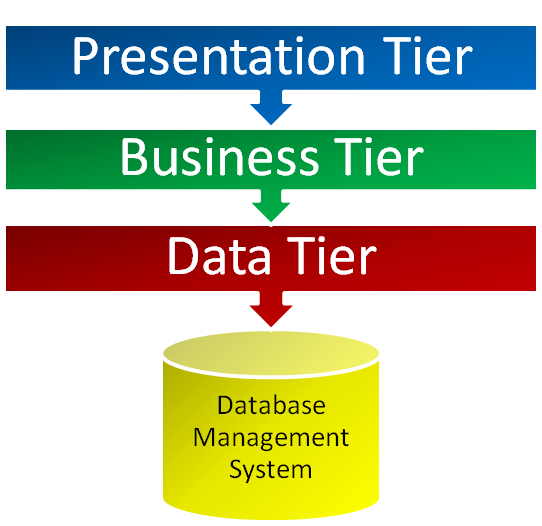
Introduction to Data Tiers in ASP .Net

**By: Prof. Thomas G. Re**

## Introduction to Multi Tier Application development

As you all have learned in class, it is very important to make all applications, including web applications, more portable by dividing the app into separate components. With Business applications, we have three basic components and that is the Presentation Tier, Business Tier and Data Tier of your application. Separating your applications into these three different objects will allow development of different platforms all at once. For example, you will have the ability to develop apps for the Web, Windows desktops, and mobile devices all at one time. Each platform will be a separate presentation Tier or UI (User Interface) to the user but will utilize the same back end by communicating with both the Business and data Tier of your application. You can see the interactions of each of these tiers as illustrated below.



With using the multi tiered approach, each of the tiers within this model can be created in a separate class within your project. These classes will have instance methods and therefore each one will have to be instantiated prior to using them within your program. It is important to note that inorder for each of the tiers to communicate, or share information with each other, proper accessors and mutators are needed in each to acomplish this task. In addition to this, proper constructors are needed in order to initialize the attributes in each of the classes.

As with all Objects in programming, attributes of these classes can also be classes (you will see this later in this documentation). If at this point you have difficulty of understanding objects, attributes, constructors, accessors, mutators and classes, you will need to review topics in object oriented programming prior to continueing with this documentation.

The Presentation tier at the top of this illustration is the UI (User Interface) for your business application. This can be a windows form or an ASP .net form web page. In either case (as you will discover later) they will both be classes within your project. This Tier is the interface between the user of your application and will either display data from a database (as a report) or allow users to enter/change data in a database (as a form). This part of an application is also called the front end of the application while the other two tiers is considered to be the backend.

The Business tier is used to enforce business rules within your application. This may include but is not limited to data validation, and calculated values. Remember, from basic database design, it is poor practices to store calculated values within a database. It is a waste of space and can cause errors when changes within the calculations are made. Examples of calculated values can include sales tax, subtotals, and totals.

For Example when creating an application for a retail store, the sales tax of a sale may change by changing data used for the sales tax calculation. If you stored the calculated sales tax, by changing this data, data integredy may be comprimised. Data may be compromised by changing the data and neglecting the stored caclulation in all of the tables it appears within the database. Therefore it is very important not to store the calculated sales tax inside of the database.

Changes can occur in a sale for the retail store at any time (as with all calulations within a business app). When these changes occur, you will have to make the changes to every location that the tax may have been stored. This is not the case, if all calculated values are done within the application instead of stored within the database. Another example of calculated values within this example can be seen with subtotals and totals of a sale. All of which should be handled on the Application within the Business Tier and not stored within the Database.

The Data Tier is used to interact directly with the Database Management System (DBMS). This is where the connection Strings, Command objects and Queries will be kept for all of your applications. All Queries and stored procedure calls should be done within the Data Tier of your application. This includes queries that are needed from the business tier for making the correct calculations and querries used for searching for specific data for the presentation tier.

Examples of methods that can be created within the Data Tier include, searching for a customer, searching for a product, searching Employees, and providing data for the totals of sales for the Business tier. You should realize at this point, that a lot of the methods that I have just mentioned, are accessors of the data tier object. That is, each of the methods will be returning a value to the calling objects. The mutators, of course, will be the methods that you create that will insert, delete, or change data of the database.

## The web.config file and connection strings

One of the attributes that are most commonly used inside of any Data Tier becomes the Connection String. The connection string will contain all the information that is needed to make the connections between the DBMS and your Data Tiers. This information is stored as a string and includes;

1. The URL of the Database Management System(Data Source) \*redacted\*.ncc.edu
2. The name of the database(Initial Catalog) Hardware Store
3. The User ID – Student1
4. Password – Password

This string in ASP .Net is stored within the web.config file. The web.config is a file that is located in the default web directory and holds all the default configuration settings for your web application. This includes a list of all the Connection Strings (yes you can have more than one) that are used to make all the connections to all of DBMS’s (Database Management Systems) . Each connection string can be accessed from the Data Tier by the name that you assign to that string within this file.

In addition to connection strings, you can have “key values” that can be used for making global settings in your application. The web.config file can also used for setting folder level permissions for your website in ASP .net. In this role, the web.config files act a lot like the .htaccess access files that are often used in Appache and Linux/Unix environments. This of course means that you can have multiple web.config files throughout your website within all of your subfolders.

What is important at this point, is to know where you place the Connection String information within the web.config file in your main web directory in your project. Most of your global settings for your site, as well as all of your connection strings are placed within this file in your web project. The following example illustrates the tags that are required, and where to place the connection string information for your website.

<configuration>

<appSettings >

</appSettings>

<connectionStrings>

<add name="MyData" connectionString ="Data Source=MyServer.MyDomain.com;

Initial Catalog=Instructor;

User ID=MyUserame;

Password=MyPassword"

providerName ="System.Data.sqlClient"/>

</connectionStrings>

<system.web>

<compilation debug="true" strict="false" explicit="true" targetFramework="4.5" />

<httpRuntime targetFramework="4.5" />

</system.web>

</configuration>

From the previous example you can see what may be found inside of a web.config file located in the default directory of your site. Of course when you start with a blank web project you may not have any of the information posted above (this changes between versions of studio). What is consistant, is that you will need to put the information into this file yourself starting with the connectionString tags.

Within the web.config file you will notice the use of “HTML Like” tags to indicate each configuration section of the file. Just like table or list tags in HTML, there is a heirarchy of placing all the tags within this file. You have the configuration Tags and nested inside of the configuration Tags you will have three other sections. One is the appSettings tags which is where the application key values can be placed and again will be explained later. The other is the connectionStrings tags which define all the Connection Strings that you will be using within your project. The last section defined in the above example is the system.web configuration section. This section should be left as it is for the time being and will be explained in more detail in a separate chapter. Please keep in mind that this is a small sample of what can be set within the web.config file and this can be seen with the intellesence of studio as you add the connectionString tag inside of your project.

Within the connectionStrings tag, you will nest the add tags to add a new connection string to the collection. Within the add tag, you will see the different attributes used for defining this tag and therefore each connection string. As stated earlier, the inside of this tag will hold all the information that is needed to make a connection with your DBMS. This information includes the conncection string name, connectionString and providerName.

The name attribute, is the name that you will be using to refer to this connection string within your application. Remember, you can have more than one connection string within your project and therefore you need different (unique) names to differentiate between each connection string. Together, all of your connection strings become a collection, and therefore, each string can be accessed by its name. You must then take care of naming each of your connection strings as to ensure that you do not have duplicate names so that each of the strings can be accessed within that collection.

The connectionString attribute is exactly that, the connection string to the DBMS. The values that are set inside of this attribute is in the form of a string and require the following information (as explained biefly earlier). The Data Source, which is the URL of your DBMS. The Initial Catalog, which is the name of the database you are connecting to. The User ID, which is the id of the user that has access to the database. The Password, which of course will be the password of the user provided in the User ID field. The figure bellow illustrates an example of a connection string that you might use. Of course you will need to insert values that are specific to your DBMS where you find the <> symbols.

"Data Source=<MyServer.MyDomnain.com>;Initial Catalog=<DB Name>;User ID=<MyUserame>; Password=<MyPassword>"

A note to make with the example provided above. Each field within the connection string is seperated by semicolons. This is very important and very similar to values that are set inside of CSS. These semicolons separate each of the fields within the one string and allows the use of spaces within each of the fields. So the space in the fields of “Initial Catalog” and “User ID” must be kept inside of the connection string and the propper use of the semicolons must be followed .

The last attribute of the connectionString tag indicates what provider objects that you are going to use in order to connect to the DBMS. In this example we are using the Microsoft SQL server for our DBMS and therefore we need to indicate that we are using the System.Data.sqlClient library. We will also be indicating this within our Data Tier when we create it. Of course, when you are using other DBMS software, such as Oracle or MySQL, this attribute would change according to the DBMS that you are using.

## Creating the Data Tier for Customers

Up to this point, regardless of the language that you use with your back end of your web application, much of the application is the same. That is, with either C# or VB, the Presentation Tier is basically the same and you will need to have created or know how to create:

1. A Master page and Content page for your site (Explained in a separate tutorial)
2. A default page for your site should be named Default.aspx
3. ASP .net tags are still used to define ASP .net web controls (although the controller code for the Presentation Tier will be different).
4. The web.config file contains your connection strings and other global variables.

The difference is the syntax of each language for creating the actual Data Tier and all of the back end code in your project. For this chapter, we will be creating a Data Tier for the customer table in our database using the C# programming language. In addition to this, all of our examples are going to be using the table provided bellow. We use this example, since most business models contain customers.

|  |  |
| --- | --- |
| **CustomerInformation** | |
| **CustID (PK)** | **int(AutoNumber)** |
| **FirstName** | **varchar(50)** |
| **MiddleName** | **varchar(50)** |
| **LastName** | **varchar(50)** |
| **Address** | **varchar(50)** |
| **Addreess2** | **varchar(50)** |
| **City** | **varchar(50)** |
| **State** | **varchar(50)** |
| **Zip** | **int** |

From the example table provided, you will see that the table name is CustomerInformation, and that it has 9 columns that are defined. The CustID column is a primary key which is an integer that is also an auto number. An auto number is a surrogate key in the table and as each new record is added to the DB table, the DBMS will generate a new unique key for that record in the form of an integer (this information will be important when we create a Data Tier). The data types of each of the columns are also listed and will also be important during the Data Tier creation for adding and updating data to this table.

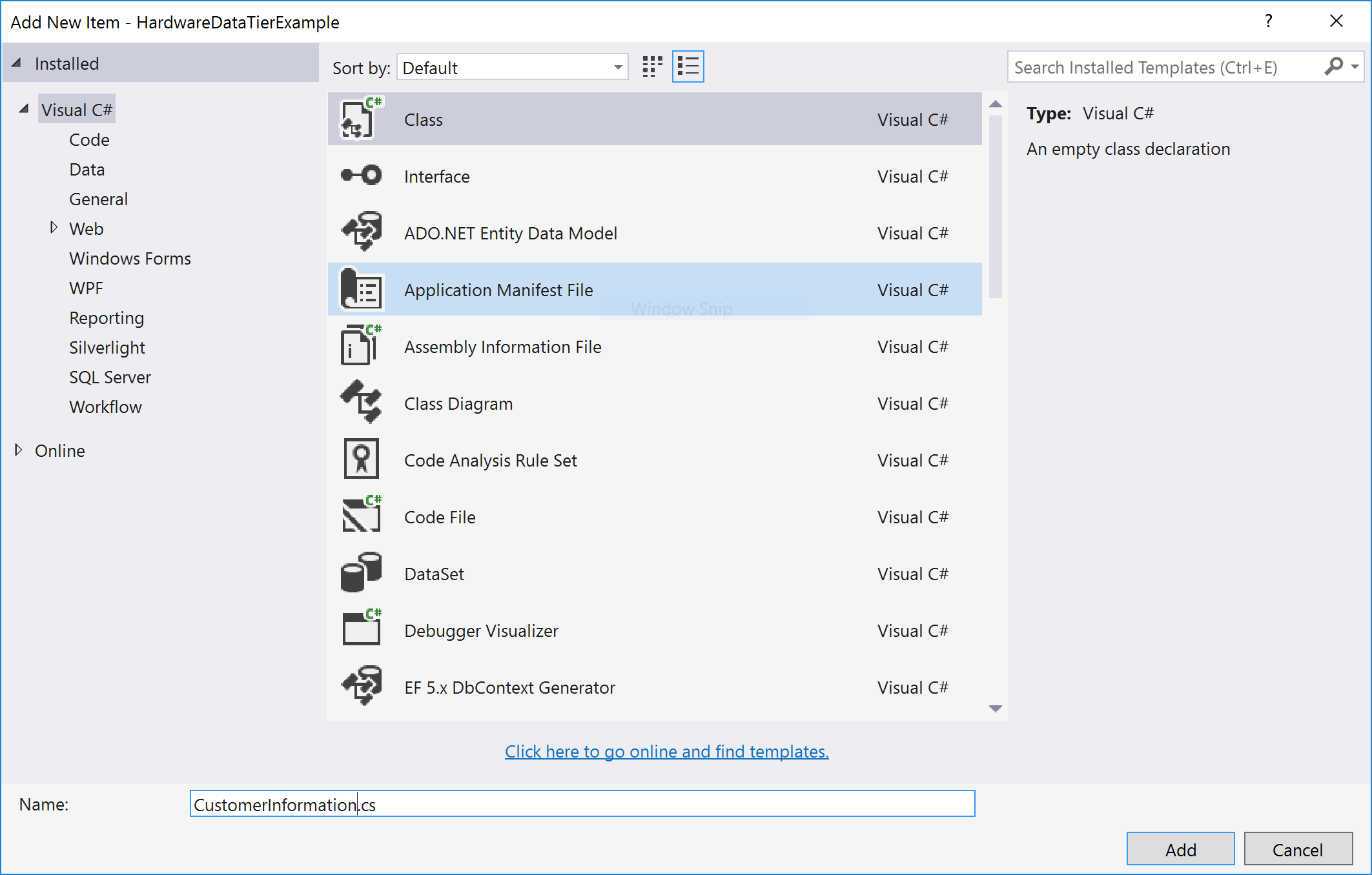
Once we take a look at the table we want to connect to, we need to ask a couple of questions. First, what type of access to this table do we want? Do we want our app to just read data from the DBMS or do we want our app to both read to and write to the database. This can be accomplished by using a Connection String that has only read access to the table or a Connection String that contains a user with both read and write access to the database. This is one of the reasons why you may have more than one Connection String in the web.config file for the same database. Once we have made our decision to read and write to a DB we have some basic methods that are going to be needed for our application’s Data Tier. This includes methods that will contain the following queries.

1. Gets information from the DBMS using a SQL SELECT statement and returns that query results.
2. Insert data into the DB using the SQL INSERT statement and returns true on success.
3. Update data that may have been edited using the SQL UPDATE statement and returns true on success.
4. Delete records from a DB using the SQL DELETE statement, which also returns true on success.

Of course you can also create custom methods within your Data Tier that will be used by your application. These custom methods include searching for specific information within each table within your DB. For example, this can be a method that will search the provided table for customers that live in a specific City. The City name would be a parameter to this function and will enable you to have a search feature, based on city within your Presentation Tier of your application.

A note at this point needs to be made. In order to move forward, the student will need to have a firm understanding of SQL. This will ensure that each student will understand the query strings within the examples provided. Prior to moving any further in this discussion, the student may want to explore additional documentation on Databases and Database Design. If you need more information on Microsoft SQL, you can find material on my website and on the web. You will find power point slides that discuss the normalization process as well as queries that you can use.

To start our Data Tier, we will be creating a class in C# and place it in a special folder that will hold all our Data Tier Classes. In order to do this, you must first create a folder to hold these classes. This is done by right clicking on the project name, then click Add -> Add ASP .Net Folder. Give the name for this folder DBClasses. Then right click on DBClasses ->Add -> Class. A new dialog box opens up like the one provided on the next page.



You will then change the name in the textbox to the name of the Data Tier that you are creating. In this case, we are naming our first Data Tier example, CustomerInformation.cs. Once you click the add button, the class file will open up and look like the following.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace HardwareDataTierExample.DBClasses

{

public class CustomerInformation

{

}

}

From this illustration, you can see that there is nothing here in the file except for the default libraries that are normally created for each class in C# and the naming of the namespace as well as the name of the class. You will have a lot of items to add at this point because all you have is a blank class. All the constructors, accessors, mutators, properties and attributes of the class are placed inside of the code block of the class. Just like like Java, C# uses curly braces to indicate the beginning and ending of its code blocks. You will also find that C#, like Java, will have the same syntax for loops, switch statements, methods, classes, constructors and if statements. The similarities between the languages, makes it easier for Java programmers to learn this new language.

To begin with our Data Tier example, we will have to add some additional libraries to the file so that we can use the appropriate objects for accessing our data in the DBMS. The following libraries should be added to the top of the data tier class. This is done before the **namespace** keyword and typing the **using** keyword for each library. After doing this, your Data Tier class should look like the one provided below.

The System, System.Data, System.Data.Sql, System.Data.SqlClient and System.Data.ProviderBase are all needed for the connecting to the Microsoft SQL Server DBMS. These libraries may be different if you are connecting to another DBMS such as Oracle or MySQL. The System.Cinfiguration library is important because this library will allow us to use tools to retrieve our connectionString information from the web.config file.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Data;

using System.Data.Sql;

using System.Data.SqlClient;

using System.Data.SqlTypes;

using System.Data.ProviderBase;

using System.Configuration;

namespace HardwareDataTierExample.DBClasses

{

public class CustomerInformation

{

}

}

Let’s now add the \_ConnectionString attribute and the constructor to the Data Tier class. We should also add a property that will be able to access or set the connection string if needed. A property in C# will take on the role of both the Accessor and Mutator for the connection String of this class. In doing this, our class should look like the one provided on the next page. As another note, please notice that the constructor of the class is the same name as the class, and it does not have a return type. In addition to this, the keyword, **this** is used to refer to the current instance of the class. These are additional “traits” that are shared between the Java programming language and C#. Again this often makes it easier for Java programmers to make the transition to using the C# language.

public class CustomerInformation

{

private String \_ConnectionString;

public CustomerInformation()

{

this.ConnectionString = ConfigurationManager.ConnectionStrings["MyData"].ToString();

}

public String ConnectionString

{

get {

return this.\_ConnectionString;

}

set {

this.\_ConnectionString = value;

}

}

}

From the above example we need to make the following notes. For one, we are using in the constructor the statement ConfigurationManager.ConnectionStrings[“MyData”].ToString() to get the connection string from the web.sonfig file. The argument “MyData” within this statement is the name of the connection string that we are trying to get. As a reminder, this is a collection (List) of connection strings and the ConfigurationManager.ConnectionStrings[] collection can access each string by its unique name (you can read more on List or collections from any C# text).

When we are done with the constructors, attributes and properties, it is now time to write our first method used to get data from the database. We will call this method getDataSet(). This method will use a SELECT statement to retrieve all the data within the CustomerInformation table and place that data into a DataSet. A DataSet in .Net, is an object that holds a local copy of the data pulled from the database. It holds this local copy as a collection (will explain more about this later) and provides methods we can use within our app to act on that data.

You can also provide a getDataSet() that will return a DataReader rather than a DataSet. A DataReader is a lot like a DataSet, with the exception of only providing a forward only result from the query. This is ideal when using the Data Tier for web applications and displaying that data within an HTML table. For simplicity of this example, however, we will be using DataSets in this example. You can find out more about DataReaders within your text and will be discussed in detail in a later chapter.

Our new method that will be inserted after the property in our class looks like the one provided on the next page. Notice the other objects that are used in this method, the exception handling that is used, the query string and the return type of this new method.

public DataSet getDataSet()

{

DataSet ds;

SqlDataAdapter da;

String Query;

Query = "SELECT \* FROM CustomerInformation;";

da = new SqlDataAdapter(Query, this.ConnectionString);

ds = new DataSet();

try{

da.Fill(ds);

}catch(SqlException ex){

throw new Exception(ex.Message);

}

return ds;

}

With this example, we are using the Query and ConnectionString as arguments within the sqlDataAdapter object. This object in this sense will open the connection of the database and use the Query for its Fill() method. The Fill method of the DataAdapter will “fill” the DataSet with the data retrieved from the Query that was executed. Notice that the exception handling was used when we are filling the data set. When an exception occurs, i.e. a failed connection, then we can handle this exception by throwing a new one up to the Tier that is using this Data Tier (in some cases is the Presentation Tier). It is very important that all ASP .net applications handle exceptions. Otherwise your web application will crash and cause problems for the end user.

The return type of this method is the DataSet. Once the DataSet is successfully filled, it is then returned to the calling method. With this example, our Presentation Tier will get its data from the Data Tier by using the getDataSet() method. The Presentation Tier can then Bind the data to a web control that is going to be using it (will explain this in the in another chapter).

It is important to note that it is important that you fully understand the code provided above prior to moving on. Without understanding this code, you will not be able to write custom methods that will be needed for your individual applications.

Our next method on our list as stated before is the insert method of inserting new records within this database table. With this example we will be using parameterized queries to insert the data. In order for us to that, we will be using two new objects, the sqlConnection object and the sqlCommand object.

The Connection Object is used to make the connection to the database. The Command Object will be used to execute the SQL query. There are many ways to do this. I chose to show you this way so that you can see how you can create a custom mutator or accessor method that can set or return data to or from the database. These examples also use a parameterized query that will use values passed into it from the parameters of the method. In any case we will be using the insert SQL statement within our query string to insert new records into the CustomerInformation table.

Our insert method looks like the one provided on the next page and should be placed after the getDataSet() method. What you will note is that every column except the primary key will be a parameter to this function. This is because the primary key is generated by the DBMS since it is an auto number and therefore needs to be omitted from the query. In addition to this, we have provided this method with a Boolean return type. This return type can be used to determine if the query executed successfully on the DBMS. We can test for this return value within the Presentation Tier of your web application.

public Boolean insertCustomer(String FirstName, String MiddleName, String LastName,

String Address, String Address2, String City, String State, int Zip)

{

Boolean Success = false;

int rows;

String Query;

SqlConnection conn;

SqlCommand cmd;

//This is a standard Query for inserting a row in a database. The multiple //lines bellow are only needed to fit the Query in this documentation. It can //all be placed on one line.

Query = "INSERT INTO CustomerInformation (FirstName, MiddleName, LastName, " +

"Address, Address2, State, City, Zip) VALUES(@FirstName, " +

"@MiddleName, @LastName, @Address, @Address2, @City, @State, @Zip);";

//Now create a connection and command object.

conn = new SqlConnection(this.ConnectionString);

cmd = new SqlCommand(Query, conn);

//place the values passed into this function into the parameters of the querry.

cmd.Parameters.Add("@FirstName", SqlDbType.VarChar, 50).Value = FirstName;

cmd.Parameters.Add("@MiddleName", SqlDbType.VarChar, 50).Value = MiddleName;

cmd.Parameters.Add("@LastName", SqlDbType.VarChar, 50).Value = LastName;

cmd.Parameters.Add("@Address", SqlDbType.VarChar, 50).Value = Address;

cmd.Parameters.Add("@Address2", SqlDbType.VarChar, 50).Value = Address2;

cmd.Parameters.Add("@City", SqlDbType.VarChar, 50).Value = City;

cmd.Parameters.Add("@State", SqlDbType.VarChar, 50).Value = State;

cmd.Parameters.Add("@Zip", SqlDbType.Int).Value = Zip;

//always use exception handling

try

{

//first open a connection to the DB

conn.Open();

//Next execute the query. The number of rows effected is returned

rows = cmd.ExecuteNonQuery();

if (rows <= 0) {

Success = false;

}else {

Success = true;

}

} catch(SqlException ex){

throw new Exception(ex.Message);

} finally {

//Close the connection, regardless if the query is successful

conn.Close();

}

return Success;

}

There is a lot to explain with the above example. For one, there is a specific order in which all of the statements are called within this method which makes a lot of sense. You first have to create a query, and then create a connection object, and then a command object. The reason for this order is that the command object will need both the query and the connection object as arguments to the command objects constructor. This is a lot like the chicken and the egg problem. The Command object is used to execute the query on the DBMS. In order for this object to do so, it needs to be able to make the connection to the DBMS. The Connection object holds all the tools to make this connection to the DBMS. In addition to this, the command object needs the query to execute as well as all the parameters for that query. Without these crucial components, the command object lacks the tools that are needed to effectively communicate with the DBMS.

Now let’s look at the query string in this example. With this query, you have the same type of query as you would see when creating stored procedures in Microsoft SQL Server (You can look up many examples of queries online that can be used within the DataTier). This is your typical parameterized queries where each of the parameters are indicated with the @ symbol. With each parameter that you add to your query, this parameter then needs to be added to the Parameters collection of the sqlCommand object.

You can add parameters to the Parameters collection by using the Parameters.Add() method that exists inside of the sqlCommand object. If you notice in the code provided before, not only do you need to provide the parameter name within the query, but you need to indicate the data type that we expect for each parameter. The data provider for Microsoft SQL server holds all the information concerning the data types that are available for that particular DBMS. The different data types of Microsoft SQL Server is provided to you with the intelli-sense of studio as you start typing these types as one of the argument within this method.

The Success variable is used to indicate whether the method was a success or failure. Sometimes a query may execute successfully and return no value for the number of rows affected. This means that the query executed successfully and no exceptions have been thrown. When this occurs, we need to have a way for our presentation tier to determine if the query executed successfully and effected one or more rows within the DBMS. So our function here will return True if 1 or more rows have been affected and False if 0 Rows have been affected. This true or false value, of course is returned once this method is completed (a query can succeed and not effect any rows within the DB and therefore no exceptions may be thrown).

Now let’s take a look into the exception handling. We start within the Try block, by opening the connection to the DBMS. We then execute the query by calling the method within the command object with the statement, cmd.ExecuteNonQuery(). Notice that we have a Row variable that catches a value from this method. This variable is an integer and it will hold the total number of rows that have been affected by the query. We set the value of success based on the number of rows that have been returned. We are not expecting to receive data from the DBMS and therefore, this is the reason why we are not using a DataSet within this method.

The catch statement within this example will catch any sqlExceptions that may occur. This can be a connection error to the DBMS. We throw a new exception when this error occurs as for our presentation tier to be able to handle it (Our web form). The argument that we use when we throw a new exception is ex.Message which is the object that holds the error that caused the exception. This information that is contained within the Message can be used by the Presentation Tier to determine the cause of the exception.

We use within this example the keyword **finally** for exception handling. Finally is very important, since within this code block, all connections to the DBMS are closed. This will be done whether an exception is thrown or not. Most commercial based DBMS software can be licensed on a per connection bases, and all unused connections should be closed. Other than licensing issues, it is just common best practices, to always close what we have just opened.

Our update method is almost exactly the same as our Insert method. This method of course will not need a DataSet and will need to have a Success value as well. This is because the UPDATE SQL query will either return 0 for the number of rows affected or 1 or more rows for the number of rows that have been affected. Sometimes when you do an update, your query will affect more than one row. When this is the case, you may want to design a method which returns the number of rows affected so that your Presentation Tier can know the exact number of records that have been updated by the query (in contrast of using a Boolean value). An example of the update method is provided on the next page.

From this example, you can see that it is identical to the previous insert method. The only difference to this is the fact that we added the CustID parameter to our method heading and to the parameterized query. This is because with this update method to our DBMS, we need to provide the primary key in order to update a specific record or row of data within the DB. Therefore, from this example, this update method will only affect one row of data every time this query executes. Of course you can, if needed, create an update method that will affect multiple rows of data. This would be a custom query and care should be given when creating such a query.

public Boolean updateCustomer(int CustID, String FirstName, String MiddleName, String LastName,

String Address, String Address2, String City, String State, int Zip)

{

Boolean Success = false;

String Query;

SqlConnection conn;

SqlCommand cmd;

int rows;

Query = "UPDATE CustomerInformation " +

"SET FirstName = @FirstName, MiddleName = @MiddleName, " +

"LastName = @LastName, Address = @Address, Address2 = @Address2, " +

"City = @City, State = @State, Zip = @Zip " +

"WHERE CustID = @CustID;";

conn = new SqlConnection(this.ConnectionString);

cmd =new SqlCommand(Query, conn);

cmd.Parameters.Add("@FirstName", SqlDbType.VarChar, 50).Value = FirstName;

cmd.Parameters.Add("@MiddleName", SqlDbType.VarChar, 50).Value = MiddleName;

cmd.Parameters.Add("@LastName", SqlDbType.VarChar, 50).Value = LastName;

cmd.Parameters.Add("@Address", SqlDbType.VarChar, 50).Value = Address;

cmd.Parameters.Add("@Address2", SqlDbType.VarChar, 50).Value = Address2;

cmd.Parameters.Add("@City", SqlDbType.VarChar, 50).Value = City;

cmd.Parameters.Add("@State", SqlDbType.VarChar, 50).Value = State;

cmd.Parameters.Add("@Zip", SqlDbType.Int).Value = Zip;

cmd.Parameters.Add("@CustID", SqlDbType.Int).Value = CustID;

try

{

conn.Open();

rows = cmd.ExecuteNonQuery();

if (rows <= 0)

{

Success = false;

}

else

{

Success = true;

}

}catch (SqlException ex)

{

throw new Exception(ex.Message);

}

finally

{

conn.Close();

}

return Success;

}

The delete method of your Data Tier is the final method that we will be discussing within this tutorial. This method will require one parameter that is used to delete a unique record within your DB. This parameter of course will be the primary key of the table. With our CustomerInformation example, this primary key is the CustID column of our table. You can however create a delete method that can delete multiple records within your DB, but this can be very dangerous and care must be give when doing so.

As with our previous examples, we will need to use the SQL statements for deleting records from a database. This Statement is provided within the Query string and illustrated in the example bellow.

public Boolean deleteCustomer(int CustID)

{

Boolean Success = false;

String Query;

SqlConnection conn;

SqlCommand cmd;

int rows;

Query = "DELETE FROM CustomerInformation WHERE CustID = @CustID;";

conn = new SqlConnection(this.ConnectionString);

cmd = new SqlCommand(Query, conn);

cmd.Parameters.Add("@CustID", SqlDbType.Int).Value = CustID;

try

{

conn.Open();

rows = cmd.ExecuteNonQuery();

if (rows <= 0)

{

Success = false;

}

else

{

Success = true;

}

}catch(SqlException ex)

{

throw new Exception(ex.Message);

}

finally

{

conn.Close();

}

return Success;

}

Again with this example, we are only returning a Boolean value to indicate success of this method’s execution. Please note, that the delete SQL statement may return more than one row when executed. This may need to be taken into account when creating your own delete methods within your Data Tiers of your application.

In other tutorials, we will explore more of what may be done in the Presentation Tier as well as the Business Tier.