# SIS – SoftUni Information Services

SIS is a combination of a Web Server and a MVC Framework. Ultimately it is designed to mimic Microsoft’s IIS and ASP.NET Core. Following several Lab documents you will build all components of the SIS.

# SIS: Simple MVC Framework

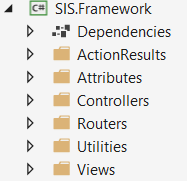
Problems for exercises and homework for the [“C# Web Development Basics” course @ SoftUni](https://softuni.bg/courses/csharp-web-development-basics).

Following to the end this document will help you to create your own very simple MVC Framework, which depends on the HTTP Server we already have. We will eventually extend the Framework, so that we can build dynamic and functional MVC Web Applications which will be hosted on the Handmade HTTP Server.

## Project Setup

Create a project, named SIS.Framework, and link the SIS.WebServer project to it.

Then create the following **folder structure** in the SIS.Frameworkproject**:**

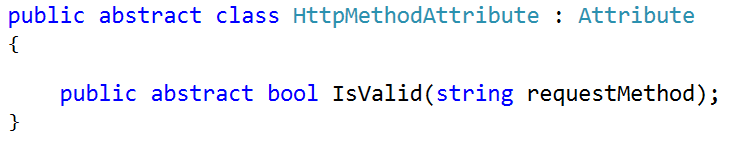


From now on it will be easier to place the new classes and interfaces in the right folders.

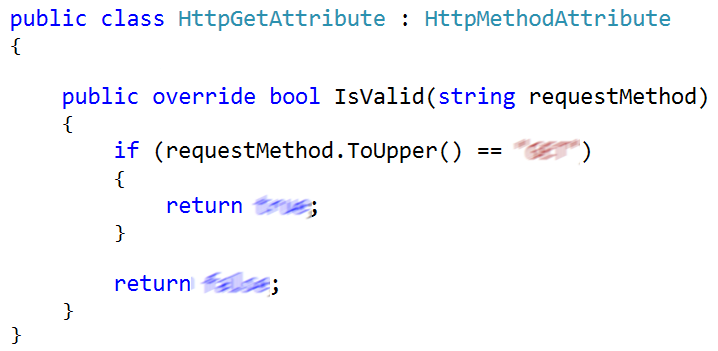
## Attributes

Our framework needs several Attributes that would be placed on the **Methods** of our **Controllers** to annotate whether the **method** can be invoked by GET, POST, PUT or DELETE requests. These attributes will validate whether the **Method** of the **Controller** can be executed. For example, if a method in the controller is marked as [HttpPost] it can be only invoked by a POST request.

In the **Attributes folder**, create a **Methods sub-folder**. In that, create an **abstract** attribute HttpMethodAttribute that inherits Attribute and has **one abstract boolean** method IsValid(string requestMethod).



Create a class, named HttpGetAttribute that inherits HttpMethodAttribute and overrides the IsValid() method. The overridden method should return true if the provided request method equals GET, otherwise returns false.



Create a class, named HttpPostAttribute, like the HttpGetAttribute but this time the IsValid() method should validate whether the request method is POST.

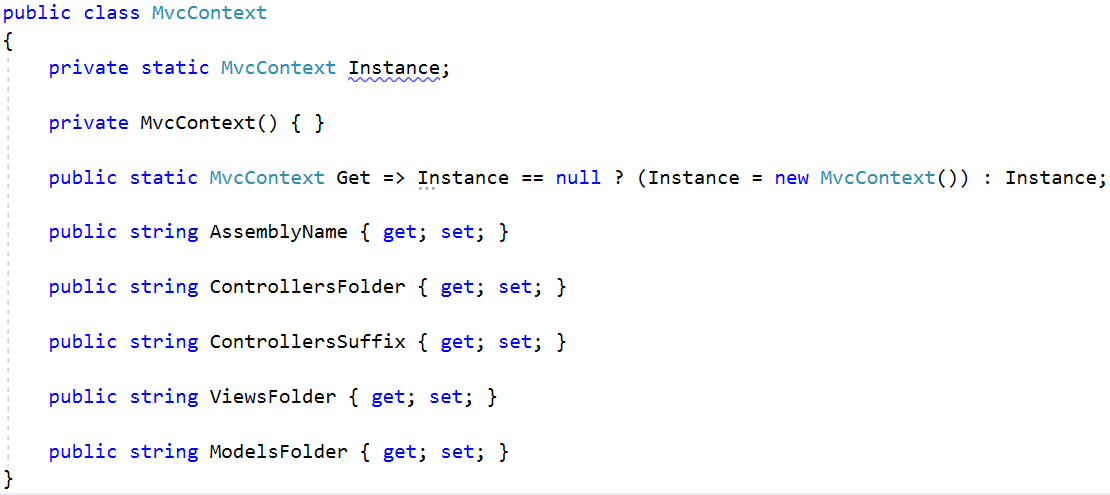
Create a class, named HttpPutAttribute, like the HttpGetAttribute but this time the IsValid() method should validate whether the request method is PUT.

Create a class, named HttpDeleteAttribute, like the HttpGetAttribute but this time the IsValid() method should validate whether the request method is DELETE.

## MVC Context

When the application is running we need to store and access information about the **context** where our **MVC Apps** are **executed**, such as the **Name** of the **Assembly,** the **Path** to the **Controllers Folder,** the **Paths** to the **Views and Models folders**, the **Suffix** of our **Controllers**, etc.   
When we start our application, we need to instantiate that Context only once. We can use the **Singleton pattern**.   
In C#, that is easily done.

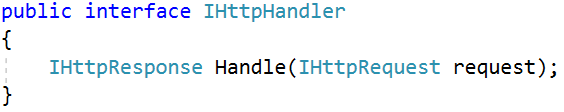
Create a class, named MvcContext in the SIS.Framework project.



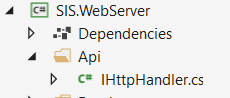
## Extending the Server

### IHttpHandler

The classes implementing this **Interface** would be responsible to **Handle** a **HTTP request** and build a **HTTP response**. It should implement **1** method Handle(IHttpRequest request) that returns a IHttpResponse. (our **router** will implement that **Interface**).



**IMPORTANT**: Create this **interface** on the WebServer project, in a folder called Api.



### Instatiation

Configure the WebServer to be able to use an IHttpHandler for its routing. Do **NOT** **delete** the ServerRoutingTable, just think of a way to **combine** them.

**For examle**: **create 2 constructors**, one with an IHttpHandler and another with the ServerRoutingTable. **Depending** on **which one** of them was **given**, **configure** the ConnectionHandler to work with a **different routing mechanism**.

**Note**: The example above is **not mandatory**. If you can think of a **better way** to implement it, feel free to do so.

## Action Results Interfaces

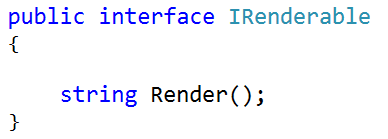
The Framework should provide applications with a generic Controller Action Result functionality. Let’s implement some **interfaces** for those classes.

Create a namespace, called ActionResults. We will use that namespace for the **following 2 tasks**.

Implement the following interfaces in the ActionResults namespace.

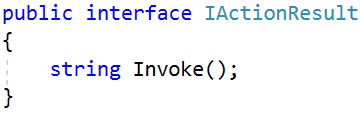
#### IRenderable

This **interface** has only 1 method Render() that returns **string**. The class implementing that method should be responsible for providing and **structuring** the **content** of a **Response** to the server.



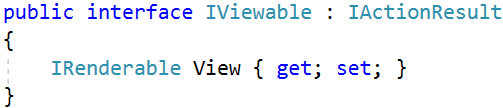
#### IActionResult

This interface will be used to define the **results** of our Controllers’ Actions. We will have different ActionResults, based on the Controller’s Action invocation.



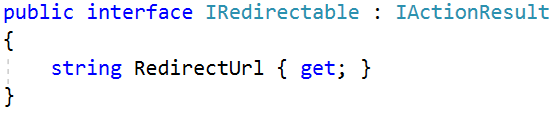
#### **IViewable**

This interface will extend the IActionResult. It will be used for our **View results**, which return **HTML pages**.



#### IRedirectable

This interface will extend the IActionResult. It will be used for our **Redirect results**, which **redirect** the **Client**, to another **location**.

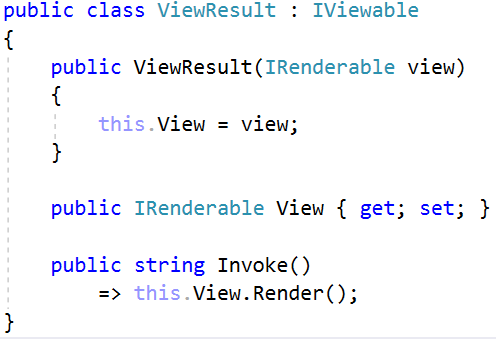


## Action Results Implementations

Now we need to create the classes which would implement the ActionResult interfaces, so that we could actually make **Result Objects**.

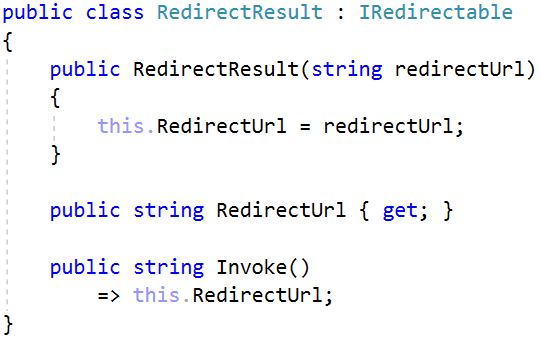
### ViewResult

Create a class, named ViewResult, which implements the IViewable interface. We will use this class for our **View Results** – the HTML Pages. The class should be implemented like this:



### RedirectResult

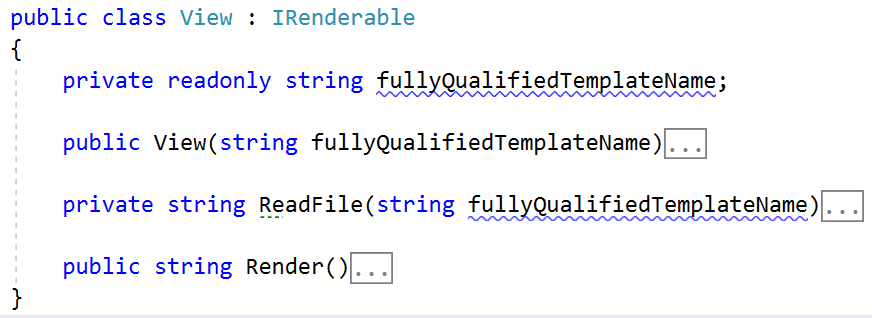
Create a class, named RedirectResult, which implements the IRedirectable interface. We will use this class for our **Redirect Results** – the **Client redirection**. The class should be implemented like this:



## View

But to represent data to the clients, we need to have a class for the Views. Thus, you should create a class, named View, in the Views **namespace** in the Framework project. It should implement the IRenderable interface.

The class should hold the following members:

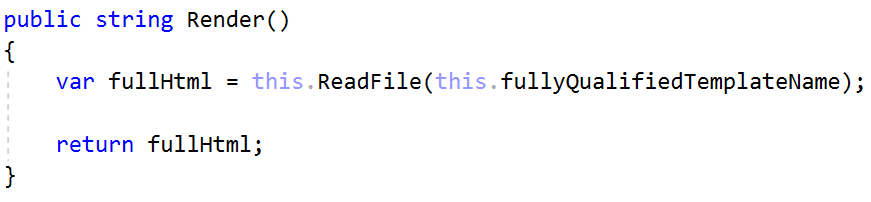


The constructor should only initialize the fullyQualifiedTemplateName, which is the name of the Template with the **Assembly**, **Views folder**, **Controller Name** and **View name**.

The ReadFile() method should extract all the text from the .html file with the **given name** if the **file exists**.

* In case it **does** **NOT** **exist**, throw a FileNotFoundException.

The Render() method is **overridden** from the IRenderable interface, and should look like this:



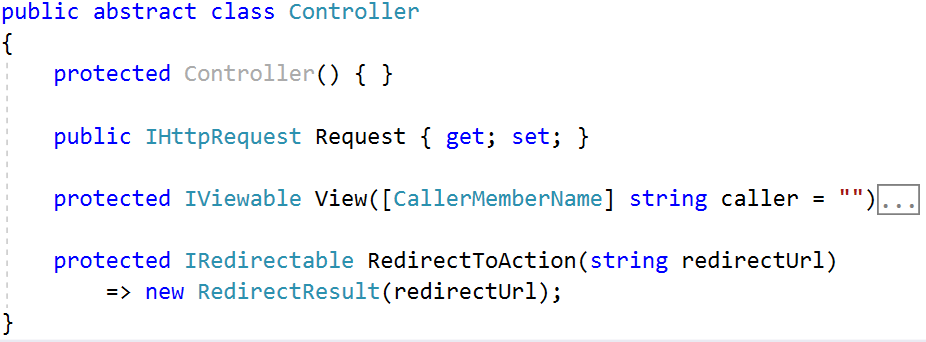
Now that we are ready with the View we can implement our Controller class.

## Controller

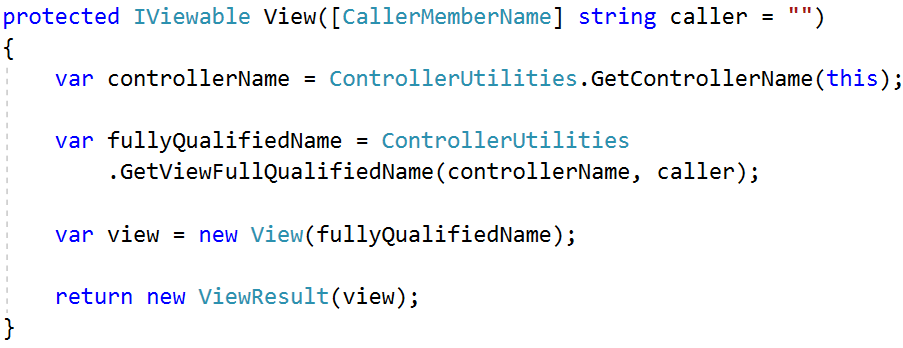
Every MVC Framework provides its consumers with a **base controller functionality** to **implement**. Create a **main** Controller class in the Controllers namespace, that would have **several methods** to help us **Handle Requests**. Every Controller class in our **application** (for example HomeController, UsersController etc.) will inherit from the **main** Controller class. For now, there would be **2 methods** in the base controller class:

* View() – would just generate the view for the method that called the View() method. We would use the [CallerMemberNameAttribute](https://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.callermembernameattribute(v=vs.110).aspx). For example, if method Index() in HomeController class call that View() method it would return <assembly>.Views.Home.Index.html.
* RedirectToAction(redirectUrl) – would return a RedirectResult with the given **URL**.

Here is how the class’s inner implementation should look:



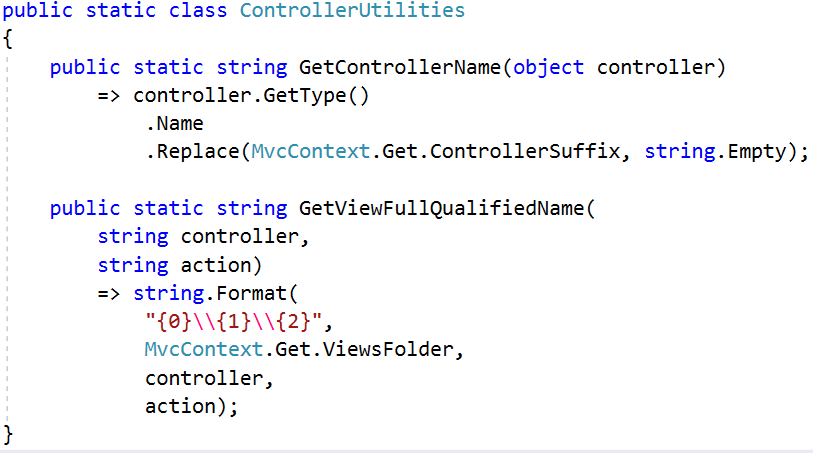
The View() method should be implemented like this:



Hmm, but we see something foreign here – the ControllerUtilities class. Well, let’s implement it right away!

### ControllerUtilities

Create a class called ControllerUtilities, in the Utilities namespace. This class will hold helpful methods for our Controller class’s inner functionality.

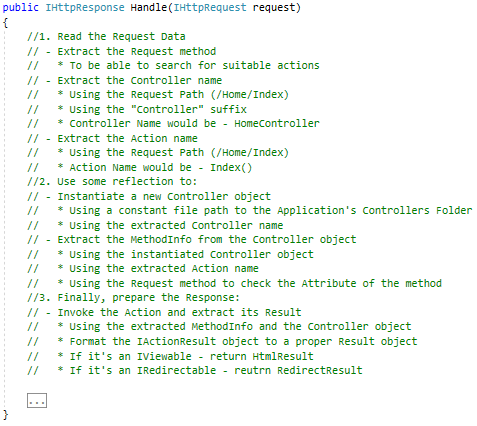


And with this we are ready with the main components of our MVC Framework. Now it’s time to connect everything.

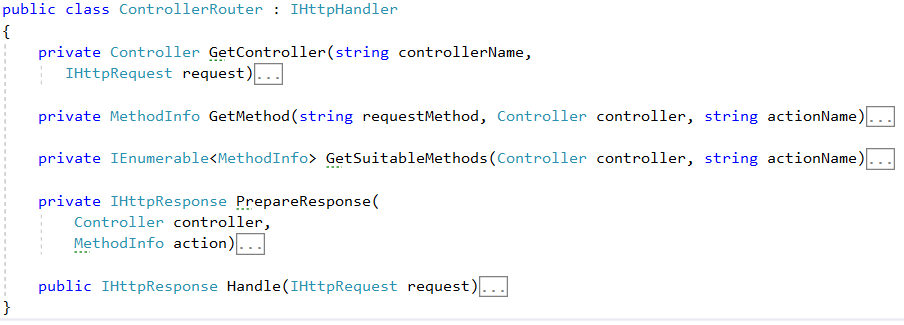
## Controller Router

Create a class, named ControllerRouter in the Routers namespace. The class should **implement** the IHandleable interface from the WebServer project.

The main purpose of this class will be to transform the direct the **incoming** Request to its **corresponding handling** Controller Action. That would be possible by following this algorithm:



The **class itself** should look like this:



### **Request & Mapping Examples**

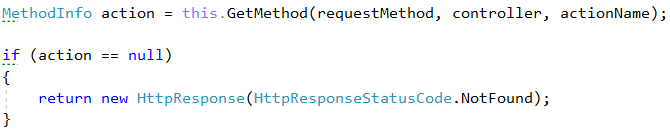
Here are some examples of how Actions should be mapped using the methods shown above.

|  |  |  |
| --- | --- | --- |
| **Request URL** | **Controller** | **Action** |
| test.com/home/index | HomeController | Index |
| test.com/users/profile?id=2 | UsersController | Profile |
| test.com/users | *Invalid URL* | *Invalid URL* |

Notice how we **capitalize the first letter** of the **controller name** and the **action**. Also, we append Controller suffix to the **controller name**.

### Retrieve Method

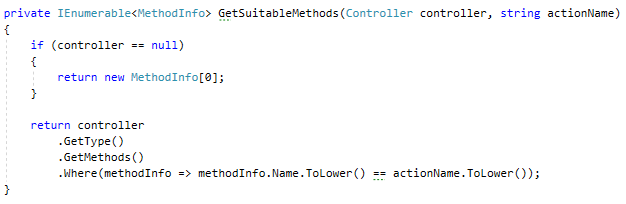
To obtain the Controller Action we should create a method that **extracts** theController’s method, **corresponding** to the Request, based on the Controller and or **NULL** if no such method is found.



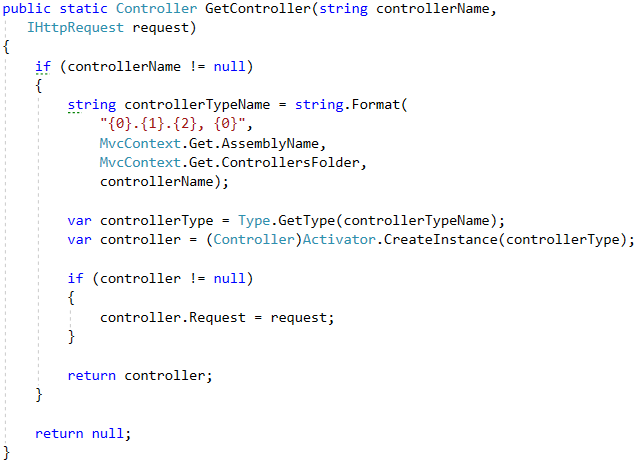
Because of the **overloading** of methods in one Controller class, there might be **several methods** with the **same** **name,** so we need to obtain all of them. Then we need to **iterate** over **every one** of them and **check** if they are **annotated** with some HttpMethodAttribute. If the **method** is **not annotated** with any HttpMethodAttribute and the request method is **GET** we should **return it**. Otherwise we check if the **attribute** on the **method** is the **same** as the Request’s **method**. If the attribute of the method and the requested method are the same => **that’s our method** and we should return it.



The GetSuitableMethods() method **extracts all methods** from the **requested** Controller.



The GetController() method dynamically creates an **instance** of the **requested** Controller using the **full path** to the Controller in the project.



Notice how every method has a check and returns a **NULL** if the check **fails**. At the end in the Handle() method, we will return a NotFound HttpResponse if even one of those **checks fails**. This is to **ensure** that the **Client** uses the correct routes. Later, we will **implement** better **error handling**.

Data for **POST**/**PUT**/**DELETE** requests should be **extracted directly** through the Request’s FormData / QueryData. Later, we will implement **data binding** and **models**.

### Requests and Actions Examples

Here are some examples of Requests, and their corresponding Actions.

#### Example #1

We have a page that should show the profile of a user by given id.  
**Data** should be **extracted** through the Request’s QueryData.

* **Request**
  + **Method:** GET
  + **URL:** test.com/users/profile?id=1
  + **Content:** (no content)
* **Action Method signature**
  + [HttpGet] public IActionResult Profile()

#### Example #2

We have a page that should register a new user to our application.   
**Data** should be **extracted** through the Request’s FormData.

* **Request**
  + **Method:** POST
  + **URL:** test.com/users/register
  + **Content:** User=John&Password=123
* **Action Method Signature**

[HttpPost] public IActionResult Register()

#### Example #3

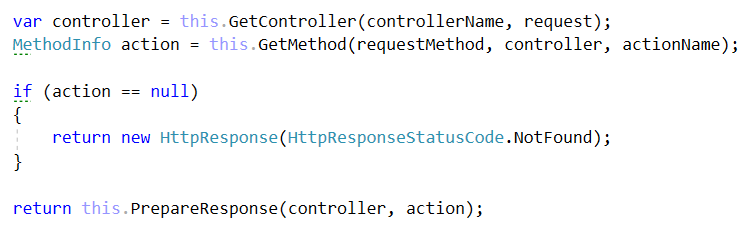
Users in our application has capabilities to create notes and add them to their profile. A note is just a simple text.  
**Data** should be **extracted** through the Request’s FormData & QueryData.

* **Request**
  + **Method:** POST
  + **URL:** test.com/users/AddNote?id=1
  + **Content:** Text=simple new note
* **Action Method Signature**

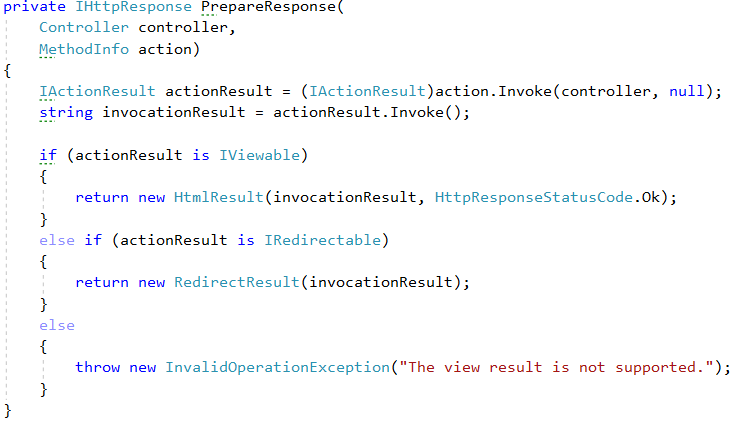
[HttpPost] public IActionResult AddNote()

#### Invoke Action Method

When we have extracted the **method**, **controller**, **action** **etc.**, we should **prepare** the **response** and return it:



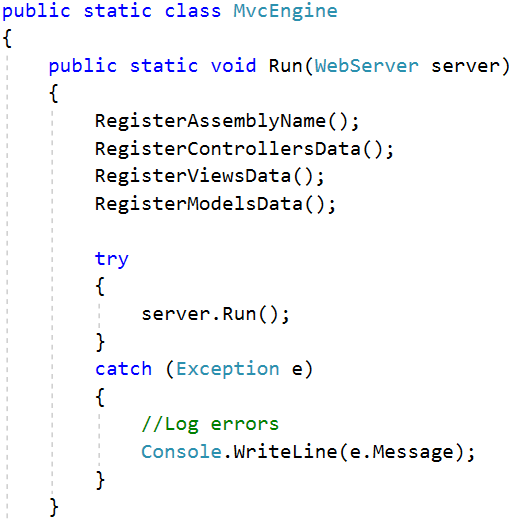
Then what is left is to implement the PrepareResponse() method.



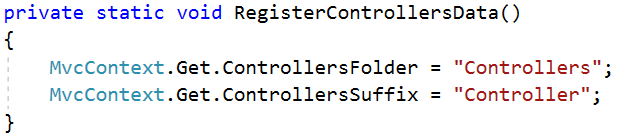
## **MVC Engine**

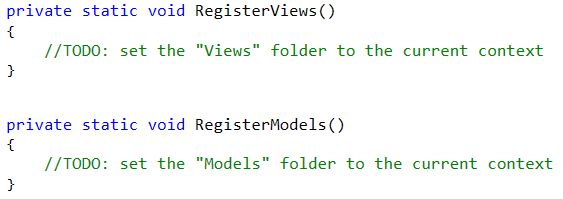
The Framework needs an Engine – a heart, a core, which would **run** the **main processes**.

Create a class, named MvcEngine, that would setup our MvcContext and run our WebServer.

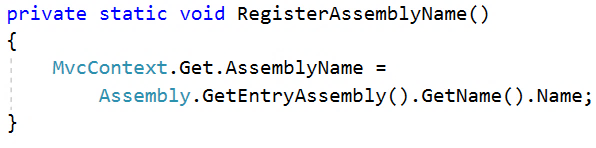


The register methods would setup the current executing assembly and the file structure of our project (set the names of the **Controllers**, **Views** and **Models** folders).





Also, we need to set the current **Assembly**, so that we can access the folders in it. Careful which assembly you are assigning.



## Application Start

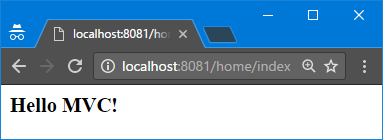
It’s time to test our framework in an application. Create a Project SIS.Demo and rename the Program.cs class to Launcher.cs.

In the Launcher class in the Main() method instantiate new server listening on some port with our ControllerRouter.   
Then run our MvcEngine class.



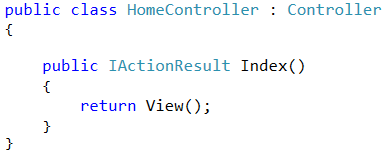
## Test Our MVC Framework

Time to test our framework. To create a page using our MVC framework there are several things you need to do. For example, lets imagine we need to do simple home page with greeting message that would be located in Home controller and the name of the page will be Index.



To create that page, we need to follow these steps and strictly following the name and folder order conventions for our classes

1. In the SIS.App project, in the Controllers folder, create a class called HomeController that inherits base Controller class
2. Inside of it create public method Index()



1. In the Views folder create **subfolder** called Home and inside of it create class Index.html
2. The Index.html should hold “<h1>Hello MVC!</h1>”
3. Run the application and in the browser, try to open your page at localhost:8000/Home/Index