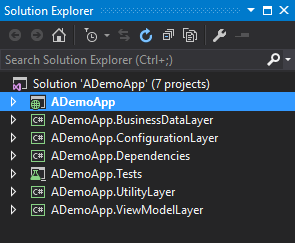
Application Development Standards

This chapter discusses our design philosophy for how to develop applications.

## Solution Structure

When creating a new Visual Studio application, there is a certain design pattern to follow. Your solution should contain numerous projects and references to several DLLs to foster a good N-Tier, SOA architecture. The projects should consist of the following.



|  |  |
| --- | --- |
| Project Name | Description |
| {ProjectName} | The main UI interface project. |
| {ProjectName}BusinessDataLayer | The data layer and your business rule layer. |
| {ProjectName}ConfigurationLayer | A class or set of classes for working with global application settings. These are usually contained within a .config file or a database table. |
| {ProjectName}Dependencies | The DLLs you reference from all other projects. This will include all DLLs from your common framework and domain frameworks. |
| {ProjectName}Tests | A unit test project. |
| {ProjectName}UtilityLayer | A collection of classes that are customizable for this application. |
| {ProjectName}ViewModelLayer | A collection of View Model classes for this application. |

## UI Project

This is your main UI project such as Web Forms, MVC, WPF, etc. You should keep the code in this layer to a bare minimum. You want most of your logic in the other assemblies in the solution. Keep the folder structure for this UI project in line with what Visual Studio gives you. We recommend keeping the folder structure shallow. Don’t go too many layers deep.

## Business Data Layer Project

The following describes the data layer project.

### Entity Framework Usage

If you are using the Entity Framework, we recommend you break up your DbContext classes. Don’t use a single DbContext for your complete database. Break it up based on modules or schemas.

If you create a DbContext called PTC, also add another file called PTC-Extension.cs. This is a partial class for the DbContext class PTC in which you can add override the ValidateEntity method in order to add custom validation.

public partial class PTC

{

// Override ValidateEntity() method

}

You can add additional classes as necessary to this project to extend any of the other generated classes.

## Configuration Layer Project

This project typically only has one class and that is to reference your configuration settings from a .config file or a database table. This class name is AppSettings. This class follows a singleton pattern. You should never use the .NET ConfigurationManager class to retrieve settings from within your applications.

ConfigurationManager.AppSettings[“DefaultStateCode”]

Instead, you should use the AppSettings class.

AppSettings.Instance.DefaultStateCode

There are many advantages to using this approach.

* You get IntelliSense on all property names
* You get strong typing
* You get compile-time errors instead of run-time errors if you misspell a setting name
* You can change the location of where you store settings without having to modify code all over your application

## Dependencies Project

This project is not a project that is compiled. It is merely a folder within your application folder to contain all your common assemblies used in building this project. By copying all your common assemblies into this folder allows you to version this application to a specific version of your common assemblies. These assemblies are also checked into your source control system. If you referenced assemblies from a common network location, if someone made a breaking change, your application could break the next time you compile, and you would not know why. You can now choose the time you want to upgrade this application to the new version of your common framework.

## Unit Tests Project

All unit tests for your application should reside in this project.

## Utility Layer Project

This project contains many “App” classes that are specific classes for this application. Many of these application classes inherit from classes in your common framework. Below is a list of some of the classes you might have in this project.

|  |  |
| --- | --- |
| Class Name | Description |
| AppController | All your MVC controllers inherit from this class |
| AppLog | All application logging goes through this class. This class will use the logging engine in your common framework. |
| AppViewModelBase | The base class from which all your view model classes inherit. |
| AppWebApiController | All your Web API controllers inherit from this class |

## View Model Layer Project

In this project, add the appropriate view model classes for your application. In general, follow these rules.

* One view model per page.
* One property, or an Entity class that has one property for each UI field
* A collection property for each UI data bound list.
* Inherit from AppViewModelBase class.
* Do not put business logic in the view model class.
  + Business logic belongs in “Manager” and “Validation” classes in the business/data layer

Controller / Code-Behind Usage

All code within the UI layer should be kept to a minimum. However, there are some basic things you should write.

Each controller or code-behind should

* Create an instance of the view model.
* Call methods and set properties of view model.
* Wrap all calls within exception handling.

CSS Usage

Below are some general standards for using CSS in your application.

1. Put all custom styles in the **site.css** file.
   1. Do not put custom styles in individual pages unless it overrides a custom style.
2. Do not use inline styles, e.g. style=”color: red;”
3. Follow the bootstrap style naming convention:
   1. Use all lower case style names with words separated by dashes.
   2. Example: glyphicon-asterisk
4. When naming custom styles, describe the purpose of the element being named.
   1. Example: dealer-number-input, btn-cancel

Summary

In this chapter, you learned some best practices for creating .NET applications.