

Data Access Application Block Hands-On Lab for Enterprise Library

patterns & practices

proven practices for predictable results

This walkthrough should act as your guide for learning about the Enterprise Library Data Access Application Block and for practicing how to leverage its capabilities in various application contexts. This lab demonstrates requires a (localDB)\v11.0 instance of SQL Server. Microsoft SQL Server 2012 Express **LocalDB** is an execution mode of SQL Server Express targeted to program developers.

The following five labs are included in this set:

- [Lab 1: Dynamic SQL with the Data Access Application Block](#)
- [Lab 2: Stored Procedures and Updates with the Data Access Application Block](#)
- [Lab 3: Encrypting Connection Information](#)
- [Lab 4: Retrieving data using Data Accessors with the Data Access Application Block](#)
- [Lab 5: Performing asynchronous data access with the Data Access Application Block](#)

The estimated completion time for this hands-on lab is **60 minutes**.

Authors

These Hands-On Labs were produced by the following individuals:

- Program Management: Grigori Melnik and Madalyn Parker (Microsoft Corporation)
- Development/Testing: Fernando Simonazzi (Clarius Consulting), Chris Tavares (Microsoft Corporation), Mariano Grande (Digit Factory), and Naveen Pitipornvivat (Adecco)
- Documentation: Fernando Simonazzi (Clarius Consulting), Grigori Melnik, Alex Homer, Nelly Delgado and RoAnn Corbisier (Microsoft Corporation)

In this release of Enterprise Library, the recommended approach to creating instances of Enterprise Library objects is to use the programmatic approach and instantiate the objects directly. You may decide to store some configuration information externally (such as in a custom configuration section or in the Windows Azure service settings) and use this information when you create the Enterprise Library objects programmatically. In this way, you can expose just those settings that you want to make available, rather than all the settings, which is the case when you use declarative configuration. Each block is self-contained and does not have dependencies on other blocks for basic operations. Typically, creating an instance is a simple operation that takes only a few lines of code.

You could use Unity, or another dependency injection container, to manage your Enterprise Library objects and their dependencies with the corresponding lifecycles. Unlike in the previous release, it is now your responsibility to register and resolve the types you plan to use. Unity 3 now supports the registration by convention to make it easier to do so. See the [Developer's Guide to Dependency Injection Using Unity](#) for more info.

However, to simplify the examples and make it easier to see the code that uses the features of each of the Enterprise Library Application Blocks, the Hands-On Labs examples use the simpler approach for resolving Enterprise Library objects from the container by using the **GetInstance** method of the container service locator. You will see this demonstrated in each of the examples.

To learn more about using dependency injection to create instances of Enterprise Library objects, see the documentation available on MSDN at <http://msdn.microsoft.com/entlib/>.

Lab 1: Dynamic SQL with the Data Access Application Block

This lab demonstrates how to perform basic database access by using the Data Access Application Block from the Enterprise Library. It also shows how to configure the application block and provide run-time database selection. The database is accessed through an alias in the code.

To begin this exercise, open the DataEx1.sln file located in the ex01\begin folder. Ensure an instance of SQL Server 2012 Express LocalDB is installed on your computer.

To create the QuickStarts database

1. Run the batch file SetUpQuickStartsDB.bat located in the **DataAccess\setup** subfolder.

The database will be installed in the (localDB)\v11.0 instance.

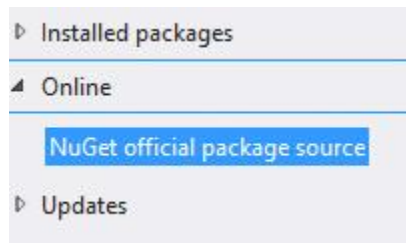
To review the application

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

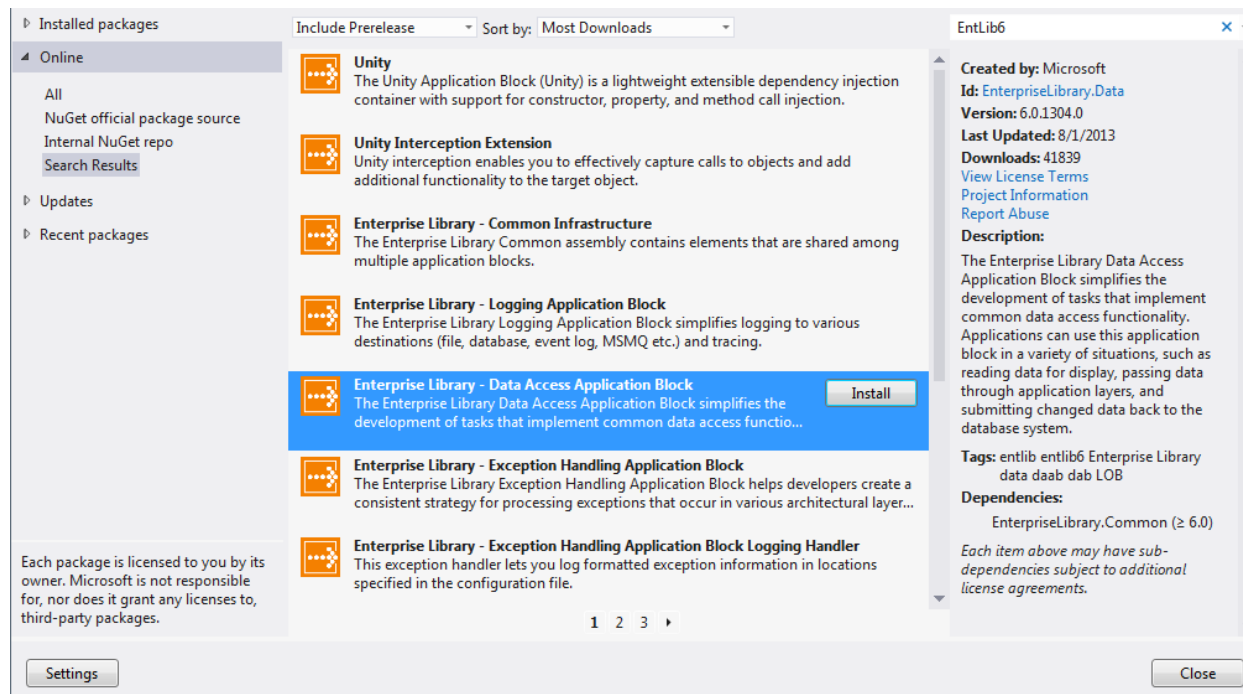
This application contains a **DataGrid** and several menus. You will implement the functionality for counting the available customers in the database, and then loading the customers into the **DataGrid**, using the Data Access Application Block.

To implement the customer menu items

1. Select the **CustomerManagement** project. Select the **Project | Manage NuGet Packages** menu command.
2. Select the “Online” option to view NuGet packages available online.



3. Search for **EntLib6** in the search bar. Select **Enterprise Library – Data Access Application Block** and click install.



4. Click **I Accept** on the License Acceptance window that pops up.

To add a database

1. Select the **MainForm.cs** file in the Solution Explorer. Select the **View | Code** menu command.
2. Add the following namespace inclusions to the list of namespaces at the beginning of the file:

```
using Microsoft.Practices.EnterpriseLibrary.Data;  
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

3. Find the **mnuCount_Click** method (the **Customer | Count** menu click event handler), and add the following bold and highlighted code.

```
private void mnuCount_Click(object sender, System.EventArgs e)  
{  
    // TODO: Count Customers  
  
    DatabaseProviderFactory factory = new DatabaseProviderFactory();  
    Database db = factory.Create("QuickStarts Instance");  
  
    int count = (int)db.ExecuteScalar(  
        CommandType.Text,  
        "SELECT COUNT(*) FROM Customers");  
  
    string message = string.Format(  
        "There are {0} customers in the database",  
        count.ToString());
```

```
    MessageBox.Show(message);  
}
```

The above code first obtains an Enterprise Library database instance by using the data access configuration for the database instance named **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 you to specify some literal SQL to execute, and returns the result in a manner similar 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 for performing any required instrumentation.

4. Find the **mnuLoad_Click** method (the **Customer | Load** menu click event handler), and add the following bold and highlighted code.

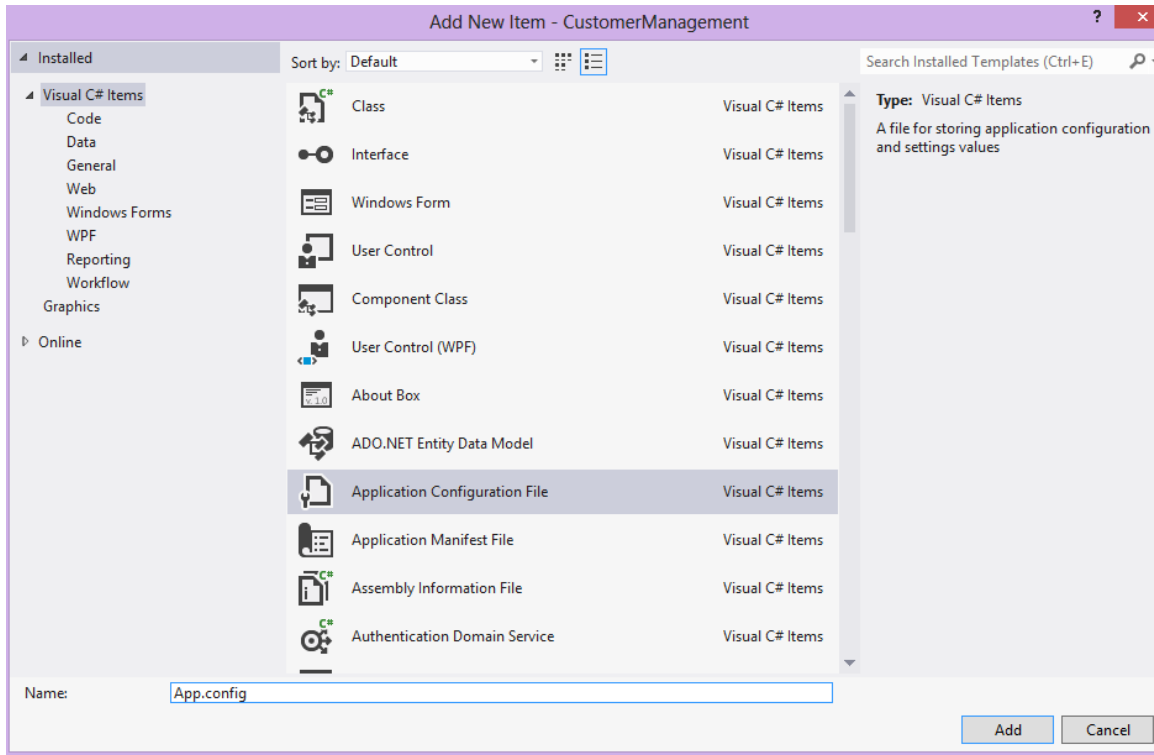
```
private void mnuLoad_Click(object sender, System.EventArgs e)  
{  
    // TODO: Load Customers  
  
    DatabaseProviderFactory factory = new DatabaseProviderFactory();  
    Database db = factory.CreateDefault();  
  
    DataSet ds = db.ExecuteDataSet(  
        CommandType.Text,  
        "SELECT * From Customers");  
  
    dataGrid1.DataSource = ds.Tables[0];  
}
```

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

In this exercise you have not passed a database instance name to the **CreateDatabase** method. Rather, the default database defined in the application configuration file will be used.

To configure the application declaratively

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. Open the **App.config** file. To start out, you should see the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

3. Inside the **configuration** tags, add the text highlighted and bold below:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="dataConfiguration" type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
Microsoft.Practices.EnterpriseLibrary.Data" requirePermission="true"/>
  </configSections>
  <connectionStrings>
    <add name="QuickStarts Instance" connectionString="Database=EntLibQuickStarts;Server=(localDB)\v11.0;Integrated
Security=SSPI;" providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <dataConfiguration defaultDatabase="QuickStarts Instance"/>
</configuration>
```

The **section** is named “dataConfiguration” and uses the Data Application Block you previously included. The **connectionStrings** section specifies databases to be used. You have added one with the **name** property set to “QuickStarts Instance,” the **connectionString** property set to

"Database=EntLibQuickStarts;Server=(localDB)\v11.0;Integrated Security=SSPI;" and the providerName set to "System.Data.SqlClient."

Finally the defaultDatabase is set to "QuickStarts Instance," the one you just added to the configuration file.

4. Save the application configuration.
-

To run the application

1. Select the **Debug | Start Without Debugging** menu command or press Ctrl+F5 to run the application without debugging enabled.
 2. 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**.
 3. Close the application and Visual Studio.
-

To verify that you have completed the exercise correctly, you can use the solution provided in the ex01\end folder.

Lab 2: Stored Procedures and Updates with the Data Access Application Block

This lab demonstrates how to use the Data Access Application Block to wrap stored procedure access and how to perform updates using a strongly typed **DataSet**.

To begin this exercise, open the DataEx2.sln file located in the ex02\begin folder.

To add the Categories table to the QuickStarts database

1. Run the batch file, **SetUpEx02.bat**, which can be found in the following lab directory:
ex02\DbSetup.

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

The modifications will be made to the **(localDB)\v11.0** instance.

To review the application

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

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

To implement database retrieval

1. Select the **MainForm.cs** in the Solution Explorer. Select the **View | Code** menu command. Add the following namespace inclusions to the list of namespaces at the beginning of the file. The required references to the **Data** and **Common** assemblies have already been added.

```
using Microsoft.Practices.EnterpriseLibrary.Data;  
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

2. Add the following private fields to the form. You will reuse this database instance in multiple places.

```
private static DatabaseProviderFactory factory = new DatabaseProviderFactory();  
private Database _db = factory.Create("QuickStarts Instance");
```

You can maintain a reference to this database instance because it does not represent an open connection; instead it 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 wherein you would acquire resources as late as possible, immediately before you need to use them, and release them immediately after you are finished with them.

3. Find the **MainForm_Load** method, and insert the following bold and highlighted code to obtain the set of categories using a **DataReader**.

```
private void MainForm_Load(object sender, System.EventArgs e)  
{  
    this.cmbCategory.Items.Clear();  
  
    // TODO: Use a DataReader to retrieve Categories
```



```

using (IDataReader dataReader = _db.ExecuteReader("GetCategories"))
{
    // Processing code
    while (dataReader.Read())
    {
        Category item = new Category(
            dataReader.GetInt32(0),
            dataReader.GetString(1),
            dataReader.GetString(2));
        this.cmbCategory.Items.Add(item);
    }
}

if (this.cmbCategory.Items.Count > 0)
    this.cmbCategory.SelectedIndex = 0;
}

```

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 metadata (and cache it for future use), and set parameters with the values of any arguments provided.

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 bold and highlighted code which will read a subset of the products, based on the selected category drop-down list.

```

private void cmbCategory_SelectedIndexChanged(object sender,
                                           System.EventArgs e)
{
    this.dsProducts.Clear();

    Category selectedCategory = (Category)this.cmbCategory.SelectedItem;
    if (selectedCategory == null)
        return;

    // TODO: Retrieve Products by Category
    _db.LoadDataSet(
        "GetProductsByCategory",
        this.dsProducts,
        new string[] { "Products" },
        selectedCategory.CategoryId);
}

```

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.

To implement updating of the database

1. Add the following namespace inclusion to the list of namespaces at the beginning of the file.

```
using System.Data.Common;
```

2. Find the **btnSave_Click** method. Insert the following bold and highlighted code to update the database based on any changes in the dataset.

```
private void btnSave_Click(object sender, System.EventArgs e)
{
    // TODO: Use the DataSet to update the Database
    DbCommand insertCommand = null;
    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);

    DbCommand deleteCommand = null;
    deleteCommand = _db.GetStoredProcCommand("HOLDeleteProduct");
    _db.AddInParameter(deleteCommand, "ProductID",
        DbType.Int32, "ProductID", DataRowVersion.Current);
    _db.AddInParameter(deleteCommand, "LastUpdate",
        DbType.DateTime, "LastUpdate", DataRowVersion.Original);

    DbCommand updateCommand = null;
    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);

    int rowsAffected = _db.UpdateDataSet(
        this.dsProducts,
        "Products",
        insertCommand,
        updateCommand,
        deleteCommand,
        UpdateBehavior.Standard);
}
```

When updating a database, you must manually create the wrappers around the stored procedures, because the database needs to know the mapping between the columns in the **DataTable** and the stored procedure arguments.

With this overload of the **UpdateDataSet** method, you can set the Data Access Application Block to execute all the updates transactionally, by setting the **UpdateBehavior** to **Transactional**. For more information, see the [Enterprise Library](#) documentation.

To run the application

1. Select the **Debug | Start Without Debugging** menu command or press Ctrl-F5 to run the application without debugging enabled.
 2. Select a category from the drop-down, and observe how it loads and saves the data.
-

To verify that you have completed the exercise correctly, you can use the solution provided in the ex02\end folder.

Lab 3: Encrypting Connection Information

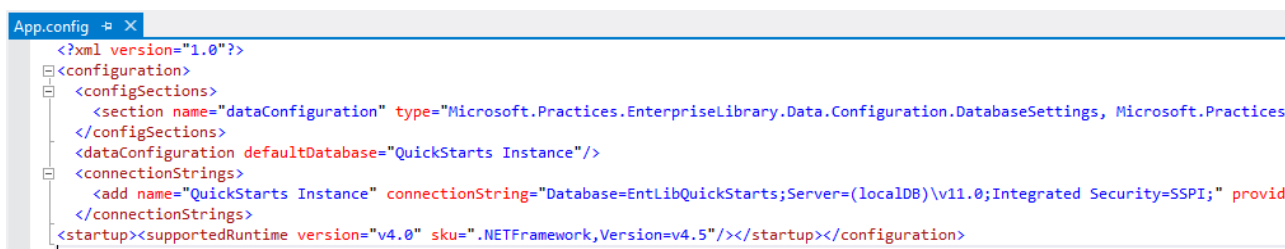
Enterprise Library supports encryption of configuration information. In this lab, you will encrypt the configuration to prevent connection string discovery by an adversary copying the application configuration file. To learn more about encryption of configuration information, see [Appendix B - Encrypting Configuration Files of the Enterprise Library Developer's Guide](#).

Before you start this exercise, you should complete the setup instructions which consist basically of downloading the [Config Console v6](#). There are no additional lab specific setup instructions for this lab.

To begin this exercise, open the DataEx3.sln file located in the ex03\begin folder.

To review the unencrypted configuration file

1. Select the **App.config** file in the Solution Explorer.
2. Select the **Open** command in the context menu. The connection string in the configuration file is plain text and easily readable.



```
App.config  X
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="dataConfiguration" type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings, Microsoft.Practices.EnterpriseLibrary.Data" />
  </configSections>
  <dataConfiguration defaultDatabase="QuickStarts Instance"/>
  <connectionStrings>
    <add name="QuickStarts Instance" connectionString="Database=EntLibQuickStarts;Server=(localDB)\v11.0;Integrated Security=SSPI;" provider="System.Data.SqlClient" />
  </connectionStrings>
  <startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/></startup></configuration>
```

3. Close the file.

To encrypt the database configuration

1. Open the application configuration in the configuration editor :
 - In Visual Studio, right-click on the **App.config** file in Solution Explorer and click **Edit configuration file v6**.
2. Click the chevron icon to the right of the **Database Settings** title to show the properties for this section if they are not already visible. Set the **Protection Provider** property to **DataProtectionConfigurationProvider**.

Database Settings

Protection P...: DataProtectionConfigurationProvider

Require Per...: True

Default Data...: QuickStarts Instance

Database Instances

LocalSqlServer

QuickStarts Instance

Name: QuickStarts Instance

Connection...: <Hidden>

Database Pr...: System.Data.SqlClient

3. Save the application configuration, and close the file.

To review the encrypted configuration file

1. Select the **App.config** file in the Solution Explorer.
2. Select the **Open** command in the context menu. The configuration information is now encrypted.

```

1 <?xml version="1.0"?>
2 <configuration>
3   <configSections>
4     <section name="dataConfiguration" type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings, Microsoft.Practices.EnterpriseLibrary.Data" />
5   </configSections>
6   <dataConfiguration configProtectionProvider="DataProtectionConfigurationProvider">
7     <EncryptedData>
8       <CipherData>
9         <CipherValue>AQAAANCMnd8BFdERjHoAwE/CL+sBAAAAQAWVPX3DMESo74Cf8cZbuwQAAAACAAAAAADZgAAwAAAAABAAAAD0a4jsLtcXyIiw1Npsg3kiAAAAA
10      </CipherData>
11    </EncryptedData>
12  </dataConfiguration>
13  <connectionStrings configProtectionProvider="DataProtectionConfigurationProvider">
14    <EncryptedData>
15      <CipherData>
16        <CipherValue>AQAAANCMnd8BFdERjHoAwE/CL+sBAAAAQAWVPX3DMESo74Cf8cZbuwQAAAACAAAAAADZgAAwAAAAABAAAABt3980jemsK0c9bjcGwkn+AAAAA
17      </CipherData>
18    </EncryptedData>
19  </connectionStrings>
20 </configuration>

```

To run the application

Select the **Debug | Start Without Debugging** menu command or press Ctrl+F5 to run the application without debugging enabled. The application behaves the same as in the previous exercise.

To verify that you have completed the exercise correctly, you can use the solution provided in the ex03\end folder.

You will not be able to open the configuration file with the Enterprise Library configuration editor nor will you be able to run the application, since the configuration file was encrypted with a key not available to you. If you only intend to deploy the encrypted configuration file to the server where you encrypted the file, you can use the **DataProtectionConfigurationProvider**. However, if you want to deploy the encrypted configuration file on a different

server, or on multiple servers in a Web farm, you should use the **RsaProtectedConfigurationProvider**. You will need to export the RSA private key that is required to decrypt the data. You can then deploy the configuration file and the exported key to the target servers, and re-import the keys. For more information, see "[Importing and Exporting Protected Configuration RSA Key Containers](#)."

If you deploy your application to Windows Azure, you should also carefully consider how to encrypt configuration settings stored in Windows Azure. One approach to consider is using the "[Pkcs12 Protected Configuration Provider](#)."

Lab 4: Retrieving Data using Data Accessors with the Data Access Application Block

This lab demonstrates use of Data Accessors, in order to map table rows to strongly typed entities. Data Accessors are designed for querying data only, and not for updating it. They are **not** designed to provide an Object-Relational Mapping (ORM) solution. For more information on Data Accessors, see "[Retrieving Data as Objects](#)" in the Enterprise Library 6 documentation.

To begin this exercise, open the DataEx4.sln file located in the ex04\begin folder.

To review the application

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

This application allows you to select a particular category and load the products for that category.

To implement database retrieval

1. Select the **MainForm.cs** in the Solution Explorer. Select the **View | Code** menu command. Add the following namespace inclusions to the list of namespaces at the beginning of the file. The required references to the **Data** and **Common** assemblies have already been added.

```
using Microsoft.Practices.EnterpriseLibrary.Data;  
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

2. Add the following private fields to the form.

```
private DataAccessor<Category> categoryAccessor;  
private DataAccessor<Product> productAccessor;
```

Data accessors can be expensive to create. It is recommended that you set them up once and then keep a reference to the instances.

3. First, in this exercise we will use the accessor to load the categories. Find the **MainForm_Load** method, and insert the following highlighted code to initialize the **categoryAccessor** and to read the set of categories from the database.

```
private void MainForm_Load(object sender, System.EventArgs e)  
{  
    // TODO: Create the accessors  
    InitializeAccessors();  
  
    this.cmbCategory.DisplayMember = "Name";  
    this.cmbCategory.ValueMember = "ID";  
  
    // TODO: Use a Data Accessor to retrieve Categories  
    var categories = categoryAccessor.Execute();  
    cmbCategory.DataSource = categories.ToList();  
  
    if (this.cmbCategory.Items.Count > 0)  
        this.cmbCategory.SelectedIndex = 0;  
}
```

- Next, find the **InitializeAccessors** method, and add the following highlighted code to create the accessor object.

```
private void InitializeAccessors()
{
    DatabaseProviderFactory factory = new DatabaseProviderFactory();
    Database db = factory.CreateDefault();

    // TODO: Create category accessor
    categoryAccessor = db.CreateSprocAccessor("GetCategories",
        MapBuilder<Category>.MapAllProperties()
            .Map(p => p.ID).ToColumn("CategoryID")
            .Map(p => p.Name).ToColumn("CategoryName")
            .Build());
}
```

The **CreateSprocAccessor** method creates the accessor object. This overload takes a string, which is the name of the stored procedure to call, and a row mapper. The **MapBuilder** static class is a helper that lets you easily build simple row mappers. In this case, it reads from the given columns and stores the results in the given properties in the **Category** instance. When this accessor is executed, an **IEnumerable<Category>** is returned.

- Run the application. You will now see that the combo box is populated with categories from the database.
- Next, we will add another accessor to read products in a category. In this case, we will be using dynamic SQL rather than a stored procedure. In addition, this accessor will accept a parameter, the category **ID** for the products. First, add the code to create the accessor. Find the **InitializeAccessors** method, and add the following highlighted code to create the **productsAccessor**.

```
private void InitializeAccessors()
{
    DatabaseProviderFactory factory = new DatabaseProviderFactory();
    Database db = factory.CreateDefault();

    // TODO: Create category accessor
    categoryAccessor = db.CreateSprocAccessor("GetCategories",
        MapBuilder<Category>.MapAllProperties()
            .Map(p => p.ID).ToColumn("CategoryID")
            .Map(p => p.Name).ToColumn("CategoryName")
            .Build());

    // TODO: Create product accessor
    productAccessor =
        db.CreateSqlStringAccessor(
            "SELECT ProductID, ProductName, UnitPrice, LastUpdate " +
            "FROM Products WHERE CategoryID = @CategoryID",
            new GetProductsByIdParameterMapper(db),
            MapBuilder<Product>
                .MapAllProperties()
                .Map(p => p.ID).ToColumn("ProductID")
                .Map(p => p.Name).ToColumn("ProductName")
            );
}
```



```

        .Map(p => p.UnitPrice).WithFunc(ApplyTax)
        .DoNotMap(p => p.LastReviewed)
        .Build());
    }

```

CreateSqlStringAccessor creates an accessor that uses a SQL statement that is given as a string, rather than a stored procedure. This overload takes a string, which is the SQL to execute, and two mapper objects. One mapper object is the row mapper that was used in step 4 of this exercise. The other is a *parameter mapper*. The row mapper is responsible for mapping the outputs of the database to objects. The parameter mapper is responsible for mapping the parameters for the execute method to the SQL command parameters.

You typically do not need to use parameter mappers with stored procedures. Enterprise Library is able to read the database and automatically discover the parameters to a stored procedure.

Also note the use of the **WithFunc** method in the map builder. This method enables you to run arbitrary functions in order to calculate a property value. The **ApplyTax** method that is called is already in the file.

7. Add a new class to the project. Name it **GetProductsByIdParameterMapper**.

8. Add the following using statements to the top of the new class file:

```

using System.Data;
using System.Data.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;

```

9. Add the following code to the class file:

```

class GetProductsByIdParameterMapper : IParameterMapper
{
    private readonly Database db;

    public GetProductsByIdParameterMapper(Database db)
    {
        this.db = db;
    }
    public void AssignParameters(DbCommand command, object[] parameterValues)
    {
        InitializeParameters(command);
        db.SetParameterValue(command, "@CategoryID", parameterValues[0]);
    }

    private void InitializeParameters(DbCommand command)
    {
        {
            if(!command.Parameters.Contains("@CategoryID"))
            {
                db.AddInParameter(command, "@CategoryID", DbType.Int32);
            }
        }
    }
}

```

10. Find the **cmbCategory_SelectedIndexChanged** method in the **MainForm** class and insert the following highlighted code, which will read a subset of the products, based on the selected category drop-down list.

```
private void cmbCategory_SelectedIndexChanged(object sender,
                                             System.EventArgs e)
{
    // TODO: Retrieve Products by Category
    BindProducts();
}

private void BindProducts()
{
    var selectedCategory = (Category) cmbCategory.SelectedItem;
    if(selectedCategory == null)
        return;

    var products = productAccessor.Execute(selectedCategory.ID);
    dgProducts.DataSource = products.ToList();
}
```

In this case, we are using the **Database.ExecuteSqlCommandAccessor**. The overload used here takes a SQL query as the first argument and an **IRowMapper** for the mapping.

To calculate the tax

1. Find the **btnCalculate_Click** method.
2. Insert the following highlighted line of code to update the database based on any changes in the dataset.

```
private void btnCalculate_Click(object sender, System.EventArgs e)
{
    BindProducts();
}
```

To run the application

1. Select the **Debug | Start Without Debugging** menu command or press Ctrl-F5 to run the application without debugging enabled.
2. Select a category from the drop-down and observe how it loads and saves the data.
3. Modify the Tax percentage values and observe how the unit prices are recalculated.

To verify that you have completed the exercise correctly, you can use the solution provided in the ex04\end folder.

Lab 5: Performing Asynchronous Data Access with the Data Access Application Block

This lab demonstrates how to perform asynchronous database access.

Asynchronous data access is not supported by all ADO.NET data providers. The **Database** class exposes a Boolean property named **SupportsAsync** that you can check at run time to determine if asynchronous operations are supported. If this property returns **false**, then any asynchronous methods you call will throw an **InvalidOperationException**. In Enterprise Library 6, the only database type that supports asynchronous operation is the **SqlDatabase** class. For more information, see "[Performing Asynchronous Data Access](#)" in the Enterprise Library 6 documentation.

To begin this exercise, open the DataEx05.sln file located in the ex05\begin folder.

To review the application

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

This application includes a DataGrid and menus. You will use the Data Access Application Block to implement the asynchronous loading of customers into the DataGrid.

To implement the customer menu items

1. Select the **MainForm.cs** file in the Solution Explorer. Select the **View | Code** menu command.
2. Find the **mnuLoad_Click** method (the **Customer | Load** menu click event handler), and add the following highlighted lines code.

```
private void mnuLoad_Click(object sender, System.EventArgs e)
{
    // TODO: Load Customers
    if (isLoading)
    {
        MessageBox.Show("Query is being executed.");
    }
    else
    {
        UpdateProgress(true);

        db.BeginExecuteReader(CommandType.Text,
                             "WAITFOR DELAY '0:0:1';SELECT * From Customers",
                             EndExecuteReaderCallback, db);
    }
}

private void UpdateProgress(bool state)
{
    isLoading = state;
}
```

```

    panel1.Visible = state;
}

private void FillGrid(DataSet ds)
{
    dataGrid1.DataSource = ds.Tables[0];
    UpdateProgress(false);
}

private void EndExecuteReaderCallback(IAsyncResult result)
{
    try
    {
        var database = (Database) result.AsyncState;

        var ds = new DataSet();
        using(var reader = database.EndExecuteReader(result))
        {
            ds.Load(reader, LoadOption.PreserveChanges, "data");
        }

        Invoke(new Action<DataSet>(FillGrid), ds);
    }
    catch (Exception exp)
    {
        MessageBox.Show(exp.Message);
    }
}

```

The **db.BeginExecuteReader** method is responsible for initiating an asynchronous call to a database. The **EndExecuteReaderCallBack** callback handler is executed when the call completes, and calls the **EndExecuteReader** method to retrieve the populated **DataReader**. This allows code to execute while the query is running.

To run the application

1. Select the **Debug | Start Without Debugging** menu command or press Ctrl-F5 to run the application without debugging enabled.
2. Select the **Customers | Load** menu command to fill the **DataGrid**.

To verify that you have completed the exercise correctly, you can use the solution provided in the ex05\end folder.

More Information

For more information about the Data Access Application Block, see the documentation in the [Enterprise Library 6 Developer's Guide](#) and the [Enterprise Library 6 Reference documentation](#).

patterns & practices

proven practices for predictable results

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes

© 2013 Microsoft. All rights reserved.

Microsoft, MSDN, Visual Studio, Windows, SQL Server, and Windows Azure are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.