



# enterprise library hands on labs

## lab 1: data access block

After completing this lab, you will be able to:

- Query a database using encrypted configuration settings.
- Use the data access block to read and update a database.

### scenario

This lab demonstrates the use of the Enterprise Library Data Access Application Block. It requires a (local)\SQLEXPRESS instance of SQL Server or SQL Server Express.

**estimated time to complete this lab: 30 minutes.**

### exercise 1: dynamic sql with the data access application block

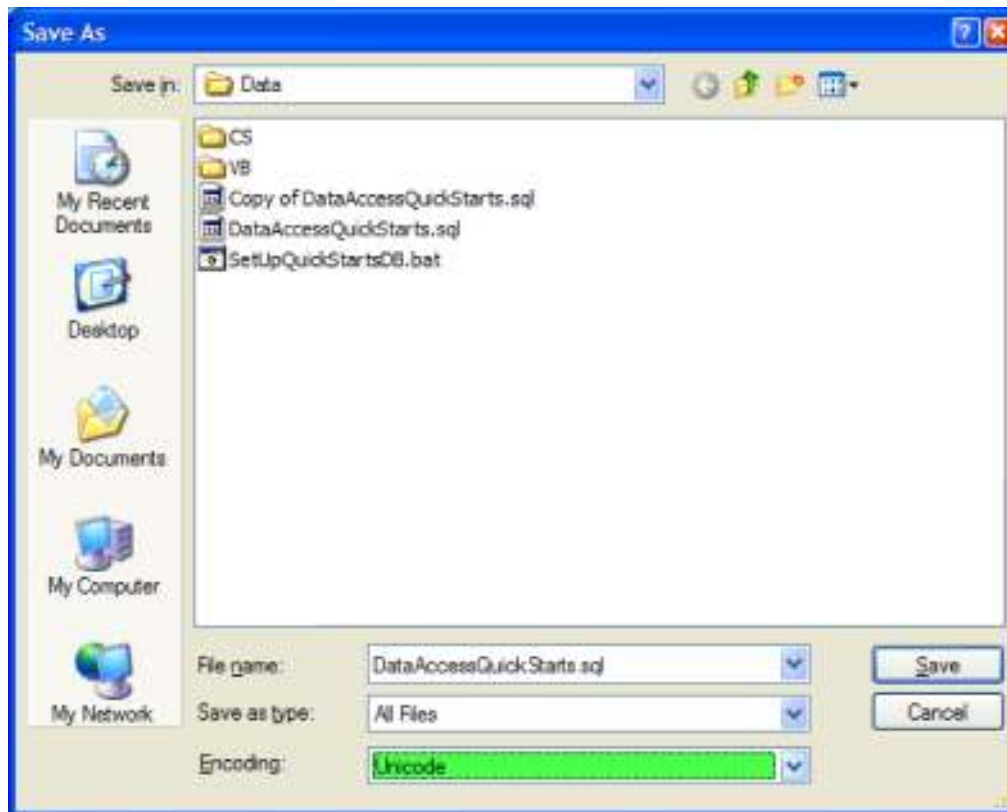
This exercise demonstrates how to do basic database access, using the Data Access Block from the Enterprise Library. It will also show how to configure the block, providing runtime database selection, accessed via an alias in the code.

#### first step

1. Open the [SimpleData.sln](#) file.

#### create the quickstarts database

1. As part of the installation of Enterprise Library, there is a batch file to set up the Quick Starts database. Unfortunately, the deployed .sql file is in ANSI, while OSQL uses the OEM code page. When you run the script, you may find that the data entered into the database has its accents distorted. To fix this problem, re-save the file as UTF16, and then re-run the batch file.
2. Open the file **DataAccessQuickStarts.sql** in **Notepad**, which can be found in the [\[Enterprise Install Dir\]\QuickStarts\Data](#) directory. Choose **File | Save As**. Select **Unicode** as the encoding. Should the Read only attribute warning appear, then select the file in Window Explorer. Choose **File | Properties** and uncheck the **Read only** checkbox at the bottom of the window.



3. Click **Save**, and click **Yes** to overwrite the file.
4. Run the batch file **SetUpQuickStartsDB.bat**, from the same directory.

**Note:** the database will be installed into the **(local)\SQLEXPRESS** instance.

### review the application

1. Select the **MainForm.vb** file in the Solution Explorer. Select the **View | Designer** menu command.

This application contains a DataGridView and some menus. You will implement counting the available customers in the database, and then loading the customers into the DataGridView, using the data access block.

### implement the customer menu items

1. Select the **CustomerManagment** project. Select the **Project | Add Reference ...** menu command. Select the **Browse** tab and select the following assemblies located [here](#):
  - Microsoft.Practices.EnterpriseLibrary.Common.dll, and
  - Microsoft.Practices.EnterpriseLibrary.Data.dll.

*You will need to add the reference to the Common assembly, because the Data Block is instrumented and the IInstrumentationEventProvider is defined in the Common assembly.*

2. Select the **MainForm.vb** file in the Solution Explorer. Select the **View | Code** menu command.
3. Add the following namespace inclusion to the list of namespaces at the top of the file:

```
Imports Microsoft.Practices.EnterpriseLibrary.Data
```

4. Find the **mnuCount\_Click** method (the **Customer | Count** menu click event handler), and add the following code (inserted code in bold):

```
Private Sub mnuCount_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles mnuCount.Click

    ' TODO: Count Customers
    Dim db As Database = Nothing
    db = DatabaseFactory.CreateDatabase("QuickStarts Instance")

    Dim count As Integer = CType(db.ExecuteScalar( _
        CommandType.Text, _
        "SELECT COUNT(*) FROM Customers"), Integer)

    Dim message As String = String.Format( _
        "There are {0} customers in the database", _
        count.ToString())

    MessageBox.Show(Message)

End Sub
```

The code above first obtains an Enterprise Library database instance using the data access configuration for the database instance name "**QuickStarts Instance**" within the configuration file. The real database connection is not opened at this point. The **db.ExecuteScalar** command has multiple overloads. The selected overload allows specifying some literal SQL to execute, and returns the result in a similar manner to the **SqlCommand.ExecuteScalar** method. The **db.ExecuteScalar** method call is responsible for opening and closing the connection to the real database defined in the configuration file, as well as performing any instrumentation required.

5. Find the **mnuLoad\_Click** method (the **Customer | Load** menu click event handler), and add the following code (inserted code in bold):

```
Private Sub mnuLoad_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles mnuLoad.Click

    ' TODO: Load Customers
    Dim db As Database = Nothing
    db = DatabaseFactory.CreateDatabase()

    Dim ds As DataSet = db.ExecuteDataSet( _
        CommandType.Text, _
        "SELECT * From Customers")

    Me.DataGridView1.DataSource = ds.Tables(0)

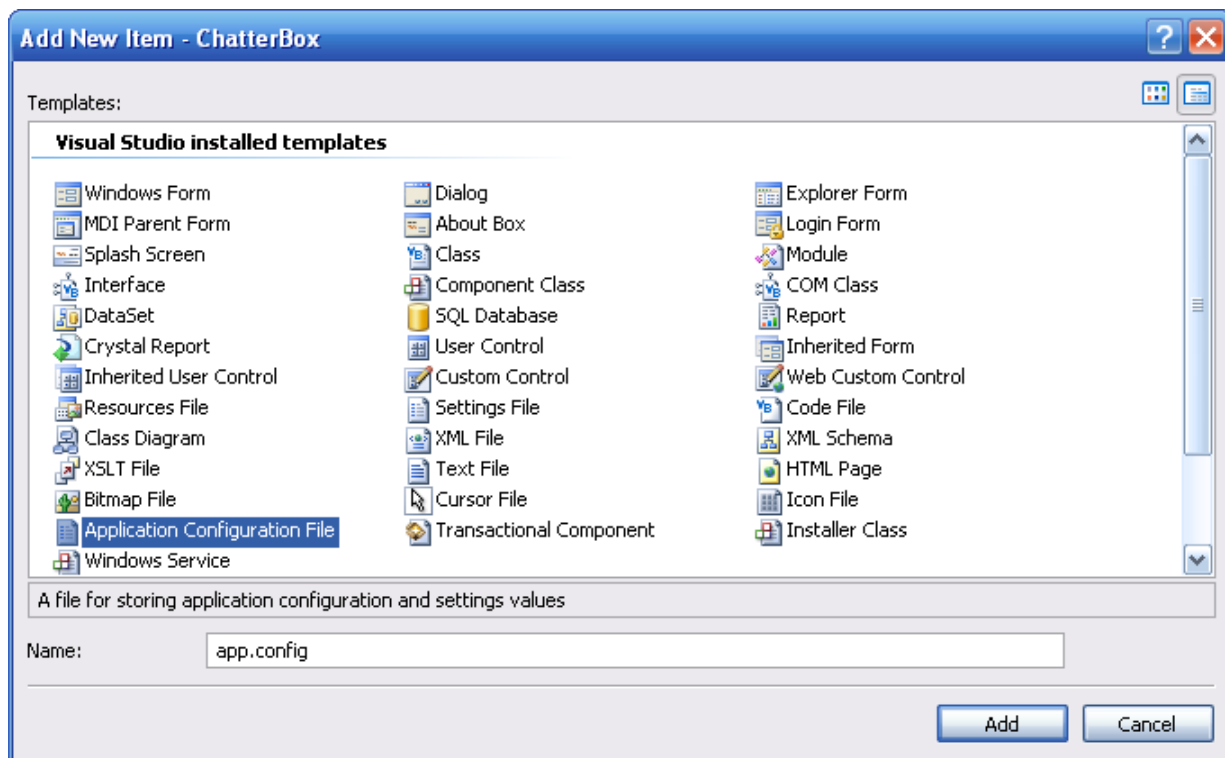
End Sub
```

The **db.ExecuteDataSet** method is responsible for opening and closing the connection, as well as returning a new dataset filled with the results of the SQL Query, which may include multiple tables.

Note: This time we have not passed a database instance name to the CreateDatabase method. Rather the default database defined in the application configuration file will be used.

## configure the application

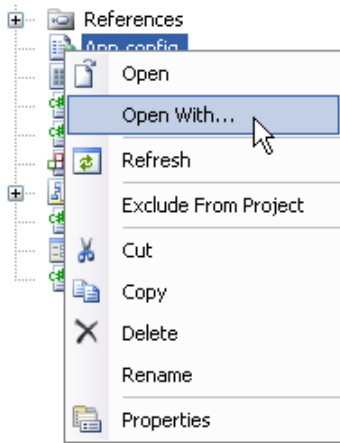
1. Add a new Application configuration file (App.config) to the **CustomerManagement** project. Click on the **CustomerManagement** project. Select the **Project | Add New Item...** menu command. Select **Application configuration file** template. Leave the Name as **App.config**.



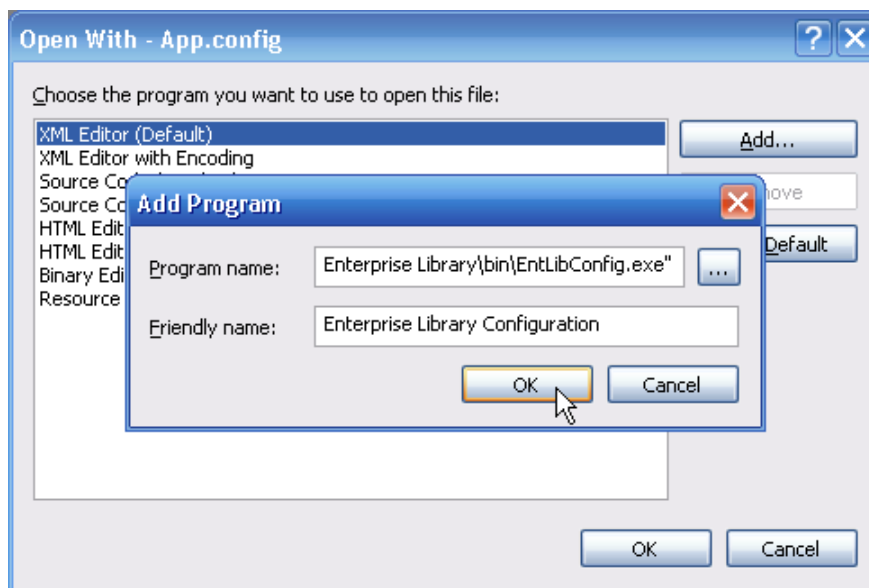
2. We will use the Enterprise Library Configuration tool to configure our application. You may either start this tool from the Windows Start menu (select **All Programs | Microsoft patterns and practices | Enterprise Library | Enterprise Library Configuration**) and open the App.config file located [here](#).

Alternatively you may configure Visual Studio to open the configuration file with the tool, as described below.

3. Select the **App.config** file in the Solution Explorer. Select the **View | Open With...** menu command. The **OpenWith** dialog is displayed. Click the **Add** button.

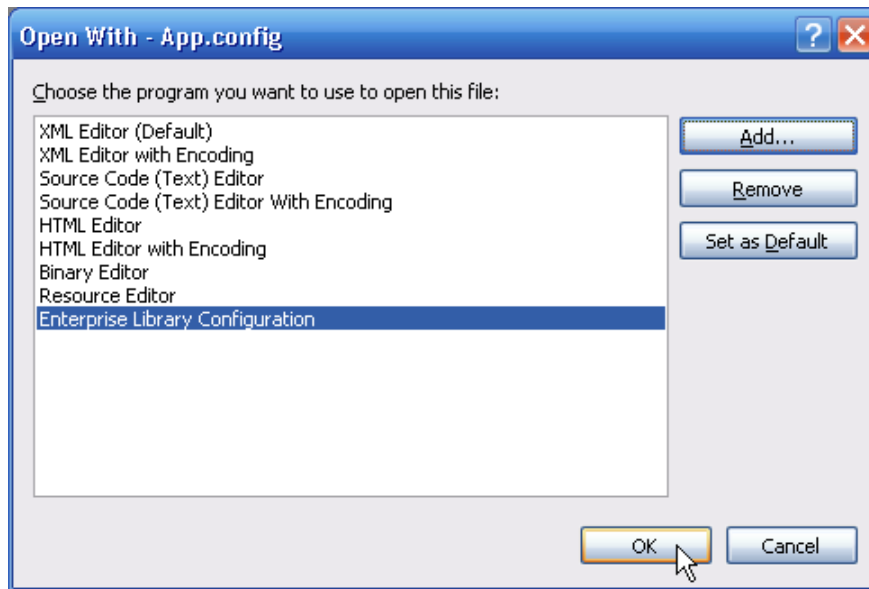


4. In the **Add Program** dialog, set the **Program name** to the **EntLibConfig.exe** file found [here](#). Set the **Friendly name** to **Enterprise Library Configuration**. Click the **OK** button.



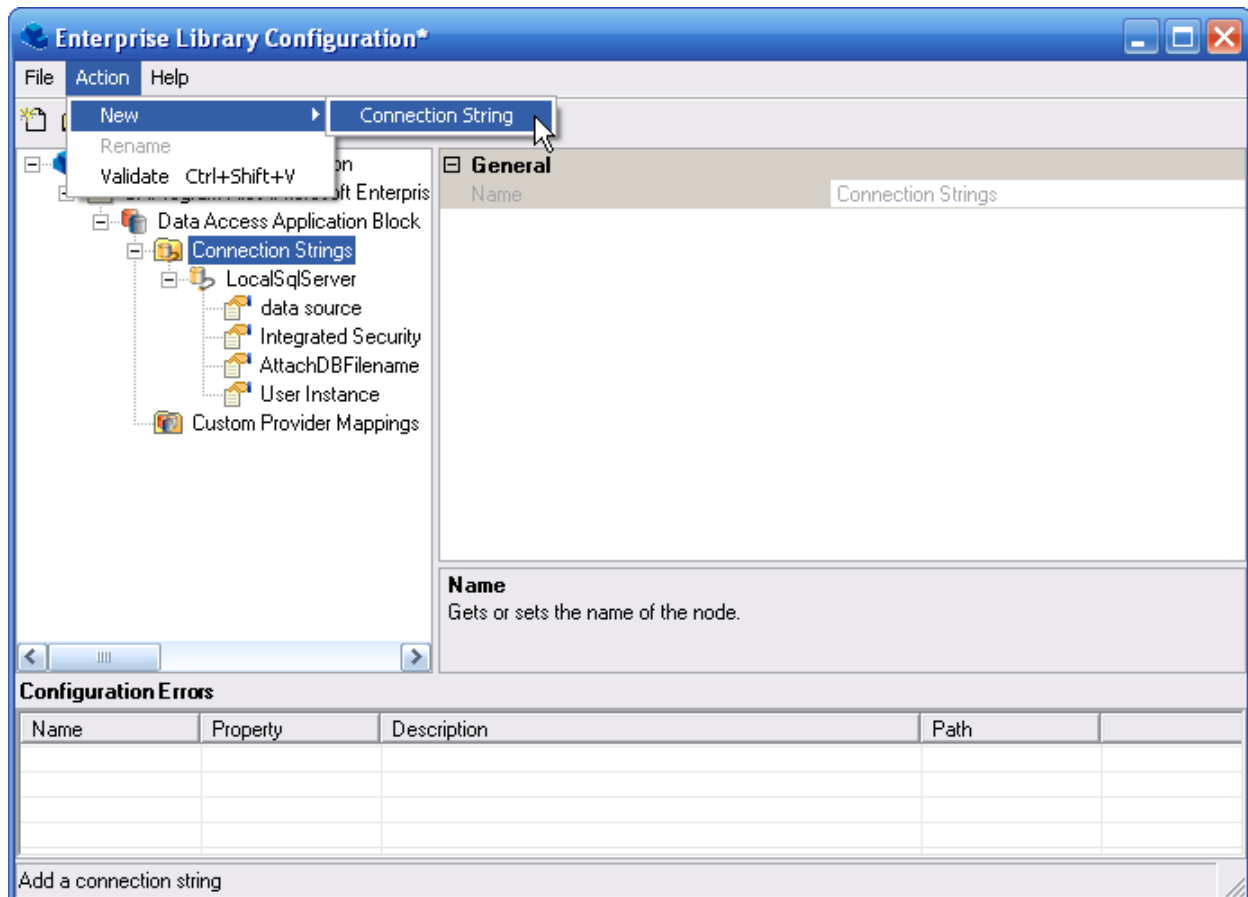
*Visual Studio will pass the configuration file (App.config) to the EntLibConfig.exe program as a command line parameter.*

5. In the **Open With** dialog, select the newly entered **Enterprise Library Configuration** program and click the **OK** button.

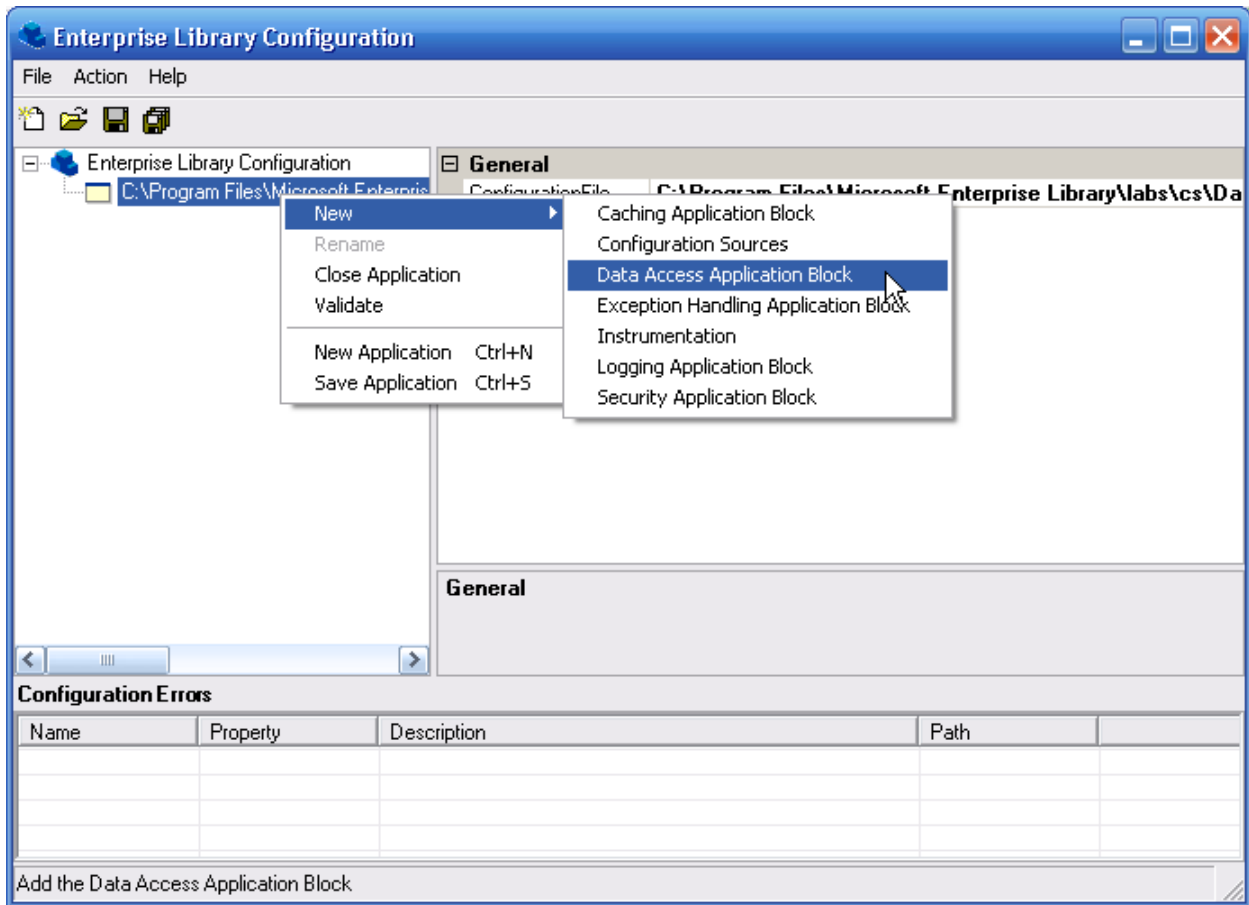


6. If you have a **connectionStrings** section defined in your **machine.config** file, you will find that the **Enterprise Library Configuration** tool automatically creates a Data Access Application Block for you.

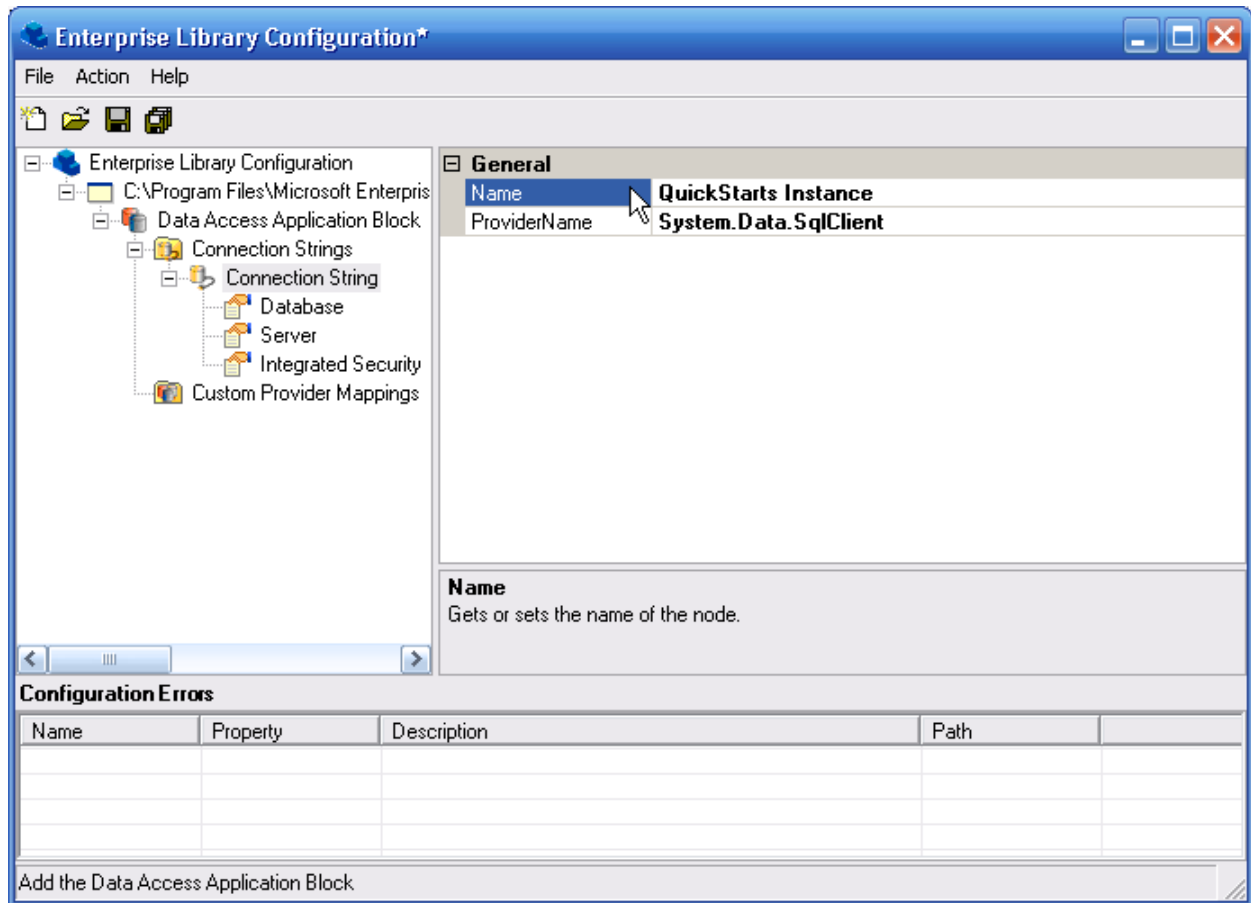
If so select the **Data Access Application Block | Connection Strings** node and select the **Action | New | Connection String** menu command.



- Alternatively, if you do not have a Data Access Application Block defined, right click on the Application and select **New | Data Access Application Block**.

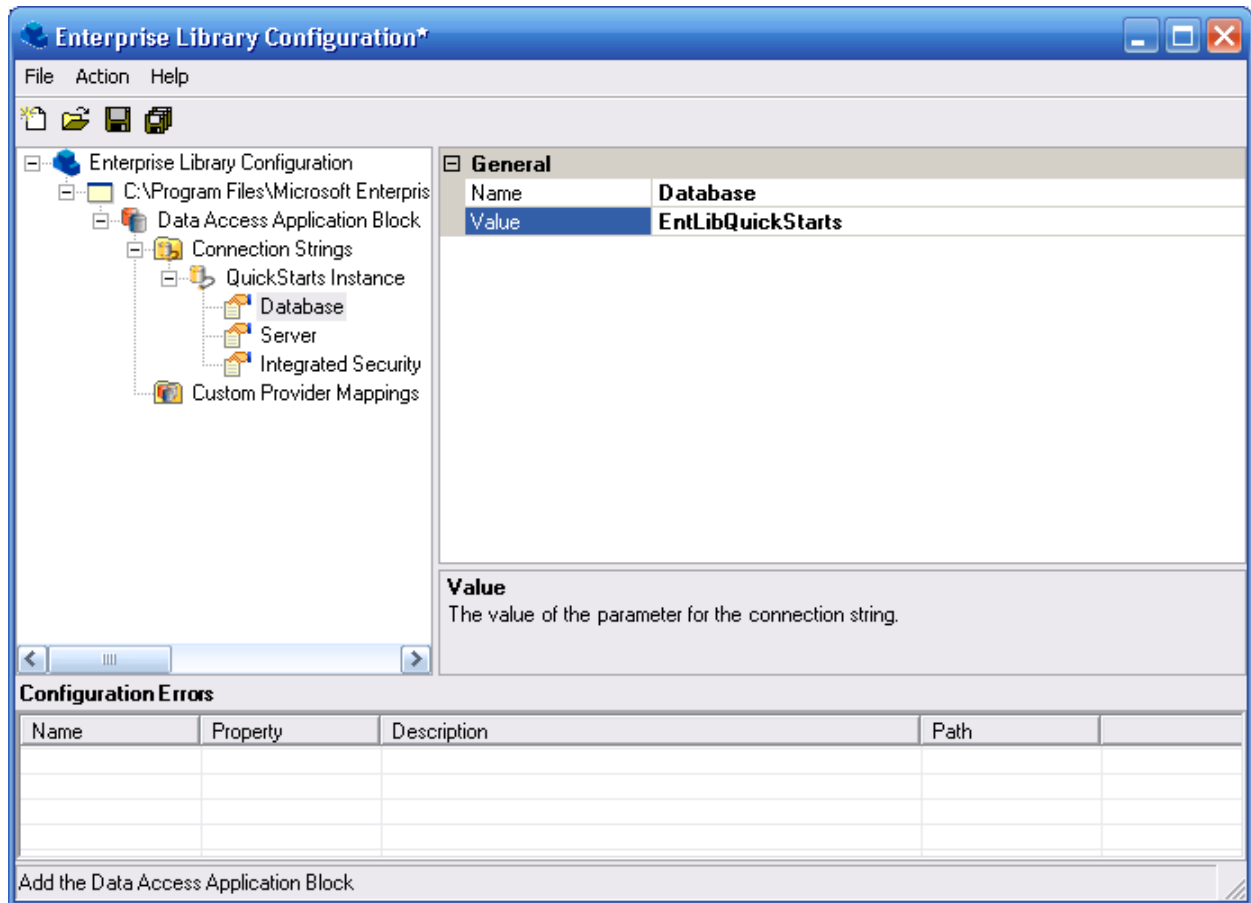


- Select the **Data Access Application Block | Connection Strings | Connection String** node. Change the **Name** property to **QuickStarts Instance**.

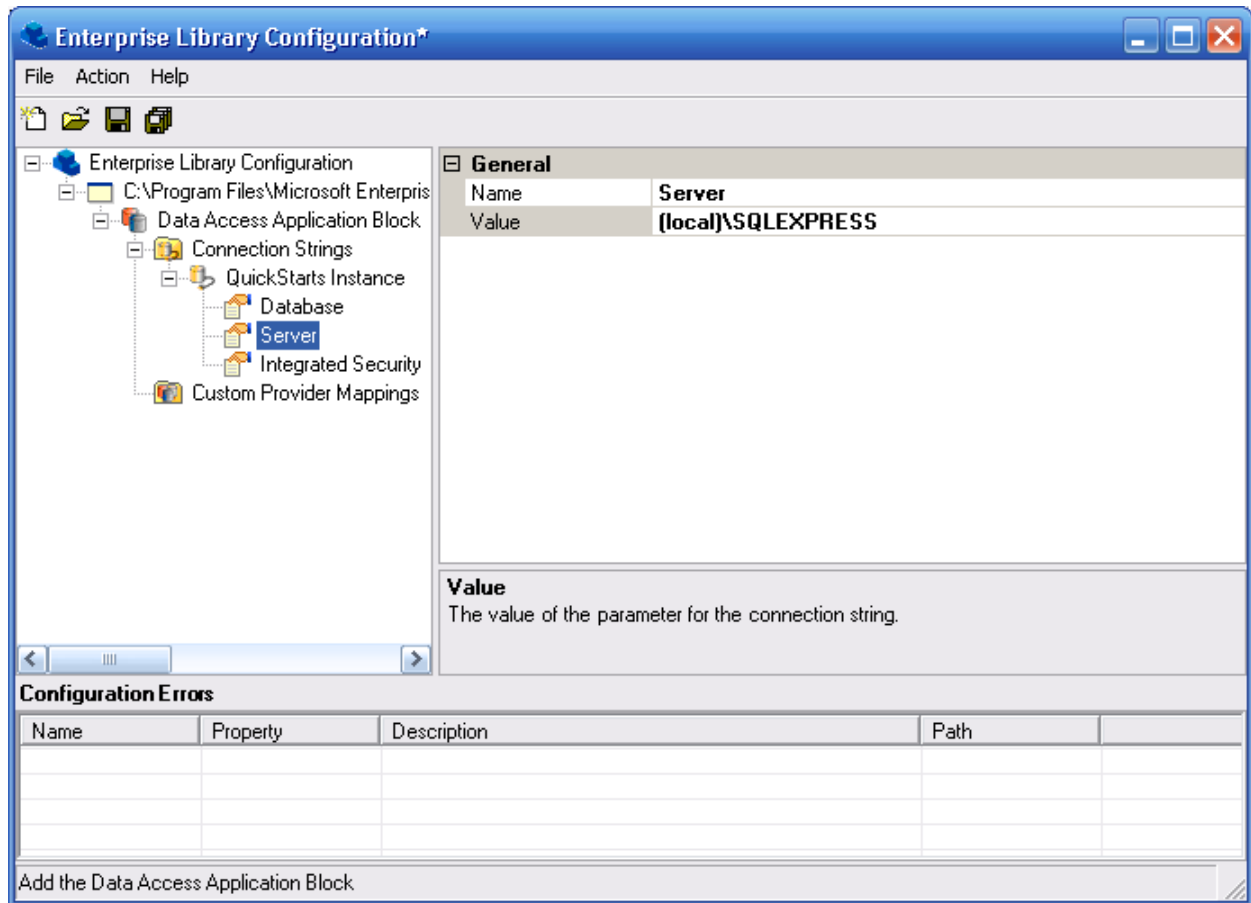


*This name has to match the name you used in the code. This is how you can create an alias in code that maps to the concrete database instance you wish to use at runtime.*

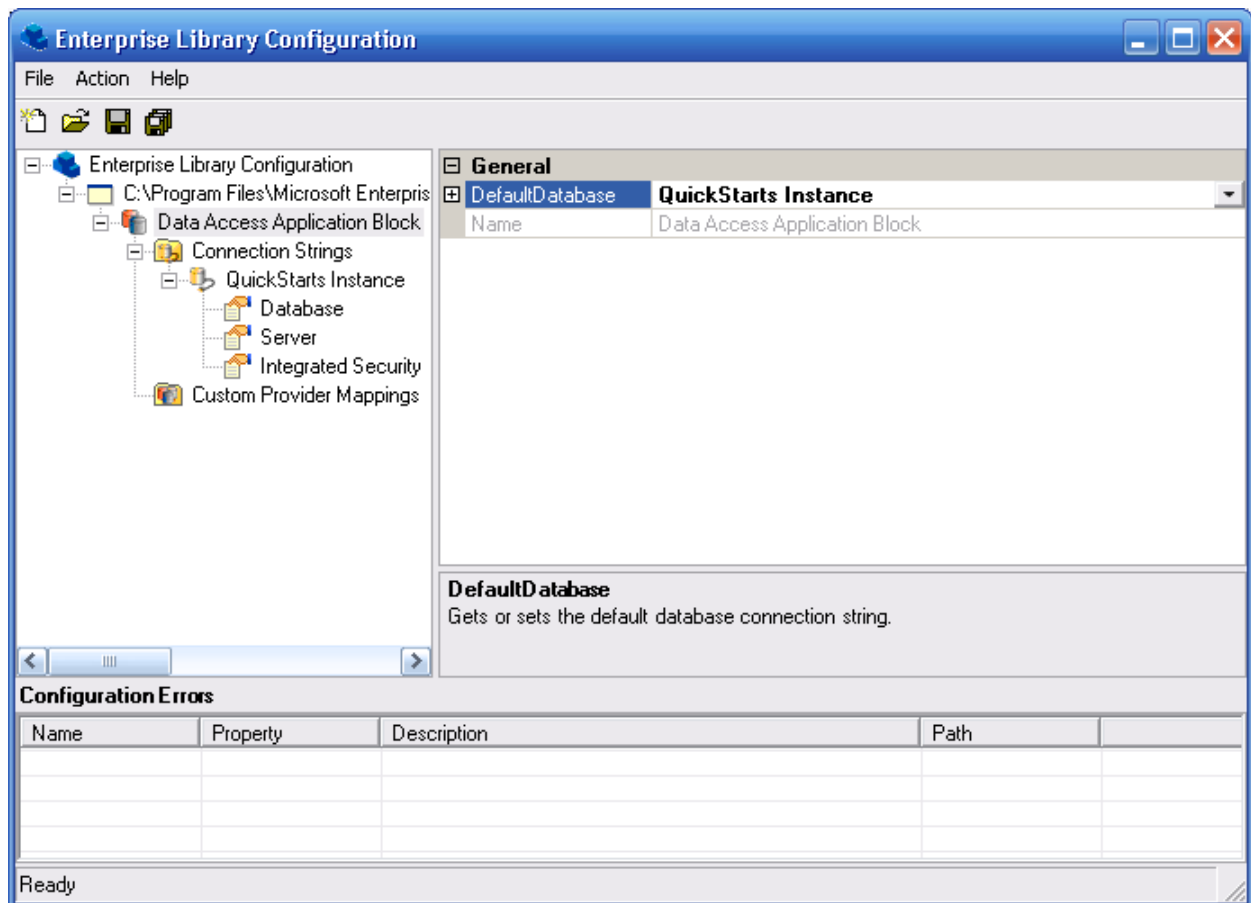
9. Select the **Database** node for this connection string. Change the **Value** property on the right hand side to **EntLibQuickStarts**.



- Similarly, select the **Server** node, and set its **Value** to "**(local)\SQLEXPRESS**".



11. Select the **Data Access Application Block** node. Set the **DefaultDatabase** property is to the **QuickStarts Instance**.



12. Select the **File | Save All** menu command to save the application configuration. Close the Enterprise Library Configuration tool.

### run the application

1. Select the **Debug | Start Without Debugging** menu command to run the application. Click on **Yes** in the dialog box warning about the **app.config** being modified outside the source editor.

Select the **Customers | Count** menu command to view the number of customers in the database, and use the **Customers | Load** menu command to fill the DataGrid.

2. Close the application and Visual Studio .NET.

To check the finished solution, open the [SimpleData.sln](#) file.

## exercise 2: stored procedures and updates with the data access block

This exercise demonstrates using the data access block to wrap stored procedure access, as well as performing updates using a strongly typed DataSet.

### first step

1. Open the [DataEx2.sln](#) file.

## add the categories table to the quickstarts database

1. Run the batch file **SetUpEx02.bat**, which can be found in the lab directory: [labs\cs\Data Access\exercises\ex02\DbSetup](#).

*This adds a set of categories that you can use to filter the set of products you retrieve and manipulate.*

**Note:** the modifications will be made to the **(local)\SQLEXPRESS** instance.

## review the application

1. Select the **MainForm.vb** in the Solution Explorer. Select the **View | Designer** menu command.

This application will allow us to select a particular category, load the products for that category, and then save any changes you make.

## implement database retrieval

1. Select the **MainForm.vb** in the Solution Explorer. Select the **View | Code** menu command. Add the following namespace inclusion to the list of namespaces at the top of the file:

The required references to the Data and Common assemblies have already been added.

```
Imports Microsoft.Practices.EnterpriseLibrary.Data
```

2. Add the following private field to the form, as you'll re-use this database instance in multiple places.

```
' TODO: Create private field for Database  
Private _db As Database = DatabaseFactory.CreateDatabase("QuickStarts Instance")
```

**Note** that you can hold onto this, as the Database instance does not represent an open connection, but instead represents a reference to a database. If you had a **SqlConnection** object instead, then you would not be complying with the *Acquire Late, Release Early* model.

3. Find the **MainForm\_Load** method, and insert the following code to obtain the set of categories using a DataReader:

```

Private Sub MainForm_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    cmbCategory.Items.Clear()

    ' TODO: Use a DataReader to retrieve Categories
    Using dataReader As IDataReader = _db.ExecuteReader
("GetCategories")
        ' Processing code
        While (dataReader.Read())
            Dim item As Category = New Category( _
                dataReader.GetInt32(0), _
                dataReader.GetString(1), _
                dataReader.GetString(2))
            cmbCategory.Items.Add(item)
        End While
    End Using

    If (cmbCategory.Items.Count > 0) Then
        cmbCategory.SelectedIndex = 0
    End If

End Sub

```

One of the overloads of the **Database.ExecuteReader** method takes a string, and an optional set of parameters. This overload will locate the stored procedure given by the string, read its meta-data (and cache it for future use), and set parameters with the values of any arguments provided.

**Note:** You are not doing any connection management here, but it is very important to Dispose the data reader returned. This is accomplished by the using statement in the code above. When the data reader is disposed, the underlying DbConnection is also closed.

4. Find the **cmbCategory\_SelectedIndexChanged** method and insert the following code, which will read a subset of the products, based on the selected category drop down.

```

Private Sub cmbCategory_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
cmbCategory.SelectedIndexChanged

    dsProducts.Clear()

    Dim selectedCategory As Category = CType(cmbCategory.SelectedItem,
Category)
    If (selectedCategory Is Nothing) Then
        Return
    End If

    ' TODO: Retrieve Products by Category
    _db.LoadDataSet( _
        "GetProductsByCategory", _
        dsProducts, _
        New String() {"Products"}, _
        selectedCategory.CategoryId)

End Sub

```

There are two methods on the Database class that populate a DataSet; **ExecuteDataSet** and **LoadDataSet**. **ExecuteDataSet** returns a newly created DataSet, while **LoadDataSet** populates an existing one. The overload used here takes a stored procedure as the first argument, the dataset to populate, a set of tables to map the result of the procedure into, and an arbitrary number of arguments. The additional arguments are mapped to any stored procedure arguments retrieved from the database metadata.

## implement updating the database

1. Find the **btnSave\_Click** method. Insert the following code, which will update the database based on any changes in the dataset.

```
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnSave.Click

    ' TODO: Use the DataSet to update the Database
    Dim insertCommand As DbCommand = Nothing
    insertCommand = _db.GetStoredProcCommand("HOLAddProduct")
    _db.AddInParameter(insertCommand, "ProductName", _
        DbType.String, "ProductName", DataRowVersion.Current)
    _db.AddInParameter(insertCommand, "CategoryID", _
        DbType.Int32, "CategoryID", DataRowVersion.Current)
    _db.AddInParameter(insertCommand, "UnitPrice", _
        DbType.Currency, "UnitPrice", DataRowVersion.Current)

    Dim deleteCommand As DbCommand = Nothing
    deleteCommand = _db.GetStoredProcCommand("HOLDeleteProduct")
    _db.AddInParameter(deleteCommand, "ProductID", _
        DbType.Int32, "ProductID", DataRowVersion.Current)
    _db.AddInParameter(deleteCommand, "LastUpdate", _
        DbType.DateTime, "LastUpdate", DataRowVersion.Original)

    Dim updateCommand As DbCommand = Nothing
    updateCommand = _db.GetStoredProcCommand("HOLUpdateProduct")
    _db.AddInParameter(updateCommand, "ProductID", _
        DbType.Int32, "ProductID", DataRowVersion.Current)
    _db.AddInParameter(updateCommand, "ProductName", _
        DbType.String, "ProductName", DataRowVersion.Current)
    _db.AddInParameter(updateCommand, "CategoryID", _
        DbType.Int32, "CategoryID", DataRowVersion.Current)
    _db.AddInParameter(updateCommand, "UnitPrice", _
        DbType.Currency, "UnitPrice", DataRowVersion.Current)
    _db.AddInParameter(updateCommand, "LastUpdate", _
        DbType.DateTime, "LastUpdate", DataRowVersion.Current)

    Dim rowsAffected As Integer = _db.UpdateDataSet( _
        dsProducts, _
        "Products", _
        insertCommand, _
        updateCommand, _
        deleteCommand, _
        UpdateBehavior.Standard)

End Sub
```

When updating a database, it is required to manually create the wrappers around the stored procedures, as it needs to know the mapping between the columns in the DataTable and the stored procedure arguments.

With this overload of the **UpdateDataSet** method, it is possible to get the Data Access Block to execute all the updates transactionally, by setting the **UpdateBehaviour** to Transactional. For more information, see the Enterprise Library documentation.

## run the application

1. Select the **Debug | Start Without Debugging** menu command to run the application.

Select a category from the drop down, and observe how it loads and saves the data.

## more information

1. For more information on designing and implementing data access layers, and knowing whether to create strongly-typed data access layers (via code generation) versus weakly typed data access layer (e.g. the data access block) see the "Developing Microsoft® .NET Applications Using Code Generation, UI Process and Abstraction" course which is part of the *Mastering Industrial Strength .NET* series. This uses Enterprise Library as one of the options for abstracting data access, as well as implementing code generation to generate transaction aware data access layers.

To check the finished solution, open the [DataEx2.sln](#) file.

## exercise 3: encrypting connection information

In this exercise, you will encrypt the configuration to prevent 'connection string discovery' by someone copying the application configuration file.

### first step

1. Open the [DataEx3.sln](#) file.

### encrypt the database configuration

1. The .NET Framework version 2.0 natively supports protected configuration via the Configuration namespace.

You can use protected configuration to encrypt sensitive information, including user names and passwords, database connection strings, and encryption keys, in an application configuration file. Encrypting configuration information can improve the security of your application by making it difficult for an attacker to gain access to the sensitive information even if the attacker gains access to your configuration file.

Encryption and decryption is performed using a **ProtectedConfigurationProvider** class. The following list describes the protected configuration providers included in the .NET Framework:

- **DPAPIProtectedConfigurationProvider**. Uses the Windows Data Protection API (DPAPI) to encrypt and decrypt data.
- **RsaProtectedConfigurationProvider**. Uses the RSA encryption algorithm to

encrypt and decrypt data.

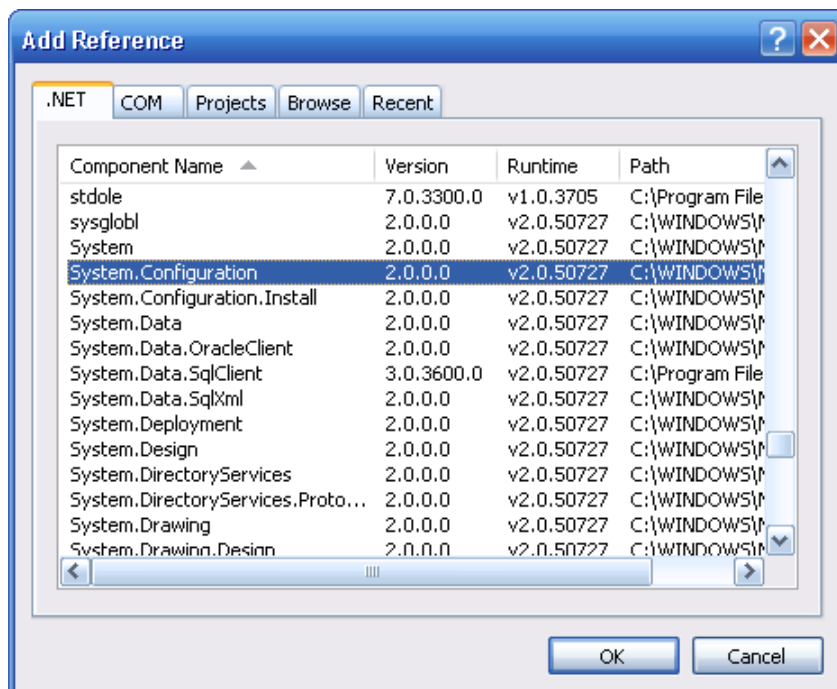
Both providers offer strong encryption of data; however, if you are planning to use the same encrypted configuration file on multiple servers, such as a Web farm, only the RsaProtectedConfigurationProvider enables you to export the encryption keys used to encrypt the data and import them on another server.

By default, RSA key containers are tightly protected by NTFS access control lists (ACLs) on the server where they are installed. This improves the security of the encrypted information by restricting who can access the encryption key.

**Note:** Encrypted configuration information is decrypted when loaded into the memory that is used by your application. If the memory for your application is compromised, the sensitive information from your protected configuration section might be compromised as well.

2. Select the **ProductMaintenance** project. Select the **Project | Add Reference ...** menu command. Select the **.NET** tab and select the following assembly.

- System.Configuration.dll.



3. Select the **Program.vb** file in the Solution Explorer. Select the **View | Code** menu command. Add the following namespace inclusion to the list of namespaces at the top of the file:

```
' TODO: Use the Configuration namespace
Imports System.Configuration
```

4. Add the following code to the **ProtectConfiguration** method.

```

Private Shared Sub ProtectConfiguration()
    ' TODO: Protect the Connection Strings
    Dim provider As String = "RsaProtectedConfigurationProvider"

    Dim config As Configuration = Nothing
    config = ConfigurationManager.OpenExeConfiguration
    (ConfigurationUserLevel.None)

    Dim section As ConfigurationSection = config.ConnectionStrings

    If ((section.SectionInformation.IsProtected = False) AndAlso _
        (section.ElementInformation.IsLocked = False)) Then

        ' Protect (encrypt) the "connectionStrings" section.
        section.SectionInformation.ProtectSection(provider)

        ' Save the encrypted section.
        section.SectionInformation.ForceSave = True
        config.Save(ConfigurationSaveMode.Full)
    End If
End Sub

```

### run the application

1. Select the **Debug | Start Without Debugging** menu command to run the application.

Note that it behaves the same as in the previous exercise.

2. Open the **ProductMaintenance.exe.config** file in the project bin\Debug directory located [here](#). Note the connection strings are encrypted.

To check the finished solution, open the [DataEx3.sln](#) file.

---

Copyright © 2005 Microsoft. All Rights Reserved.