Visual Studio

CSC310-A

Module 3

Implementing MVC & MVVM Based Applications

Copyright ©
Symbolicon Systems
2008-2018

1 Models & Controllers

Even though each application provide different functionality, it is simple to develop all applications if there is a standard application architecture that all applications can be applied to. The architecture will determine the steps that a developer can follow to be able to implement all applications regardless of size and complexity. In this module, we will implement an application using the Model-View-Controller (MVC) application architecture.

1.1 Creating Database & Model

A model represent one or more data-oriented classes that you can use to store data that can then be displayed in views and processed by controllers. A model should provide persistence features that will allow controllers to be able to save and restore the model from persistent storage.

Example of a model class representing a single entity

```
public class Account {
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Balance { get; set; }
}
```

Example of a model class representing a set of entities

```
public class AccountList : List<Account> {
    public void Save(string path) { ... }
    public static AccountList Load(string path) { ... }
}
```

If you wish to persist models to a database, there is no need to actually create the entire data model manually since .NET version 3.5. Microsoft has added data access and model generation technologies like *ADO.NET Entity Data Model* and *LINQ to SQL Classes*. In this chapter, you will use LINQ to SQL Classes to generate a data model to store and access data in a SQL Server database.

We will now create a database for our *SymBank* application and generate a data model for *SymBank.Banking* module. Run *SQL Server Management Studio to* execute *SymBank.sql* script provided to create a database that has two tables **Accounts** and **Transactions** and a set of stored procedures. Once the database is ready, add a *Data Connection* to the database using the *Server Explorer* window.

Data Connection properties

```
Server Name      :    .\SQLEXPRESS
Authentication   :    Windows Authentication
Database         :    SymBank
```

Add a **Models** folder to **SymBank.Banking** project. To generate a data model, first add a *LINQ to SQL Classes* template named **SymBank.dbml** in the folder. Then drag and drop all tables in *SymBank* database to the left pane and drag and drop all the stored procedures on the right pane. LINQ to SQL will generate one class for each table in your database to store individual entities and also generate an additional data context class to store and access the entities in the database. You can now compile the project to complete the library.

While the data model does provide all the functionality that we need in order to store and access data in the database, you should not perform data operations directly in your application as each data operation may be still a bit complex or tedious to perform. You should implement a controller to help you perform the data operations on the data model.

1.2 Implementing a Controller

Controllers provide services that views can utilize to perform tedious or complex operations. Even though it is not compulsory that controllers require interfaces, you do get extra functionality and also more security if you abstract access to controllers. We will first declare interfaces that represent all the services that a controller will provide to the application. Add a **Controllers** folder to the project to declare the following interfaces.

Interface for account services : Services\IAccountController.cs

```
public interface IAccountController : IService {
    void Add(Account item);
    Account GetAccount(int code);
    List<Account> GetAccountList();
    List<Account> GetAccountsForName(string name);
}
```

Interface for transaction services: Services\ITransactionController.cs

```
public interface ITransactionController : IService {
    int Debit(int source, decimal amount);
    int Credit(int source, decimal amount);
    int Transfer(int source, int target, decimal amount);
}
```

Implementing the controller: Controllers\BankingController.cs

```
public class BankingController : BaseService,
    IAccountController, ITransactionController {
    public void Add(Account item) { }
    public Account GetAccount(int code) { return null; }
    public List<Account> GetAccountList() { return null; }
    public List<Account> GetAccountForName(string name) { return null; }
    public int Debit(int source, decimal amount) { }
    public int Credit(int source, decimal amount) { }
    public int Transfer(int source, int target, decimal amount) { }
}
```

To use the data model, create an instance of a data-context class generated by LINQ to SQL. It has one property for each table; Accounts and Transactions. Each table implements IQueryable<T> and IEnumerable<T> which means that you can apply all available LINQ operators and **foreach** to access the entities in the table. The data-context also exposes stored procedures as methods that you can call to execute the procedures in the database.

Implementing IAccountController contract: BankingController.cs

```
private IAuthorization _auth;
public string UserName {
    get {
        if (_auth == null) _auth =
            ServiceRepository.Get<IAuthorization>();
        return _auth.UserName;
    }
}

public void Add(Account item) {
        var dc = new SymBankDataContext();
        dc.AccountAdd(item.Code,
            item.Type, item.Name,
            item.ZipCode, UserName,
            DateTime.Now, item.Balance);
}

public Account GetAccount(int code) {
        var dc = new SymBankDataContext();
        return dc.Accounts.Single(a => a.Code == code);
}

public List<Account> GetAccountList() {
        var dc = new SymBankDataContext();
        return dc.Accounts.ToList();
}

public List<Account> GetAccountsForName(string name) {
    name = name.ToLower();
    var dc = new SymBankDataContext();
    var query = from account in dc.Accounts
        where account.Name.ToLower().Contains(name)
        orderby account.Name
        select account;
    return query.ToList();
}
```

```
public int Debit(int source, decimal amount) {
    int? transactionCode = null;
    var dc = new SymBankDataContext();
    dc.AccountDebit(source, amount, UserName,
        DateTime.Now, ref transactionCode);
    return (int)transactionCode;
}

public int Credit(int source, decimal amount) {
    int? transactionCode = null;
    var dc = new SymBankDataContext();
    dc.AccountCredit(source, amount, UserName,
        DateTime.Now, ref transactionCode);
    return (int)transactionCode;
}

public int Transfer(int source, int target, decimal amount) {
        int? transactionCode = null;
        var dc = new SymBankDataContext();
        dc.AccountTransfer(source, target, amount, UserName,
            DateTime.Now, ref transactionCode);
        return (int)transactionCode;
}
```

The controller is now completed. You can use the controller in any kind of application to easily store and access data from the **SymBank** database. In the **BankingModule** class, we can pre-create the controller and register it with our **ServiceRepository** so that multiple views can access and use it. Alternatively you can use **ServiceAttribute** to mark the controller to allow **ServiceRepository** to help you create and register it.

Preparing controllers in code: SymBank.Banking\BankingModule.cs

```
public override void Init() {
    var controller = new BankingController();
    controller.Add<IAccountController>();
    controller.Add<ITransactionController>();
                :
}
```

Automation with ServiceAttribute: SymBank.Banking\BankingController.cs

```
[Service(typeof(IAccountController))]
[Service(typeof(ITransactionController))]
public class BankingController : BaseService,
    IAccountController, ITransactionController {
                :
}
```

Now that you have both model and controller, you can finally complete the views that you have created in the previous module.

2 Views

2.1 Using Services & Controllers

Views can use services and controllers to perform required operations. Both of the views will need to access a controller for retrieving and updating data in the database. You can pre-fetch the services and controllers in the constructor.

Accessing required services in AddAccountView

```
private IAccountController _accountController;
public void AddAccountView() {
    _accountController = ServiceRepository.Get<IAccountController>();
}
```

Accessing required services in SearchAccountsView

```
private IAccountController _accountController;
public void SearchAccountsView() {
    _accountController = ServiceRepository.Get<IAccountController>();
}
```

2.2 Models & Data-Binding

Certain operations require us to pass objects to methods to complete the operations. For example, to add an account to the database, we will need to pass an **Account** object to the **Add** method in **IAccountController**. You can bind controls to objects so that the content of the object can be displayed in controls and if the user changes the data in the controls, the object will automatically be updated. This is called data-binding. In WPF you can instantiate and assign your data model to the **DataContext** property of any UI element. The element and all of its children can then bind to the properties of the object. If you only need to use a single object, you can assign it to **DataContext** of the entire view.

Instancing and exposing a model to the view

```
private Account _account;
public AddAccountView() { DataContext = _account = new Account(); }
```

You can also directly instantiate the object in XAML and then fetch it from the view in the constructor. You will need to register a prefix (**m** is used in the example below) to reference the data model in XAML.

Instantiating data model in XAML

```
<fx:BaseView.DataContext><m:Account /></fx:BaseView.DataContext>
```

Retrieving data model in view

```
public AddAccountView() { _account = (Account)DataContext; }
```

Data-binding to the model

```
<Label>Code</Label><TextBox Text="{Binding Code}" />
<Label>Customer Name</Label><TextBox Text="{Binding Name}" />
<Label>Account Type</Label>
<ComboBox SelectedIndex="{Binding Type}">
    <ComboBoxItem>Savings</ComboBoxItem>
    <ComboBoxItem>Fixed Deposit</ComboBoxItem>
    <ComboBoxItem>Checking</ComboBoxItem>
</ComboBox>
<Label>Zip Code</Label><TextBox Text="{Binding ZipCode}" />
<Label>Opening Balance</Label><TextBox Text="{Binding Balance}" />
```

You can now complete the view by programming the event handler for the *Add* button to use the controller to perform the operation and use the shell to show the results to the user. Rather than using models and data-binding, you can also retrieve and store information in controls. In the other view, we will not use any data model or binding but directly access the content of the control by name. To display the search result we can directly assign it to the properties of the control.

Using services to complete an operation

```
private void btnAdd_Click(object sender, RoutedEventArgs e) {
    try {
        _accountController.AddAccount(_account);
        Shell.Success("Account added successfully.");
    //  Close();    // auto-close view when completed
        _account = new Account(); DataContext = _account;
    }   catch (Exception ex) { Shell.Failure("Cannot add account. " + ex.Message); }
}
```

Search and display results in a ListBox

```
private void btnSearch_Click(object sender, RoutedEventArgs e) {
    try {
        lsbAccounts.ItemsSource = _accountController.GetAccountsForName(txtSearch.Text);
    }   catch (Exception ex) { Shell.Failure("Search account failed. " + ex.Message); }
}
```

2.3 Data Templates

When you execute the application, you may notice that while the search is correctly performed, the results is the object rather than the contents of the object. By default, the ListBox has no idea how to display the content of custom data objects so it simply calls **ToString** method to convert the object to a string and display the string instead. You can easily customize the display by providing a **DataTemplate** that can contain any WPF elements. You can assign the data template to the **ItemTemplate** property. Use binding expressions to display content of the data model.

Assigning a DataTemplate: AddAccountView.xaml

```
<ListBox x:Name="lsbAccounts">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Border Margin="4" Padding="4" Background="Azure"
                BorderThickness="1" BorderBrush="SteelBlue">
                <StackPanel Orientation="Horizontal">
                    <Label Content="{Binding Code}" Width="80" />
                    <Label Content="{Binding Name}" Width="260" />
                    <Label Content="{Binding Balance}" Width="80"
                       HorizontalContentAlignment="Right" />
                </StackPanel>
            </Border>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

2.4 Validation

Applications should always validate user input and external data before usage. Since validation is something that all applications need to implement, we begin by adding some helper methods to simplify validation operations and reporting of validation errors. Add a class named **Guard** to our *Symbion* library and mark it **static**. Since the class is static, all members must also be static as well. Add the following constants and static methods to the class. To make it easier to use the methods, we will turn them into extension methods.

Implementing a validation helper class: Symbion\Guard.cs

```
using System;
using System.Text.RegularExpressions;

namespace Symbion {
    public static class Guard {
        public const string CannotBeNull = "{0} cannot be null.";
        public const string CannotBeEmpty = "{0} cannot be empty.";
        public const string OutOfRange = "{0} is must be between {1} and {2}.";
        public const string IsNotValid = "{0} is not valid.";
        public static void NotNull(this object value, string name) {
            if (value == null) throw new Exception(string.Format(
                CannotBeNull, name));
        }
        public static void NotNullOrEmpty(this string value, string name) {
            NotNull(value, name);
            if (value.Length == 0) throw new Exception(
                string.Format(CannotBeEmpty, name));
        }
        public static void InRange(this int value,
            string name, int minValue, int maxValue) {
            if (value < minValue || value > maxValue) throw new Exception(
                string.Format(OutOfRange, name, minValue, maxValue));

        }
```

As like normal methods, extension methods can be overloaded to support more types. You can overload the **InRange** method to support other commonly used types like **Double**, **Decimal** and **DateTime**. You can also use any generic type and parameter arrays in the methods. Generics allow us support for multiple types but using a single method, class or interface. You do not need to overload one for each type.

Overloading extension methods

```
public static void InRange(this double value,
       string name, double minValue, double maxValue) { … }
public static void InRange(this decimal value,
       string name, decimal minValue, decimal maxValue) { … }
public static void InRange(this DateTime value,
       string name, DateTime minValue, DateTime maxValue) { … }
```

Supporting generic collections and parameter arrays

```
public static void Exists<T>(this T value,
       string name, ICollection<T> collection) {
        if (!collection.Contains(value))
            throw new Exception(string.Format(IsNotValid, name));
}

public static void Exists<T>(this T value, string name, params T[] array) {
    int count = array.Length;
    for (int index = 0; index < count; index++)
        if (array[index].Equals(value)) return;
    throw new Exception(string.Format(IsNotValid, name));
}
```

We will add in one more extension method to validate a string value against a regular expression pattern. This can be used to perform simple and complex text validations using patterns like zip code or email.

Method to simplify regular expression validations

```
public static void Matches(this string value, string name, string pattern) {
    if (!new Regex(pattern).IsMatch(value))
        throw new Exception(string.Format(IsNotValid, name));
}
```

You can now add validation to the add account event handler to validate data in the model before passing it onto the controller as shown in the following page. Exceptions will be automatically generated on validation errors.

Validating input data: SymBank\AddAccountView.xaml.cs

```
private void btnAdd_Click(object sender, RoutedEventArgs e) {
    try {
        _account.Name.NotNullOrEmpty("Name");
        _account.Name.Length.InRange("Length of name", 1, 30);
        _account.ZipCode.Matches("Zip code", @"^\d{5}$");
        _account.Balance.InRange("Balance", 100m, Decimal.MaxValue);
                    :
}
```

3 View Models

3.1 View Model

The concept of a view-model is to separate data and operations away from the view. The view can use data-bindings to access or store data into the view model. Since the data is the view-model, it can provide commands to the view to initiate operations on the data. Whenever data in the view-model changes, a **PropertyChanged** event is required to be fired to refresh the UI. The UI uses the **INotifyPropertyChanged** interface to listen to this event.

Initial base class for view models: Symbion\BaseViewModel.cs

```
using System;
using System.ComponentModel;
using System.Windows;

namespace Symbion {
    public class BaseViewModel : INotifyPropertyChanged {
        public event PropertyChangedEventHandler PropertyChanged;
        public void NotifyPropertyChanged(string propertyName) {
            if (PropertyChanged != null) PropertyChanged(this,
                    new PropertyChangedEventArgs(
                        propertyName));
        }
        private IShell _shell;
        public Shell Shell { get { return _shell; }}
        public BaseViewModel() {
            _shell = ServiceRepository.Get<IShell>();
        }
    }
}
```

We can provide better integration between the view and the viewmodel. A view can fetch the view model from **DataContext**. It is then possible for the view to pass itself to the view model so that the view model can access the view. Add following code to both view and view model to support this integration.

Provide access to view from viewmodel: BaseViewModel.cs

```
private BaseView _view;
public BaseView View {
    get { return _view; }
    set { _view = value; }
    }
}
```

Integration to view model from view: BaseView.cs

```
private BaseViewModel _vm;
public BaseViewModel ViewModel { get { return _vm; }}
public BaseView() {
    Loaded += (s, e) => {   // must only run this code after view is completely loaded
        if (_vm == null) {  // check if view model already retrieved
            _vm = (BaseViewModel)DataContext;    // try to retrieve the view model
            if (_vm != null) _vm.View = this;    // connect view to the view model
    };
}
```

Since the view-model now has access the view we can add in a method that can be used from the view-model to close the view. To allow bindings to be used as well to call the operation, you can expose a command as a property.

Method to close the view from view-model: BaseViewModel.cs

```
public void CloseView() {
    if (_view != null) _view.Close();
}
```

Exposed command to close views

```
public DelegateCommand CloseViewCommand { get; set; }

public BaseViewModel() {
        CloseViewCommand = new DelegateCommand(p => CloseView());
}
```

3.2 The MVVM Pattern

To demonstrate the usage of view-models, add a **ViewModels** folder and implement one view model for each view in **SymBank.Banking** module. Add a new class named **AddAccountViewModel** extended from **BaseViewModel**. The role of a view model is to help provide content and operations to the view through data-binding. Expose content and commands through properties. If the content can be changed during the view, ensure **PropertyChanged** event is invoked so that the UI will be refreshed for controls binded to the property. Following is view model for **AddAccountView**.

View-model for AddAccountView: ViewModels\AddAccountViewModel.cs

```
namespace SymBank.ViewModels {
    public class AddAccountViewModel : BaseViewModel {
        private Account _account;
        private AccountController _accountController;

        public Account Account {
            get {   return _account; }
            set { _account = value;
                NotifyPropertyChanged("Account");
            }
        }
        public void Add() {
            try {
                _account.Name.NotNullOrEmpty("Name");
                _account.Name.Length.InRange("Length of name", 1, 30);
                _account.ZipCode.Matches("Zip code", @"^\d{5}$");
                _account.Balance.InRange("Balance", 100m, Decimal.MaxValue);
                _accountController.Add(_account);
                Shell.Status = string.Format("Account {0} added.", _account.Code);
            //  Account = new Account();
                CloseView();
            }
            catch (Exception ex) {
                Shell.Failure("Error adding account.\n" + ex.Message);
            }
        }

        public DelegateCommand AddCommand { get; private set; }

        public AddAccountViewModel() {
            AddCommand = new DelegateCommand(p => Add());
            _accountController = ServiceRepository.Get<IAccountController>();
            _account = new Account();
        }
    }
}
```

3.3 Binding to View Model

Instead of assigning the model to **DataContext** you should now assign view model as the DataContext instead. You need to register a prefix (**vm** is used in the following example below).

Assigning view-model to DataContext

```
<fx:BaseView.DataContext>
    <vm:AddAccountViewModel />
</fx:BaseView.DataContext>
```

You will need to update all the binding expressions to binding to **Account** properties rather than directly to DataContext. This allows you to expose multiple data models in one view model.

Binding to Account in view model

```
<TextBox Text="{Account.Code}" />
```

To allow user to activate operations from view, you can bind commands in your view-model to **Command** property of the buttons in the view. When the user click on the button, the command will be executed. The button will be automatically disabled if the command cannot be executed.

Binding to commands in view-model

```
<Button Content="Add" Command="{Binding AddCommand}" />
<Button Content="Cancel" Command="{Binding CloseViewCommand}" />
```

You have now completed the view, you can now start using the view to add one or more accounts. We can now use the same techniques to implement and use a view-model for **SearchAccountsView**. This view is much simpler as you just need to store a single text string and provide a read-only collection of **Account** objects as result for the search. There is only one command for searching as the command for closing the view is already provided by the base class.

View-model for SearchAccountsView: SearchAccountsViewModel.cs

```
namespace SymBank.ViewModels {
    public class SearchAccountsViewModel : BaseViewModel {
        private AccountController _accountController;
        private List<Account> _results;
        private string _name;

        public string Name {
            get {   return _name; }
            set { _name = value; NotifyPropertyChanged("Name"); }
        }
        public List<Account> Results {
            get { return _results; }
            private set { _results = value; NotifyPropertyChanged("Results"); }
        }
        public void Search() {
            try {
                Results = _accountController.GetAccountsForName(_name);
                Shell.Status = string.Format(
                    "Search completed. {0} items found.", _results.Count);
            }   catch (Exception ex) {
                Shell.Failure("Error searching for accounts.\n" + ex.Message);
            }
        }
        public DelegateCommand SearchCommand { get; private set; }
        public SearchAccountsViewModel() {
            SearchCommand = new DelegateCommand(p => Search());
            _accountController = new AccountController();
            _name = string.Empty;
        }
    }
}
```

Instancing view-model as DataContext: SearchAccountsView.xaml

```
<fx:BaseView.DataContext>
    <vm:SearchAccountsViewModel />
</fx:BaseView.DataContext>
```

Binding to the view model

```
<TextBox Text="{Binding Name}" />
<Button Grid.Column="1" Command="{Binding SearchCommand}">
<ListBox x:Name="lsbResults" Margin="0,4,0,0" ItemsSource="{Binding Results}"></ListBox>
```

4 Asynchronous Commands

4.1 Asynchronous Methods

You can easily create asynchronous version of any synchronous method that you have already implemented. For example you can now create an asynchronous version of **GetAccountsForName** method. All asynchronous methods must create, start and return a **Task**. You do not have to duplicate the code as the asynchronous version can still call the existing synchronous method.

Asynchronous method declaration: SymBank.Banking\IAccountController.cs

```
public interface IAccountController : IService {
        :
    Task<List<Account>> GetAccountsForNameAsync(string name);
}
```

Asynchronous method: SymBank.Banking\BankingController.cs

```
public Task<List<Account>> GetAccountsForNameAsync(string name) {
    var task = new Task<List<Account>>(() => GetAccountsForName(name));
    task.Start(); return task;
}
```

In C# 4.5 Microsoft has added two keywords that will make writing of multi-threading code as easy as single-threading. Use **await** when calling any asynchronous method from synchronous code. Methods that uses **await** can be marked with **async** as they can also be called asynchronously with **await**. Code is generated automatically to turn **async** methods into tasks and turn **await** statements into synchronization code that is executed on the caller's thread.

Simpler to write asynchronous methods in .NET 4.5

```
public async Task<List<Account>> GetAccountsForNameAsync(string name) {
    return GetAccountsForName(name));
}
```

By marking event handler as **async**, you allow code to run asynchronously but **await** calls are synchronized back to the UI thread when handler is called from UI. However this technique is more appropriate in MVC than MVVM because we usually do not use event handlers in MVVM.

Using async and await: SearchAccountsView.xaml.cs

```
private async void btnSearch_Click(object sender, RoutedEventArgs e) {
    btnSearch.IsEnabled = false;
    try {
        var results = await _accountController.GetAccountsForNameAsync(txtSearch.Text);
        lstAccounts.ItemsSource = results;
    }
    catch (Exception ex) {
            :
    }
    finally {
        btnSearch.IsEnabled = true;
        txtSearch.Focus();
    }
}
```

4.2 Asynchronous Commands

Can commands be executed asynchronously from UI? In short terms yes, but you will need to mark **Execute** as **async** and implement a **Task** to execute command code in a separate thread that Execute can **await** on. The **CanExecute** must also be coded to return **false** when the task is running and only return back to original state when it is completed. If you're using delegates in your command, you should add one more to update the UI when the task is completed. Since the task may fail, the delegate must be able to accept an exception.

Asynchronous version of DelegateCommand: Symbion\AsyncDelegateCommand.cs

```
public class AsyncDelegateCommand : ICommand {
    private Predicate<object> _canExecute;
    private Action<object> _execute;
    private Action<Exception> _completed;
    private bool _running;

    public bool Running { get { return _running; } }

    public AsyncDelegateCommand(Action<object> execute, Action<Exception> completed) {
        _execute = execute; _completed = completed;
    }

    public AsyncDelegateCommand(Predicate<object> canExecute,
        Action<object> execute, Action<Exception> completed) {
        _canExecute = canExecute; _execute = execute;
        _completed = completed;
    }

    public event EventHandler CanExecuteChanged;

    public void NotifyCanExecuteChanged() {
        if (CanExecuteChanged != null) CanExecuteChanged(this, EventArgs.Empty);
    }

    public bool CanExecute(object parameter) {
        if (_running) return false;
        if (_canExecute == null) return true;
        return _canExecute(parameter);
    }
    private Task ExecuteAsync(object parameter) {
        var task = new Task(() => _execute(parameter));
        task.Start(); return task;
    }

    public async void Execute(object parameter) {
        if (_running) return; _running = true;
        NotifyCanExecuteChanged();
        try {
            if (_execute != null) await ExecuteAsync(parameter);
            if (_completed != null) _completed(null);
        }   catch (Exception ex) { if (_completed != null) _completed(ex); }
        _running = false; NotifyCanExecuteChanged();
    }
}
```

Simpler implementation of asynchronous method

```
private async Task ExecuteAsync(object parameter) {
    _execute(parameter);
}
```

Methods executed from asynchronous command: SearchAccountsViewModel.cs

```
private List<Account> _tempResults;

private void StartSearch(object parameter) {
    _tempResults = _accountController.GetAccountsForName(_name);
}

private void SearchCompleted(Exception ex) {
    if (ex != null) Shell.Failure("Error searching for accounts.\n" + ex.Message);
    else {
        Shell.Status = string.Format(
            "Search completed. {0} items found.",
            _tempResults.Count);
        Results = _tempResults;
    }
}
```

Creating an asynchronous command

```
public AsyncDelegateCommand SearchCommand { get; private set; }

public SearchAccountsViewModel() {
    SearchCommand = new AsyncDelegateCommand(StartSearch, SearchCompleted);
            :
}
```