming *Primer Plus XML Programming* (Sams). He loves to hear from readers and can be reached at *nicholas@nicholaschase.com*.

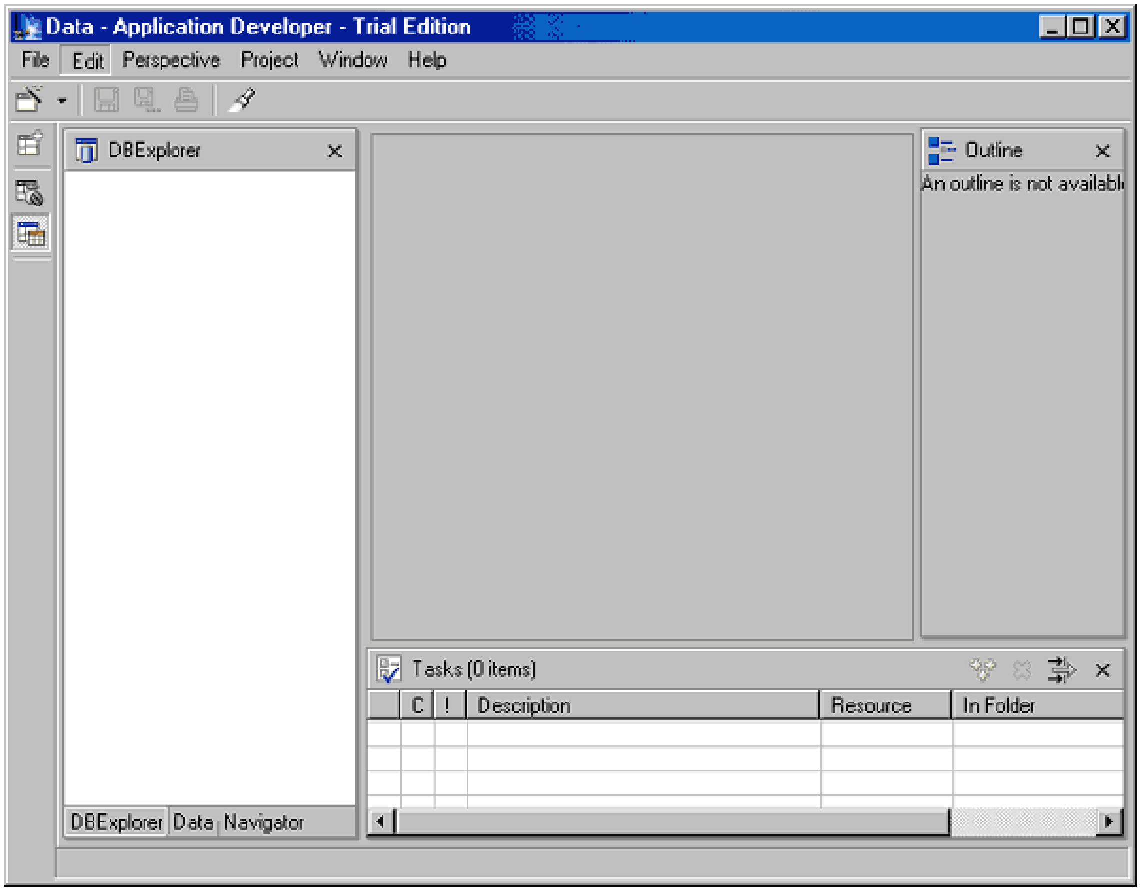
Section 2. Creating the database structure

WebSphere Studio's database tools

Organization is half the battle in a development project, and WebSphere Studio assists by organizing resources into projects that reside in individual folders within the workspace. Developers work on those resources within *perspectives*, such as the Data, Java, and Web perspectives, which include specialized views and editors for particular development tasks. This section shows you how to use WebSphere Studio to work with a DB2 database.

The Data perspective includes views such as the DBExplorer, which provides a "live" look at a remote database, the Data view, which provides a look at the "internal" version, and editors such as the SQL Statement Object editor. These tools will be invaluable in laying the groundwork for the project.

To use the database tools, open the Data perspective by choosing **Perspective=> Open=> Other** and selecting **Data**. The left-hand pane contains tabs for three windows: the DBExplorer, to manage database connections; Data, to view the actual data structures; and the Navigator, to view the project objects. The Navigator view is available in most perspectives.

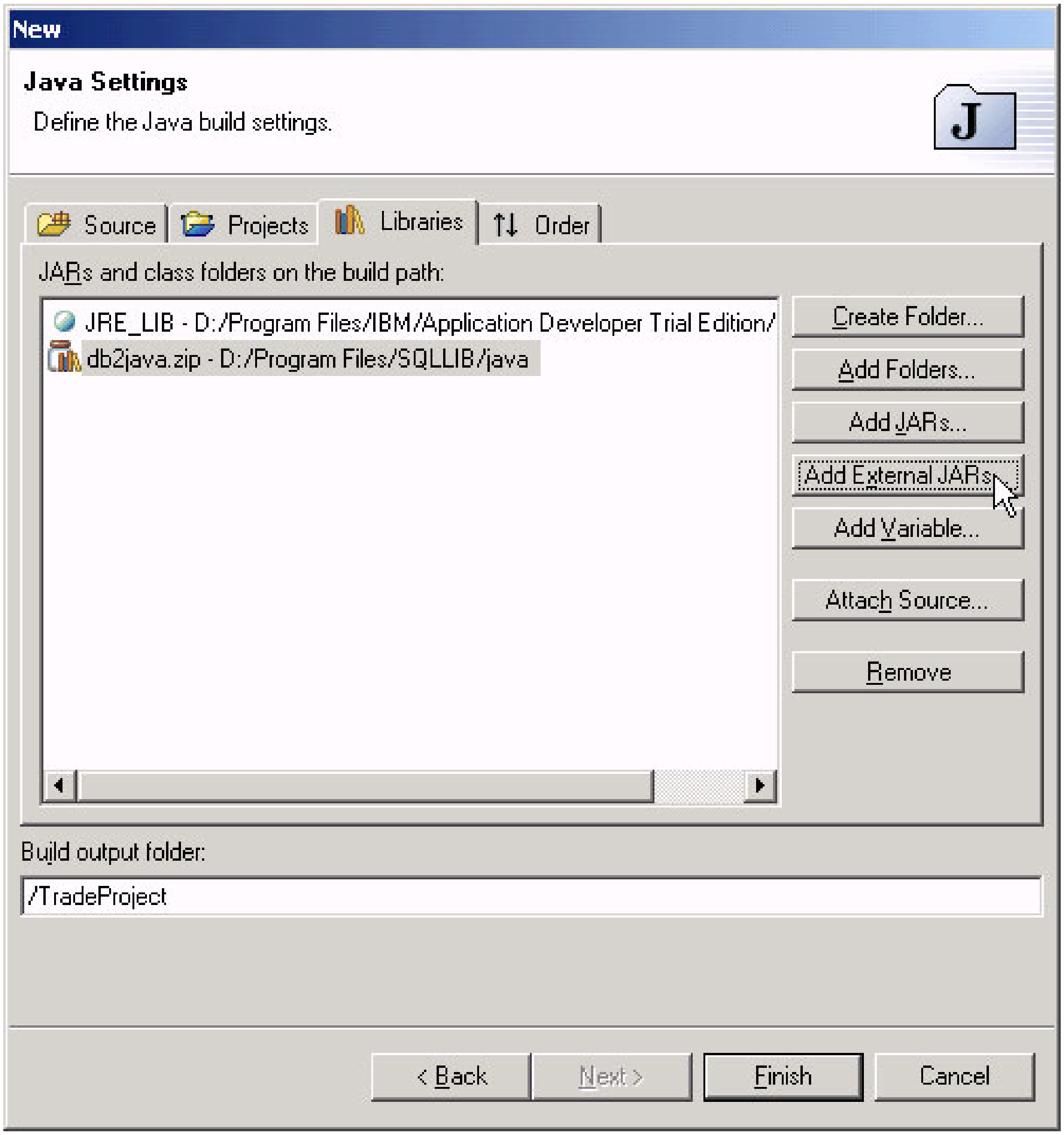

Data - Application Developer - Trial Edition
File Edit Perspective Project Window Help
DBExplorer
Outline
An outline is not availabl
Tasks (0 items)
C ! Description Resource In Folder
DBExplorer Data Navigator

Create a project

At its most basic level, the Web service needs a DB2 table in which to store the available offers and their associated prices. This section introduces WebSphere Studio tools for navigating existing DB2 resources and for creating new resources such as the TRADE schema, which will host all of the project resources, and the QtyAvailable table.

All application resources need to be part of a project, so start by creating a Java project. Choose **File=> New=> Project**, and then click **Java** in the left-hand pane and **Java Project** in the right-hand pane. Enter a name for the project in the next window. This tutorial assumes a project name of `TradeProject`.

Click **Next** to add any required libraries. Selecting data from a Java application on page 28 requires the DB2 JDBC driver classes, so click **Add External Jars** and add the `db2java.zip` file, located in `{DB2 INSTALL}/SQLLIB/java`.

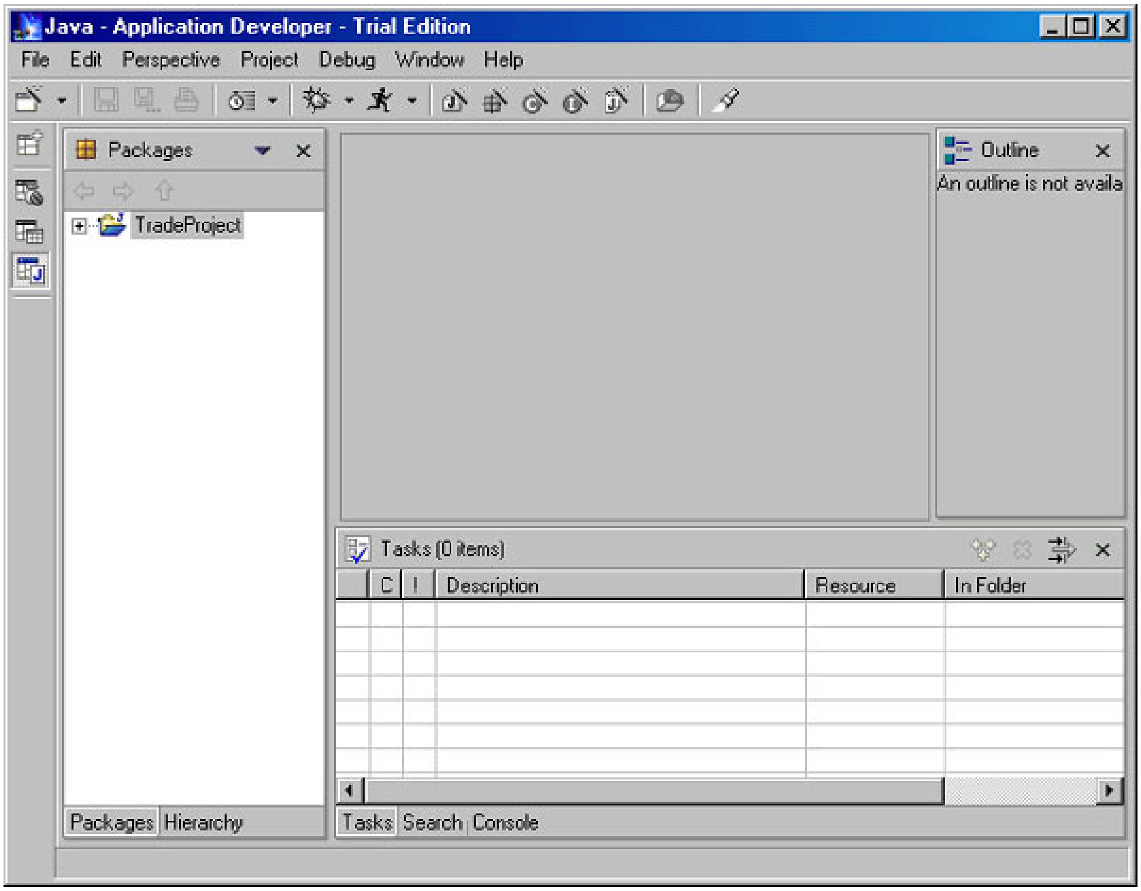


JARs and class folders on the build path:

- JRE_LIB - D:/Program Files/IBM/Application Developer Trial Edition/
- db2java.zip - D:/Program Files/SQLLIB/java

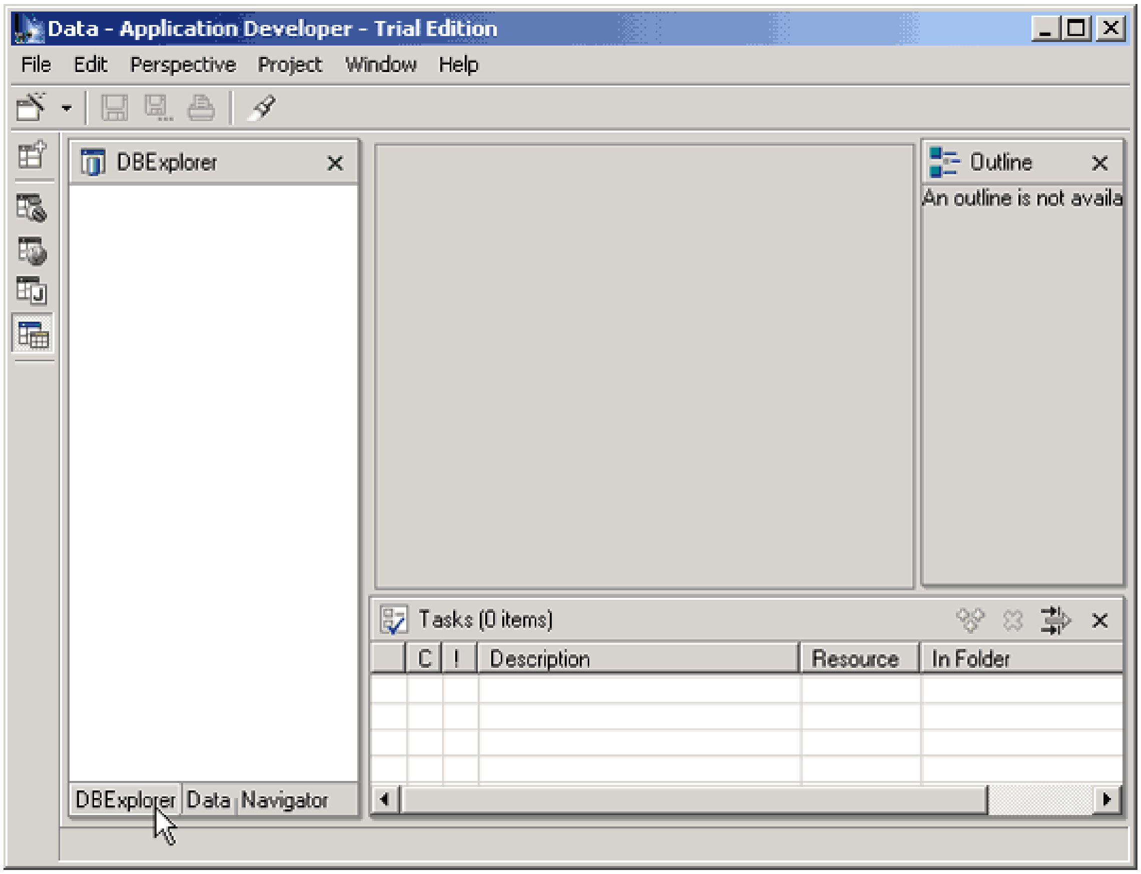
Build output folder: /TradeProject

Click **Finish** to create the project. WebSphere Studio creates the project and opens it in the Java perspective, which includes the views and tools for building Java applications.


Java - Application Developer - Trial Edition
File Edit Perspective Project Debug Window Help
Packages
TradeProject
Outline
An outline is not availa
Tasks (0 items)
C ! Description Resource In Folder
Packages Hierarchy
Tasks Search Console

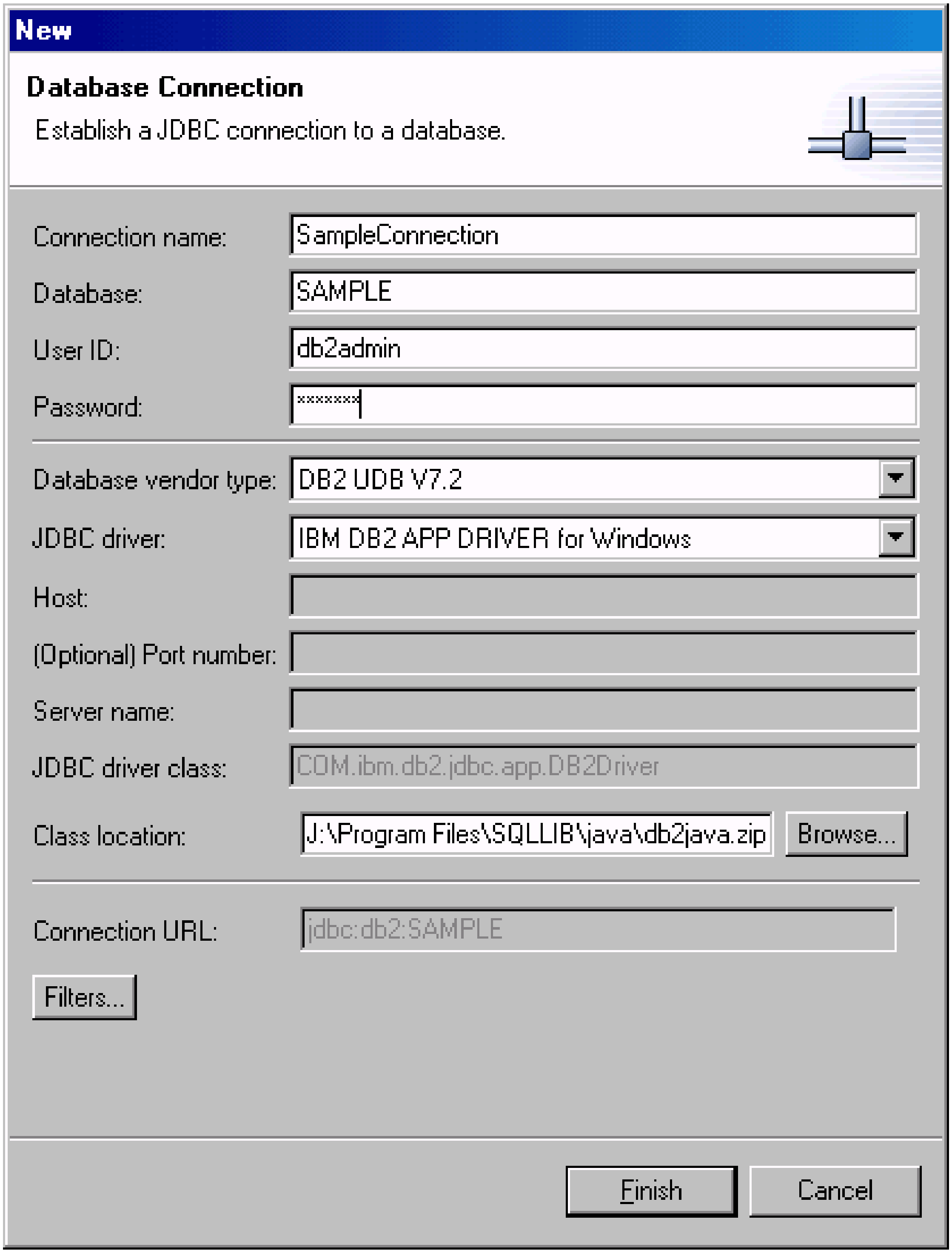
Create a connection

Working with DB2 requires a database connection. Open the Data perspective by choosing **Perspective=> Open=> Other=> Data=> OK** and choose the DBExplorer pane by clicking its tab in the left-hand window.



Right-click the DBExplorer pane and choose **New Connection**.

In the Database Connection window, enter the connection information for your database. This example shows connection information for the SAMPLE database that comes with DB2 and creates a connection called `SampleConnection`.



Database Connection

Establish a JDBC connection to a database.

Connection name: SampleConnection

Database: SAMPLE

User ID: db2admin

Password: *******

Database vendor type: DB2 UDB V7.2

JDBC driver: IBM DB2 APP DRIVER for Windows

Host:

(Optional) Port number:

Server name:

JDBC driver class: COM.ibm.db2.jdbc.app.DB2Driver

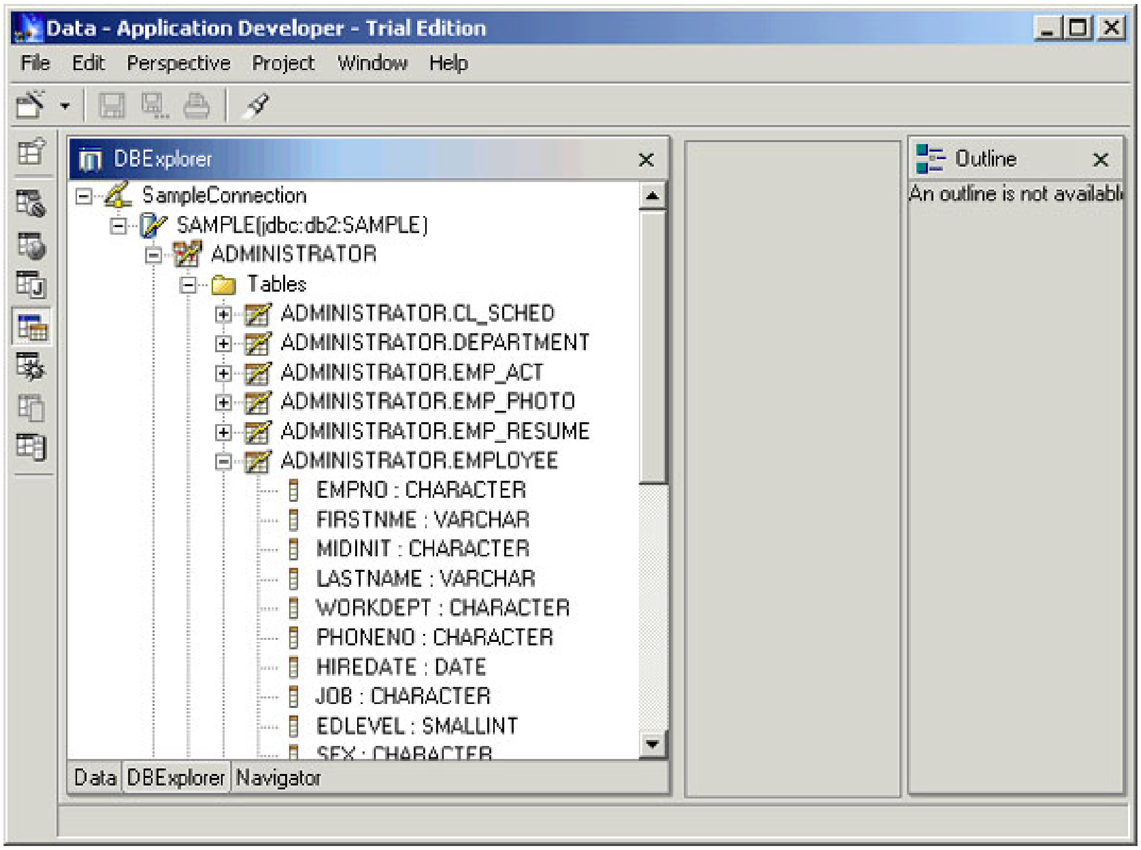
Class location: J:\Program Files\SQLLIB\java\db2java.zip

Connection URL: jdbc:db2:SAMPLE

The information for a DB2 database is pre-populated; just enter the database name, username, and password. Enter the information for your database and click **Finish** to create the connection.

Explore the database

The connection provides a way to examine the objects within the database. To expand or contract nodes within the tree, click the plus (+) or minus (-) signs to the left of the node.

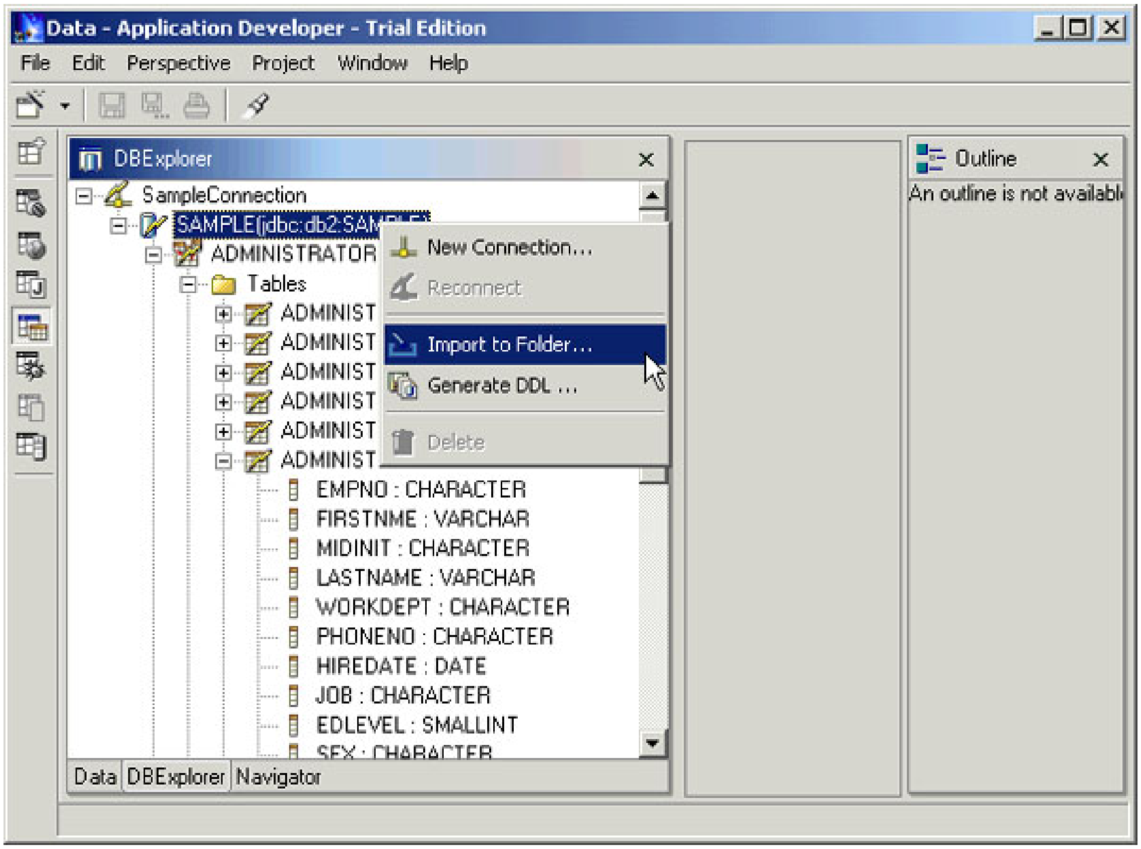


These are the live objects within the database, but the structure can't be worked on or changed until it's imported into the project.

Import the schema

All work on the database is done in the Data view, but if you click the Data tab, you'll notice that there's no information listed under the project name even though a connection to the database exists. To make the structure of the database available to WebSphere Studio, import it into the project folder.

In the DBExplorer, right-click the database and choose **Import to Folder**.



In the Import to Folder window, click **Browse** and navigate to the project folder, then click **OK**. (You can also create a new folder within the project first, if desired.) Click **Finish** to save the database information to the project. This process creates a copy of the database structure within the workbench.

Create a new schema

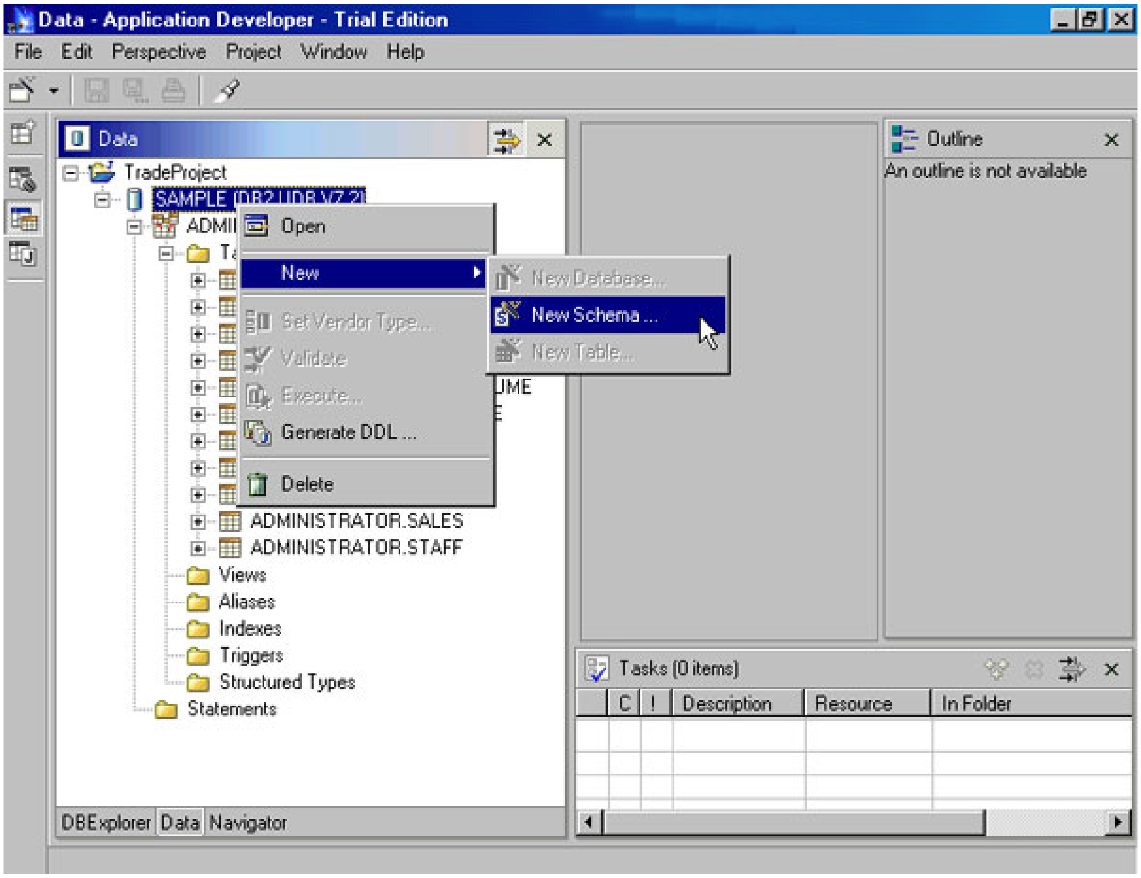
DB2 organizes information into schemas, where a single schema contains all of the database resources for a project, or for an aspect of a project. Schemas often correspond to individual DB2 users and help to prevent name conflicts. For example, the objects automatically created with the SAMPLE database belong to the ADMINISTRATOR schema, and can be identified using "dot" notation, so the full name of the EMPLOYEE table is actually:

```
ADMINISTRATOR.EMPLOYEE
```

Another schema can have an EMPLOYEE table without a conflict, because it can be referenced by the schema name, as in:

```
JOE.EMPLOYEE
```

For the trading service project, create a new schema called `TRADE` to hold the database objects. Click the Data tab to see the Data view. Right-click the database and choose **New=> New Schema**.



Enter a name for the schema (such as `TRADE`) and click **Finish**.

Create the new table

With the new schema created, it's time to create the new table. Expand the TRADE schema by clicking the plus sign (+), and right-click the Tables folder. Choose **New=> New Table**. Notice that WebSphere Studio automatically fills in the schema info. Name the table `QtyAvailable` and add a comment explaining the new table. Click **Next**.

The Table Columns dialog box enables you to add columns to the table. Start by changing the name of the default column, *col1*, to `offerid`. To add a second column, click the Add Another button under the left-hand pane. For each column, set the appropriate type. The columns for the QtyAvailable table are as follows:

```
CREATE TABLE TRADE.QTYAVAILABLE
   (OFFERID INTEGER NOT NULL,
    SELLERID VARCHAR(10) NOT NULL,
    UNITPRICE DOUBLE NOT NULL,
    MINQUANTITY INTEGER NOT NULL,
    MAXQUANTITY INTEGER NOT NULL);
```

To make sure each column is created with a `NOT NULL` constraint, make sure the Nullable box is left unchecked.

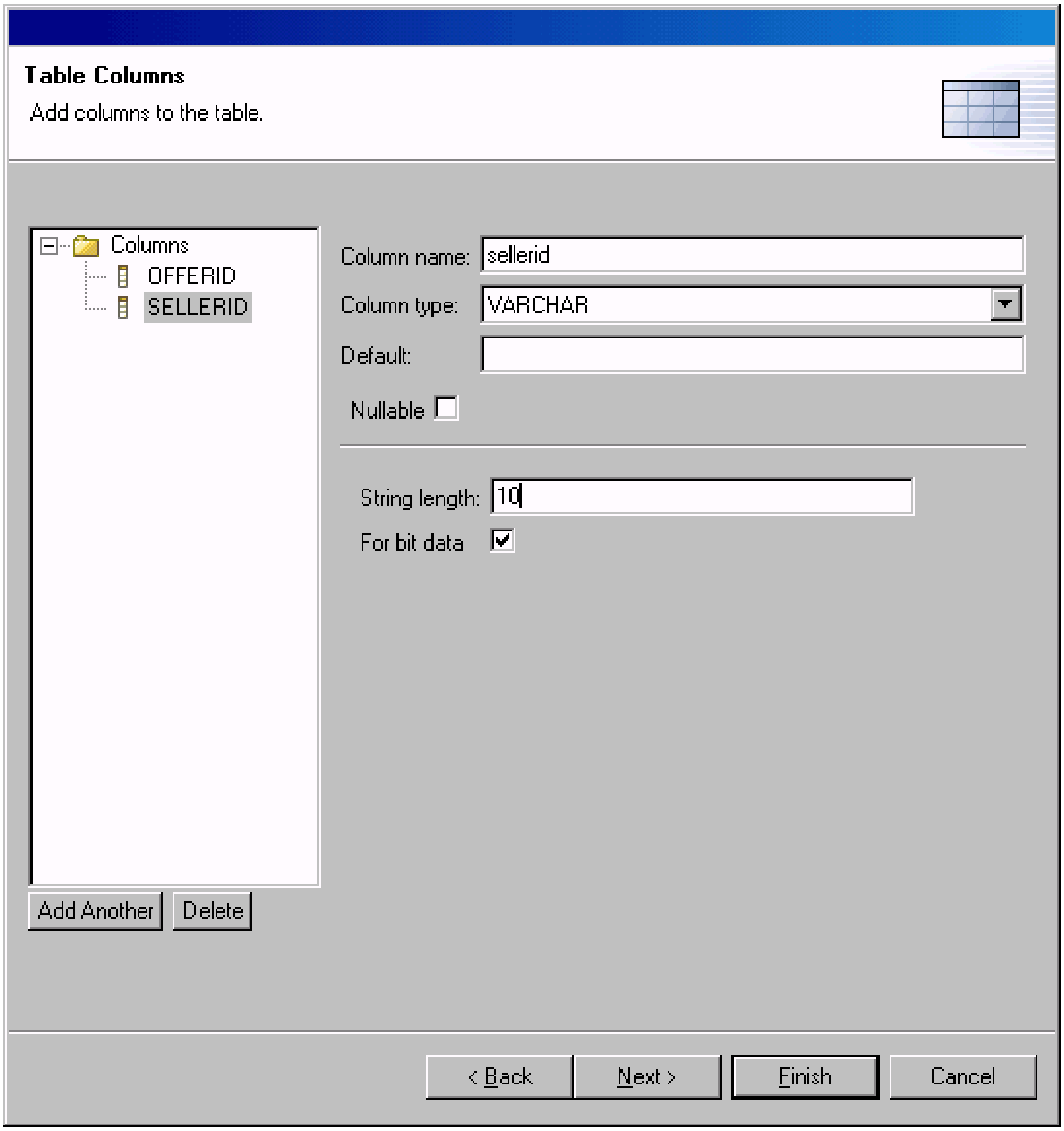


Table Columns

Add columns to the table.

Column name: sellerid

Column type: VARCHAR

Default:

Nullable ☐

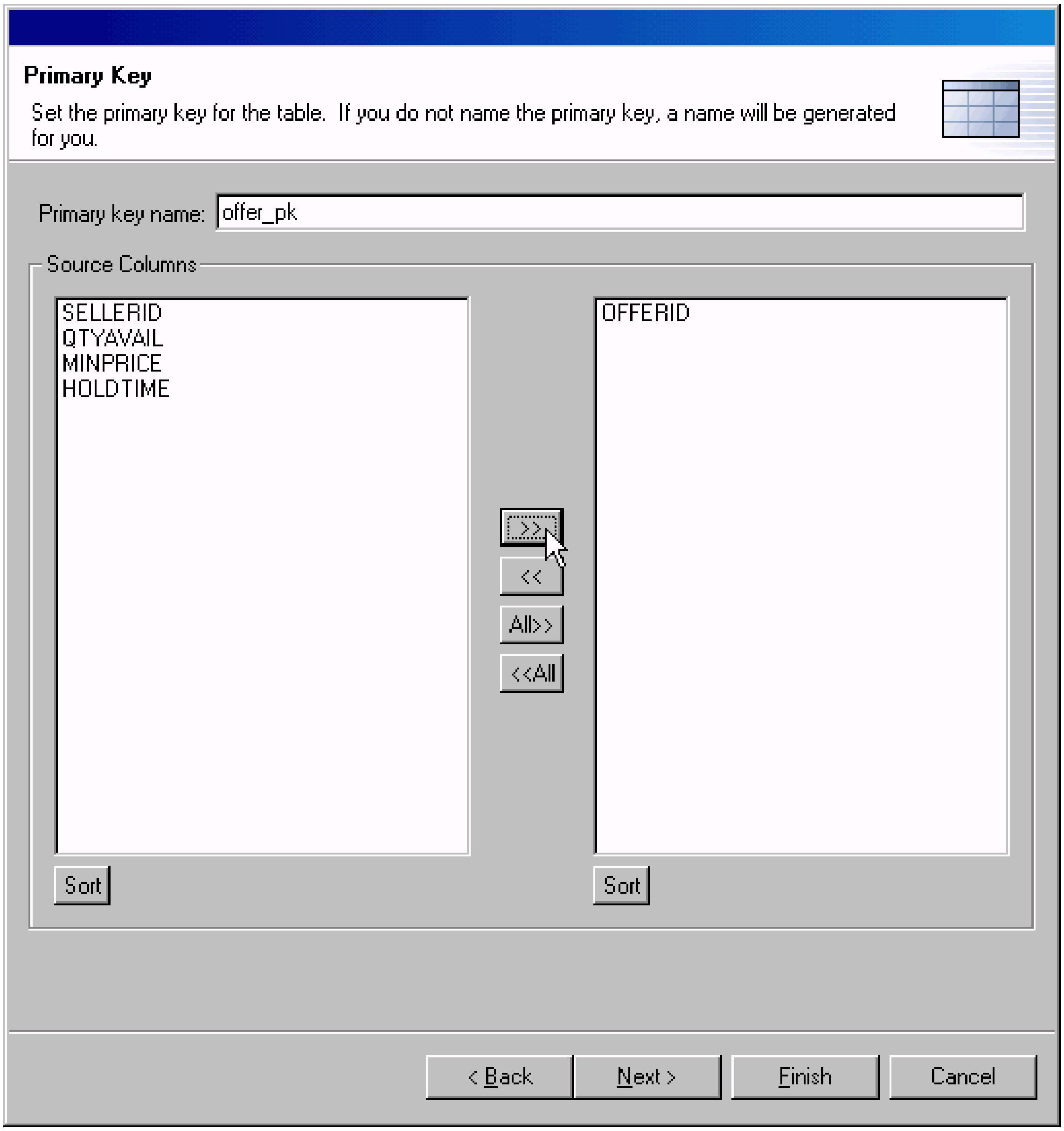
String length: 10

For bit data ☑

After adding all of the columns, click **Next**.

Add a primary key to the table

The table creation process also enables the creation of a primary key for the table. Give the key an appropriate name, such as `offer_pk`, and click the offerid column in the left pane. Click the >> button to move it to the right pane.



Primary Key

Set the primary key for the table. If you do not name the primary key, a name will be generated for you.

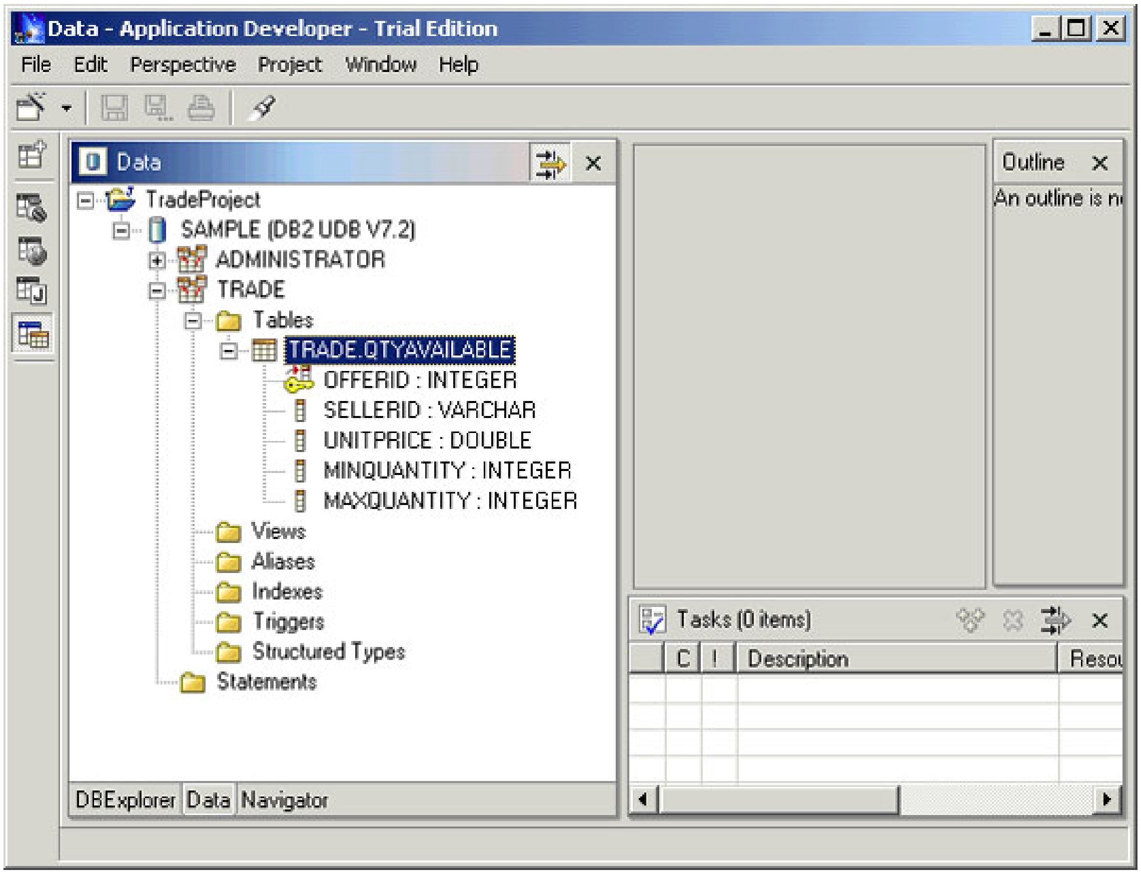
Primary key name: offer_pk

Source Columns

SELLERID
QTYAVAIL
MINPRICE
HOLDTIME

OFFERID

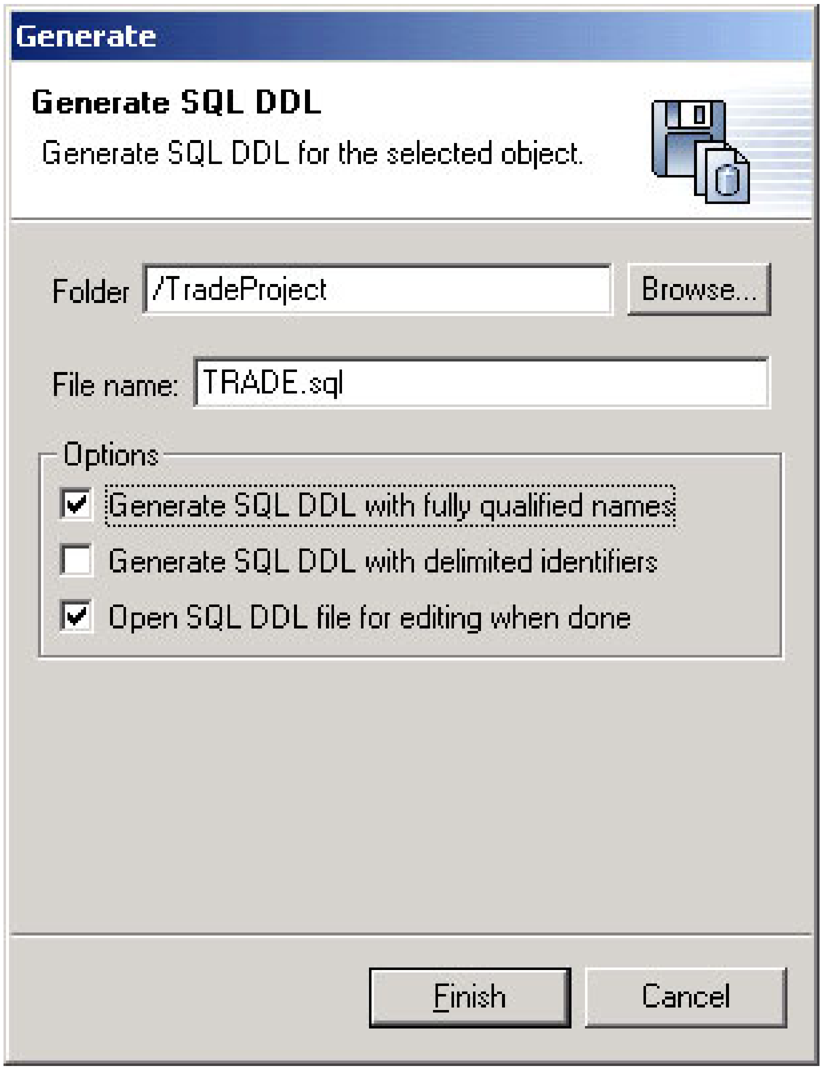
Clicking **Next** enables the creation of foreign keys, but this table doesn't have any, so click **Finish**.



Migrating changes back to the database

At this point, the new table is clearly obvious in the Data view, but doesn't appear in the DBExplorer view. In fact, a look at any of the DB2 tools will show that the table doesn't exist in the database at all. Instead, it just exists in the local copy of the schema within WebSphere Studio. This separation enables you to create tables without affecting the main DB2 database. However, the table needs to be migrated back to the database in order to run any queries and use the table.

Fortunately, this is a straightforward process. Right-click the TRADE schema (as opposed to the table) in the Data view and select **Create DDL**.



Generate SQL DDL

Generate SQL DDL for the selected object.

Folder: /TradeProject

File name: TRADE.sql

Options
☑ Generate SQL DDL with fully qualified names
☐ Generate SQL DDL with delimited identifiers
☑ Open SQL DDL file for editing when done

Click **Generate SQL DDL with fully qualified names** to create a script that identifies the schema for each object. Click **Finish** to create the actual `.sql` script.

Options for running the script

You have several options for running the script against the DB2 database. The first is to use the Command Center:

1. Choose **Start=> Programs=> IBM DB2=> Command Center**.
2. Click the Interactive tab and click **Execute** to connect to the database.
3. Copy and paste each command into the Command field and click **Execute**.

A more streamlined option is to use the Script Center.

1. Choose **Start=> Programs=> IBM DB2=> Control Center**.
2. Choose **Tools=> Script Center**.
3. Choose **Script=> Import**.
4. Navigate to the appropriate file and click **OK**.
5. Right-click the script and choose **Run Now**.

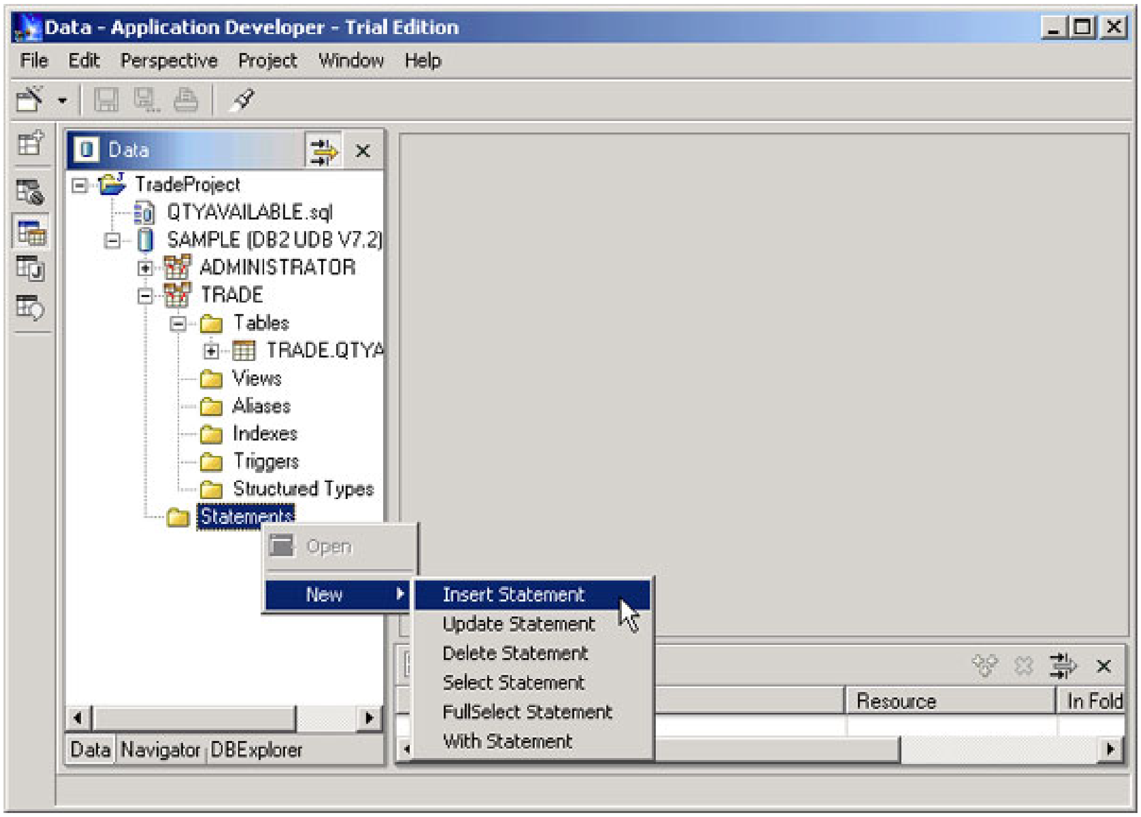
Now that the table exists in the database, the next section, Using SQL statements on page 18 , explains how to access and manipulate the data within it.

Section 3. Using SQL statements

Create a new INSERT statement object

At this point, the table exists in DB2, but it's empty. Normally, you would populate the table using SQL `INSERT` statements. WebSphere Studio helps to automate the process of creating, organizing, and executing SQL statements. This section familiarizes you with WebSphere Studio statement objects.

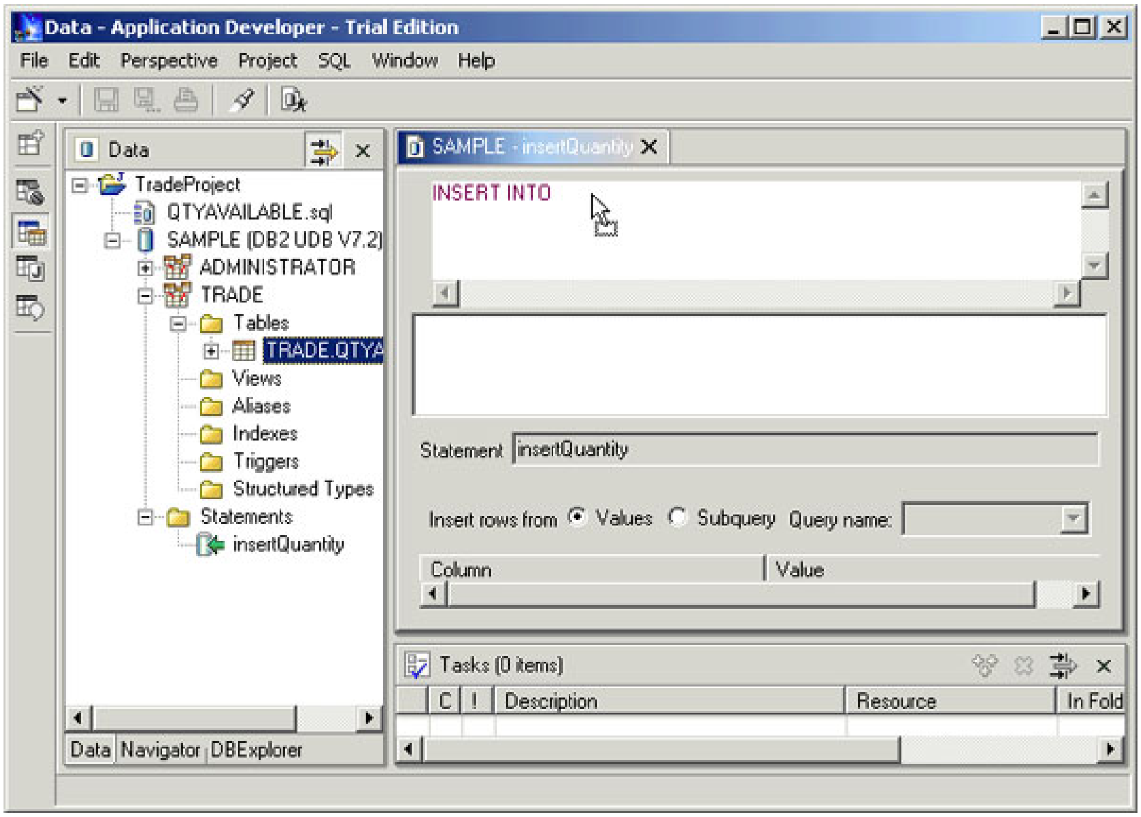
To populate the new table with available quantities of Zip-Air, create a new `INSERT` statement by right-clicking the Statements folder in the Data view and choosing **New=> Insert Statement**. Give the statement a descriptive name, such as `InsertQuantity`.



This action creates a new `INSERT` statement object and opens the Statement editor. You can simply type the statement in the editor, but WebSphere Studio provides GUI tools to make this task a bit easier and to prevent typographical mistakes, as seen in the next panel.

Set the appropriate table

Creating the object opens the statement editor with a skeleton statement already provided. In the case of an `INSERT` statement, that skeleton is simply `INSERT INTO`. Add the table to the statement by dragging it over from the Data view.



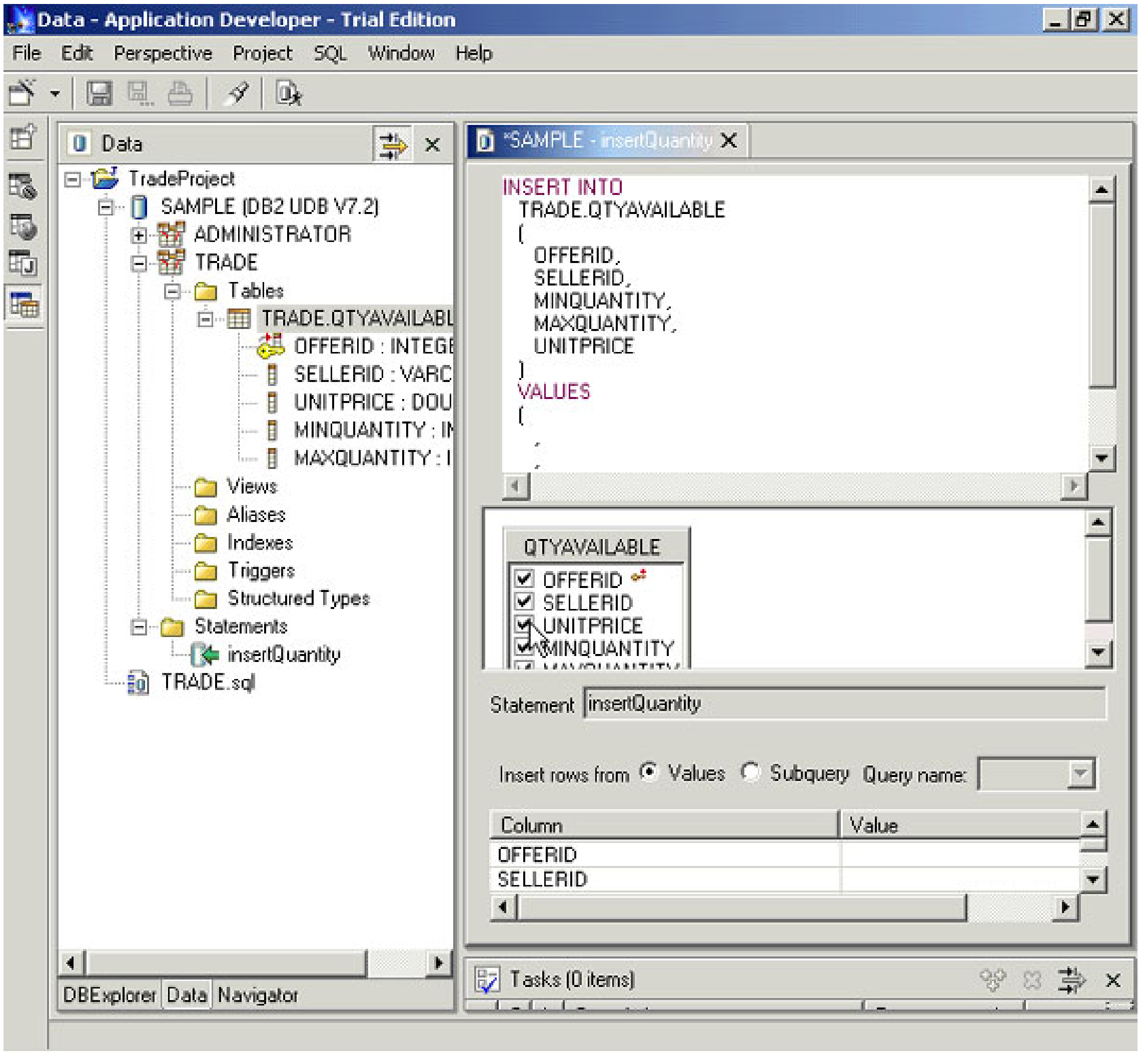
```
INSERT INTO
```

Statement: insertQuantity

Insert rows from ● Values ○ Subquery Query name:

Choose columns

Next add columns to the statement. The middle pane shows the available columns with check boxes. Checking each appropriate column adds it to the statement in the upper pane, but beware: Columns are added in the order in which they are checked. As a result, if the `UNITPRICE` column is checked last, it will appear last in the statement.



```
INSERT INTO
  TRADE.QTYAVAILABLE
  (
    OFFERID,
    SELLERID,
    MINQUANTITY,
    MAXQUANTITY,
    UNITPRICE
  )
  VALUES
  (
    ,
    ,
```

Statement: insertQuantity

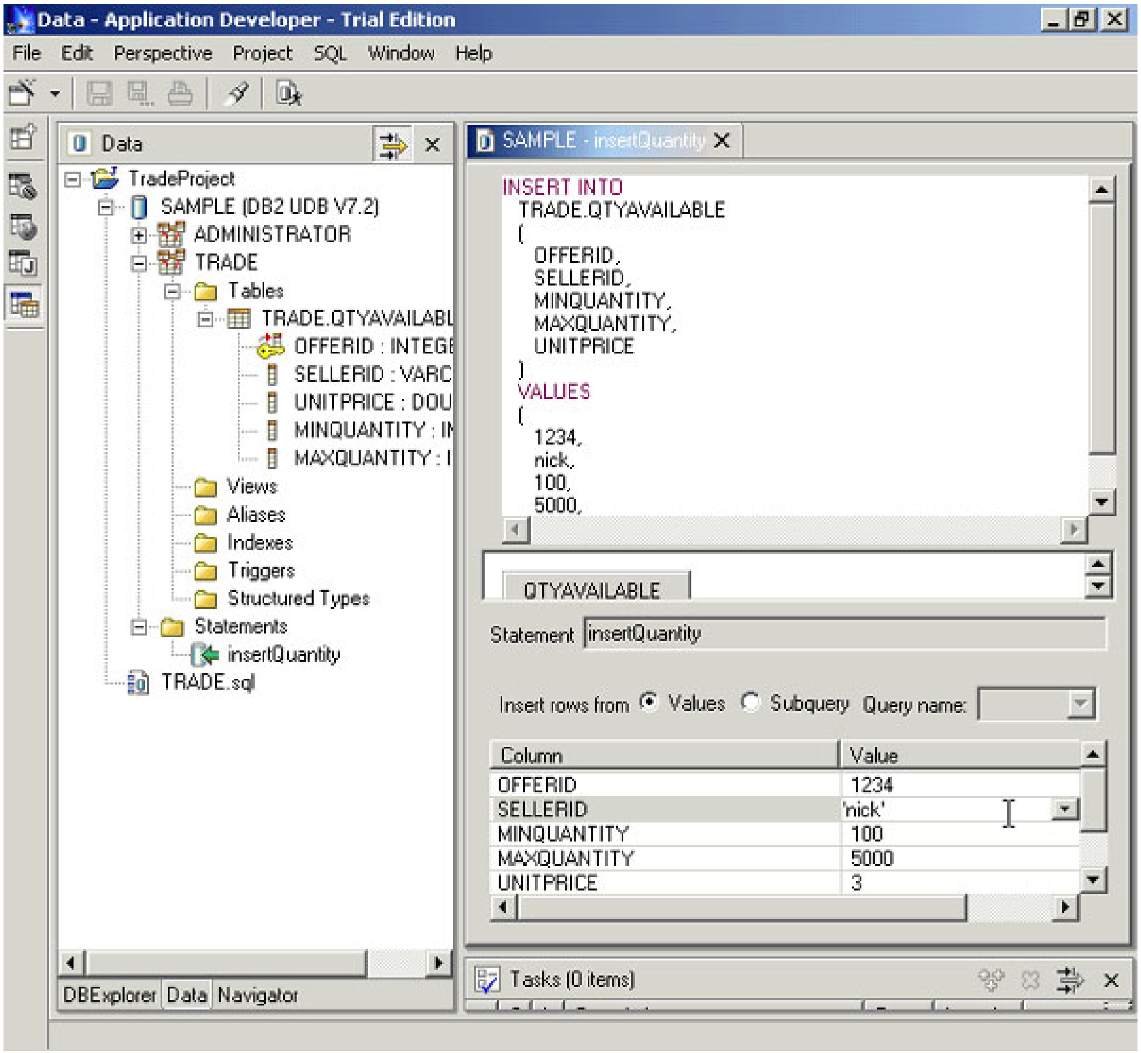
Insert rows from ● Values ○ Subquery Query name:

Column	Value
OFFERID	
SELLERID	

Add values

The last step is to add values to the statement, either by typing them directly into the upper pane, or by adding them through the editor. The editor enables the use of specific values in the Column/Value pane at the bottom, or by adding a subquery. To use a subquery, click the Subquery radio button and choose the appropriate Query name from the pulldown menu.

Add values directly to the appropriate columns. Because this is a fictional example, the actual values are not crucial, but try to leave some room between the `minquantity` and `maxquantity` values. The Column/Value pane can be useful because it also assists in the creation of expressions, as seen in Building an expression, part 1 on page 24 .



```
INSERT INTO
  TRADE.QTYAVAILABLE
  (
    OFFERID,
    SELLERID,
    MINQUANTITY,
    MAXQUANTITY,
    UNITPRICE
  )
  VALUES
  (
    1234,
    nick,
    100,
    5000,
```

Statement insertQuantity

Insert rows from ● Values ○ Subquery Query name:

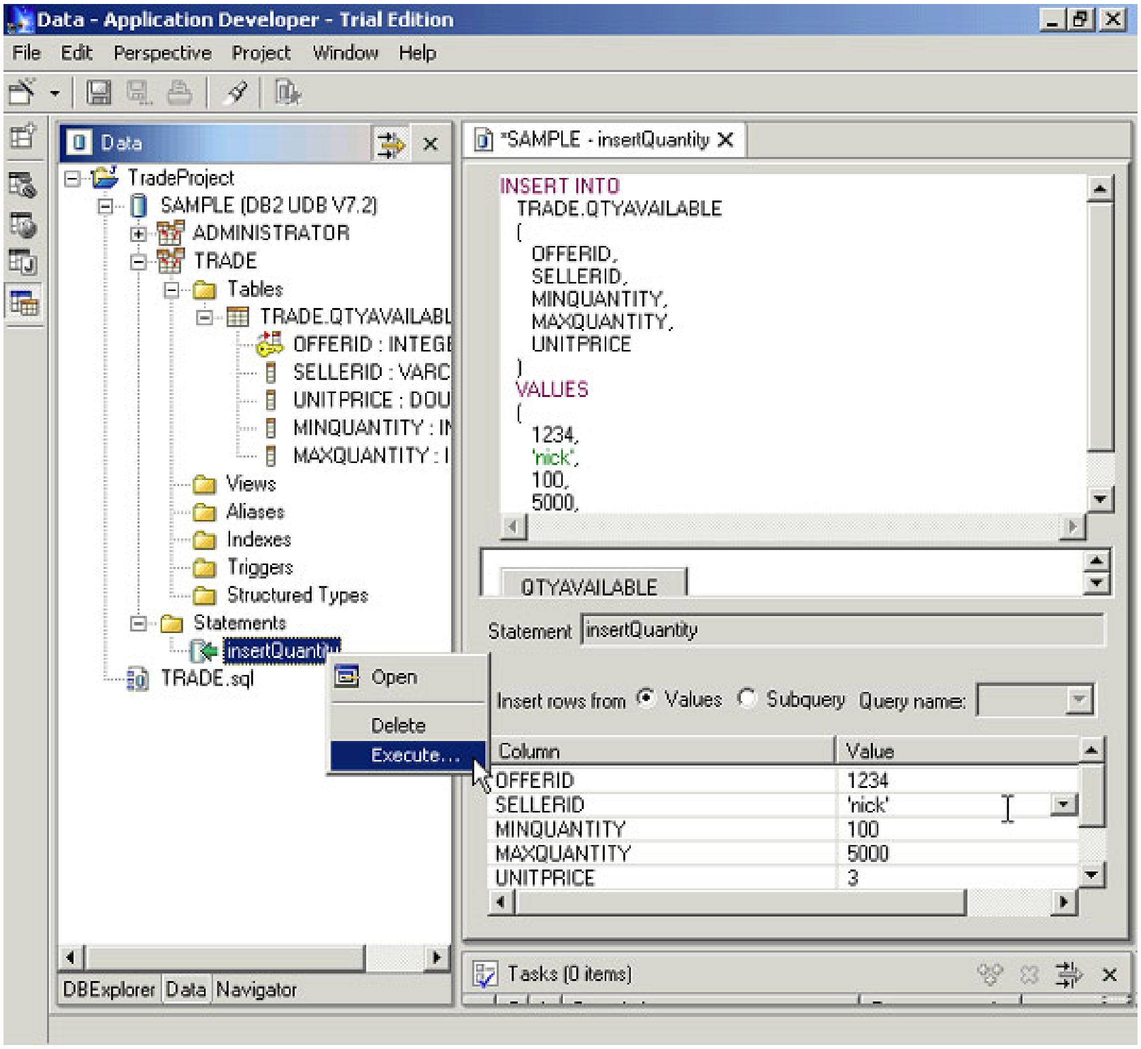
Column	Value
OFFERID	1234
SELLERID	'nick'
MINQUANTITY	100
MAXQUANTITY	5000
UNITPRICE	3

When adding a text value such as the `SELLERID`, make sure to enclose the value in quotes so the generated statement is correct.

Adding the values in the lower pane automatically adds them to the text pane at the top, and vice versa. Save the changes to the statement object by clicking **File=> Save SAMPLE - insertQuantity**.

Execute the statement object

To execute the statement, right-click the statement in the Statements folder and choose **Execute**.



```
INSERT INTO
  TRADE.QTYAVAILABLE
  (
    OFFERID,
    SELLERID,
    MINQUANTITY,
    MAXQUANTITY,
    UNITPRICE
  )
  VALUES
  (
    1234,
    'nick',
    100,
    5000,
```

Statement: insertQuantity

Insert rows from ● Values ○ Subquery Query name:

Column	Value
OFFERID	1234
SELLERID	'nick'
MINQUANTITY	100
MAXQUANTITY	5000
UNITPRICE	3

In the new window that appears, click **Execute** to execute the statement. After a few moments, you should see a message that says:

```
Query results: Statement execution successful.  No results to display.
```

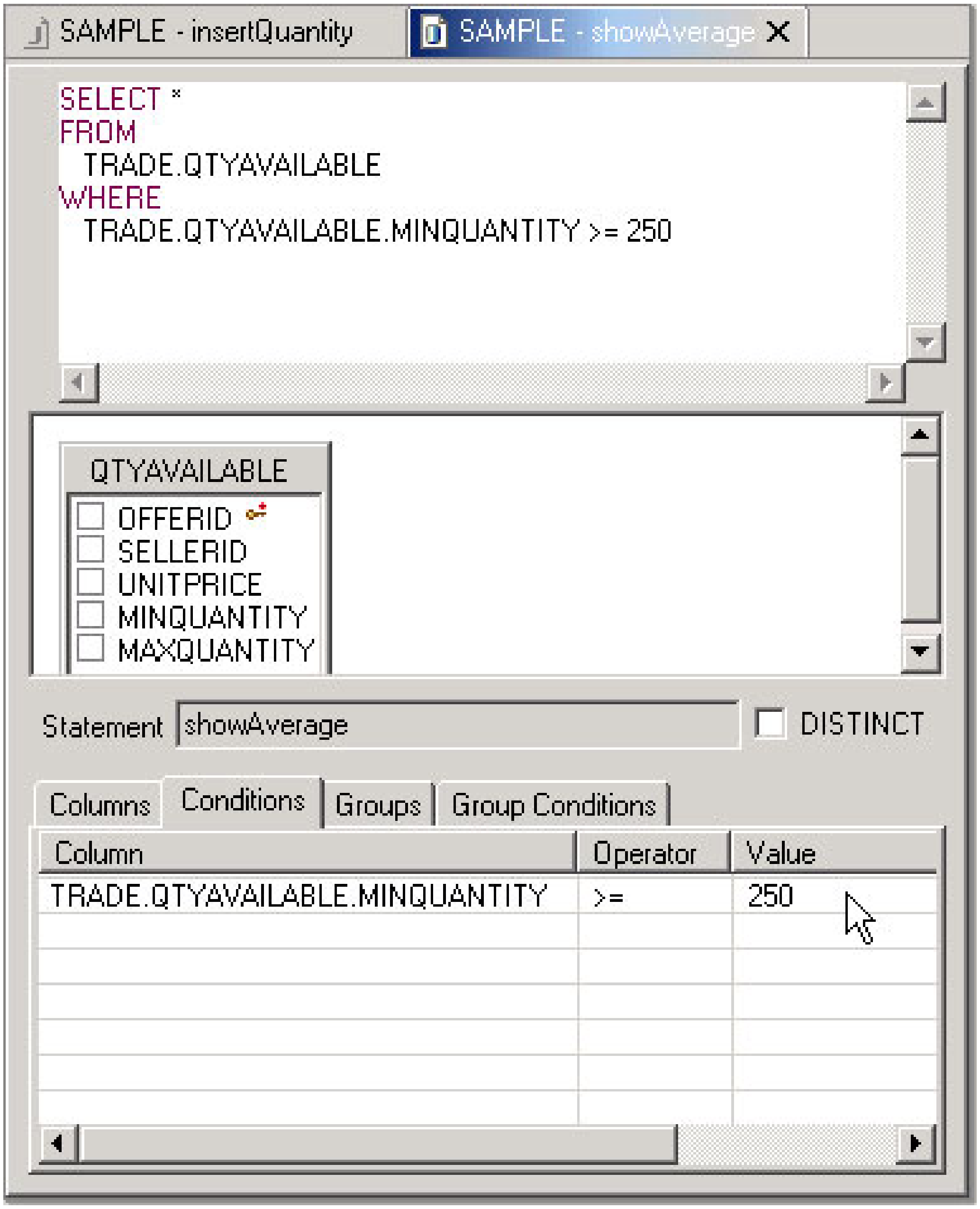
This action executes against DB2 itself, as opposed to the local copy, so running a `SELECT` statement in a tool such as the Command Line Processor should show the new offer.

Add several more entries by changing the values in the `INSERT` statement object, saving it, and executing it.

Create a select statement

Now let's build the statement that lets us retrieve the data we inserted. Create a new `SELECT` statement by right-clicking the Statements folder and choosing **New=> Select Statement**. Give the new statement a name, such as `showAverage`, and drag the QtyAvailable table into the text pane to add it to the query. By default, all columns will be selected.

To add a `WHERE` clause to the statement, click the Conditions tab and add a column, operator, and value.

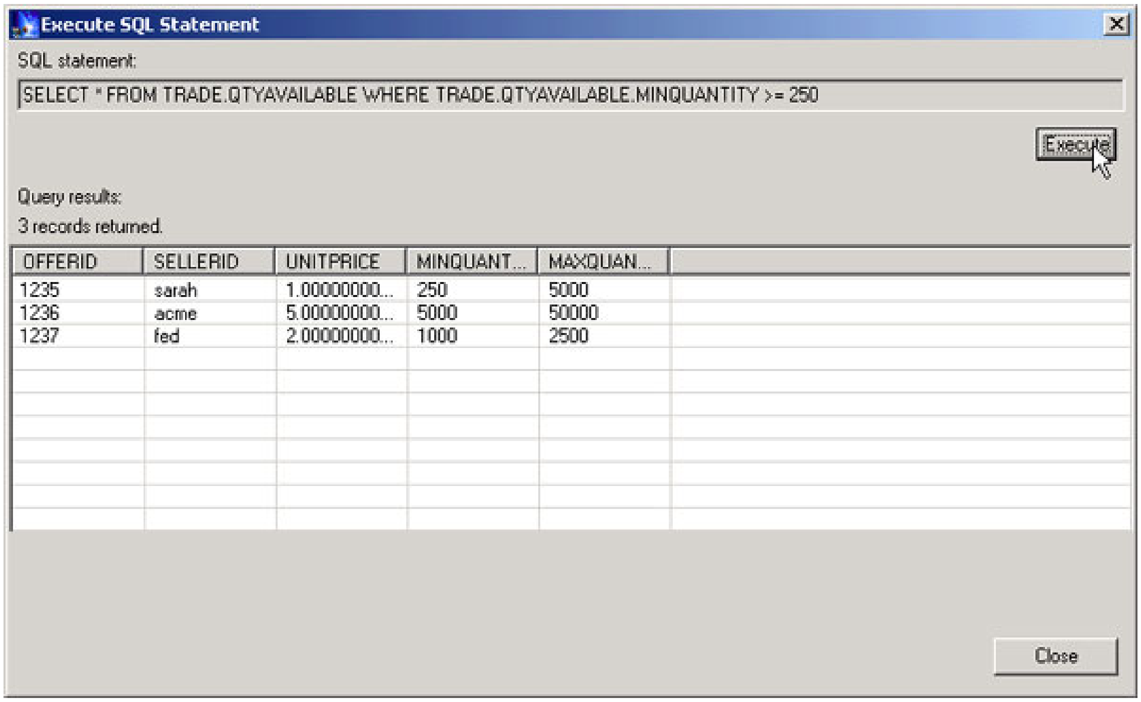


```
SELECT *
FROM
  TRADE.QTYAVAILABLE
WHERE
  TRADE.QTYAVAILABLE.MINQUANTITY >= 250
```

Statement: showAverage ☐ DISTINCT

Column	Operator	Value
TRADE.QTYAVAILABLE.MINQUANTITY	>=	250

Execute the statement to see the results of the query.



SQL statement:

```
SELECT * FROM TRADE.QTYAVAILABLE WHERE TRADE.QTYAVAILABLE.MINQUANTITY >= 250
```

Query results:

3 records returned.

OFFERID	SELLERID	UNITPRICE	MINQUANT...	MAXQUAN...
1235	sarah	1.00000000...	250	5000
1236	acme	5.00000000...	5000	50000
1237	fed	2.00000000...	1000	2500

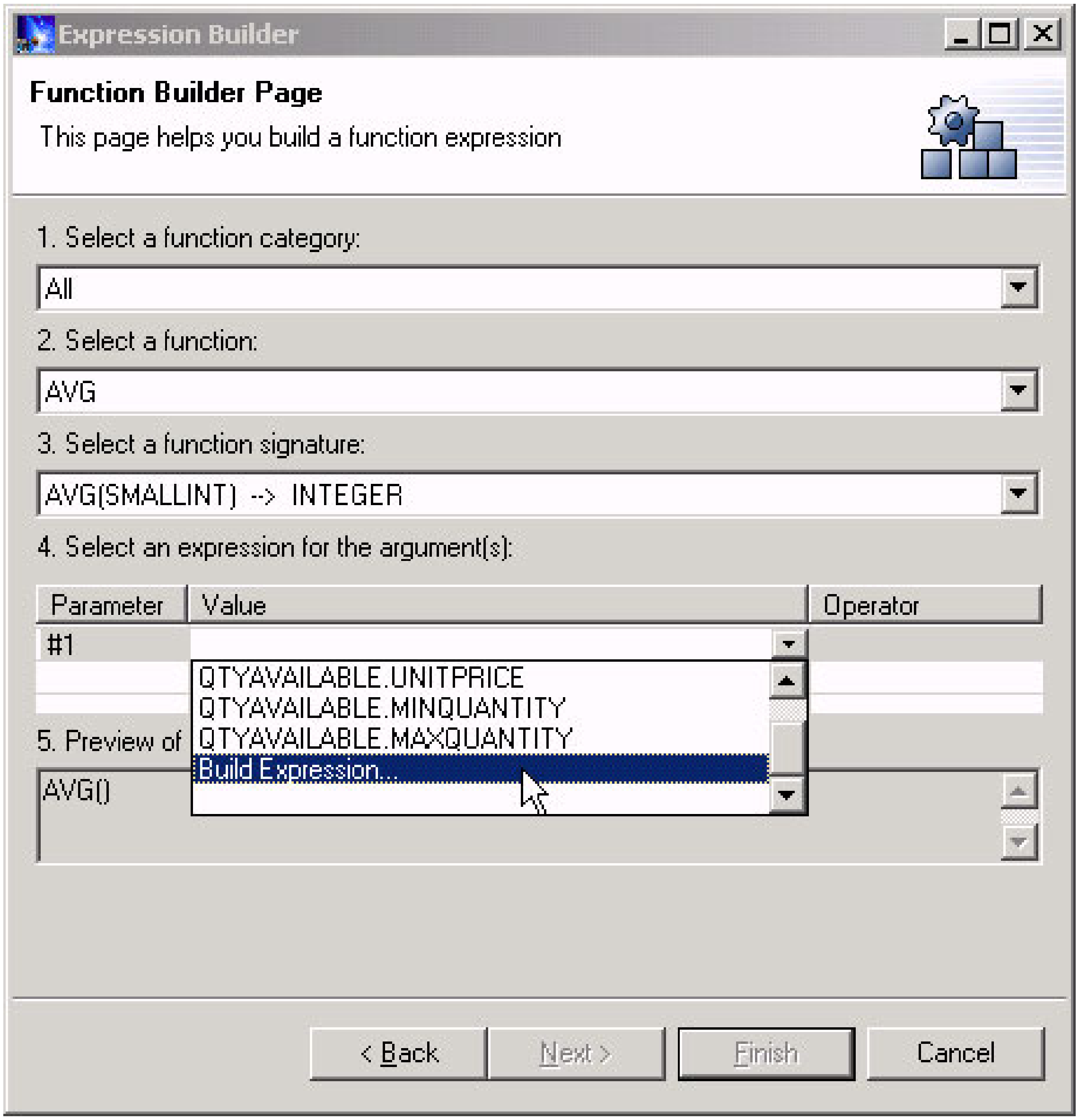
Building an expression, part 1

Notice that the statement from the previous panel doesn't actually have an average in it yet. To select the average minimum price for the group of offers, click the Columns tab, then click the first column to activate the pull-down menu. Select **Build Expression**.

An expression can be built on functions such as `avg()` or `sum()`; case statements, in which the outputted value depends on a particular expression; or casting, where a value of one type is converted to a value of another type. An expression can also consist of a constant, such as a string or numeric value, a subquery, where the values come from a second `SELECT` statement, or groups of operations on column values. Here, the desired expression is:

```
avg(unitprice * minquantity)
```

Choose **Function** and click **Next**.



Function Builder Page

This page helps you build a function expression

1. Select a function category: All

2. Select a function: AVG

3. Select a function signature: AVG(SMALLINT) --> INTEGER

4. Select an expression for the argument(s):

Parameter	Value	Operator
#1		

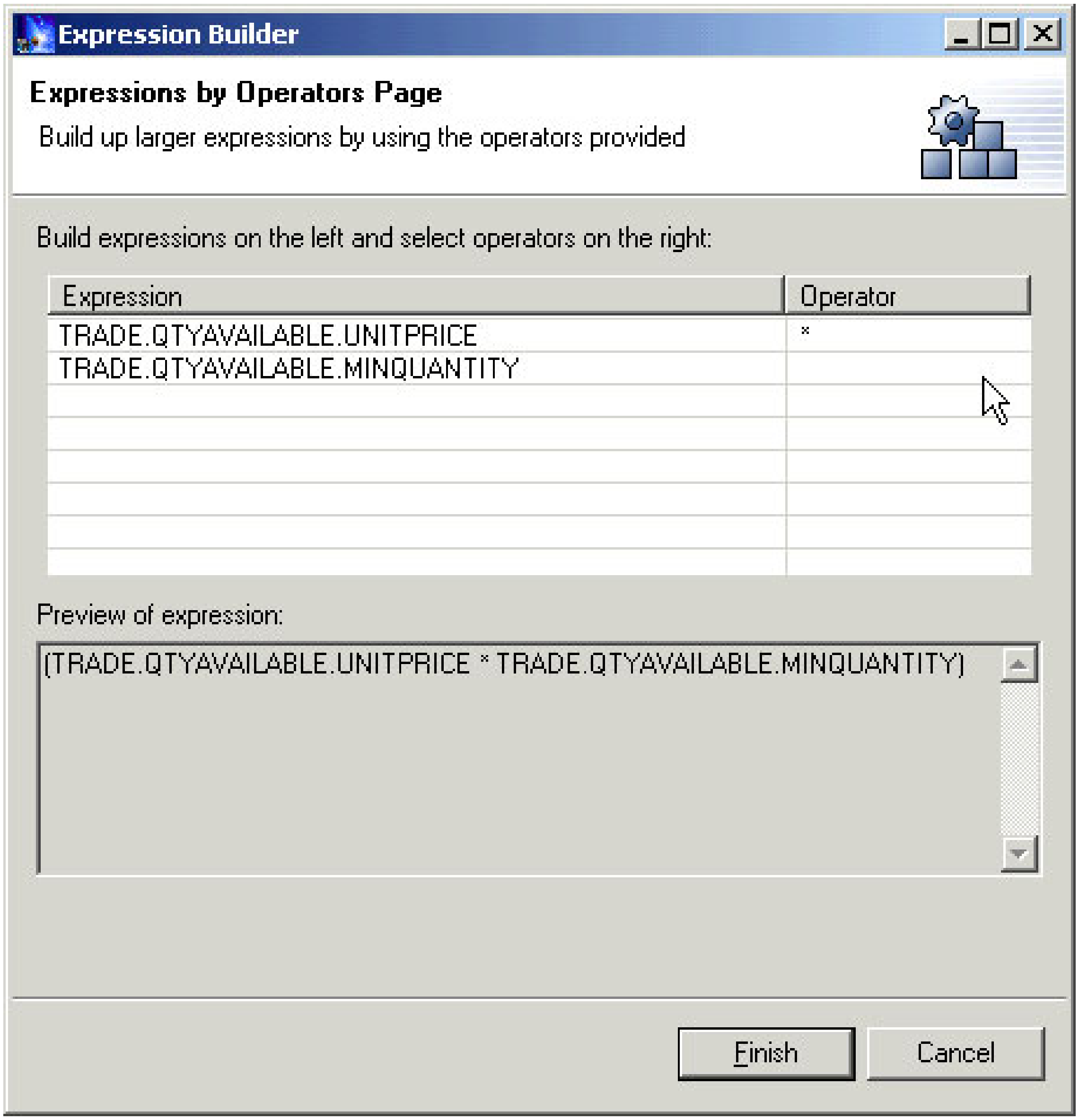
QTYAVAILABLE.UNITPRICE
QTYAVAILABLE.MINQUANTITY
QTYAVAILABLE.MAXQUANTITY
Build Expression...

5. Preview of

AVG()

Building an expression, part 2

Functions are grouped by type. Choose the `avg()` function and then specify the argument as a `double` value. The goal is to create the extended price average, so the argument for the `avg()` function is also an expression. Click the value box under step 4 to activate the pulldown menu and select **Build Expression**. Choose **Build up Expression by Operators**.

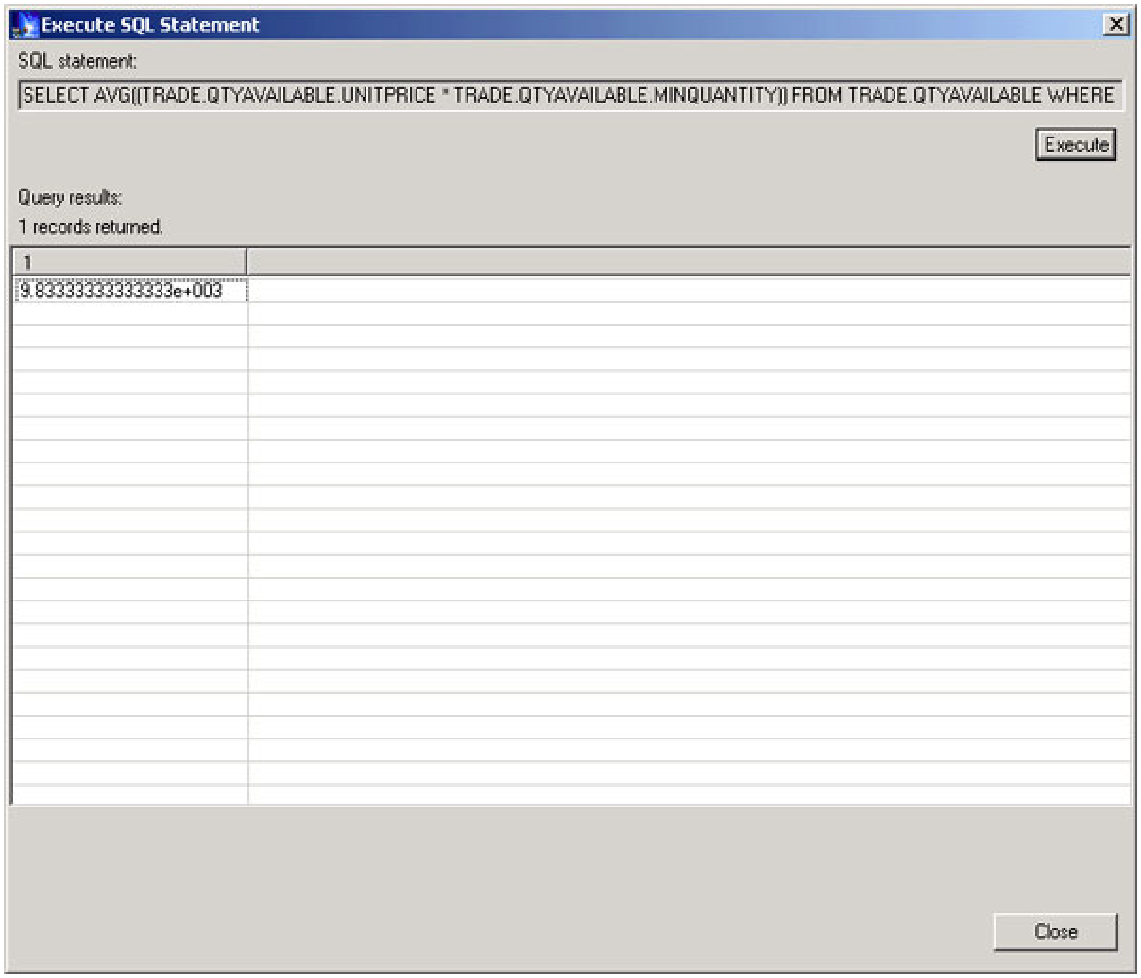

Expression Builder
Expressions by Operators Page
Build up larger expressions by using the operators provided
Build expressions on the left and select operators on the right:
Preview of expression:
Finish
Cancel

Expression	Operator
TRADE.QTYAVAILABLE.UNITPRICE	*
TRADE.QTYAVAILABLE.MINQUANTITY	

(TRADE.QTYAVAILABLE.UNITPRICE * TRADE.QTYAVAILABLE.MINQUANTITY)

Select the unitprice and minquantity columns and the `*` operator, then click **Finish** to return to the original expression.

Click **Finish** again to return to the statement. Notice that the complete expression is now part of the `SELECT` statement. Save the statement and execute it.



SQL statement:

```
SELECT AVG((TRADE.QTYAVAILABLE.UNITPRICE * TRADE.QTYAVAILABLE.MINQUANTITY)) FROM TRADE.QTYAVAILABLE WHERE
```

Query results:

1 records returned.

1
9.833333333333333e+003

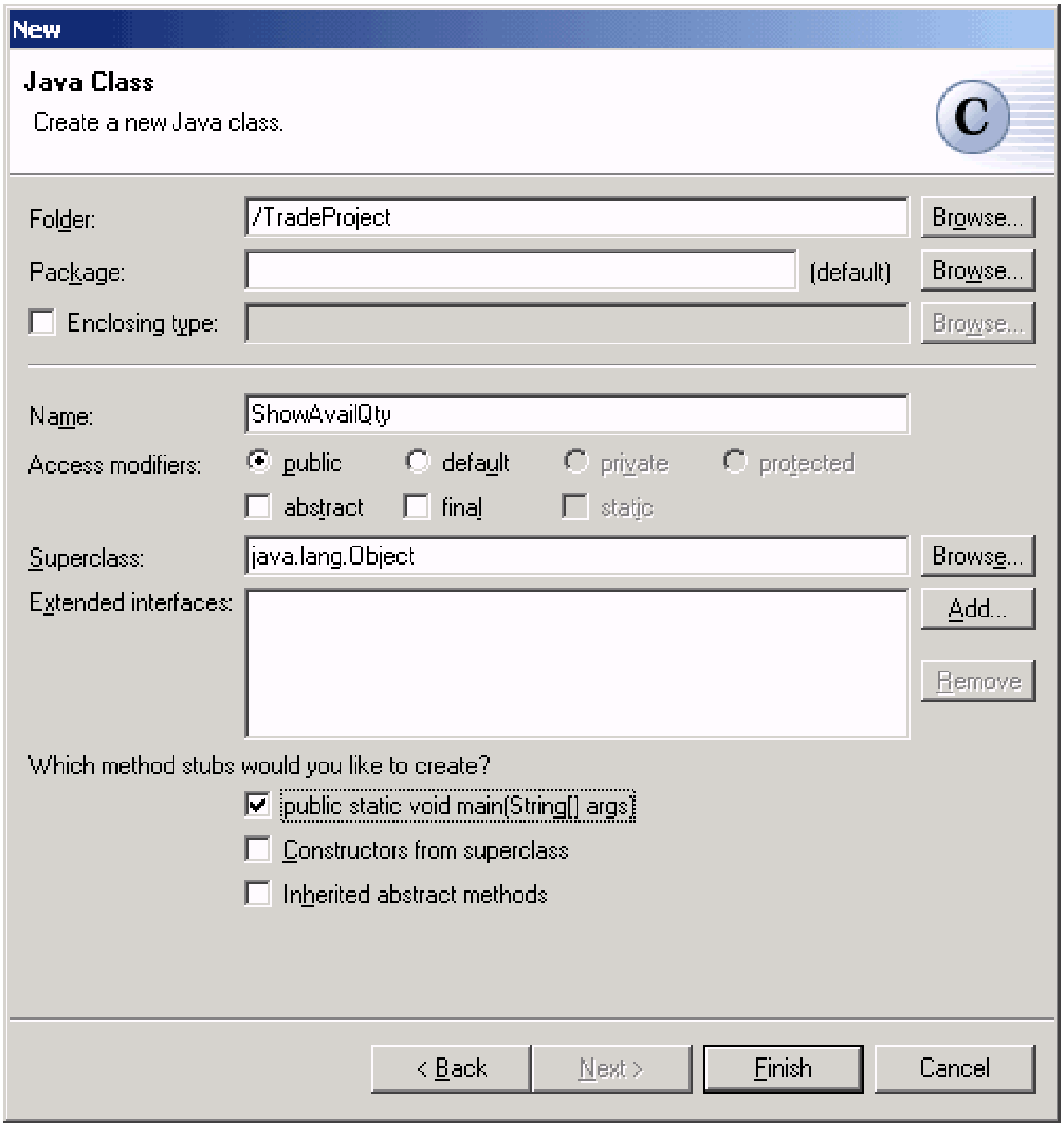
The next section, Selecting data from a Java application on page 28 , shows how to interact with DB2 directly from a Java application.

Section 4. Selecting data from a Java application

Create a Java class

In order for the data within DB2 to be accessible from the Web service, it must be accessible to an application. This section briefly demonstrates the concepts behind selecting data from and updating a DB2 database using a Java application and JDBC.

Create the Java class in WebSphere Studio by choosing **File=> New=> Other=> Java=> Java Class**. Click **Next** to create the class.



New

Java Class

Create a new Java class.

Folder: /TradeProject

Package: (default)

☐ Enclosing type:

Name: ShowAvailQty

Access modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Extended interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☐ Inherited abstract methods

If it's not pre-selected in the Folder field, click **Browse** to choose the appropriate project folder. WebSphere Studio enables you to specify the package to which the class should belong, the superclass, and any interfaces being extended, as well as method stubs. Name the class `ShowAvailQty` and click the `main` method stub. Click **Finish**.

The Java editor should appear with the basic class. Click the Navigator tab in the left-hand pane to see that both the `.java` file and the `.class` file were created. Saving the `.java` file automatically compiles the class.

Next, the class has to connect to the database.

Connect to the database

It's a good idea to switch to the Java perspective to work with the class, either by clicking the icon on the left-hand side of the screen or by choosing **Perspective=> Other=> Java**. Double-click the `.java` file to open the source code editor.

The first step in connecting to a database via JDBC is to instantiate the JDBC driver. In this case, the driver class is `COM.ibm.db2.jdbc.app.DB2Driver`. Once the driver's been instantiated, connect to the database using the appropriate URL and authentication information.

```
import java.sql.*;

public class ShowAvailQty {

   public static void main(String argv[]) {

      try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }

      Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      try {
         DBConnection = DriverManager.getConnection(DBurl,
                                                     "db2admin",
                                                     "db2pass");
         DBConnection.close();
      } catch( Exception e ) {
         e.printStackTrace();
      }
   }
}
```

The appropriate URL for a database varies with the database and driver type. Here, it's `jdbc:db2:sample` because the connection is using the `jdbc:` protocol to connect to the `db2:` database called `sample`.

The `DriverManager` class attempts to get a connection at the appropriate address using drivers that have been loaded into memory, including the driver instantiated by the `Class.forName()` method.

Once the application connects to the database, it can execute an SQL statement.

Call a SELECT statement

Actually calling a `SELECT` statement requires two objects: a Statement, and a ResultSet. The Statement populates the `ResultSet`, and the ResultSet provides the data, as seen below.

```
import java.sql.*;

public class ShowAvailQty {

   public static void main(String argv[]) {

      try {
        Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }

      Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      try {
          DBConnection = DriverManager.getConnection(DBurl,
                                                     "db2admin",
                                                     "db2pass");
         System.out.println("Getting available quantities...");
         Statement statement = DBConnection.createStatement();
          ResultSet results = statement.executeQuery("SELECT * from TRADE.QtyAvailable");
                while (results.next()) {
                        String offerid = results.getString("OFFERID");
                        String sellerid = results.getString("SELLERID");
                        String minquantity = results.getString("MINQUANTITY");
                        String maxquantity = results.getString("MAXQUANTITY");
                        String unitprice = results.getString("UNITPRICE");
                        System.out.print("Offerid" + offerid +": ");
                        System.out.println("Between "+ minquantity + " and "+
                                              maxquantity + " available for " + unitprice
                                            "("+sellerid+")");
                      System.out.print("\n");
                    }
                results.close();
                statement.close();
         DBConnection.close();
      } catch( Exception e ) {
         e.printStackTrace();
      }
   }
}
```

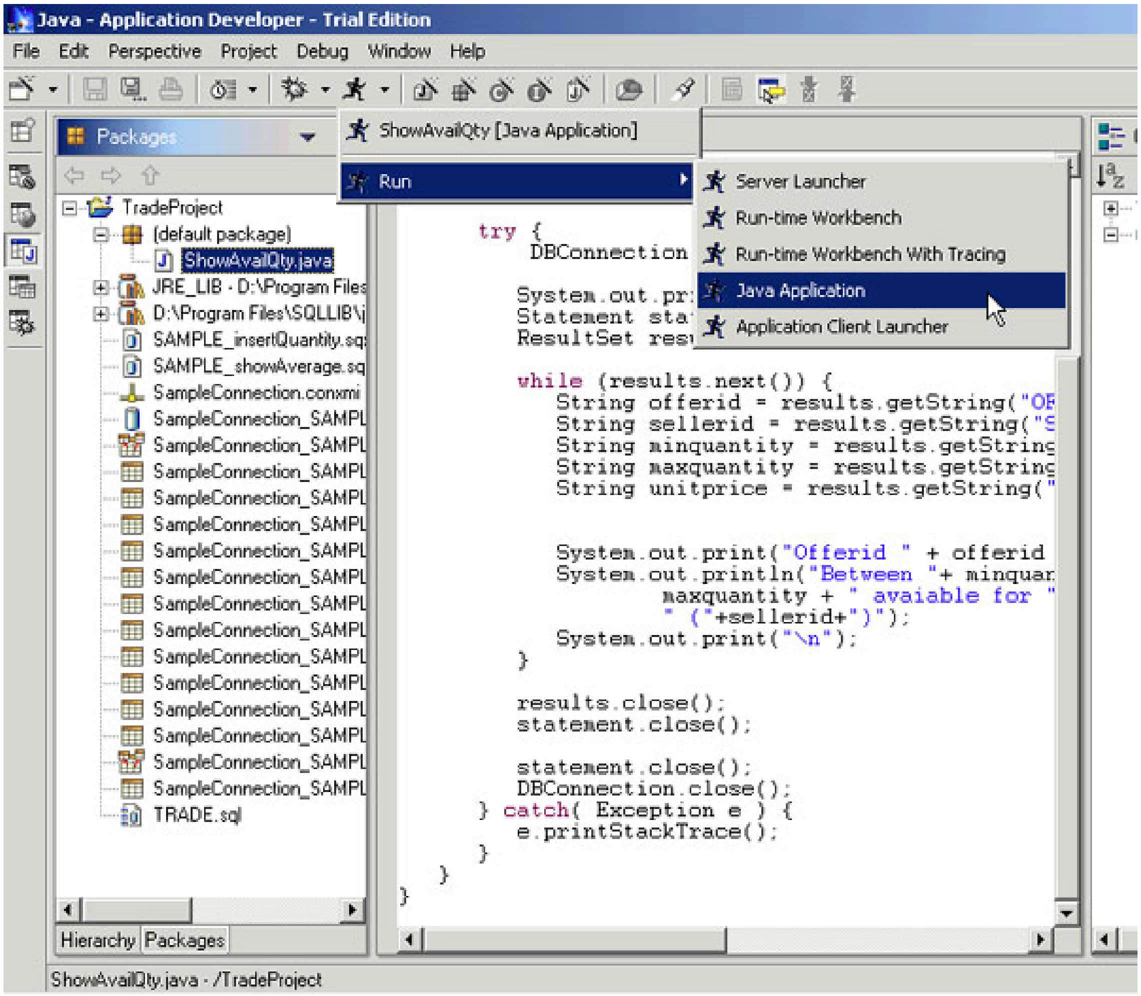
The database connection itself creates the Statement object, which can then execute a specific query. In this case, the query simply selects all of the records in the QtyAvailable table, using the results to populate the ResultSet object.

The ResultSet object enables you to loop through each record. As long as there is a `next()` record to move to, the loop executes, providing the opportunity to extract data from the current record. Data can be extracted as any compatible type using the `getXXX` methods. For example, an integer can be extracted as a String, but a String can't be extracted as a double. The `getXXX` methods accept either a column name or a column index as an argument.

The application is now ready to execute.

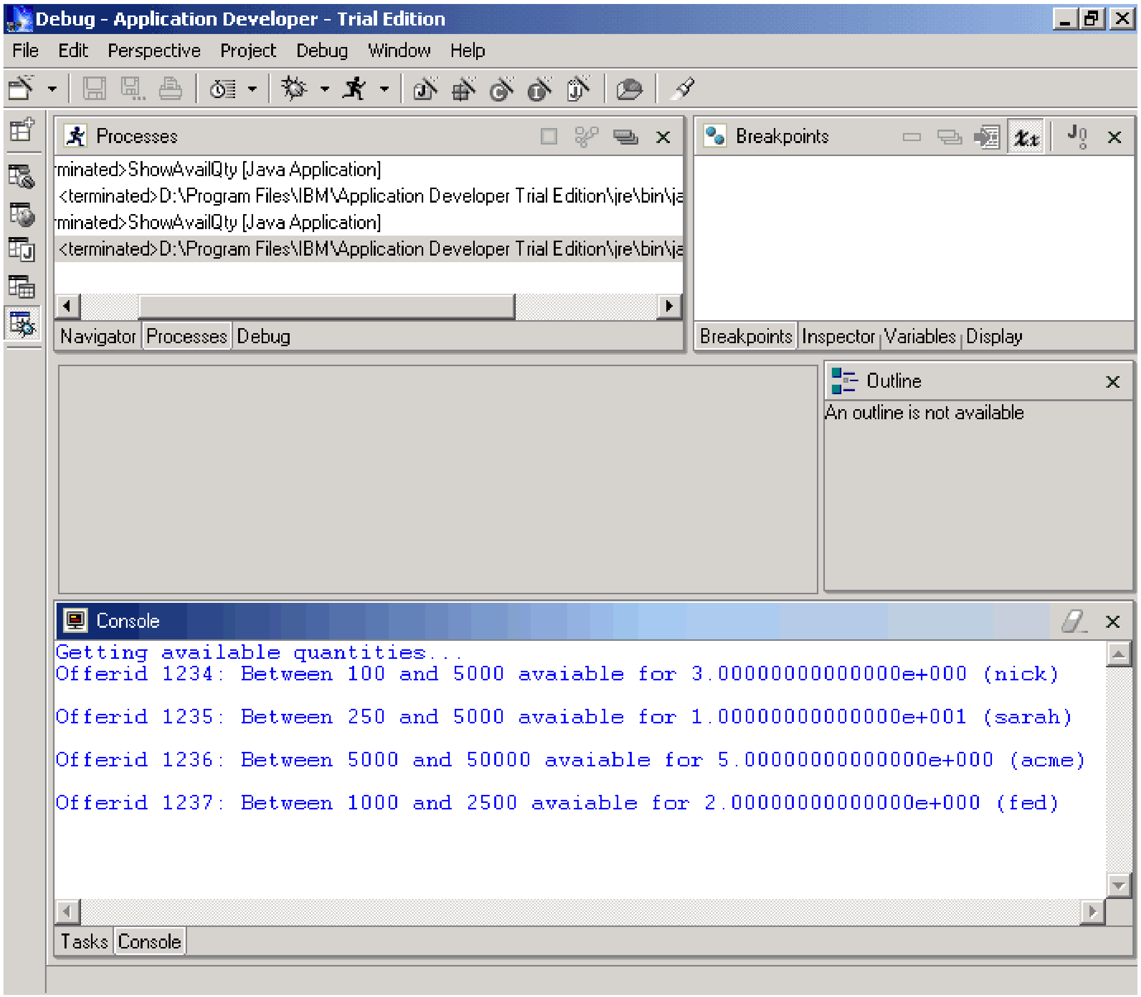
Execute the application

The Java application provides a means for accessing the DB2 data in an environment in which the DB2 tools may not be available. To see this in action, run `ShowAvailQty` by using the Run menu in the Java perspective. Select the class in the Packages view and click the triangle next to the running man. Choose **Run=> Java Application**.



```
    try {
      DBConnection

      System.out.pr
      Statement sta
      ResultSet resu

      while (results.next()) {
         String offerid = results.getString("OF
         String sellerid = results.getString("S
         String minquantity = results.getString
         String maxquantity = results.getString
         String unitprice = results.getString("


         System.out.print("Offerid " + offerid
         System.out.println("Between "+ minquan
              maxquantity + " avaiable for "
              " ("+sellerid+")");
         System.out.print("\n");
      }

      results.close();
      statement.close();

      statement.close();
      DBConnection.close();
    } catch( Exception e ) {
      e.printStackTrace();
    }
  }
}
```

WebSphere Studio opens the Debug perspective, which shows the results in the Console view. If any problems occur, they will be detailed in the console.


Debug - Application Developer - Trial Edition
File Edit Perspective Project Debug Window Help
Processes
minated>ShowAvailQty [Java Application]
<terminated>D:\Program Files\IBM\Application Developer Trial Edition\jre\bin\ja
minated>ShowAvailQty [Java Application]
<terminated>D:\Program Files\IBM\Application Developer Trial Edition\jre\bin\ja
Navigator Processes Debug
Breakpoints
Breakpoints Inspector Variables Display
Outline
An outline is not available
Console
Tasks Console

```
Getting available quantities...
Offerid 1234: Between 100 and 5000 avaiable for 3.00000000000000e+000 (nick)

Offerid 1235: Between 250 and 5000 avaiable for 1.00000000000000e+001 (sarah)

Offerid 1236: Between 5000 and 50000 avaiable for 5.00000000000000e+000 (acme)

Offerid 1237: Between 1000 and 2500 avaiable for 2.00000000000000e+000 (fed)
```

Updating DB2 from a Java application

One of the tasks the trading system Web service must accomplish is to add new offers to the database. This panel shows how to create an application that executes an `INSERT` statement against DB2. In the Creating a Web service on page 50 section, the functionality of this class will be incorporated into the Web service.

Save the current class with a new name to create a new class by choosing **File=> Save As** and selecting the project folder. Name the new file `AddOffer.java`. WebSphere Studio automatically creates the new class, compiles it, and opens it for editing.

Change the `executeQuery()` method to `executeUpdate()` to insert a new offer into the database:

```
import java.sql.*;

public class AddOffer {

   public static void main(String argv[]) {

      try {
        Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }

      Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      try {
          DBConnection = DriverManager.getConnection(DBurl,
                                                       "db2admin",
                                                       "db2pass");
         System.out.println("Adding a new offer...");
         Statement statement = DBConnection.createStatement();
          int result  = statement.execute Update (
                "INSERT INTO TRADE.QTYAVAILABLE " +
                "(OFFERID, SELLERID, MINQUANTITY, " +
                "MAXQUANTITY,UNITPRICE) VALUES "+
                "(1238, 'united', 250000, 500000, .75)"
                if (result=>0) {
                     System.out.println("Record Added.");
                     } else {
                     System.out.println("No records were added.");
                     }
         statement.close();
         DBConnection.close();
      } catch( Exception e ) {
         e.printStackTrace();
      }
   }
}
```

The application simply executes a pre-built query.

Execute the application and query the database -- either by creating a new query or by simply running `ShowOffers` -- to see the new data.

Section 5. User-defined functions

What is a user-defined function?

One of the advantages of using DB2 for your database needs is the ability to create your own functions, known as user-defined functions. These functions, which can be created in a number of different ways, can then be used as if they were built in to DB2. DB2 comes with dozens of functions built-in, but in some cases a developer might want to create an additional function so that other users can simply call it from an SQL statement in order to get the results. This eliminates the need for users to call a separate program just to write a report.

User-defined functions can be simple, such as one that is based on an existing function but merely has a different name or deals with a different data type, or they can be external applications such as Java methods.

This section demonstrates the creation of a user-defined function that adds commission and fees to a purchase to determine the "real price" of a transaction. Later, this function will be incorporated into the Web service for price quotes. The function is an external application, and is created as a Java class.

Create a function using Java

Create a new class called `ServiceUDFs` for the new functionality. The function `realprice()` should take a unit price and a quantity and calculate the extended price. It should then add 15% commission and a $9.95 transaction fee and return the result:

```
import COM.ibm.db2.app.UDF;

class ServiceUDFs extends UDF
{

  public void getRealPrice(double unitPrice, int quantity, double result)
                           throws Exception {

      double realPrice = (unitPrice * quantity * 1.15) + 9.95;
      set(3, realPrice);
      return;

  }
}
```

The class extends the `UDF` class provided by DB2, so it has all of the appropriate methods. For example, notice that even though the `realprice()` function should return a value, the `getRealPrice()` method does not. Instead, the value will always be returned by the last argument. To set that value, use the `set()` method that is part of the `UDF` superclass. It takes the index (assuming a 1-based list) and the value.

Save the class to compile it and copy the `ServiceUDFs.class` file to `{DB2 INSTALL}/SQLLIB/function` so the database knows where to find it.

All that's left is to register the function, and it will be ready to use.

Register a Java function

Before the function can be called from an SQL statement, it must be registered with the database. Registering a function can be a simple matter of referencing an existing function, as in:

```
CREATE FUNCTION myAvg(double)
     RETURNS double
     SOURCE Avg(double)
```

This statement creates a new function with the same capabilities as the original, but with a different name.

In the case of a function defined in a Java class, this is an *external* user-defined function, so the registration is a bit more complex:

```
CREATE FUNCTION TRADE.realPrice (DOUBLE, INTEGER)
   RETURNS DOUBLE
   EXTERNAL NAME 'ServiceUDFs!getRealPrice'
   LANGUAGE java
   PARAMETER STYLE db2general
   DETERMINISTIC
   FENCED
   NOT NULL CALL
   NO SQL
   NO EXTERNAL ACTION
   NO SCRATCHPAD
   NO FINAL CALL
   ALLOW PARALLEL
   NO DBINFO;
```

The `CREATE FUNCTION` statement defines the signature, or name and argument types, of the function as it will be called by an SQL statement. Notice that the name is not the same as the original Java method, and that the third parameter isn't shown. Instead, the `RETURNS` statement specifies the return type. The external name indicates the class that contains the method, an exclamation point as a delimiter, and the name of the method.

The function is considered `DETERMINISTIC` because given the same inputs it will always give the same output; it doesn't depend on external conditions such as a counter. In this case, the function is FENCED, which means that it runs in a separate process from the database, so if something goes wrong, the database is not in danger. (Fencing a function also allows you to debug and load the changed without having to restart DB2.) After a function has been thoroughly tested, changing it to `NOT FENCED` will improve performance.

This function doesn't require the ability to maintain information between calls, so `NO SCRATCHPAD` is necessary. In addition, many copies can be run simultaneously without incident, so you can `ALLOW PARALLEL` operations.

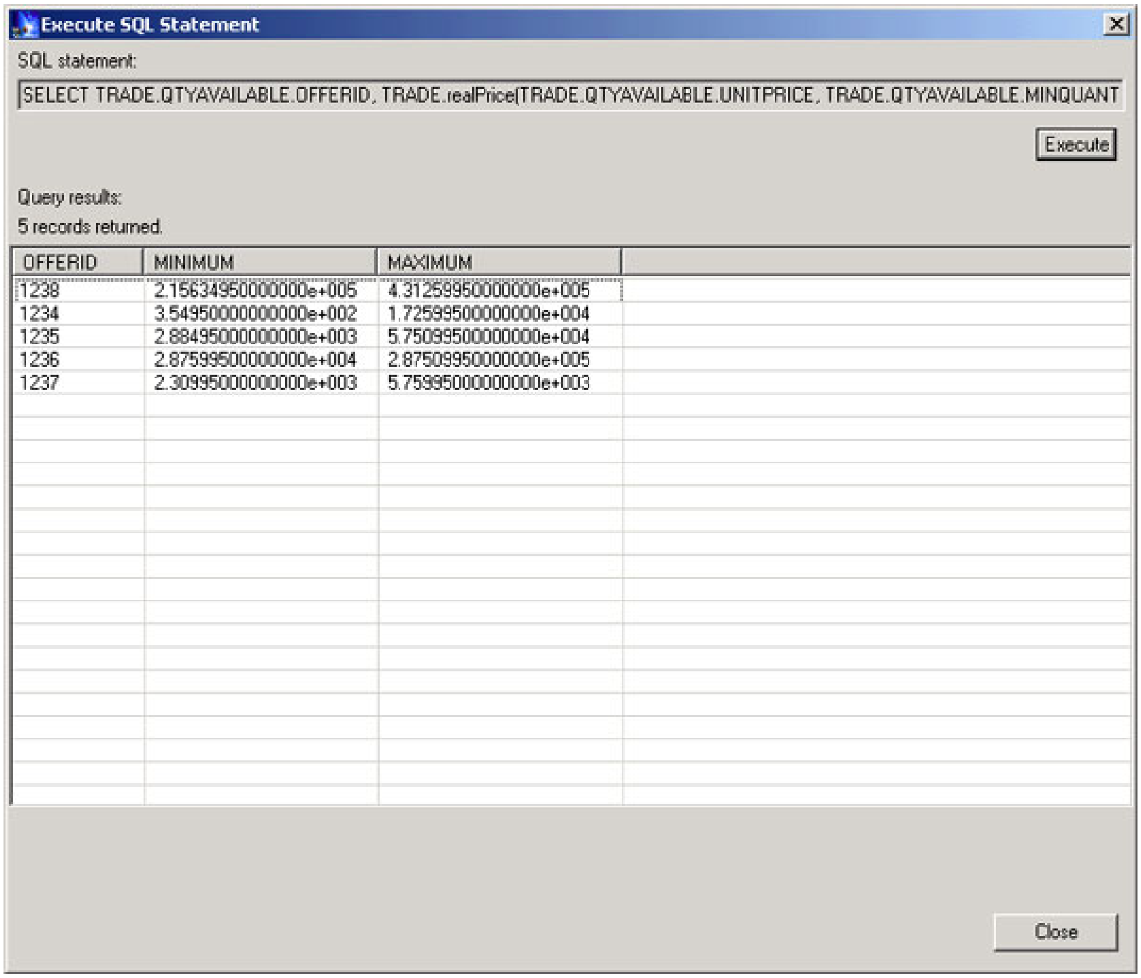
Run this command in the Command Line Processor (**Start=> Programs=> IBM DB2=> Command Line Processor**) or Command Center (**Start=> Programs=> IBM DB2=> Command Center**) to register the function with the database.

The `realPrice()` function is now available for use in an SQL statement.

Reference the function from an SQL statement

Once a function has been registered, it can be used in an SQL statement such as

```
SELECT
    TRADE.QTYAVAILABLE.OFFERID,
    TRADE.realPrice( TRADE.QTYAVAILABLE.UNITPRICE, TRADE.QTYAVAILABLE.MINQUANTITY ) AS mi
    TRADE.realPrice( TRADE.QTYAVAILABLE.UNITPRICE, TRADE.QTYAVAILABLE.MAXQUANTITY )  AS ma
FROM
   TRADE.QTYAVAILABLE
```



SQL statement:

```
SELECT TRADE.QTYAVAILABLE.OFFERID, TRADE.realPrice(TRADE.QTYAVAILABLE.UNITPRICE, TRADE.QTYAVAILABLE.MINQUANT
```

Query results:

5 records returned.

OFFERID	MINIMUM	MAXIMUM
1238	2.15634950000000e+005	4.31259950000000e+005
1234	3.54950000000000e+002	1.72599500000000e+004
1235	2.88495000000000e+003	5.75099500000000e+004
1236	2.87599500000000e+004	2.87509950000000e+005
1237	2.30995000000000e+003	5.75995000000000e+003

For even more functionality, you can create stored procedures, detailed in the next section.

Section 6. Stored procedures

Stored procedures

In addition to user-defined functions, DB2 allows the creation of stored procedures. Unlike user-defined functions, a stored procedure can call an SQL statement and can affect the contents of the database. In fact, with the exception of sending output to streams such as `System.out` and `System.err`, a Java stored procedure can do virtually anything a traditional Java application can do, but it does it within the DB2 process.

Also unlike a user-defined function, a stored procedure accepts and returns values using IN and OUT variables. For example, consider the potential stored procedure:

```
import java.sql.*;

public class ServiceSPs {

  public static void placeOrder(int qtyWanted, double[] totalPrice) {

            try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }

      Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      try {
         DBConnection = DriverManager.getConnection(DBurl, "db2admin", "db2pass");

         Statement statement = DBConnection.createStatement();
         ResultSet results = statement.executeQuery(
                             "SELECT * from TRADE.QtyAvailable "+
                             "where minquantity <= "+qtyWanted+
                             " and maxquantity >= "+qtyWanted+
                             " order by unitprice desc");

         int maxQuantity = 0;
         int offerid = 0;

         if (results.next()) {
             double unitPrice = results.getDouble("unitprice");
             totalPrice[0] = (unitPrice * qtyWanted * 1.15) + 9.95;
             maxQuantity = results.getInt("maxquantity") - qtyWanted;
             offerid = results.getInt("offerid");
         }

         results.close();

         statement.executeUpdate("update trade.qtyAvailable "+
                                           "set maxquantity = "+maxQuantity+
                                           " where offerid = "+offerid);

         statement.close();
         DBConnection.close();
      } catch( Exception e ) {
         e.printStackTrace();
      }


  }

}
```

Overall, `ServiceSPs.placeOrder()` is just like the applications built in Selecting data from a Java application on page 28 . It simply executes SQL statements to query or update DB2. The main difference is that it also sets a value for the `totalPrice` parameter that is intended to be passed back to the calling application. This is known as an `Out` parameter. An `Out` parameter can't have a value set when the stored procedure is called, but it returns a value.

By contrast, `qtyWanted` is an `In` parameter, so it must have a value set when the stored procedure is called, but it can't return a value.

A third type of parameter, an `InOut` parameter, can both take and return a value.

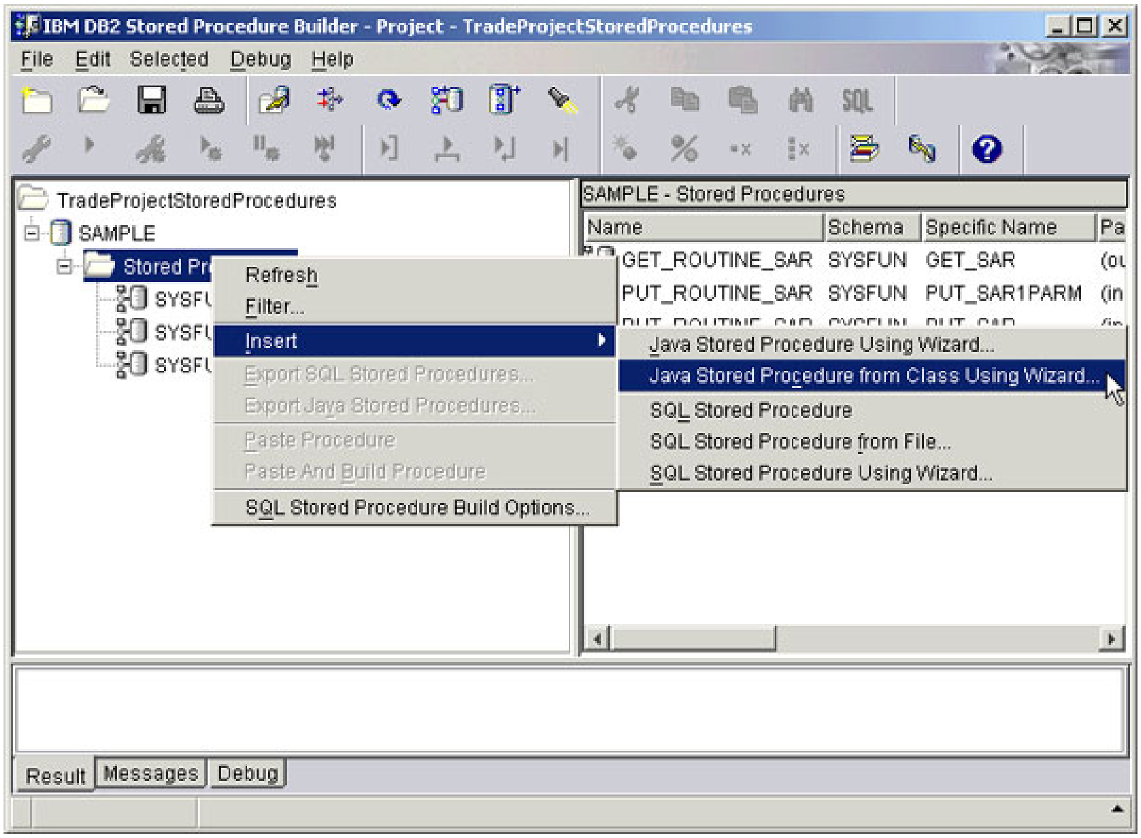
Like a user-defined function, a stored procedure must be registered with the database before it can be used.

Add the stored procedure to DB2

You can use a `CREATE PROCEDURE` statement to add a stored procedure to DB2, but there's a much easier way. Open the DB2 Stored Procedure Builder by choosing **Start=> Programs=> IBM DB2=> Stored Procedure Builder**. Even though the procedure is already built, this application makes it simple to add it to the database.

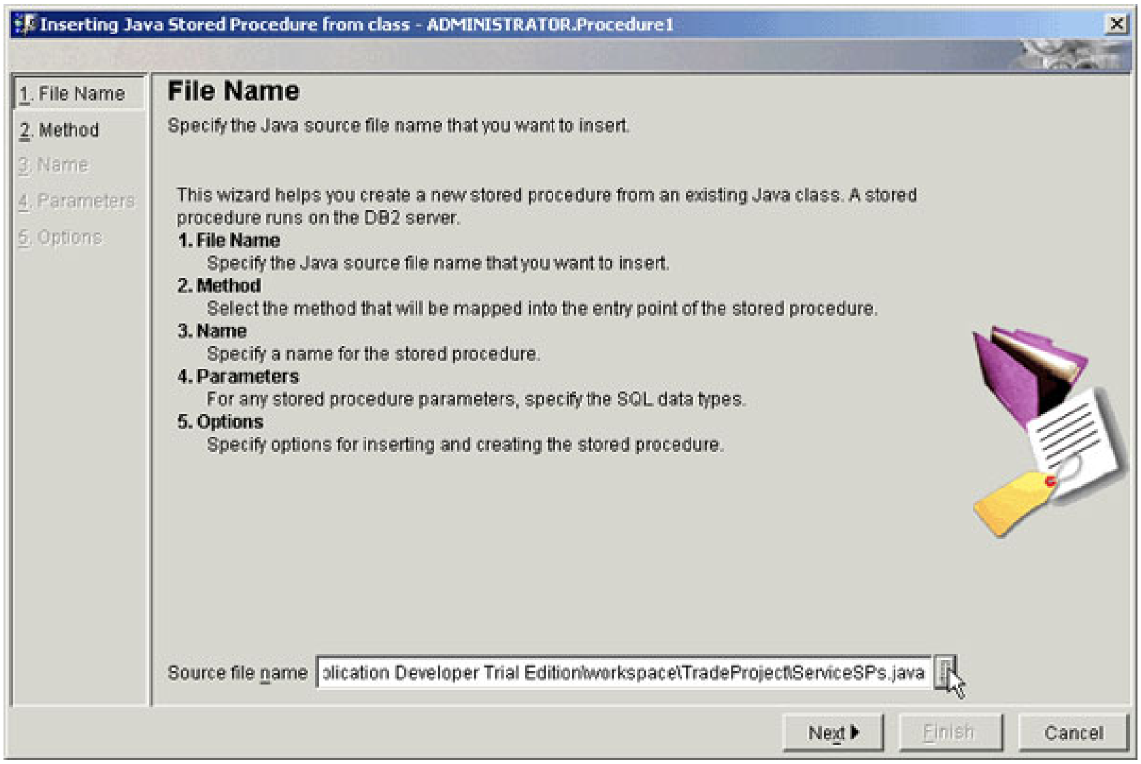
Create a new project, making sure to add the appropriate username and password. The application automatically connects to the database.

Right-click the Stored Procedures folder and select **Insert**. Stored Procedure Builder automates the process of creating both SQL and Java stored procedures, but in this case choose **Java Stored Procedure From Class Using Wizard...**.


IBM DB2 Stored Procedure Builder - Project - TradeProjectStoredProcedures
File Edit Selected Debug Help
TradeProjectStoredProcedures
SAMPLE
Stored Pr
SYSFU
SYSFU
SYSFU
Refresh
Filter...
Insert
Export SQL Stored Procedures...
Export Java Stored Procedures...
Paste Procedure
Paste And Build Procedure
SQL Stored Procedure Build Options...
Java Stored Procedure Using Wizard...
Java Stored Procedure from Class Using Wizard...
SQL Stored Procedure
SQL Stored Procedure from File...
SQL Stored Procedure Using Wizard...
SAMPLE - Stored Procedures
Name
Schema
Specific Name
Pa
GET_ROUTINE_SAR
SYSFUN
GET_SAR
(ou
PUT_ROUTINE_SAR
SYSFUN
PUT_SAR1PARM
(in
Result
Messages
Debug

Click the button at the end of the Source file name field to browse for the `ServiceSPs.java` file. The file is located in:

```
{Websphere install}\workspace\TradeProject
```


Inserting Java Stored Procedure from class - ADMINISTRATOR.Procedure1
1. File Name
2. Method
3. Name
4. Parameters
5. Options
Next
Finish
Cancel

File Name

Specify the Java source file name that you want to insert.

This wizard helps you create a new stored procedure from an existing Java class. A stored procedure runs on the DB2 server.

1. File Name
Specify the Java source file name that you want to insert.

2. Method
Select the method that will be mapped into the entry point of the stored procedure.

3. Name
Specify a name for the stored procedure.

4. Parameters
For any stored procedure parameters, specify the SQL data types.

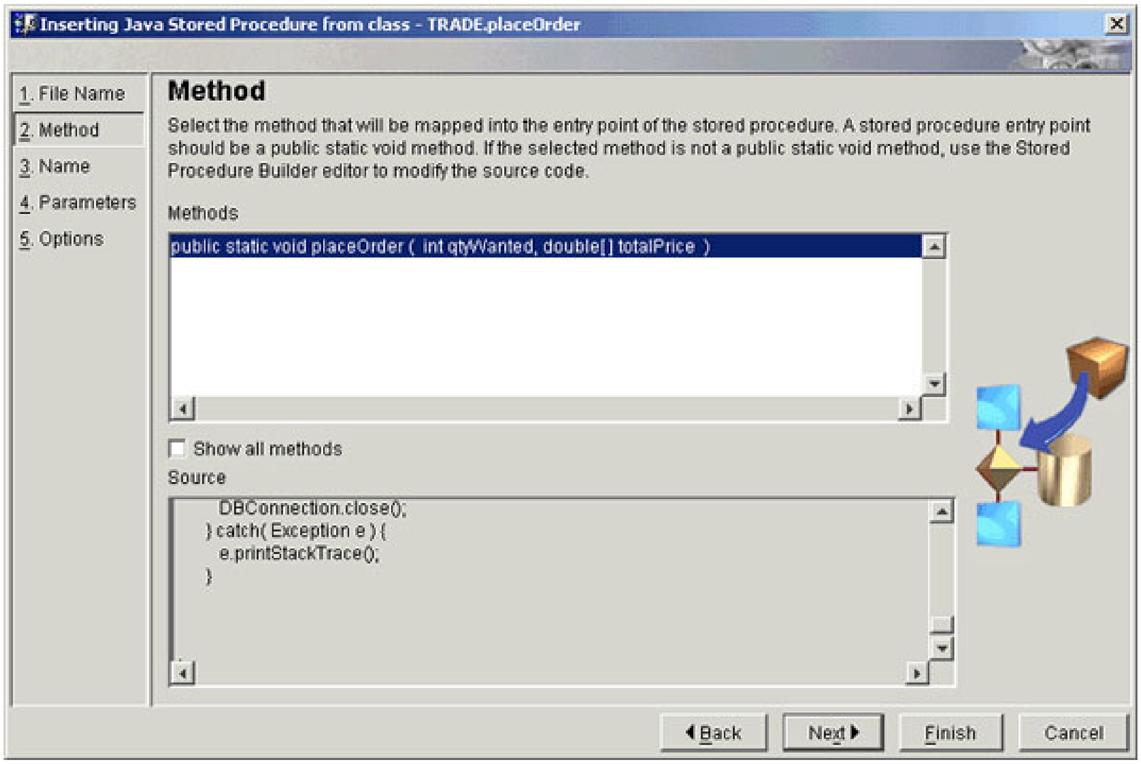
5. Options
Specify options for inserting and creating the stored procedure.

Source file name: plication Developer Trial Edition\workspace\TradeProject\ServiceSPs.java

Click **Next**.

Specify the method and name

Choose the method that will be called by this stored procedure.



Method

Select the method that will be mapped into the entry point of the stored procedure. A stored procedure entry point should be a public static void method. If the selected method is not a public static void method, use the Stored Procedure Builder editor to modify the source code.

Methods: public static void placeOrder (int qtyWanted, double[] totalPrice)

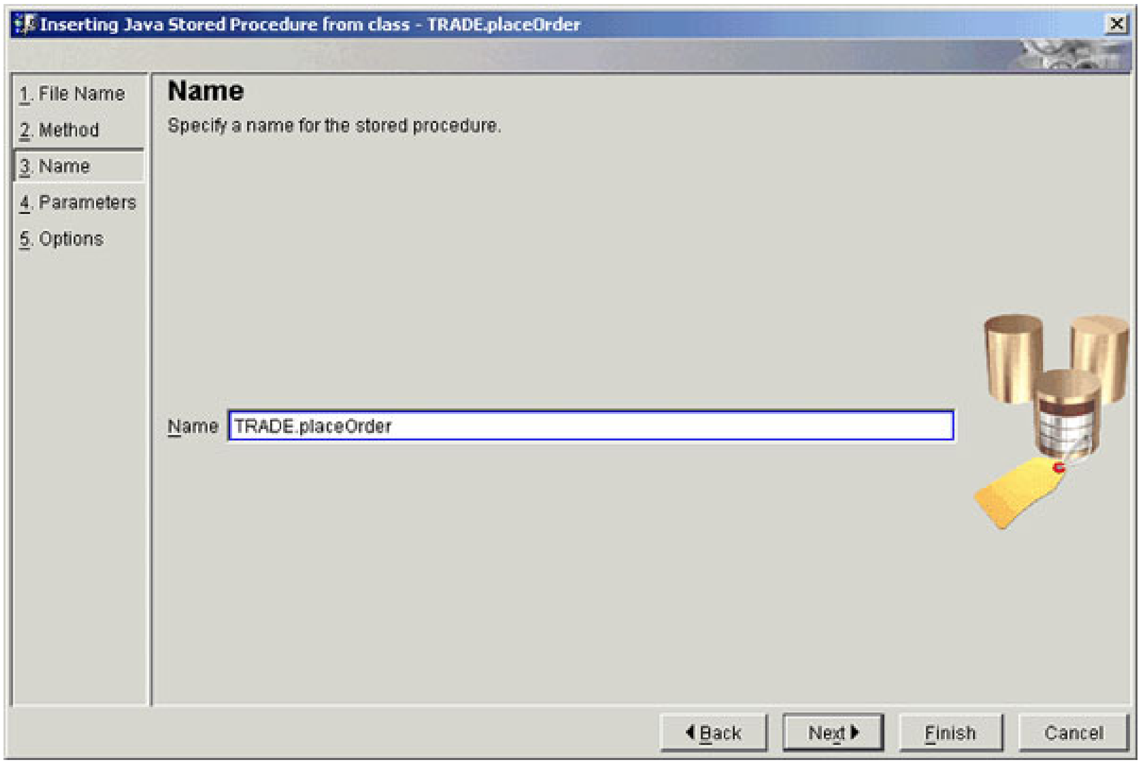
☐ Show all methods

Source

```
    DBConnection.close();
  } catch( Exception e ) {
    e.printStackTrace();
  }
```

Click **Next**.

Enter the name by which the stored procedure will be known. Note that like a user-defined function, the stored procedure doesn't have to have the same name as the method it calls. Also note that in order for the stored procedure to be stored in the TRADE schema, it needs to specified as part of the name.

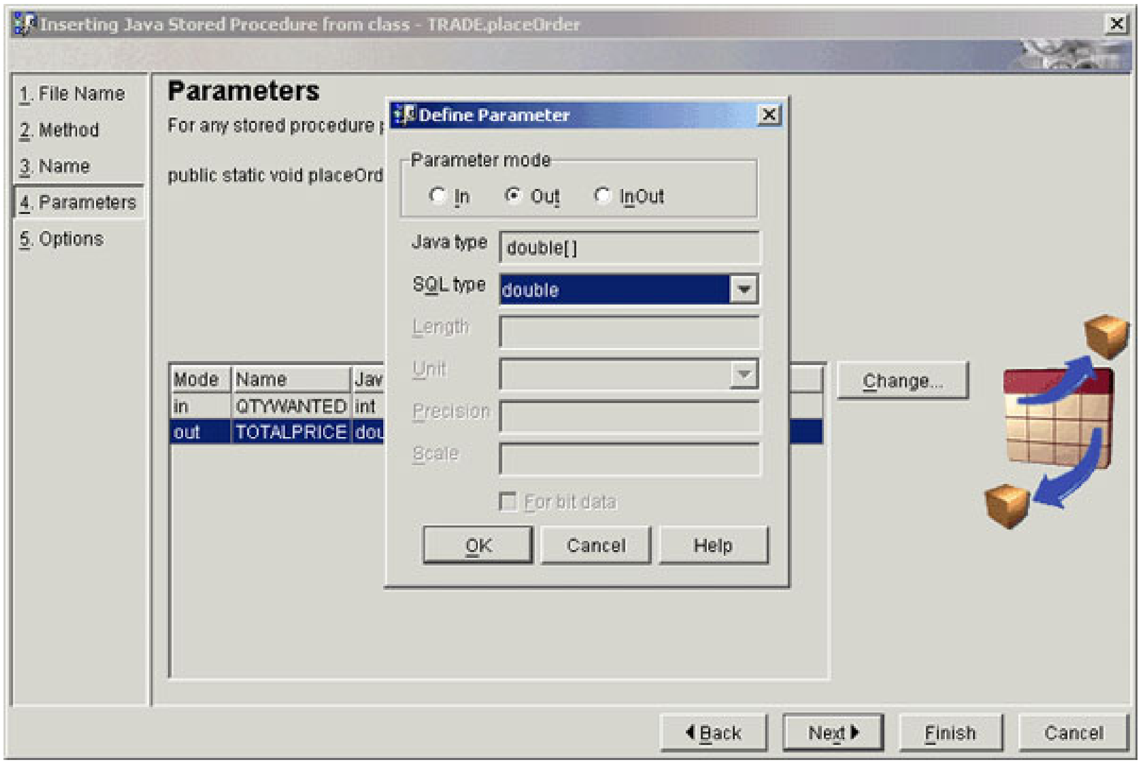

Inserting Java Stored Procedure from class - TRADE.placeOrder
1. File Name
2. Method
3. Name
4. Parameters
5. Options
Name
Specify a name for the stored procedure.
Back
Next
Finish
Cancel

Name TRADE.placeOrder

Click **Next**.

Specify parameter types

On the Parameters screen, notice that both parameters are originally specified as `In` parameters. Highlight the `totalPrice` parameter and click **Change**.



Inserting Java Stored Procedure from class - TRADE.placeOrder

Parameters

For any stored procedure p

public static void placeOrd

Mode	Name	Jav
in	QTYWANTED	int
out	TOTALPRICE	dou

Define Parameter

Parameter mode: ○ In ● Out ○ InOut

Java type: double[]

SQL type: double

Length:

Unit:

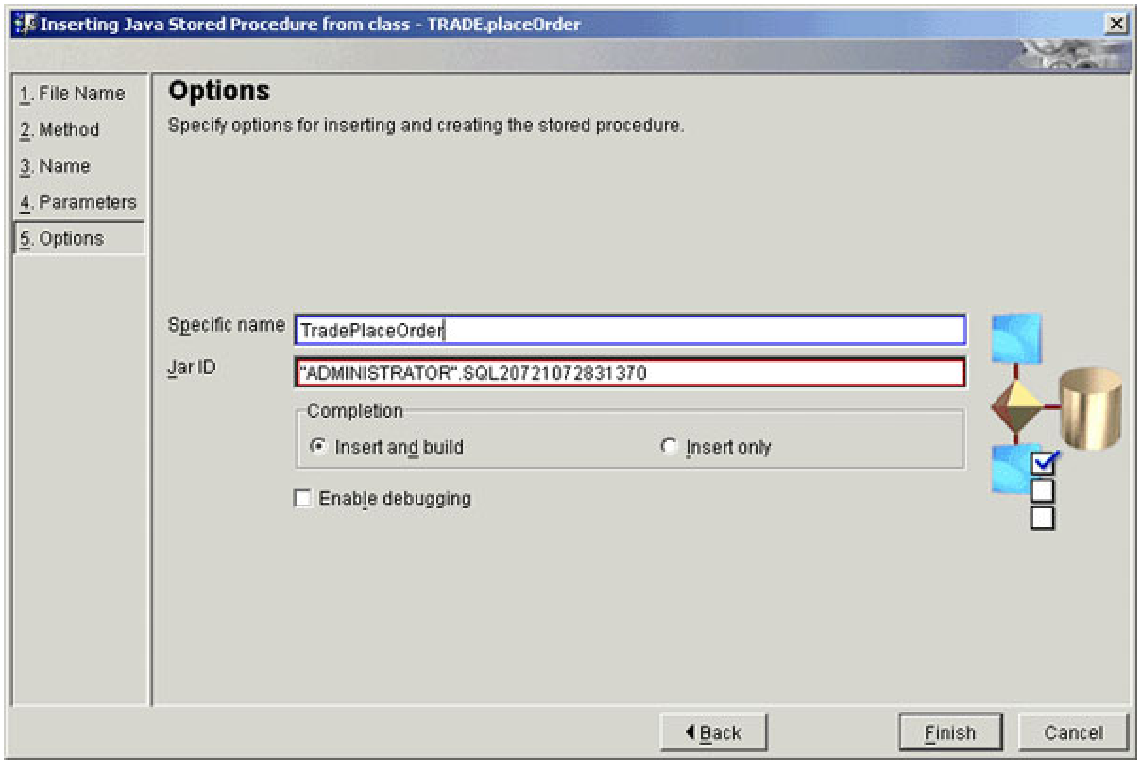
Precision:

Scale:

☐ For bit data

This is the time to map SQL types to Java types, if necessary. In this case, both are using `double`, so just click the Out radio button and click **OK**. Note that the `Out` and `InOut` parameters must be arrays in the Java application.

Finally, add a specific name for the procedure. This is not strictly necessary; DB2 will provide a random string of numbers if you don't.



Inserting Java Stored Procedure from class - TRADE.placeOrder

Options

Specify options for inserting and creating the stored procedure.

Specific name: TradePlaceOrder

Jar ID: "ADMINISTRATOR".SQL20721072831370

Completion

● Insert and build ○ Insert only

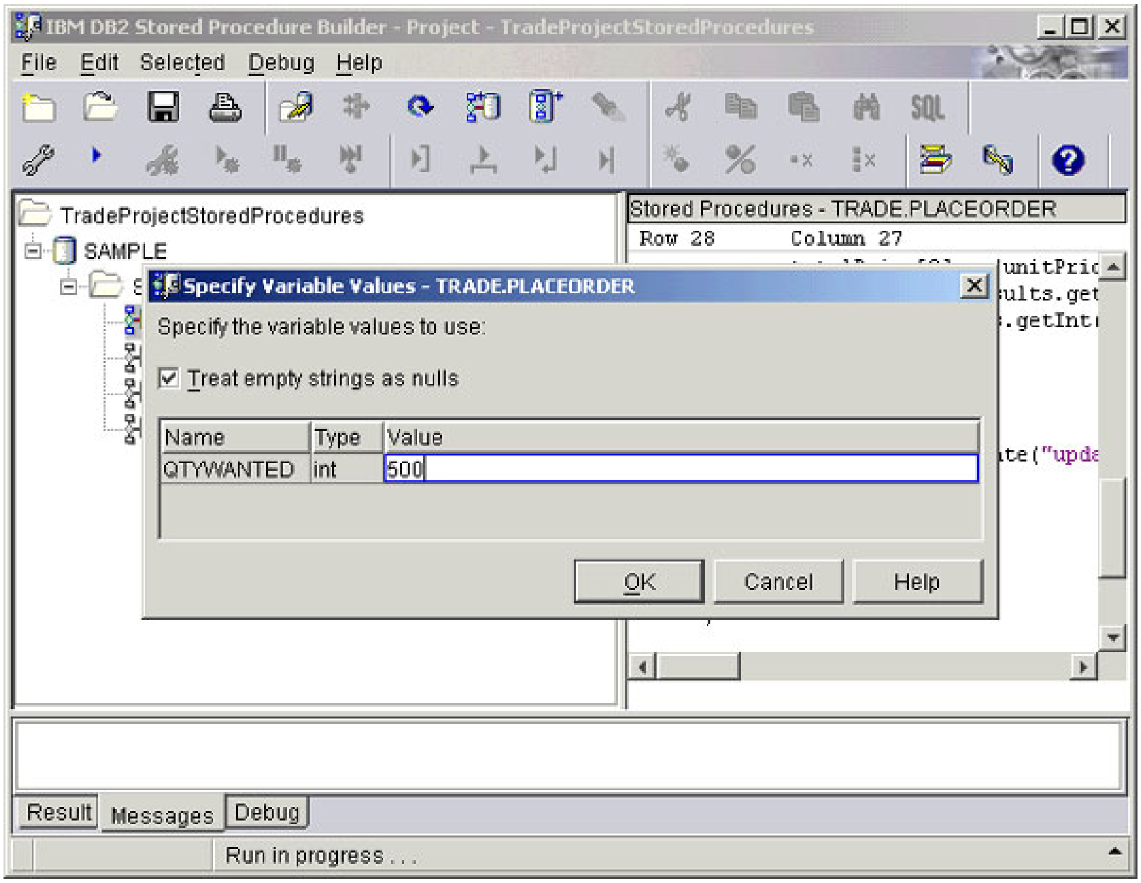
☐ Enable debugging

Click **Finish**. Stored Procedure Builder saves and builds the stored procedure, and registers it with the database.

Stored Procedure Builder also makes it easy to test the new procedure.

Test the procedure

To test the new stored procedure, right-click its listing under the Stored Procedures folder and choose **Run**. The application provides a dialog box into which you can enter the `qtyWanted` value.


IBM DB2 Stored Procedure Builder - Project - TradeProjectStoredProcedures
File
Edit
Selected
Debug
Help
SQL
TradeProjectStoredProcedures
SAMPLE
Stored Procedures - TRADE.PLACEORDER
Row 28
Column 27
Specify Variable Values - TRADE.PLACEORDER
OK
Cancel
Help
Result
Messages
Debug

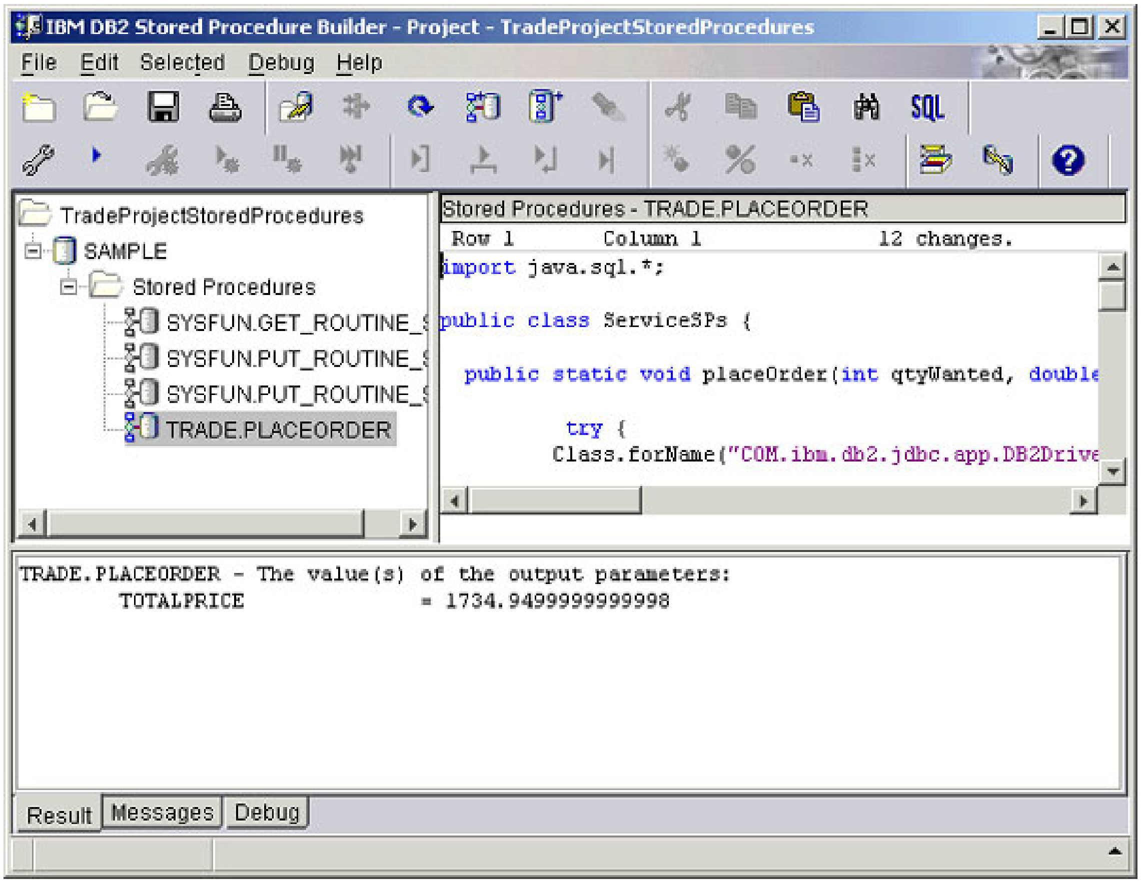
Specify the variable values to use:

☑ Treat empty strings as nulls

Name	Type	Value
QTYWANTED	int	500

Run in progress . . .

Enter a value such as `500` and click **OK**. The results appear in the Results pane at the bottom of the window.



```
import java.sql.*;

public class ServiceSPs {

  public static void placeOrder(int qtyWanted, double

          try {
        Class.forName("COM.ibm.db2.jdbc.app.DB2Drive
```

```
TRADE.PLACEORDER - The value(s) of the output parameters:
        TOTALPRICE              = 1734.9499999999998
```

Call the procedure from Java

Finally, the stored procedure is ready to be called from a Java application. The process is similar to what's been discussed previously in the tutorial, with the exception of the need to set the parameters on the statement. Consider the `Purchase` class below:

```
import java.sql.*;

public class Purchase {

    public static void main(String[] args) {
      updateQuantities(1000);
    }

    public static void updateQuantities(int qtyWanted) {

      try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }


          Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      try {
         DBConnection = DriverManager.getConnection(DBurl,
                                                    "db2admin",
                                                    "db2pass");

         String sql = "CALL TRADE.PLACEORDER( ?, ? )";
         CallableStatement callStmt = DBConnection.prepareCall(sql);

         callStmt.setInt(1, qtyWanted);

         callStmt.registerOutParameter(2, java.sql.Types.DOUBLE);

         callStmt.execute();

         double totalPrice = callStmt.getDouble(2);

         System.out.println(totalPrice);

         DBConnection.commit();
         DBConnection.close();

       } catch (SQLException e) {
         System.out.println(e);
       }

    }

}
```

Overall, the idea is the same, but the differences are significant. The application still calls a statement, but instead of an SQL statement, it's a DB2 `CALL` statement. The question marks (?) indicate parameters.

By preparing the call, the database connection makes it possible to set the `In` parameter (using the `setXXX()` method `setInt()`) and to register the `Out` parameter. Notice that when setting the `In` parameter, the type is predetermined by the method (`setInt()`). For the `Out` parameter, the type must be explicitly specified.

From there, the application executes the statement, so the `Out` parameter becomes available.

This basic framework will make the stored procedure available from the Web service in the next section, Creating a Web service on page 50 .

Section 7. Creating a Web service

How WebSphere Studio creates Web services

A Web service communicates by taking in XML messages in a predetermined format such as SOAP. It then parses them to determine the relevant information, acts on that information, and returns a similar XML message.

Rather than writing an application from scratch, you can let WebSphere Studio do the hard work for you by creating a proxy, or wrapper, for your application. The proxy accepts the SOAP messages, performs a call to the application, then formats the response and sends it back to the requester.

All you have to do is create your application and the proxy will do the rest. Other users can build your functionality into their applications by simply making the appropriate calls to the proxy, as defined by the (conveniently generated) Web Services Description Language (WSDL) files.

Complete the Java application

The complete Java application performs three functions. First, it accepts a desired quantity and returns the price that quantity would cost at that moment, taking into consideration minimum purchase requirements. Second, it accepts a purchase, determining the appropriate offer and deducting that amount from the available maximum and returning the cost of the purchase. Third, it accepts the data necessary to create a new offer in the database and inserts that data.

The complete application takes the code developed throughout the tutorial and makes minor changes. Most of these changes are necessary because a Web service must return values rather than output them.

```
import java.sql.*;
import COM.ibm.db2.jdbc.app.*;

public class TradingService {

   public double getCurrentPrice(int qtyWanted){

              try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }
      Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      double price = 0;

      try {
         DBConnection = DriverManager.getConnection(DBurl, "db2admin", "db2pass");

         Statement statement = DBConnection.createStatement();
         ResultSet results = statement.executeQuery(
                          "SELECT TRADE.realPrice(unitprice, "+qtyWanted+
                          ") as price from TRADE.QtyAvailable "+
                          "where (maxquantity >= minquantity) and "+
                          "(minquantity <= "+qtyWanted+") and (maxquantity >= "+qtyWanted
                          ") order by unitprice");

         if (results.next()) {
            price = results.getDouble("price");
         }

         results.close();
         statement.close();

         statement.close();
         DBConnection.close();

      } catch( Exception e ) {
         e.printStackTrace();
      }

      return price;
   }

        public static double placeOrder(int qtyWanted)
    {

      try {
         Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
         e.printStackTrace();
      }


            Connection DBConnection = null;
      String DBurl = "jdbc:db2:sample";

      try {
         DBConnection = DriverManager.getConnection(DBurl,
                                                    "db2admin",
                                                    "db2pass");

         String sql = "CALL TRADE.PLACEORDER( ?, ? )";
         CallableStatement callStmt = DBConnection.prepareCall(sql);

         callStmt.setInt(1, qtyWanted);

         callStmt.registerOutParameter(2, java.sql.Types.DOUBLE);
```

```
       callStmt.execute();

       double totalPrice = callStmt.getDouble(2);

       DBConnection.commit();
       DBConnection.close();

       return totalPrice;

     } catch (SQLException e) {

       System.out.println(e);
       return -1;

     }

  }

public static String addOffer(int offerid, String sellerid,
                                  double price, int minquantity,
                                  int maxquantity) {

    try {
       Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
    } catch (Exception e) {
       e.printStackTrace();
    }

    Connection DBConnection = null;
    String DBurl = "jdbc:db2:sample";

    String returnString = "";

    try {
        DBConnection = DriverManager.getConnection(DBurl, "db2admin", "db2pass");

       Statement statement = DBConnection.createStatement();
       int result = statement.executeUpdate(
               "INSERT INTO TRADE.QTYAVAILABLE "+
               "(OFFERID, SELLERID, MINQUANTITY, MAXQUANTITY, UNITPRICE)"+
               " VALUES ("+offerid+", '"+sellerid+"', "+minquantity+", "
                                        +maxquantity+", "+price+")");
       if (result > 0) {
               returnString = "Record Added.";
       } else {
               returnString = "No records were added.";
       }

       statement.close();
       DBConnection.close();
    } catch( Exception e ) {
             returnString = "Problem adding record: "+e.getMessage();
       e.printStackTrace();
    }

    return returnString;
 }
```

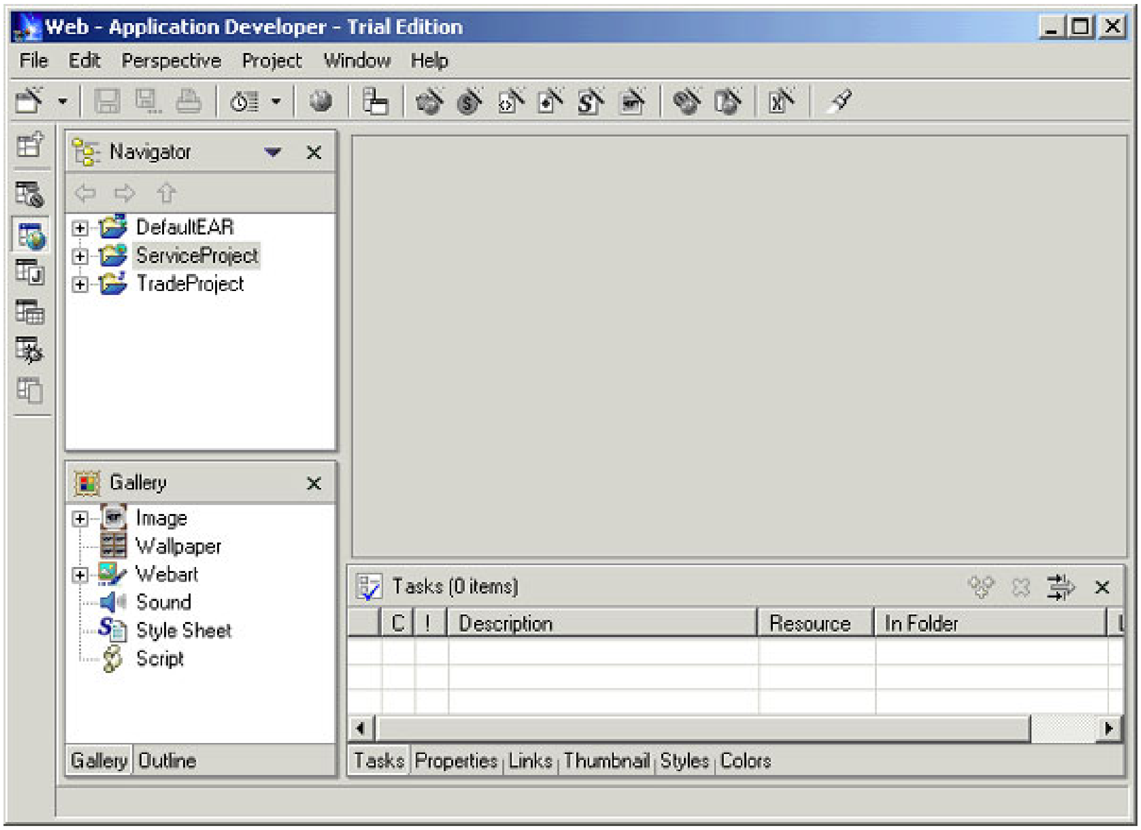
```
}
```

WebSphere Studio uses the signatures for these methods to create the proxy.

Create a Web project

Web services can only be created within Web projects, so choose **File=> New=> Project=> Web=> Web Project** and click **Next**. Name the project `ServiceProject`. Click **Next** twice to get to the Define Build Setting screen and click the Libraries tab to specify `SQLLIB/db2java.zip` as an external `.jar` file, just as you did in Create a project on page 5 .

Click **Finish** to create the project. WebSphere Studio automatically creates the directories necessary for a J2EE Web application.

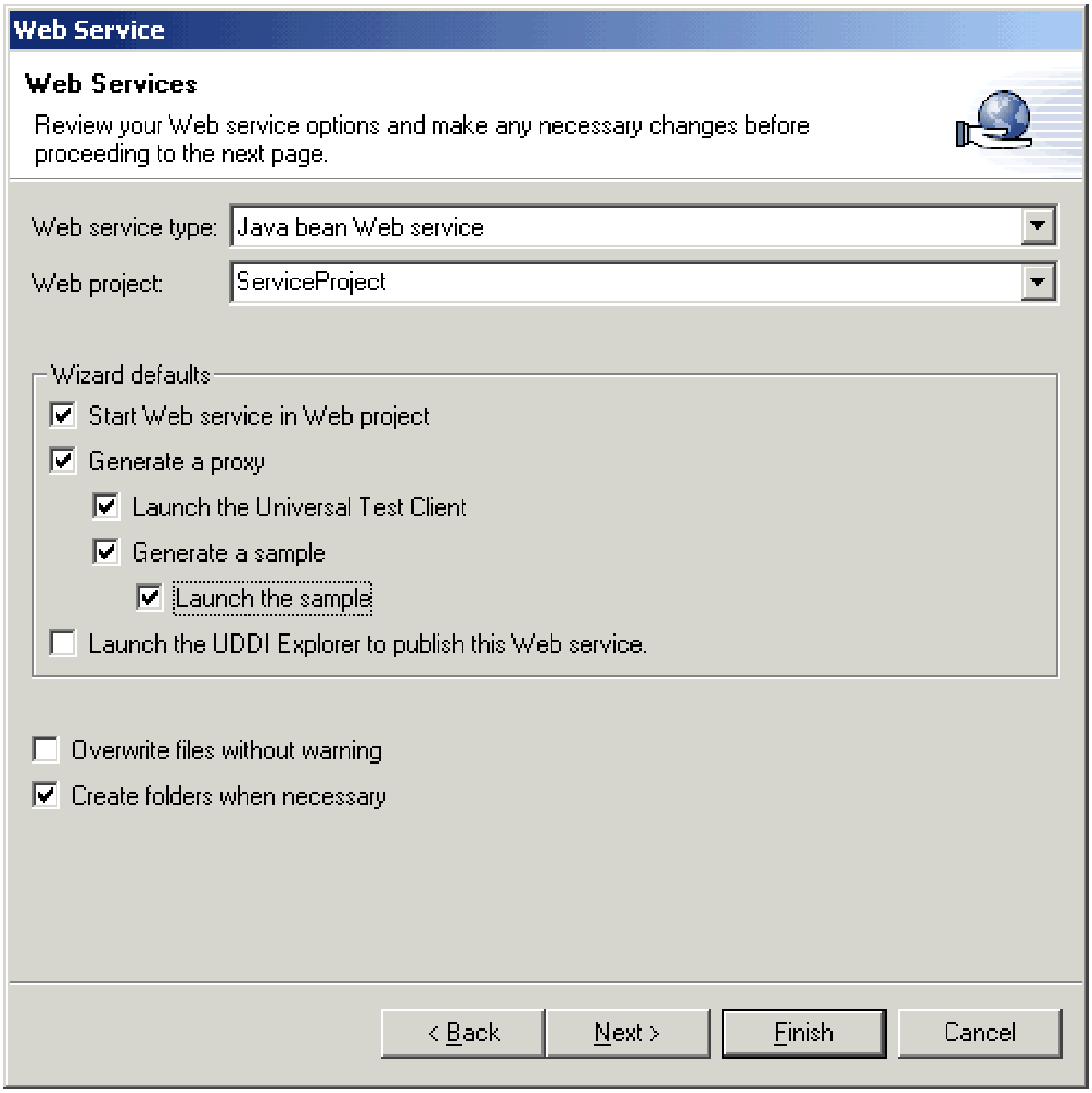

Web - Application Developer - Trial Edition
File Edit Perspective Project Window Help
Navigator
DefaultEAR
ServiceProject
TradeProject
Gallery
Image
Wallpaper
Webart
Sound
Style Sheet
Script
Gallery Outline
Tasks (0 items)
C ! Description Resource In Folder
Tasks Properties Links Thumbnail Styles Colors

Web projects can be tested on the built-in test server, which is an instance of the WebSphere server environment built into WebSphere Studio. This allows you to test applications without having to disturb a running server.

At this point the Java application to be used as the basis of the Web service is in the workspace, but it's within a different project. Expand the TradeProject folder so you can see the `TradingService.java` file, and the ServiceProject folder so you can see the source folder. Drag the `.java` file into the source folder or right-click the `.java` file and choose **Copy**, setting the destination as the source folder. WebSphere Studio automatically compiles the application to the `webapplication/WEB-INF/classes` directory.

Create the Web service

To create the Web service, choose **File=> New=> Other=> Web Services=> Web Service** and click **Next**.



Review your Web service options and make any necessary changes before proceeding to the next page.

Web service type: Java bean Web service

Web project: ServiceProject

Wizard defaults

☑ Start Web service in Web project

☑ Generate a proxy

  ☑ Launch the Universal Test Client

  ☑ Generate a sample

    ☑ Launch the sample

☐ Launch the UDDI Explorer to publish this Web service.

☐ Overwrite files without warning

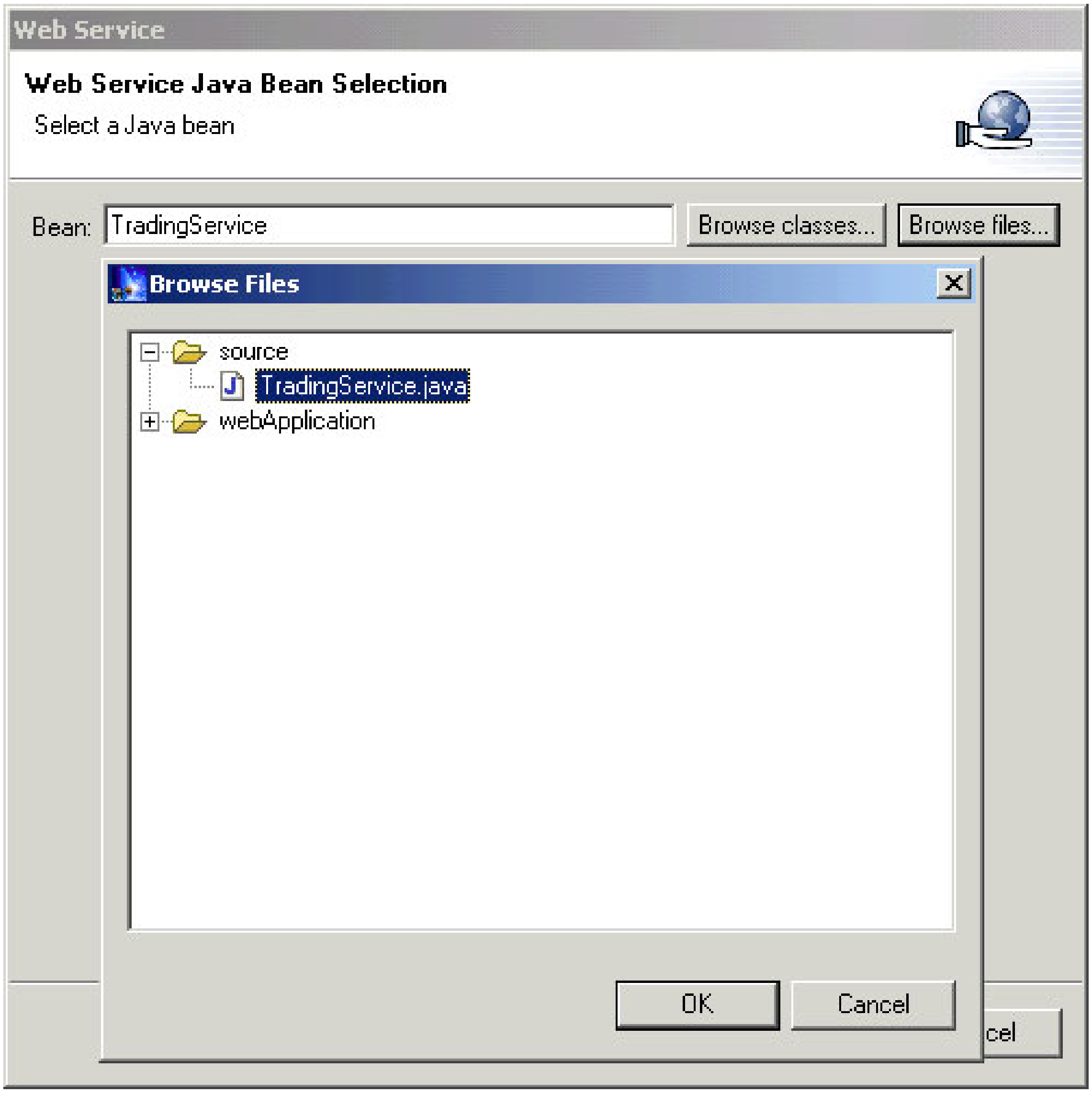
☑ Create folders when necessary

Make certain that **Java Bean Web Service** and **ServiceProject** are selected. WebSphere Studio can also create a test client, so make sure **Generate a proxy** is checked and check **Launch the Universal Test Client**, **Generate a sample**, and **Launch the sample**.

Click **Next**.

Choose the Java bean

Click **Browse files** and navigate to the `TradingService.java` file.



Click **OK**.

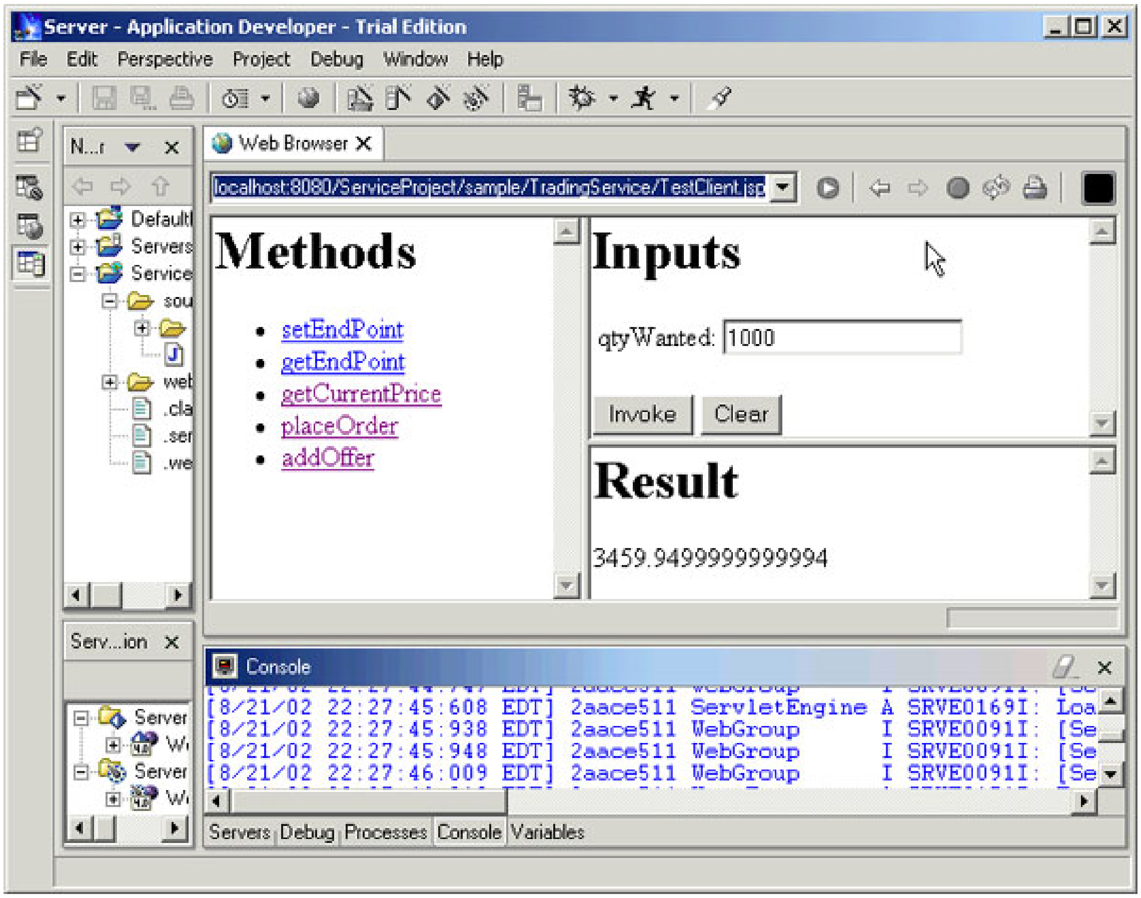
WebSphere Studio enables a developer to control aspects of Web service generation, such as which methods are made available, type mappings, and so on. To examine these possibilities click the **Next**, or click **Finish** to accept the defaults.

Test the application

The application takes several moments to create the Web service and the sample, then opens the sample for use. It also creates the test environment server instances and starts them. With the test server running, the sample can be accessed from within WebSphere Studio, or it can be accessed via browser at:

`http://localhost:8080/serviceProject/sample/TradingService/TestClient.jsp`

Notice that the left-hand column shows the calls available to the Web service, and that they match the method names for the application. Click the getCurrentPrice link to get a form that accepts the desired quantity. Enter a quantity and click **Invoke** to see the resulting price information.


Server - Application Developer - Trial Edition
File Edit Perspective Project Debug Window Help
N...r
Default
Servers
Service
Web Browser
Serv...ion
Server
Server
Console
Servers Debug Processes Console Variables
Invoke
Clear

localhost:8080/ServiceProject/sample/TradingService/TestClient.jsp

Methods

- setEndPoint
- getEndPoint
- getCurrentPrice
- placeOrder
- addOffer

Inputs

qtyWanted: 1000

Result

3459.9499999999994

```
[8/21/02 22:27:45:608 EDT] 2aace511 ServletEngine A SRVE0169I: Loa
[8/21/02 22:27:45:938 EDT] 2aace511 WebGroup      I SRVE0091I: [Se
[8/21/02 22:27:45:948 EDT] 2aace511 WebGroup      I SRVE0091I: [Se
[8/21/02 22:27:46:009 EDT] 2aace511 WebGroup      I SRVE0091I: [Se
```

Test both methods and watch for price variations as certain sellers sell out. All of these changes are reflected in the actual database.

Section 8. Conclusion

Summary

We have covered a considerable amount of material here. This tutorial chronicled the creation of a Web service that accesses a simple commodity trading service built on DB2. WebSphere Studio greatly simplifies the process of creating a Web service that accesses a DB2 database by providing an environment for data and query design, by providing editing and debugging resources for Java applications, and by actually creating a Web service wrapper around a Java application. This tutorial discussed:

- Using WebSphere Studio to access an existing database
- Creating a new schema and database table and migrating the design back to the database
- Accessing a database from a Java application
- Creating a Java-based user-defined function that could then be used in a SQL statement
- Creating DB2 stored procedure out of a Java class, then calling the stored procedure from a Java application
- Creating a Web service out of a Java application

Resources

This tutorial covers a lot of ground, so you'd do well to further investigate any of the topics that particularly interest you. Here are some good places to start:

- See the *developerWorks* tutorials *Web services with WebSphere Studio: Build and test* and *Web services with WebSphere Studio: Deploy and publish* for information on creating a Web service with WebSphere Studio Site Developer.
- Also see this tutorial on *integrating applications with Web services using WebSphere*.
- Get a feel for building applications with WebSphere Studio with *Developing and Testing a Complete "Hello World" J2EE Application with IBM WebSphere Studio Application Developer for Linux*.
- Get a different perspective on creating Web services that use SQL statements and stored procedures with the tutorials *Creating Web services on Windows to access DB2* and *Creating Web services on Linux to access DB2*.
- Read the official *SOAP recommendation* from the W3C or look in on other *Web Services activities*.
- See the tutorial *Manipulating XML and SQL data with WebSphere Studio* for

information on integrating SQL into XML applications.

- Take a look at IBM's new resource center for information on *DB2 Web services*.
- For a technical overview of the new Development Center, read the article *DB2 Development Center -- The Next-Generation AD Tooling for DB2* .
- IBM's WebSphere Application Developer includes tools that assist in the building and debugging of Java applications, database manipulation, Web sites, and other J2EE applications. Download an evaluation version for *Windows* or *Linux*.
- IBM's DB2 Universal Database includes everything you need to build powerful, robust database applications. See the *DB2 Universal Database page* for more information.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.