# Lab: Simple MVC Framework

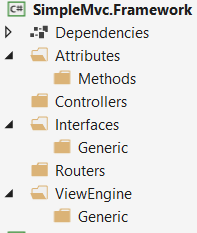
Problems for exercises and homework for the [“C# Web Development Basics” course @ SoftUni](https://softuni.bg/courses/csharp-web-development-basics).

Following the steps of this document at the end of it you will made all by yourself a custom simple MVC Framework simliar to ASP.NET Core framework and would understand the basics of how the MVC framework works under the hood.

## Project Setup

Open the provided skeleton and reference WebServer project to SimpleMvc.Framework project, so our MVC framework and application would use our WebServer.

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

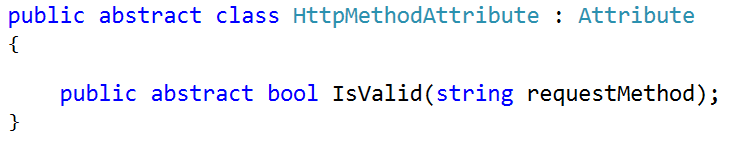


From now on it will be easier to place the new classes and interfaces in the right folders.

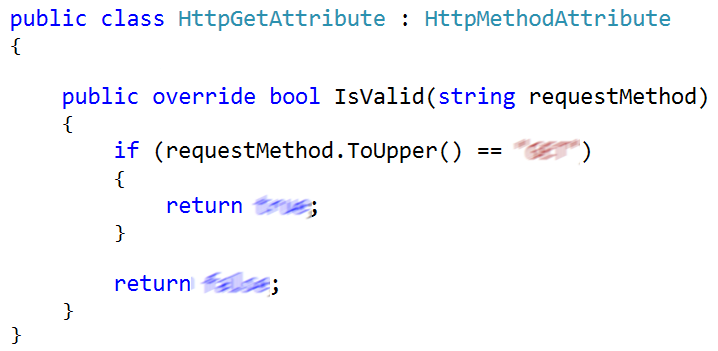
## Attributes

Our framework needs several attributes that would be placed in the methods of our controllers to annotate whether the method can be invoked by GET or POST request. 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/Methods folder** create abstract attribute HttpMethodAttribute that inherits Attribute and has one abstract boolean method IsValid(string requestMethod).



Create HttpGetAttribute that inherits HttpMethodAttribute and overrides the IsValid() method and returns true if the provided request method equals GET otherwise returns false.

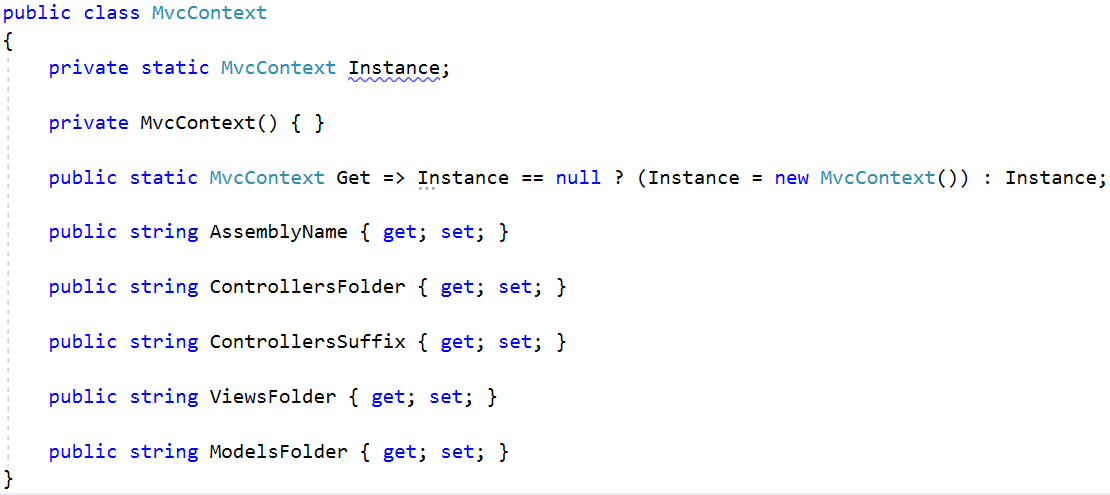


Create HttpPostAttribute similar to the HttpGetAttribute but this time the IsValid() method should validate whether the request method is POST.

## MVC Context

When the application is running we need to keep, and have access to information about the **context** where our **MVC framework** is executed such as the **name of the assembly, the path to the controllers, views and models folders** and the **suffix of our controllers**. When we start our application, we need to instantiate only once that context so we can use the **Singleton pattern**. In C#, that is easily done.

Create a class – MvcContext in the Framework project.

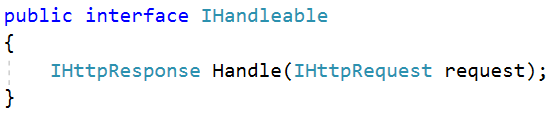


## Interfaces

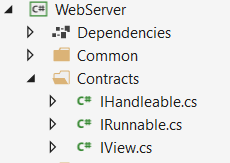
There are several interfaces required for our framework. Implement them now. Place all non-generic interfaces in the Interfaces folder and all generic interfaces in the Interfaces/Generic folder.

#### IHandleable

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



**IMPORTANT**: Create this interface on the WebServer project, in the Contracts.

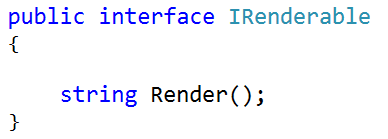


This is the **ONLY** **interface** that will be needed on the server. Later on we will change even that, to ensure maximum quality of our architecture.

**ALL OTHER Interfaces** should be created in the Contracts folder of the Framework project.

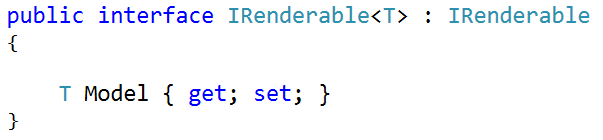
#### IRenderable

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 (our **views** will implement that class. Also, it would be enough to create a view for **static pages**)



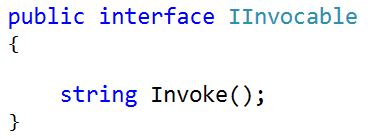
#### IRenderable<T>

This interface will be the same as the IRenderable but this will have 1 more property called **Model** and would be of **generic** type. We can use that model to generate **dynamic content** to be displayed **on the page**.



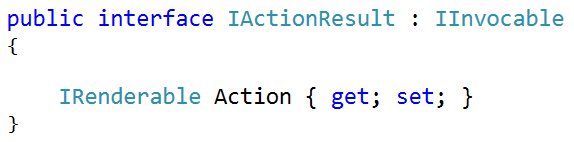
#### IInvocable

This interface has only 1 method Invoke() that returns string. A class that implements that method would be responsible for calling the Render() method of a view. (that would be implicitly our **view engine**).



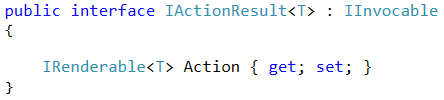
#### IActionResult

This interface inherits IInvocable and has only 1 property called Action of type IRenderable. Our **view engine** will implement that interface.



#### IActionResult<T>

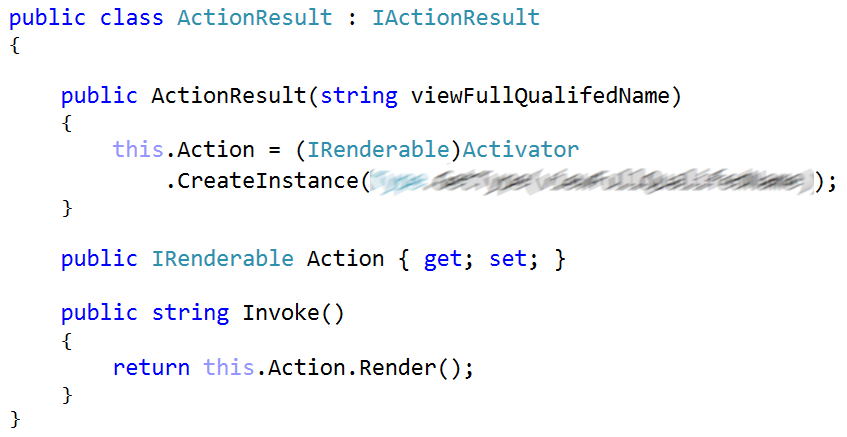
this interface is the same as the IActionResult but it is generic and its property **Action** is of type IRenderable<T>.



## View Engine

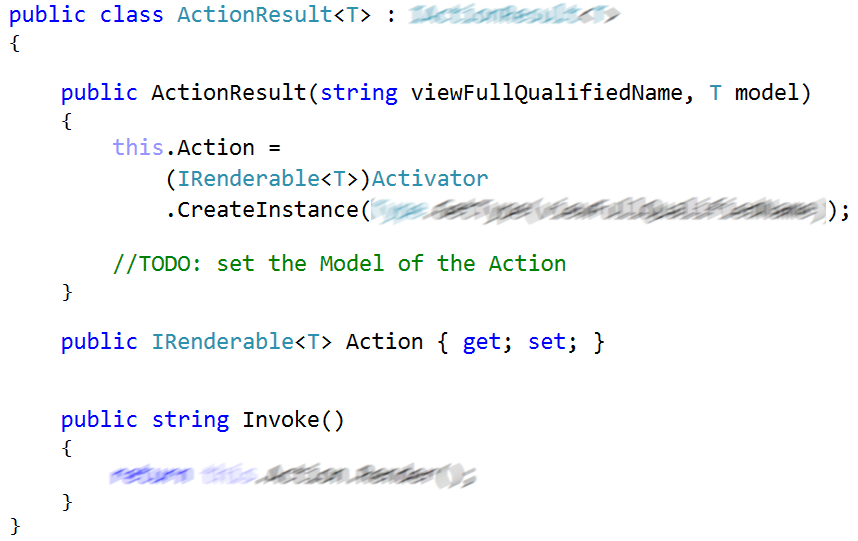
Our view engine would be just a simple class responsible for generating view by creating an instance of the desired view class and calling its Render() method. The class should be named ActionResult and should implement the IActionResult. When we make instance of the class we pass as parameter to the constructor the full qualified name of the view we want to instantiate. For example, if we want to instantiate a view located in **Views/Home/Index.cs** we should use the following example code:

var indexView = new ActionResult(“SimpleMvc.App.Views.Home.Index”);



Place the ActionResult class in ViewEngine folder.

Inside the ViewEngine/Generic folder create class ActionResult<T> that implements IActionResult<T>. It would be the same as ActionResult but generic and its constructor also accepts a model and set the Model of the Action to the model provided.

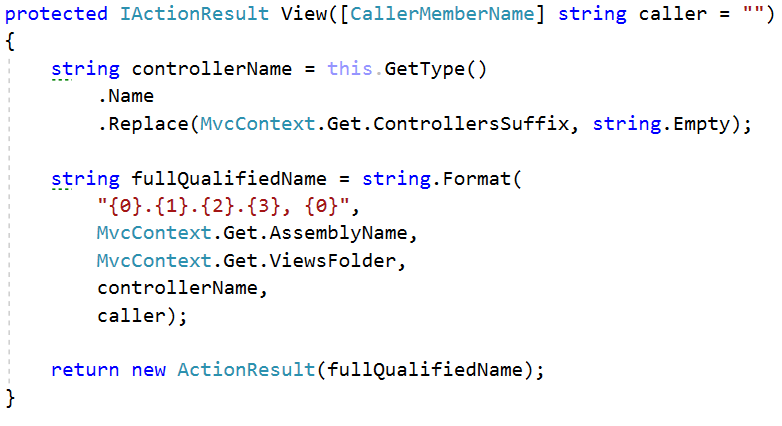


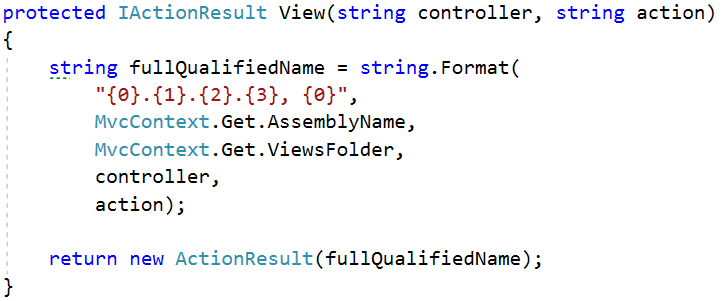
## Controller

We would create main Controller class that would have **several methods to help us create views**. Then every controller class in our application (for example HomeController, UsersController etc.) would inherit that base Controller class so it would be easier to create Views. For now, there would be **4 methods** in the base controller class:

* View() – would just generate the view for the method that called that 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 instantiate <assembly>.Views.Home.Index.cs class
* View(controller, action) – would generate a view of another controller and action by provided names. For example, this method might be used when we want to redirect the user to different page.
* View(T model) – same as the View() method but also a model is provided to the view so dynamic content can be created.
* View(T model, controller, action) – same as View(controller, action) but also a model is provided to the view so dynamic content can be created.

Here are the implementations of the non-generic methods.





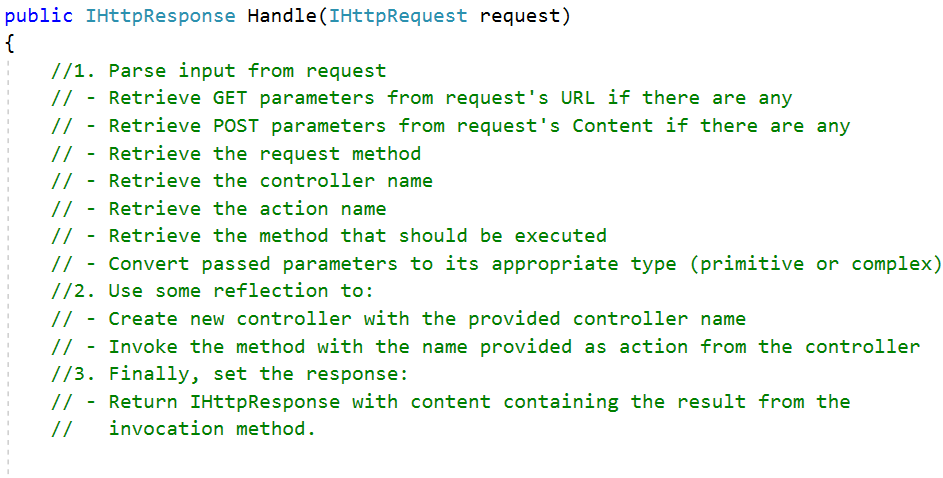
Implement the **generic** View() methods by yourself. Put the Controller class in **Controllers** folder.



## Controller Router

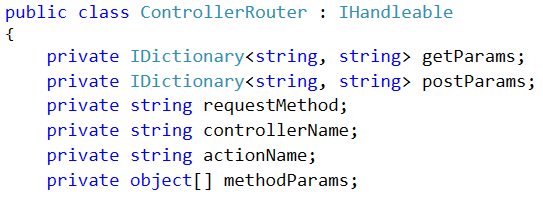
In the **Routers** folder add new class ControllerRouter that should implement the IHandleable interface from the WebServer project.

The main purpose of this class would be to transform the incoming request to a response. That would be possible by following this algorithm:



We would require some variables to keep the information required to create instance of a controller class and invoke any of its methods.

* getParams – dictionary containing transformed query string to name and value pairs
* postParams – dictionary containing transformed content of POST request to name and value pairs
* requestMethod – type of the request method (GET or POST)
* controllerName – name of the controller
* actionName – name of the action method in the controller
* methodParams – array of parameters which will be passed to the action method when it is executing



#### Retrieve GET parameters

GET parameters are coming from the query string of the request’s URL. You need to transform it like the examples below.

|  |  |
| --- | --- |
| **Request’s URL** | getParams |
| test.com/users/profile?id=3 | [{“id”, “3”}] |
| test.com/users/profiles?startWith=g&isActive=True | [  {“startWith”,”g”},  {“isActive”, “True”}  ] |
| test.com/users/profile | [] |

#### Retrieve POST parameters

POST parameters are contained in the content of the request. You need to transform them like the examples below.

|  |  |
| --- | --- |
| **Request’s Content** | postParams |
| id=3 | [{“id”, “3”}] |
| Username=JohnD&Password=1234Secure | [  {“Username”,”JohnD”},  {“Password”, “1234Secure”}  ] |
| *(no content)* | [] |

#### Retrieve Request Method

To retrieve the request method simply convert the HTTP request’s method property to string.



#### Retrieve Controller and Action Names

The name and the action of the controller can be retrieved from the request’s URL. The URLs must be in the following format se we can process them

<host>/{controllerName}/{actionName}

or

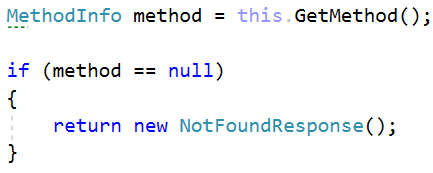
<host>/{controllerName}/{actionName}?{query\_string}

|  |  |  |
| --- | --- | --- |
| **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* |

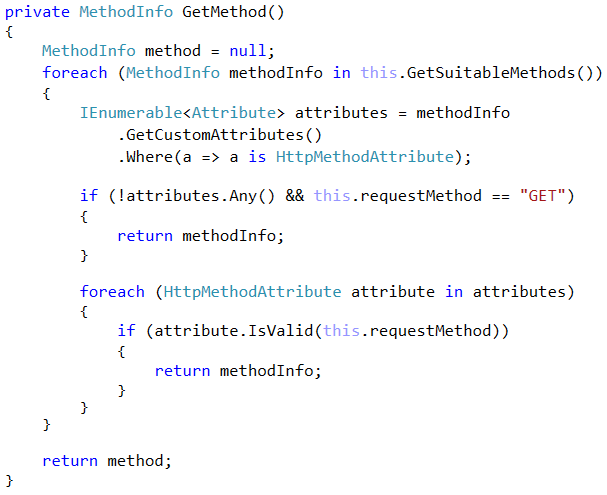
Notice how we **capitalize the first letter** of the **controller** and the **action**. Also, we append Controller word to the controller.

#### Retrieve Method

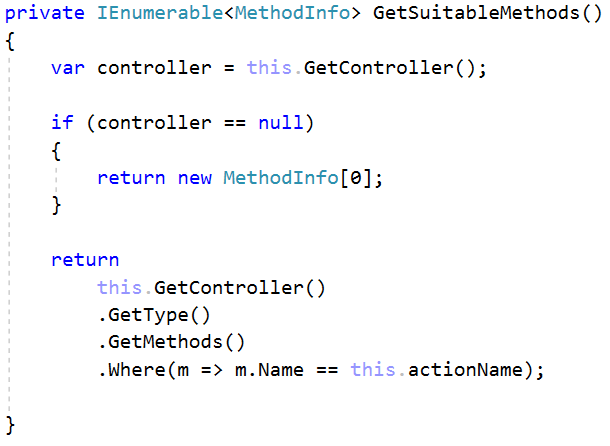
To obtain the method we would create another method that would returns the requested method from the controller or null if no such method is found.



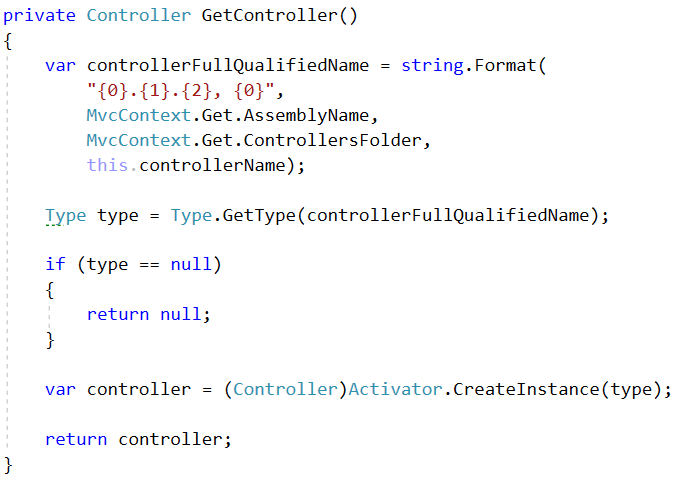
Because of the overloading of methods in one controller class there might be several methods with same name so we need to obtain all of them. Then we need to iterate over every one of them and check if they are annotated with some HttpMethodAttribute. If the method is not annotated with any HttpMethodAttribute 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 get all methods of the requested controller.



The GetController() method 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 NotFoundResponse if even one of those check fails. This is to ensure that the client uses the correct routes.

#### Prepare Method Parameters

To simplify our framework, we would assume that when the action method in the controller is annotated with [HttpGet] and expects parameters they should be only of primitive types (int, string etc..). Otherwise if the action method in the controller is [HttpPost] and expects parameters the parameters of the action method should be encapsulated in a binding model class.

##### Example #1

We have a page that should show the profile of a user by given id.

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

##### Example #2

We have a page that should register a new user to our application

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

[HttpPost] public IActionResult Register(RegisterUserBindingModel model)

* **Binding Model**

class RegisterUserBindingModel{

public string User {get;set;}

public string Password {get;set;}

}

##### Example #3

Users in our application has capabilities to create notes and add them to their profile. A note is just a simple text.

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

[HttpPost] public IActionResult AddNote(int id, AddNoteBindingModel model)

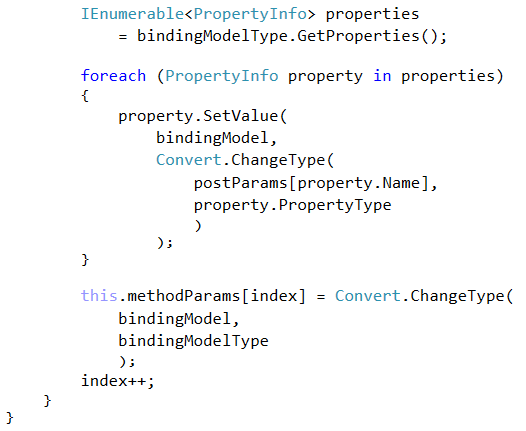
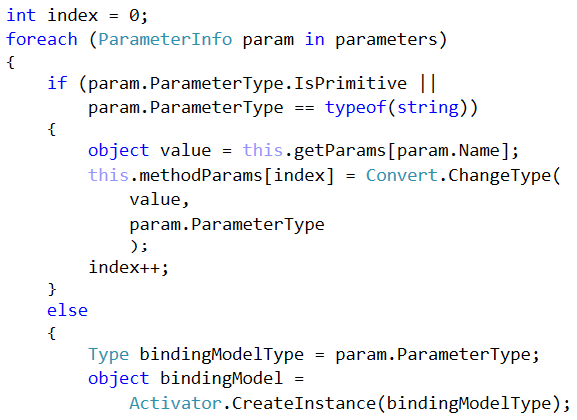
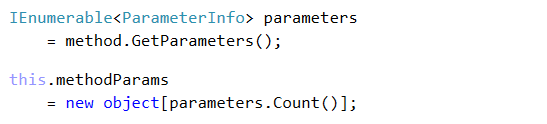
* **Binding Model**

class AddNoteBindingModel{

public string Text {get;set;}

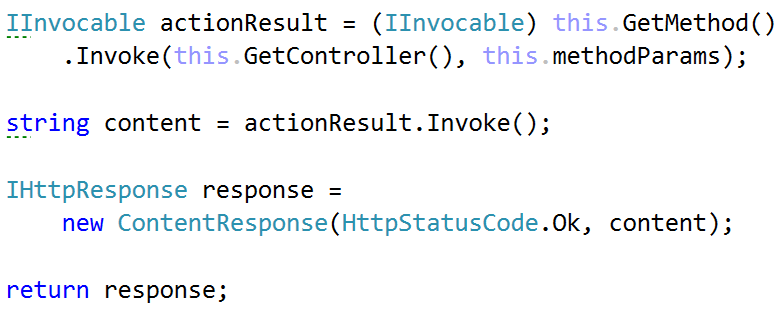
}

When we obtain the method of the controller we need to convert the string parameters from getParams or postParams dictionaries to their appropriate type that the method in the controller expects.



#### Invoke Action Method

Finally, we need to invoke the method of that controller with the provided parameters. The execution of the method will return the content of the response. We would create new response and add the generated content to it.

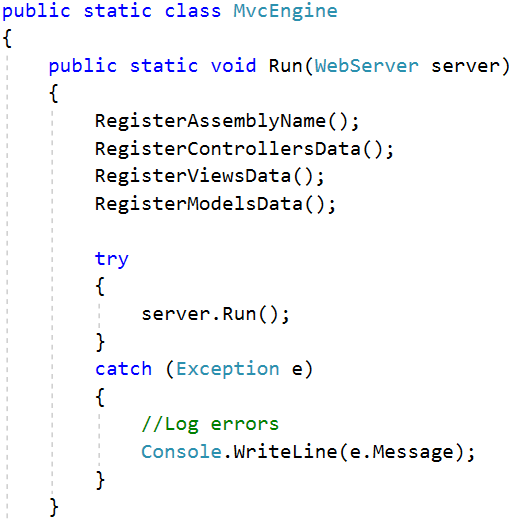


But wait... What is this ContentResponse? There was no such class last time we checked. Well … The casual ViewResponse, which the WebServer had, was changed to ContentResponse, so that it could suit our purposes. The only change is that instead of an IView, the class accepts a string, and renders it in its ToString() method.

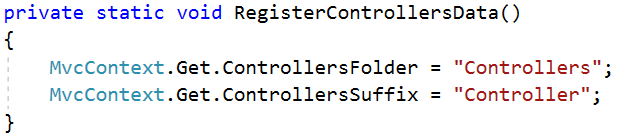
You can see the changed class in the given skeleton.

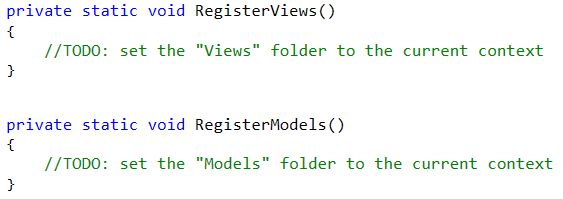
## **MVC Engine**

We need to create MvcEngine class that would setup our MvcContext and run our WebServer.

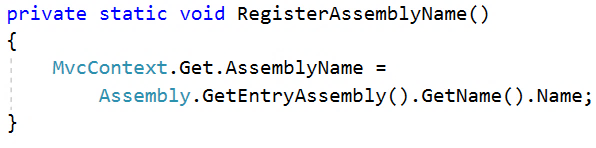


The register methods would setup the current executing assembly and the file structure of our project (set the names of the Controllers, Views and Models folders).





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



## Routing

Now, you might have noticed that the ControllerRouter class works as a RequestHandler with a specific routing. But we already have a ServerRouteConfig, a HttpHandler and some other classes…

Well, that changes now. If you look at the WebServer provided in the skeleton, you’ll see that it is changed. Now it works mainly with an IHandleable object, which our ControllerRouter is.

That being said, the only thing we need to do, is pass the new ControllerRouter to the Server upon initialization, and run the MvcEngine.

## Application Start

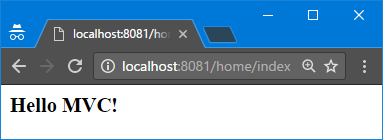
It’s time to test our framework in an application. Create a Project SimpleMvc.App, and rename the Program.cs class to Launcher.cs.

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



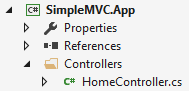
## Test Our MVC Framework

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.

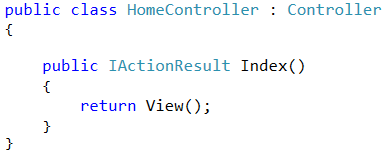


To create that page, we need to follow these steps and strictly following the name and folder order conventions for our classes

1. In the SimpleMvc.App project, in the Controllers folder, create a class called HomeController that inherits base Controller class



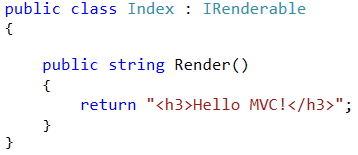
1. Inside of it create public method Index()



1. In the Views folder create **subfolder** called Home and inside of it create class Index.cs



1. The class Index inherits IRenderable interface
2. Make the Render() method return simple heading level 3 HTML text containing the greeting



1. Run the application and in the browser, try to open your page at localhost:8000/home/index

## Notes Application

Let’s create more advanced application using our MVC framework. The application should have the following capabilities:

* Register new user to the application
* Show a list containing all registered users
* Show profile page of each user where he/she can see list of all of his notes as well as form to add new note

(optional delete user with all of its notes)

We would use database to keep all the user and their list of notes. Each user has username and password and list of notes. A single note has title and content.

#### Step 1 – Design Database

You know the drill. Create 2 projects:

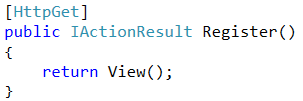
* SimpleMvc.Data project
* SimpleMvc.Domain project

In the SimpleMvc.Domain project, implement the following 2 Entities (EntityFrameworkCore Entities)

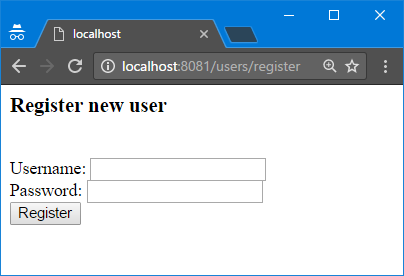
* **User** – username, password, list of Notes
* **Note** – title, content, owner (User)

#### Step 2 – Register New User

First thing we need to do to register new user in our application is to create UsersController and add Register() method that would **show the page with the form** to register new user.

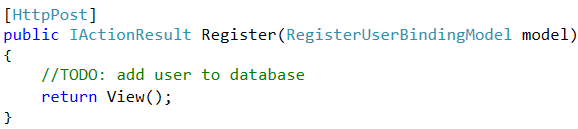


We now need a view that will show the form on the browser. In the Viewsfolder create subfolder User. Inside of it create new class Register that implements the IRenderable interface. **Make a simple form** that would accept **username** and **password** and **submit button.**

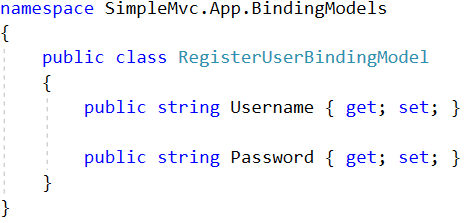




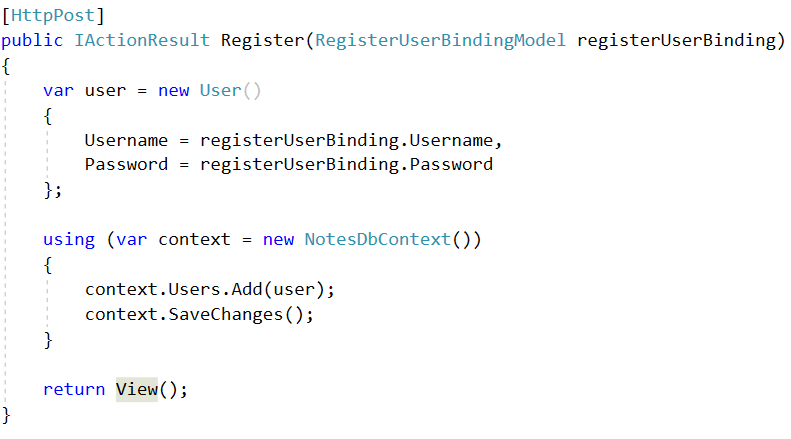
When we click on the submit button the form is send and **we need another method to handle it**. So, in the UsersController class we create another method Register() but this time annotated with HttpPostAttribute



The framework is designed in a such way that if we need to **create a binding model to obtain the content of the POST request**. So, we create a folder in the root of the Application project called **BindingModels** and inside of it add class RegisterUserBindingModel. The binding model is just a class with several properties.



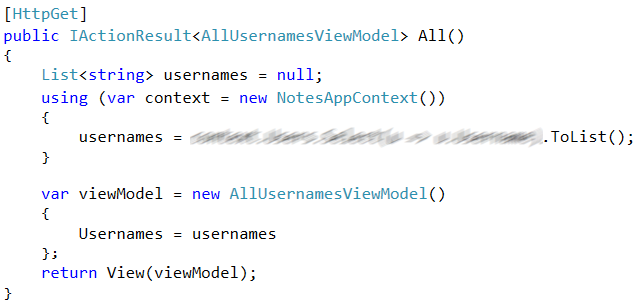
In the Register() method that handles the POST request we need to create new User with the data provided from the RegisterUserBindingModel model and add it to the database.



Finally **test it in the browser** to see if the user is successfully added to the database.

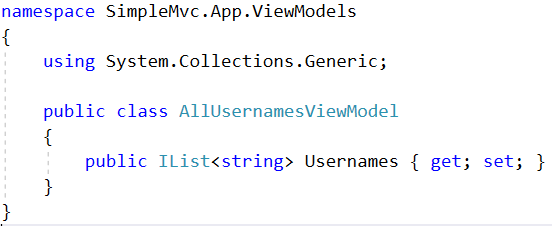
#### Step 3 – Show All Users

Showing a list of users is even easier that registering we need just to **obtain all usernames from the Users table** in the database and **print them**. The list of all user would be accessible at **/users/all** address. So, in the UsersControllers create new method All().

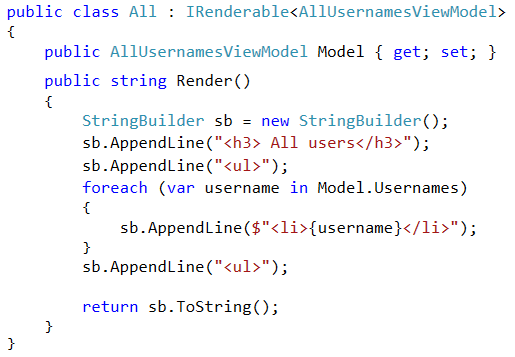


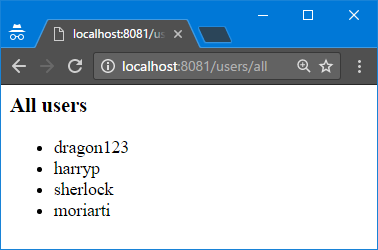
This time we need to send some data to the View so we would use the generic version of IActionResult interface. It is **not good idea to use the User model** (that contains all information about the user) to be transferred to our view. So, we **create a view model** that would **contain only the information the view needs to display**. In our case that would be only the **usernames of all users**. Also, provided in such way the **object is** **smaller** and would be **faster transferred on the network**.

The AllUsernamesViewModel class is located in the ViewModels folder in the root of our Application project.



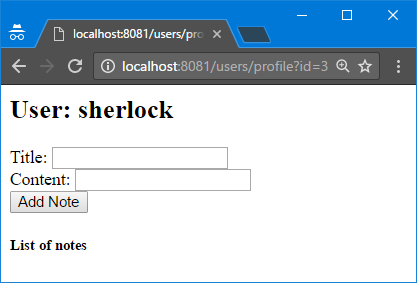
Now to display the list of users we need to create **View** in the **Views/Users folder** called All.cs that implements IRenderable<AllUsernamesViewModel> interface. And in the render method we should just print all of the provided usernames contained in the view model object.





#### Step 4 – Users Profile Page

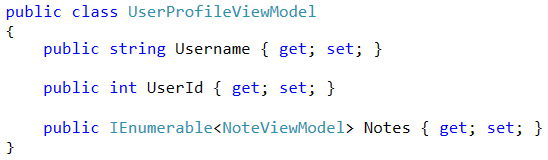
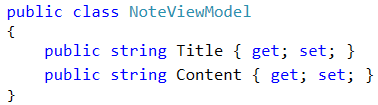
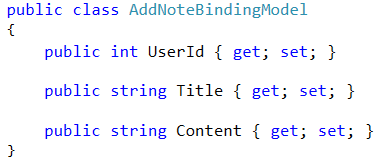
The profile page of a user should be accessible by get request containing the user ID. For example, from the sample users added in the previous step if we try to reach /users/profile?id=3 we should open the **sherlock**’s profile page.



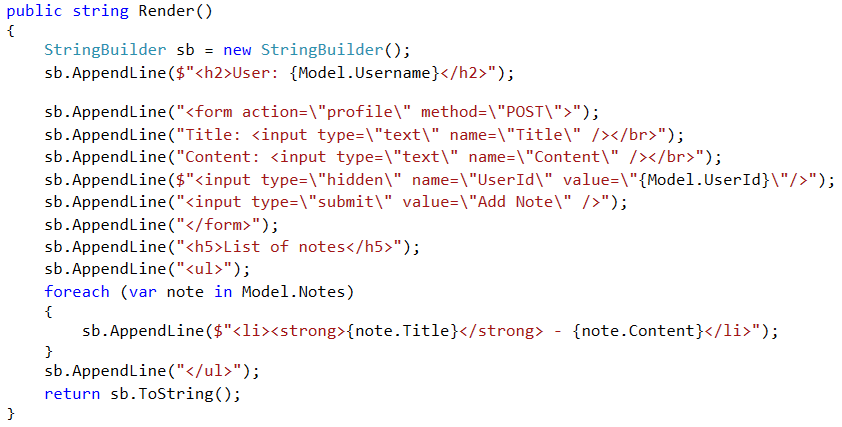
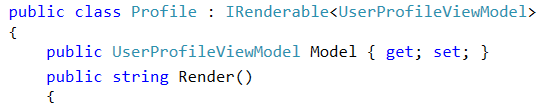
The profile page should contain the username of the user, form from where we can add notes to that user and list of his notes.

We will need some binding and view models:

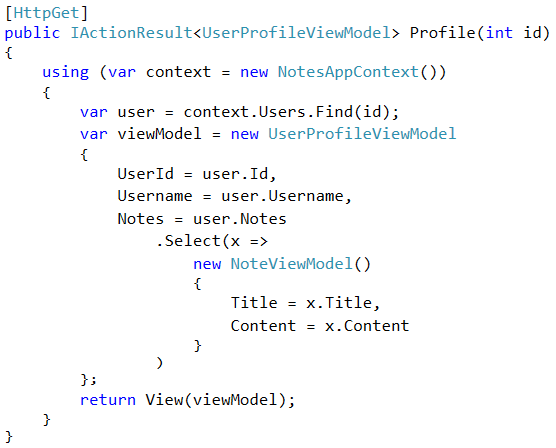
* **AddNoteBindingModel** – UserId, Title, Content
* **NoteViewModel** – Title, Content
* **UserProfileViewModel** – UserId, Username, collection of NoteViewModel



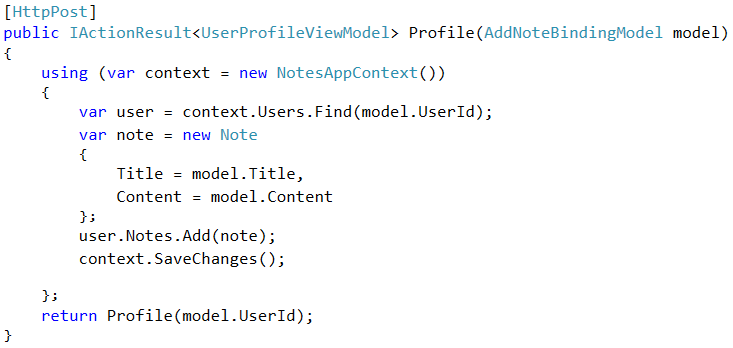
Now let’s create the view of the page. In the Views/Users folder create view Profile that implements IRenderable<UserProfileViewModel>



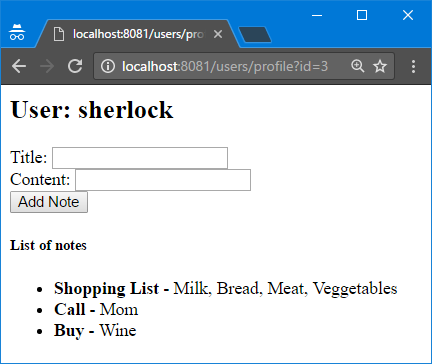
In the UsersController class create **Profile()** method that would **show the profile page** of a user **by** **provided id**



To **add a new note to the user** we need another method that would handle the POST request of the profile page.



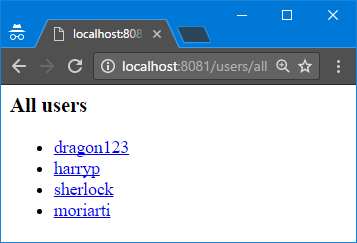
Finally **test in the browser** if the user can add notes and they are displayed on his profile page.



#### Step 5 – Create Links to Profile Pages

At this point you should know **how the framework works and how to use it**. So, the next features you should be able to complete it all by yourself.

This final task for this lab is in the users/all page. You need to **modify the list of usernames to be a links that lead to the user’s profile** page instead of plain text.



#### Step 6 – Page Navigation

Create **home page** that contains 2 links that lead to **users/all** and **users/register** page. Also on each page **add links to go back to the home page**.

#### Examples

