Accessing Data Using ADO .Net

Objectives
The aim of this tutorial is to provide a practical introduction to using ADO .NET for accessing data contained in remote databases. We will consider the fundamentals of ADO .NET objects, data binding, retrieving and updating data from a remote database and using ADO .NET in conjunction with XML Web services.

Prerequisites
You should have read the accompanying lecture notes on the use of databases in distributed applications and .NET database technologies before attempting this tutorial. You should also be familiar with SQL.

Lab Setup
All the exercises in this tutorial are based around a pre-built Microsoft SQL Server database (the ePizzeria database) which must be installed on your development machine. Your tutor will supply you with the URL from which you can download the installation application for this database. Note that you will also need to install a version of Microsoft SQL Server 2000 or MSDE before installing this application. You will also need to have the IIS web server running on your machine if you wish to try the Web service exercises.

For More Information
There are several .NET books available which cover the various techniques for using databases from .NET applications, and from mobile devices in particular. The following titles are directly relevant to ADO .NET and data access from mobile devices in particular but there are many other books that you could consult.

**Database Access using ADO .NET**

ADO .NET is a database technology available within the .NET framework which has been designed to support sophisticated multi-tiered applications. It provides a *disconnected architecture* which can operate over the Internet and is based on XML. It also supports binding of data to user interface controls in .NET Windows Forms applications. ADO .NET is available in the full .NET Framework and in the .NET Compact Framework, albeit in a slightly more limited form.

Disconnected operation in a database application is extremely useful when connection time is at a premium and where connections have intermittent availability or can be disconnected at any time. As a result, ADO .NET is particularly suited to mobile device and dial-up network applications. Essentially ADO .NET works by creating an in-memory copy of relevant data from the data source. ADO.NET objects can be created independently of the data source and then connected to it for as long as it takes to obtain necessary data. Once the data has been obtained it can be manipulated and updated offline and then merged into the data source when a connection becomes available again. This is essentially a form of caching which enables the application to continue operating when a network connection is not present.

ADO.NET objects hold versioning and update status details to allow very fine control over the merging process. ADO .NET objects are also able to write their contents and their schema to a text file or an I/O stream. This makes it very easy to save local persistent copies of ADO .NET objects and to transfer them across the network as arguments to, or return values from, remoting and Web service calls. ADO .NET objects can also be used with Windows Forms data binding in the user interface layer of an application. This allows the state of user interface controls to be updated based on the state of bound data items and allows the data items to be updated when the state of the controls changes. Data binding enables very fast development of a user interface that manipulates data.

The most important ADO .NET objects are, *DataSet, DataTable, DataColumn, DataRow, DataRelation* and *DataView*.

**DataSet**

This is the most important object in the ADO .NET hierarchy and it is fundamental to the disconnected nature of ADO .NET. A *DataSet* is simply a container for data being held in memory. It allows data to be cached and changed locally when a connection to the persistent data store is not available. A *DataSet* object contains a collection of one or more *DataTable* objects and is capable of representing multiple tables.
of data in a structured and relational way that is familiar to database programmers. It represents the data to the programmer in terms of the familiar concepts of tables, rows and columns. When a connection to the data store becomes available, the local data in a *DataSet* object can be read from or written to using other ADO .NET objects. The *DataSet* object can also be written to, or read from, an external source in XML format. *DataSet* objects are useful for containing and handling data in both the user interface and business layers of a distributed application. In their XML representation, *DataSet* objects can be passed between part of an application or between different applications.

**DataTable**

A *DataTable* object contains the rows and columns of data held in memory inside a *DataSet* object. Conceptually it is similar to a database table in a relational database, except that all the data is available at once and there is no concept of a current row. A *DataTable* object has a collection of *DataColumn* objects which define the format of each column in the *DataTable*. The *DataTable* also has a collection of *DataRow* objects through which the data in the table can be accessed and updated.

**DataColumn**

A *DataColumn* object is used to define the format of a column in a *DataTable* in terms of its data type, length, etc. Instances of the *DataColumn* class, one for each column in the table, are added to the collection exposed by the *Columns* property of a *DataTable* to define the table structure.

**DataRow**

A *DataRow* object contains a single record from the *DataTable* to which it belongs. All operations on the data on a record are performed using the *DataRow* object which contains the data. A *DataTable* object has a *Rows* property which exposes a collection of all the *DataRows* in the table. Records can be added to, or deleted from, the *DataTable* by using adding or removing *DataRows* from this collection.

**DataRelation**

Relational databases allow relationships to be defined between multiple tables. These relationships, which are based on the use of foreign keys, are used to ensure the consistency and integrity of data entered in to the database tables. This ability to define relations between items in tables is so useful that it is also included in ADO .NET. A *DataRelation* object can be used to define and enforce the relationship between *DataColumns* in different *DataTable* objects. Proper use of *DataRelations* helps ensure referential integrity and can also provide automatic removal of all related data from a *DataSet* when one item of data is deleted.
**DataView**

A DataView object is used to provide a layer on top of a DataTable object which offers sorting, filtering, and data binding of data in that DataTable instance. Multiple DataView objects can be used to create different views on a single table of data. Sorting and filtering do not change the underlying data, and filtering can use criteria based on the data values in the table.

**The e-Pizzeria Database**

All of the practical exercises in this tutorial are based on a simple Microsoft SQL Server 2000 database used by a hypothetical online pizza retailer. The database contains seven tables and its structure is illustrated in Figure 1. As supplied, this database contains some data in all the tables except for the Orders and OrderLines tables.

In the practical exercises that follow we shall use ADO .NET objects to interact with this database. We shall also create Web services which allow clients to access and update the data in this database.
**Exercise 1: Creating a DataSet object**

A `DataSet` is created by constructing a new instance of the `System.Data.DataSet` class. This class has a default constructor and no arguments are needed. Once you have created the `DataSet`, you can add tables to it by creating `DataTable` instances and adding them to the `DataSet`'s `Tables` property.

A `DataTable` is created by constructing an instance of the `System.Data.DataTable` class. Again there is a default constructor and no arguments are required. The structure of the table must be defined by creating new `System.Data.DataColumn` instances, one for each column in the table, and adding them to the `DataTable`'s `Columns` property. Once the table structure has been specified, the table may be added to the `DataSet` that owns it, using the `DataSet.Tables` property.

Using Visual Studio, create a new C# Windows (not Compact Framework) Application project. Add some code to the main form class to create a private `DataSet` member variable. Add some code to the constructor which instantiates a new `DataSet` and stores a reference to it in the private member variable that you created. Although it won't do much at this stage you should verify that your code compiles correctly.

We are going to implement a `DataSet` to which can be used to store information from the Customers table of our sample database. Create a `DataTable` object called `Customers`; you can pass the table name as a string argument to the `DataTable` constructor. The Customers table has seven columns with data types and lengths as shown in the diagram in Figure 1. Create a `System.Data.DataColumn` instance for each of the columns in the Customers table, setting the data type and length correctly, and add each `DataColumn` to the `Columns` property of the Customers `DataTable`. The code sample in Figure 2 illustrates how the Surname column is added to the Customers `DataTable`. Once you have added all the columns to the table, add the Customers `DataTable` instance to the `Tables` property of your `DataSet` and check that your code still compiles correctly.

```csharp
ePizzeriaData = new DataSet("ePizzeria");
DataTable cust = new DataTable("Orders");
DataColumn surname = new DataColumn("Surname",typeof(string));
cust.Columns.Add(surname);
ePizzeriaData.Tables.Add(cust);
```

**Figure 2: Defining a table structure with DataColumns**

**Exercise 2: Defining Relationships with the DataRelation class**

The `DataRelation` class is used to define relationships between items in different tables. Using `DataRelations` objects to define relationships helps keep items that they remain consistent and correct and allows all
related data items to be automatically removed when a particular data item is removed from a table. A relationship is defined by creating a new instance of the `System.Data.DataRelation` class which references the linked columns in each of the tables. This `DataRelation` instance is then added to the `Relations` collection of the `DataSet` to which the related items belong.

The `DataRelation` class has numerous overloads but the easiest way to construct a relationship is to use the following constructor:

```csharp
DataRelation(string name, DataColumn parent, DataColumn child);
```

All that is required is a name for the relationship and a reference to the two related `DataColumn` instances from the appropriate tables. Figure 3 shows how a `DataRelation` object might be used to define the relationship between the OrderID column of the OrderLines table and the ID column of the Orders table. Note how indexers are used in this example to retrieve members from the `Tables` property of a `DataSet` and the `Columns` property of a `DataTable`. .NET overloads the index operator `[ ]` to allow string indexes to be used with these properties to retrieve the named table or column. This is an extremely powerful feature and one that you will come to rely on when using ADO .NET.

```csharp
DataColumn parent = myDataset.Tables["Orders"].Columns["ID"];
DataColumn child = myDataset.Tables["OrderLines"].Columns["OrderID"];
DataRelation r = new DataRelation("OrderLinesOrderID", parent, child);
myDataset.Relations.Add(r);
```

**Figure 3: Adding a relationship using a DataRelation**

Modify your code from exercise 1 by creating two further `DataTable` instances, one to store the Reviews table and one to store the FoodItems table. Use `DataColumns` to define the structure of these tables to be the same as that shown in Figure 1.

In the database, there is a relationship between the Customers column of the Reviews table and the ID column of the Customers table. Create a `DataRelation` instance which defines this relationship. You need to pass `DataColumn` instances referring to these columns to the `DataRelation` constructor. You can obtain these instances either by maintaining a reference to them in a variable when you construct them, or by using string indexers as shown in the example above.

**Exercise 3: Adding, updating, and deleting data**

A data record is added to a particular table by creating a new instance of the `System.Data.DataRow` class and adding this to the `Rows` collection of the relevant `DataTable` instance. Again the technique of using string indexers can be applied to easily set the values stored in each column of the `DataRow`. Figure 4 shows an example of adding a new row to the OrderLines table. Note that the ID column of the OrderLines table contains a unique ID (in the database, this is the
primary key of the table); we use the Count property of the DataTable’s Rows collection to retrieve the number of rows currently in the table and use this to set the ID value appropriately. However this is not a very good solution; if we subsequently delete a row from the table then add a new row, we may end up with two rows having the same value in the ID column. A better solution would be to define this column of the OrderLines table to be an autoincrement column by setting the AutoIncrement property to true in the DataColumn that defines this column. If the AutoIncrementSeed and AutoIncrementStep values are also set correctly then ADO .NET will automatically create the values in this column for us and we will not need to manually add them to the table. However we should ensure that when we finally upload our data to the database we do not upload the values for the ID column which ADO .NET automatically generated for us. This is because they may conflict with existing unique ID values in the corresponding database table; it is much better to let the database take care of creating unique ID values when we upload the data.

```csharp
DataRow dr = ds.Tables["OrderLines"].NewRow();
int numOrderLines = ds.Tables["OrderLines"].Rows.Count;
dr["ID"] = numOrderLines + 1;
dr["OrderID"] = 12;
dr["Item"] = 2;
dr["Quantity"] = 2;
dr["Subtotal"] = 7.90;
ds.Tables["OrderLines"].Rows.Add(dr);
```

Figure 4: Adding a row a DataTable

Modify the main form of your application to allow a customer to record a review comment about a particular food item sold by the e-Pizzeria. Add two dropdown list boxes and a textbox, with appropriate labels, to the form. The listboxes should display the numeric values in the range 1 to 10 and will be used to enter FoodItem ID numbers and Customer ID numbers. The textbox will be used to enter the review comment. Your form should look something similar to that shown in Figure 5. Place an ‘Add’ button on the form; when this is clicked a new review should be added to the DataSet with the values of the FoodItem, Customer and Review columns being derived from the contents of the form controls.

Add the code behind the Add button to insert a new record into the Reviews table of your DataSet. You should ensure that the ID field of Reviews table has a unique integer value.

Compile your application and start it running in the debugger. Set a breakpoint in the event handler for the Add button’s click method just before the statement which adds the new row in to the Reviews table. Using the debugger’s Autos window you can expand the entry corresponding to your DataSet. By expanding entries under its Tables collection you should be able to find the DataTable instance
corresponding to the Reviews table and, under this, its Rows collection. You should see a new DataRow appearing in the Rows collection. Expand this DataRow instance and satisfy yourself that it contains the correct data.

![Figure 5: Review Form User Interface](image)

**Exercise 4: Finding data in a DataSet**

The `DataTable` class provides a `Select()` method which enables you to find a particular row or rows in the `DataTable`. The `Select()` method is given a single string parameter which is the criteria for the search. The format of the criteria is similar to that of a SQL WHERE clause, without the WHERE keyword and can include criteria based on multiple columns combined with SQL logical operator keywords. The `Select()` method returns an array of `DataRow` objects, one for each row in the table that matches the search criteria. Remember though that there may be no matching rows in the table, in which case a null array will be returned. You should write your code appropriately to deal with this scenario. Figure 6 shows how the `DataTable.Select()` method can be used to retrieve all rows of the OrderLines table for which the FoodItem column contains the value 6.

```csharp
DataRow[] dRows;
string criteria = "FoodItem = 6";
dRows = ds.Tables["OrderLines"].Select(criteria);
```

![Figure 6: Finding rows using DataTable.Select()](image)

You can remove a row from a `DataTable` by obtaining a reference to the `DataRow` instance corresponding to the desired row. You can then call `Remove()` on the `Rows` collection of the table, passing in the relevant `DataRow` instance. Figure 7 shows how to remove all the rows from the OrderLines table for which the FoodItem column contains the value 6.

Add a 'Remove Review' to the simple form application which you created in the previous exercise. When clicked, this button should remove any reviews from the Reviews table which have values in the FoodItem and Customer columns that correspond to the values currently displayed in
the dropdown boxes in the interface. Build your application and add some sample reviews. Test that 'Remove Reviews' removes the rows you expect it to.

```csharp
DataRow[] dRows;
string criteria = "FoodItem = 6";
dRows = ds.Tables["OrderLines"].Select(criteria);

foreach (DataRow dr in dRows)
{
    ds.Tables["OrderLines"].Rows.Remove(dr);
}
```

![Figure 7: Removing rows from a DataTable]

**Exercise 6: Windows Forms Data binding**

One of the most powerful and time-saving features of Windows forms is the ability to bind objects to many Windows Forms controls (e.g. list boxes, combo boxes, data grids etc). This is known as data binding. If the value of an object bound to a control changes, the control immediately updates to display the changed value. Similarly, if the value in the control is changed, the value in the bound object is immediately changed to the same value. Although data binding can be used with all sorts of objects, it is at its most useful when coupled with ADO .NET objects.

Simple data binding can be achieved with any Windows Forms control that has DataSource and DisplayMember properties. An object is bound to the control by assigning a reference to the object to the control’s DataSource property. The control’s DisplayMember property is assigned a string which defines which property of the object should be used to retrieve the value that is displayed by the control (most controls can only display a single value).

We can bind Windows Forms controls to ADO .NET object using the technique described above. With this approach it is possible to bind DataRow instances to controls that display only a single value and DataTable instances to controls that can display multiple values (e.g. combo boxes, data grids). The DataSource member is assigned a reference to the ADO .NET object containing the DataRow or DataTable we want to bind. The DisplayMember property is assigned a string which corresponds to the name of the column of the table/row that we want the control to display and update. When the value(s) in the column of the bound row/table changes, the values displayed by the control change. When the value in the control changes, the bound value in the row/table also changes.

Add code to the constructor of your form application which simply adds some data values into the FoodItems and Customers tables of your dataset. It does not matter what values you enter, we just need some values in these tables to test out the data binding.
Bind the combo boxes which display customers and food items to the FoodItems and Customers tables of your dataset respectively. The food items combo should display values from the Name column of the FoodItems table. The customers combo should display values from the Surname column of the Customers table. The sample code in Figure 8 illustrates how the food items combo box may be bound to FoodItems table and configured to display values from the ID column of that table.

```csharp
FoodItemComboBox.DataSource = ePizzeriaData.Tables["FoodItems"]; FoodItemComboBox.DisplayMember = "Name";
```

Figure 8: Binding the FoodItems combo box to the Name column of the FoodItems table

Compile and run your code. You should now see textual names rather than numeric values appearing in the combo boxes of your form; this is obviously more intuitive for the user and is a great improvement. Now try adding a review using your application. Does it work?

When you created the code behind the ‘Add’ button in exercise 3 you probably used the Text property of the combo boxes to retrieve the numeric value that they were displaying, and used this as the integer value to be inserted into the appropriate column of a new row added to the Reviews table. Now that you have added data binding to the form, the combo boxes display textual values, not numeric values and you probably encounter an exception when you try inserting them into a new row in the table.

The Reviews table expects FoodItems and Customers to be represented as integer values and we need some way to convert the text displayed in the combo box into its corresponding numeric value. We can use the `SelectedItem` property of the `ComboBox` class to retrieve the object that has been selected. We bound a table to each of the combo boxes and we might reasonably expect that each of the objects in the combo boxes is a `DataRow` instance. In fact the data binding layer automatically creates a `DataRowView` instance for each `DataRow` in the bound table and it is these `DataRowView` instances that are the objects in the combo box. Calling `SelectedItem` on the combo box will return a `DataRowView` instance and we can use the Row property of the `DataRowView` class to return the `DataRow` with which it is associated. We can then use the normal string-based column indexers on this `DataRow` to retrieve any of the values we are interested in, in this case the value in the ID column. The code fragment in Figure 9 shows how the `ComboBox.SelectedItem` property is used in the code behind the ‘Add’ button to determine the correct FoodItems ID value to insert into the new row of the Reviews table.
Modify the code behind your 'Add' button so that it uses the
SelectedItem property of the combo boxes to retrieve the correct ID
values for inserting into a new review. Make similar modifications to the
code behind the 'Remove Review' button so that it too obtains the
correct numeric ID's. Compile and test your code again. Does it now
work without exceptions?

Finally, expand the size of your review form and add a DataGrid control
to the foot of it. Set the Reviews table from your DataSet to be the
DataSource for the DataGrid. Compile and test your application again.
Now when you add and remove reviews, you should see the contents of
the Reviews table changing in the DataGrid.

**Connecting to a DataSource**

So far in this tutorial we have seen how to create and use ADO .NET
objects for manipulating data. Although ADO .NET objects can be
created and used in isolation, as we have done in the preceding
exercises, they are really designed to be used when interacting with a
database. ADO .NET objects are used to provide locally cached
copies of data which is held in a database elsewhere on the network.
The cached data is held in a familiar relational format by the ADO
.NET objects and can be manipulated and updated even when there is
no connection to the database server. At a later time, when a network
connection becomes available again, the changes to the cached data
can be merged back in to the data held by the database server.

A mechanism for interacting with a networked database server is first
required before we can cache data in ADO .NET objects. .NET
provides facilities to support interaction with several types of database

- Microsoft SQL Server - System.Data.SqlClient namespace
- Oracle 8i - System.Data.Oracle namespace
- Many other database servers through Open Database Connectivity
  (ODBC), provided a suitable ODBC driver is available – System.Data.Odbc
- Other databases through OLE database connectivity – System.Data.OleDb namespace

```csharp
DataRow dr = ePizzeriaData.Tables["Reviews"].NewRow();
dr["FoodItem"] = ((DataRowView)ItemComboBox.SelectedItem).Row["ID"];
Figure 9: Using Combo Box.SelectedIndex to determine the ID value of a
displayed food item
```

Further information on data binding may be found in *Windows Forms in a Nutshell*, Ian Griffiths & Matthew Adams, O'Reilly, 2003.
Each of these namespaces provides a common set of classes for interacting with databases.

- **Connection** object - represents a connection to data source; used by other objects to interact with data source
- **Command** object - represents queries passed to the database or calls to stored procedures
- **DataReader** object - allows forward-only iteration through rows returned from a query
- **Transaction** object - provides transactional support
- **Parameter** object - represents parameters to queries or stored procedures
- **DataAdapter** object - bridge between a database and the disconnected data objects of ADO .NET. It is used in passing data between ADO .NET objects and data source

Each namespace gives a specific name prefix to each class depending on the database technology it is supporting. For example, the `System.Data.SqlClient.SqlConnection` class is the connection class for a Microsoft SQL database whilst the connection class for an ODBC database is `System.Data.Odbc.OdbcConnection`.

The use of all of these objects is outside the scope of this tutorial. Instead we will consider the Connection, Command and DataAdapter classes as these are most commonly used in passing data to and forth between ADO .NET objects and a network database server. The DataAdapter is particularly important as this is used to fill the ADO .NET objects with data and to update the networked database with changes from the ADO .NET objects.

A database has been provided to use with the remaining exercises in this tutorial. This database is hosted by a Microsoft SQL Server database engine so we will be using classes from the `System.Data.SqlClient` namespace. However the principles are the same whichever database technology you use, the only difference is the actual class names/namespaces used for the database classes.

**Exercise 7: Connecting to a database**

The first step in establishing a connection to the database on the network server is to create a `SqlConnection` object for the connection. The SqlConnection class has methods which allow you to control when you connect and disconnect from the database and a SqlConnection instance is required to create SqlCommand and SqlDataAdapter objects.

When we construct a `SqlConnection` instance we are required to pass a **connection string** to the constructor. The connection string consists of a set of name-value pairs which provide information about the server
which hosts the database, what the database is called, how users are to be authenticated, log-in credentials, etc. For example, the connection string required to connect to the database used for the remaining exercises is

“user id=giovanni; password=giovanni; initial catalog=PizzeriaDB; data source=\"localhost\"”

This string specifies the login credentials for the database user (these must always be UID: giovanni, PWD: giovanni), the server on which the database resides (localhost) and the name of the actual database to be used (PizzeriaDB).

The easiest way to create a database connection is to open the Server Explorer window in Visual Studio. At the top of the windows you will see a Data Connections entry; right-click on this and select ‘Add new connection…’. You will be presented with the dialog box shown in Figure 10. Using this dialog you can specify which server you want to connect to, the required login credentials and the name of the database you wish to use. You may need to select the type of database you are using from the Provider tab before you can correctly configure the connection. Once you have created the connection it will appear under the Data Connections entry in the server explorer and can be used in your applications by dragging it on to form designers.

![Data Link Properties dialog box](image)

**Figure 10: The Visual Studio dialog box used to create database connections**

`SqlCommand` objects are used to execute queries against a database. They can be created programmatically or by dragging a `SqlCommand` component from the Visual Studio toolbox on to a form designer. If you choose to create a `SqlCommand` command in code you pass the text of the SQL query to the `SqlCommand` constructor. You must also specify the database connection to use when executing this command; this can be done either by passing a `SqlConnection` instance as a
constructor parameter or assigning it to the `SqlCommand.ConnectionString` property. If you choose to create a `SqlCommand` by dragging a toolbox component to your form, a new `SqlCommand` item will appear in the component tray of the form. You must set the database connection for this command using the `Connection` property in the properties window. You must also specify the SQL text of the command using the `CommandText` property. If you click the small triangle in the `CommandText` property box, the Query Builder dialog will open. Query Builder allows you to visually design and test SQL queries, which can be useful if you are not very familiar with SQL.

You can perform a command on the database by calling the `Execute()` method of the `SqlCommand` instance that describes the command. You must ensure that you have first called the `Open()` method of the `SqlConnection` object used by the command so that the connection to the database is open and ready for use. When you no longer require the database connection you can use the `Close()` method to close it. Note that this method of operation is the traditional `connected` mode of operation; an open database connection is required to make changes to data in the table.

Modify the review form application to create a database connection to the ePizzeria database which is hosted on the machine you are using. To access this database you must use the user name `giovanni` and the password `giovanni`.

Temporarily comment out the existing code in the event handler for the Add button’s `On_Click` event (Do NOT delete the existing code, you will need it again later). Insert code to create a `SqlCommand` which will add a new row into the Reviews table. The SQL query which inserts the data is as follows:

```sql
INSERT INTO Reviews (FoodItem, Customer, Review)
VALUES ('1', '1', 'Rubbish!')
```

Finally add the code to open the database connection, execute the command and close the connection. Compile and test your code. When you click the 'Add' button you should see a new review appearing in the Reviews table in the database on the local machine.

**TIP:** The SQL Servers element of the Visual Studio Server Explorer windows allows you to inspect SQL Server databases on the local machine. If you expand the Databases element you will see all the databases on the local machine. Expanding the desired database will reveal several other elements, including Tables. Expanding the Tables element will reveal all of the tables that make up the database. Right-clicking on any table name and selecting 'Retrieve data from table' will open a window which shows the data currently held in that table. You can use this facility to check that your `SqlCommand` correctly adds a new row to the Reviews table.
Exercise 8: Using ADO.NET data adapters

The SqlDataAdapter is the bridge between the traditional connected mode of access to network databases and the cached, disconnected mode of operation which ADO .NET objects support.

A SqlDataAdapter is used to retrieve data from a database and cache it locally in ADO .NET objects. To do this, the SqlDataAdapter must make use of a valid database connection and it retrieves the required data by issuing SQL queries against the database. The results of those queries are stored in the ADO .NET objects. Given a DataSet object, that dataset can be filled with copies of data from the database by passing the dataset to the SqlDataAdapter.Fill() method.

A SqlDataAdapter can be created programmatically or by dragging a SqlDataAdapter component from the Toolbox on your form. The latter is by far the easiest. When you drop a SqlDataAdapter on to your form the Data Adapter Configuration wizard appears. Using this wizard you can select the database connection which the data adapter should use and determine whether the adapter should use stored procedures in the database to retrieve data or whether it should send SQL queries to the database each time. In a real application you would probably choose to use stored procedures for efficiency but in our exercises here you should choose to ‘Use SQL statements’. You must also specify the queries that the data adapter will use to retrieve, update, insert and delete data from the database. You can specify this as raw SQL queries, or if you prefer, you may use the graphical query builder to specify and test the queries. The query builder allows you to create the SELECT query used to retrieve data from the database; the wizard then attempts to construct corresponding INSERT, UPDATE and DELETE queries automatically.\(^1\) When you complete the wizard, a SqlDataAdapter component will have been created and added to your form.

Add a SqlDataAdapter to your review application by dragging it from the Toolbox on to your form. You should choose to use the database connection your created in exercise 8 and use the Query Builder to construct a query that returns all columns of the Customers, FoodItems and Reviews tables.

Exercise 9: Retrieving data to a DataSet

We can use the data adapter to copy data from the database in to the ADO .NET objects that we created in earlier exercises. Recall that we created a DataSet and three DataTables; the structure of these tables exactly mirrors the structure of the FoodItems, Customers and Reviews tables in the database. We can copy data from the database

\(^1\) The DataAdapterWizard may warn you about the dangers of storing the database password in the source code. If this is the case, for these exercises simply click the option to include and store the password. In other circumstances you should carefully consider the security risks posed by embedding the password in the code.
into our DataSet by calling the data adapters Fill() method and passing it a reference to our dataset, as shown in Figure 11.

```csharp
   ePizzeriaData = new DataSet("ePizzeria");
   sqlDataAdapter1.Fill(ePizzeriaData);
```

**Figure 11: Filling a DataSet using a DataAdapter**

Note that there is no need for us to explicitly open the database connection, the DataAdapter will take care of controlling the connection on our behalf. It is opened before the data adapter fills the dataset and closed as soon as the fill is complete.

Modify your application by removing the code which populates your DataSet with values. Replace this code with code similar to that in Figure 11 so that data is retrieved from the database into your DataSet. Compile and run your application. Does your review application work as before, showing intelligible values in the FoodItems and Customers combo boxes? If you do not see intelligible values, try using the debugger to peek inside your DataSet after the Fill() operation to see if it contains any data and in which tables.

You will find that your DataSet does indeed contain data but only in the first of the three tables and that each row contains 17 entries. Essentially what happens is that the DataAdapter retrieves all the columns from the tables in the database and inserts them in a single (the first) table in your DataSet. Although the columns came from three distinct tables in the database the DataAdapter cannot automatically determine which tables are the corresponding ones in your DataSet. If you are building a desktop application you can avoid use the Visual Studio Data->Generate DataSet menu command. This creates a strongly typed dataset which automatically creates the correct table mappings by inspecting the schema of the tables in the database.

Unfortunately, strongly typed datasets are not supported by the Compact Framework. If you are developing for this platform you must either set up table mappings (see *Microsoft ADO .NET*, David Sceppa, Microsoft Press, 2002) or use multiple data adapters, one adapter for each table that you wish to retrieve.

Reconfigure your existing SqlDataAdapter and change the query it uses so that it returns just the ID, Surname and Forename fields of the Customers table in the database. You can reconfigure a data adapter by right-clicking on it and selecting the appropriate menu option. Create two further SqlDataAdapter. Configure one to retrieve the ID and Name fields from the FoodItems table and the other to retrieve all the fields from the Reviews table.

In the constructor of your application form add code to call the Fill() method of all three of your data adapters, passing in the same DataSet every time. Your code should appear similar to that in Figure 12. Compile and test your application. This time you should find that all...
three tables of your DataSet are populated with information from the database and that your application now behaves correctly.

```csharp
ePizzeriaData = new DataSet("ePizzeria");
// Fill the dataset from the database using
// the data adapters
daCustomer.Fill(ePizzeriaData);
daFoodItems.Fill(ePizzeriaData);
daReviews.Fill(ePizzeriaData);
```

**Figure 12: Using multiple DataAdapters to retrieve multiple tables**

**Exercise 10: Using a DataSet to update the data source**

When a DataAdapter is created using the wizard, it automatically creates SQL statements to insert, delete and update items in the database tables. These SQL statements are used by the SqlDataAdapter.Update() method to update the information in the database based on changes on the cached data held locally in a DataSet.

In our sample application we would like to be able to update the contents of the Reviews table in the database to reflect reviews that have been added or deleted whilst we have been using our application. All that is required is for us to call the Update() method on the SqlDataAdapter that we initially used to extract the information from the Reviews table, passing it a reference to our DataSet.

Add a new button to your application labelled 'Save Changes'. Add an OnClick event handler for this button and add code to the event handler which uses an appropriate data adapter to merge changes made to the Reviews table in your DataSet back into the database. Build and execute your application. Add and/or remove some reviews then click the 'Save Changes' button. Using the Server Explorer, check that the Reviews table has been updated correctly.

**Using ADO .NET with XML Web Services**

One of the most useful features of the ADO .NET DataSet type is that it is serializable. This means that a DataSet instance can be passed as arguments to and from remoting calls. More importantly, DataSet objects are natively supported by XML Web services. A Web service method can return a DataSet. The DataSet is serialized to XML and passed back to the caller; the caller can use and update values in it. A DataSet can also be passed in as a parameter to a Web method, thus allowing a client to send a modified DataSet back to the Web service. The Web service can then use this DataSet to merge changes back in to the underlying database. This is an extremely attractive technique to use in mobile applications which need to interact with data sources held elsewhere in the network.
As Web services are a cross-platform technology it is possible for a Web method written in C# to populate a DataSet with data from a back-end database and return this to, for example, a Java client running on a Unix platform. The Java client can modify the DataSet and return it to the Web service using another Web method call. The Web method takes the returned DataSet and merges the changes into the back-end database using a DataAdapter as described above. This is a very useful technique for enabling data access in an application that has to support heterogeneous clients.

**Exercise 11: Interacting with a Database using a Web Service**

Using Visual Studio, create a new Web service project and create a single Web method called `GetMenu()` which takes no parameters and returns a DataSet as a result.

Adapt the code you have used in earlier exercises in this tutorial to create the body of the `GetMenu()` method. This method should retrieve all the required information about food items in the Pizzeria database and place them in a DataSet which is passed back as the return value from the Web method.

Compile your Web service and test that it returns a DataSet containing the data values you expect by calling it from a browser on the development machine.

Mobile applications built using the .NET Compact Framework can consume Web services. Create a simple mobile Web service client by creating a new Pocket PC Smart Device Application using Visual Studio. The client should consist of a single form with a DataGrid or other Windows Forms control that can display multiple data values (e.g., ListBox). Add a Web reference to your application that references the Web service you created earlier. Add some code to the main form of your application that causes it to invoke the `GetMenu()` method of the Web service when the form loads, thus retrieving the DataSet containing the menu information. Bind the control of your form to this DataSet so that you can display the data values it contains. If you are unsure how to do this, refer back to the notes from the ‘Creating User Interfaces with Windows Forms’ tutorial. Compile and test your application in the emulator, and verify that it does indeed retrieve and display the expected data values.