# Lab: Advanced MVC Framework

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

We’ve created our simple MVC Framework in the previous exercise. In this exercise we will make it even better and more dynamic. Follow the steps in the document closely and be careful not to make mistakes.

Most of the things will get removed, and new ones will be added on their place. This process is essential in development... Things get deprecated over time.

The process of reconfiguring the framework will be complex and delicate, some things might break due to direct deletion. Do not worry, just follow the steps trough, till the end, and you will reach a point at which everything works wonderfully.

## HTML View Engine

The old view engine was cool, we actually created dynamic web content… With StringBuilders… Which is not very impressive. What will be very impressive though, is if we make our framework to work with actual HTML Views.

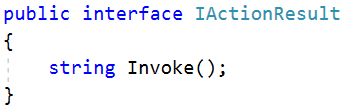
First, we will need to reconfigure the Interface Hierarchy. Go into the Contracts folder of the Framework project. **REMOVE** the Generic sub-folder, we **won’t** be needing any more **generic views**.

Let’s recreate the Interfaces, so they match our new idea.

### Interfaces

#### IActionResult

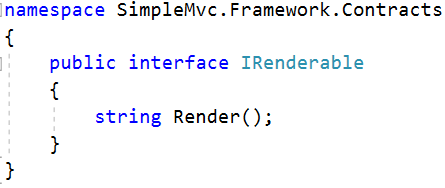
This interface is mainly used as a return type for the actions of the different controllers. It must always stay at the top of the hierarchy chain because many things depend on it. Naturally, the interface will have different implementation than before. This is the implementation you need to build:



It has only one method – Invoke() which is all that is needed.

#### IRenderable

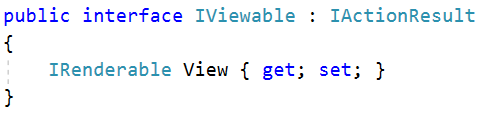
This interface **remains the same** … It has a single method in it – Render() which is all that is needed.



This is where the fun begins. Now here are the main 2 interfaces, that our Controllers will use as the results of their actions.

#### IViewable

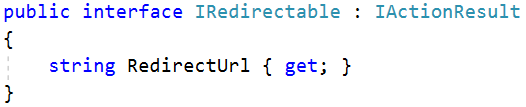
This interface will be the implementation contract of all action results which can be rendered in some way.



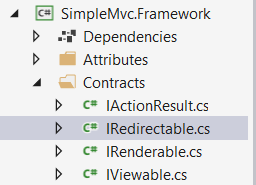
But aren’t all of our action results – renderable? Well, what happens if we want to Redirect? This is where the other type of action result comes.

#### IRedirectable

This interface will help us create another type of View which is for the Redirect functionality. We need to be able to redirect in order to make our framework the most efficient.



With that we should summarize all the work on the Contracts folder. The IInvocable interface is also unneeded, so you can just, **remove** it. The folder’s layout should be like this:



Now, after these steps, most of the things in your application and framework should be relatively broken. For example the Views in the Application project will not work, because they used the generic interfaces.   
Rest assured, we will fix that in a moment.

### ViewModel

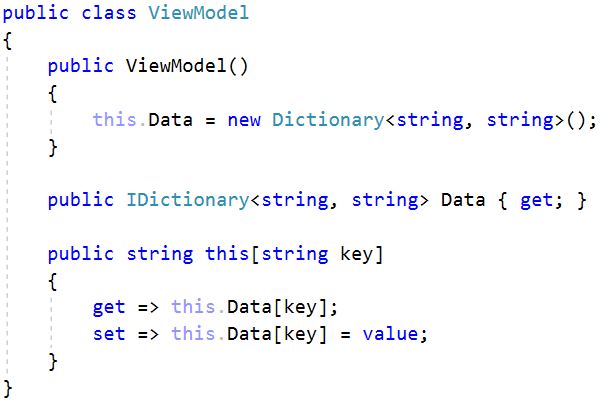
Remember how we did that cool **ViewData** dictionary in the WebServer and we could render parameters in our HTML with the following syntax:

{{{parameter}}}

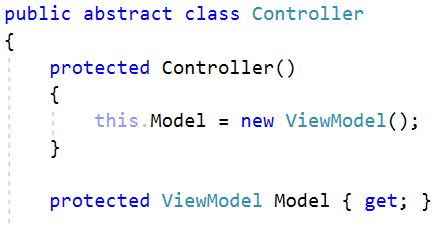
Well, let’s do it again, but a little more elegantly.

Create a folder called Models in the Framework project and create a class ViewModel in the folder. This class will be used by our base Controller for the View Data. It will accept several parameters, which can be configured in any controller, and will render them to the View upon returning the HTML.

The ViewModel class holds a **Dictionary**, which holds **string data**, and access it by **string keys**, so it is limited to only being able to **render HTML** in **strings**, but you will see how it will significantly optimize the StringBuilder mess.



Now that the ViewModel, we can integrate it into the base Controller in the Framework project, so that we can use it.



We’ll integrate it later in our Views. Now it’s time for a centralized View.

### View Class

Create a folder Views in the Framework project and create a class View in it. This class will be initialized with a path to a certain View and ViewData as a Dictionary. Its job is to load a base Layout.html file, determined by the Application, and then load a HTML View, and **render** the View into the Layout file, composing one full View. When the full View is composed the class traverses the whole ViewData Dictionary, and replaces all parameters within the composed HTML with their respective values in the Dictionary. Upon finishing with all the job, the class returns a fully rendered HTML content as a string. This is all done by the Render() method which comes from the IRenderable interface.

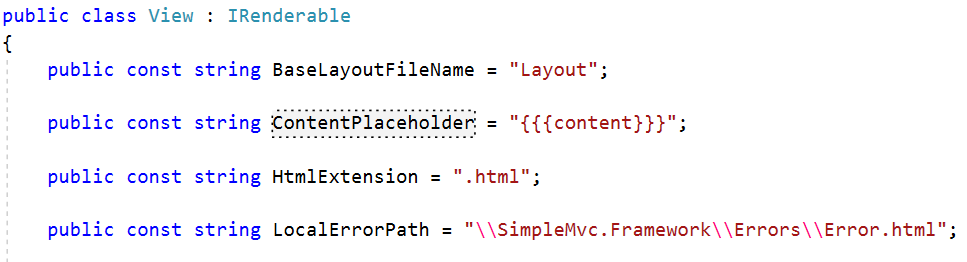
Let’s create the View class.

#### Constants

First, we need some constants for all the paths, and variables we will be using. Those are mainly:

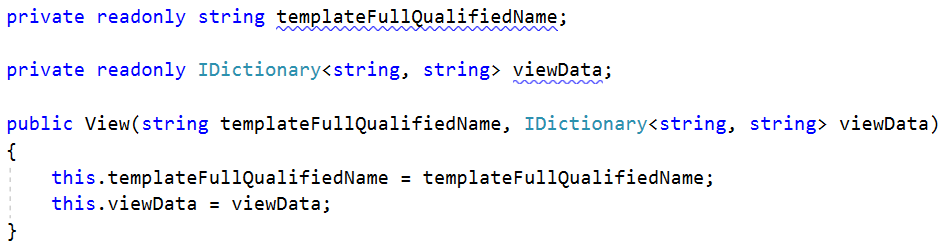
* BaseLayoutFileName – The framework determines with this **constant**, how the Layout file must be called in the application using it.
* ContentPlaceholder – The framework determines the placeholder for the Views, in the Layout file. This is the placeholder that the View class will look for in the Layout file, to render the partial View.
* HtmlExtension – Just a string “.html” which we will use for the loading of files.
* LocalErrorPath – This is the path to the Error folder, which we will create later. The Error folder will contain a simple HTML partial View, which will be returned instead of the actual View, in case of an error.

And here is the code:



#### Initialization

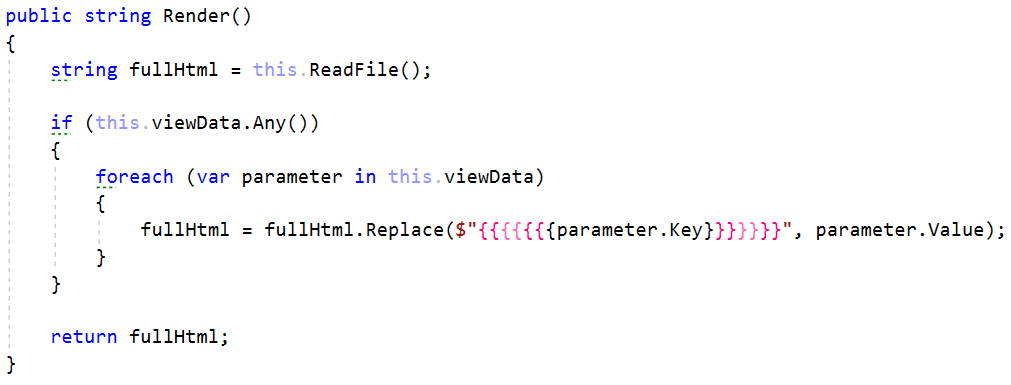
Now that we are ready with all the constants, we can start with the main part of the class, the Constructor and the members.



As you can see we receive the View Data directly in the constructor. We also receive a full path to the partial View we must render.

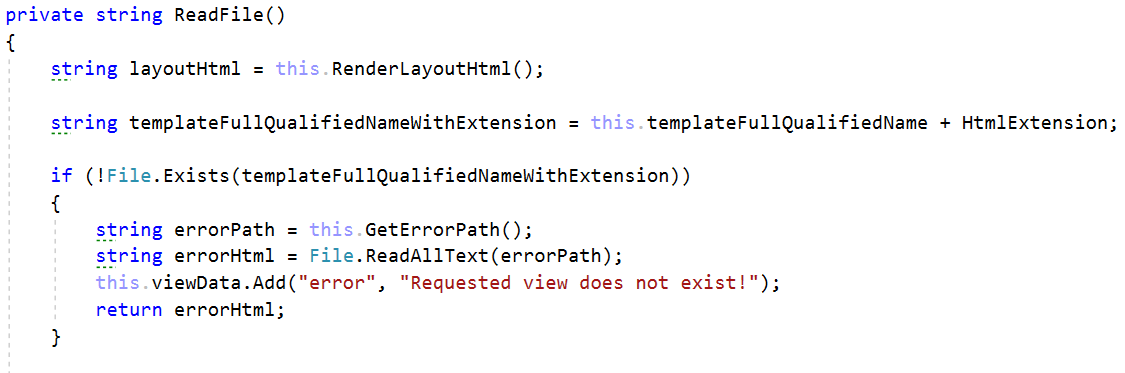
#### Functionality

And here comes the Render() method – the main component of the View class.



Familiar syntax, right? You’ve seen this in the WebServer, it is exactly the same. We get the full composed HTML, and we render the parameters into it, if there are any of course.

Let’s see how the HTML composition is done in the ReadFile() method.



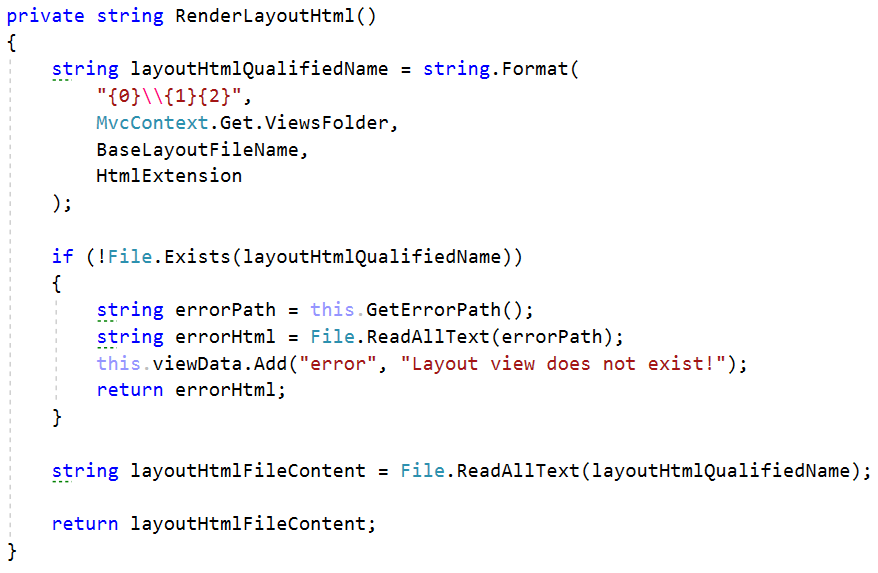
The first thing we do is render the Layout HTML, which we will see in a moment. Then we get the **full template** **name** **with** **extension** by using the template name with which the View class was initialized and the HTML extension. We get for example “Views\Home\Index.html”.

Then we check if the file exists, and if it does not exist, we return an Error View instead of the actual one.

First, we get the path to the Error (we still don’t have the folder and the Error.html view, but we’ll do that after we finalize the View class).

Notice how we add to the View Data a parameter “error” and an error message. That means that the Error.html view, most likely holds a “{{{error}}}” placeholder.

Now let’s go check the RenderLayoutHtml() method. It is very simple.



First we have to initialize the complete Layout view name, which is determined by the Framework (and by this method in particular). This design must be followed by any Application that uses our Framework. And the design is that the Layout file must be called exactly Layout.html and it must be in the Views folder of the Application project.

Notice that we are **checking** for an **error** here too.

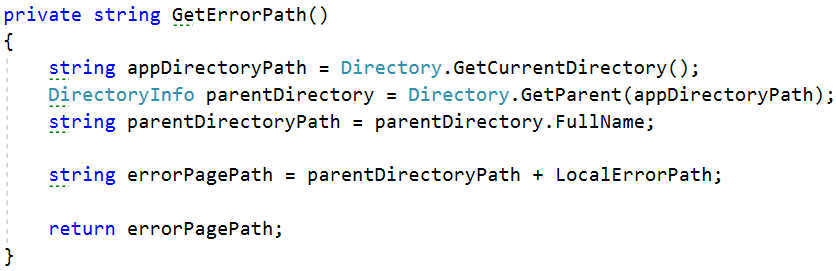
At the end, when all checks pass, we use the File.ReadAllText() method and return the HTML as a string.

Let’s return to the ReadFile() method, we still haven’t composed the HTML. That will be your task.

* **First**, you have to read the file HTML as string using the given **full qualified name with extension**.
* **Second**, you need to replace the content placeholder, in the Layout HTML **string**, with the string **HTML** of the View file.
* **Third**, you have to return the result HTML.



The last thing we need to do is implement the GetErrorPath() method.

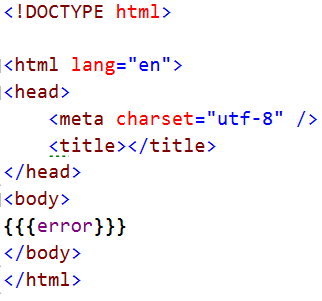


Due to the constant **switching between assemblies** and the fact that the work with Directories is inevitable.

With that our View Class is finished, but we left some holes – The Errors. Let’s do them really quickly.

### Errors

Create a folder Errors in the Framework project. Add an Error.html file to it. The file should have the following structure:



Notice how we have the {{{error}}} placeholder. That is where our error message will go from the View Data upon rendering.

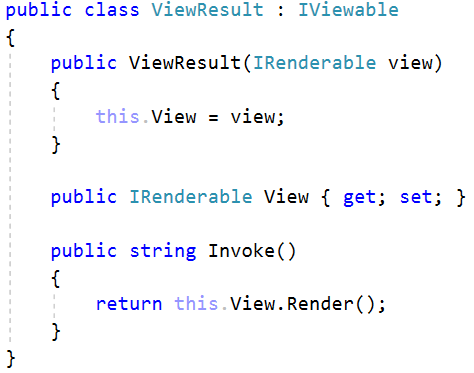
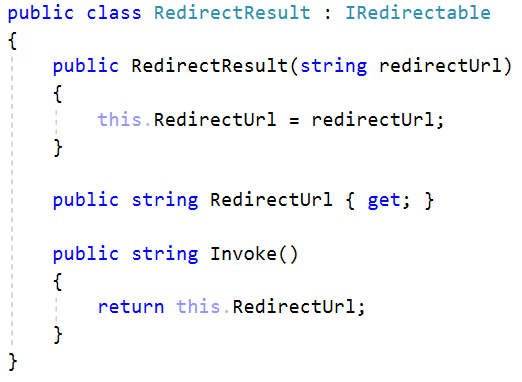
That was all you needed to do. We are done with the error handling, for now. 😊

### ActionResults

Remember the folder ViewEngine in the Framework project? Well, it contains some ActionResults and some Generics, which we won’t be needing anymore. So you can basically remove it, completely.

On that place, create a folder called ActionResults. This folder will hold the different implementation of the IActionResult Interface. We will have mainly 2 classes of that type:

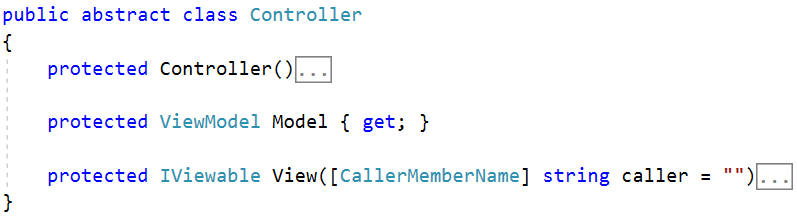
* ViewResult – A normal string content view result, which can be used to compose a ContentResponse.
* RedirectResult – A view result with a redirect url, which can be used to compose a RedirectResponse.

With that we can finalize the View Engine. But there is other functionality in the other components of the Framework that waits to be changed. Let’s directly jump to that.

## Controller Class

The second main component is the base Controller class. This class needs to be changed a little. For example, first, delete all generic methods from it and leave only the view with the single caller parameter, like this:



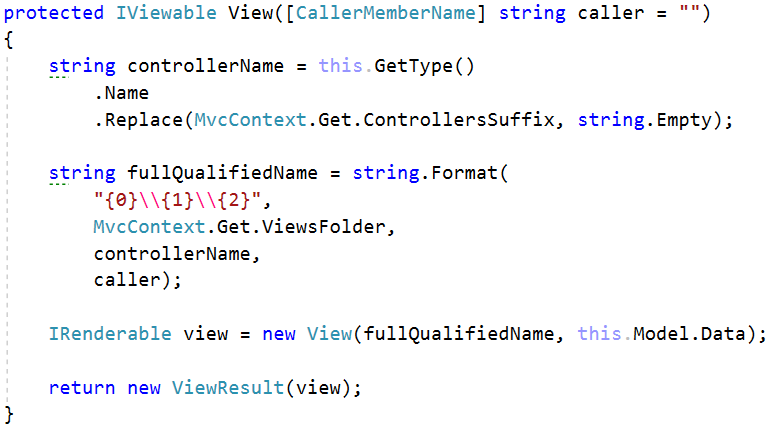
The controller has the ViewModel which we set up earlier, and a simple method which it will use. The method should return IViewable. The method should extract the Controller’s name (without the “Controller” suffix) and it should look in the Views folder for a View with name – the caller.

Example:

* “Controller = HomeController”
* “Action = Index”
* “View Path = Views\Home\Index.html”

### View() method

That being said, the method should look like this:



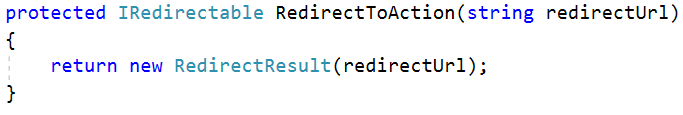
Notice how we create a View and we give it the path to the HTML file, by forming it from the **controller’s name** and the **caller’s name**.

Then we just return a ViewResult with the IRenderable view.

And with this we have almost finished the functionality that provides the applications the ability to render HTML Views from their Controllers.

But is returning ViewResults the only functionality we want in our controllers? What about the Redirect functionality? Well, we’ve already made it quite simple. Just add the following method to the Controller class:

### RedirectToAction() method



Aaand boom, we have RedirectToAction functionality.

Now the base Controller should provide more functionality than it currently does. For example, we need to have a functionality of checking if the data sent to our action and captured in our BindingModel is valid. This is a necessity which most frameworks support.

But how do we validate a BingindModel? Naturally, we will have to check it’s properties. But we do not know the type of the BindingModel, so we do not know what its properties types are. Well, that is why we will work with Attributes!

### Property Attributes

Go to the Attributes folder of the Framework project, and add a subfolder Property. In it, create an abstract class PropertyAttribute, which inherits from the Attribute class (like we did with the MethodAttributes).

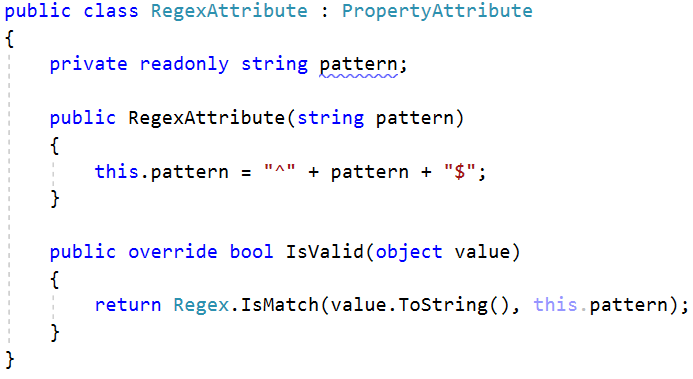
In the class, create an abstract boolean method IsValid() which accepts an object parameter – value.

Now let’s create the PropertyAttributes.

#### RegexAttribute

Create a class in the same folder, called RegexAttribute, and make it inherit the abstract PropertyAttribute.

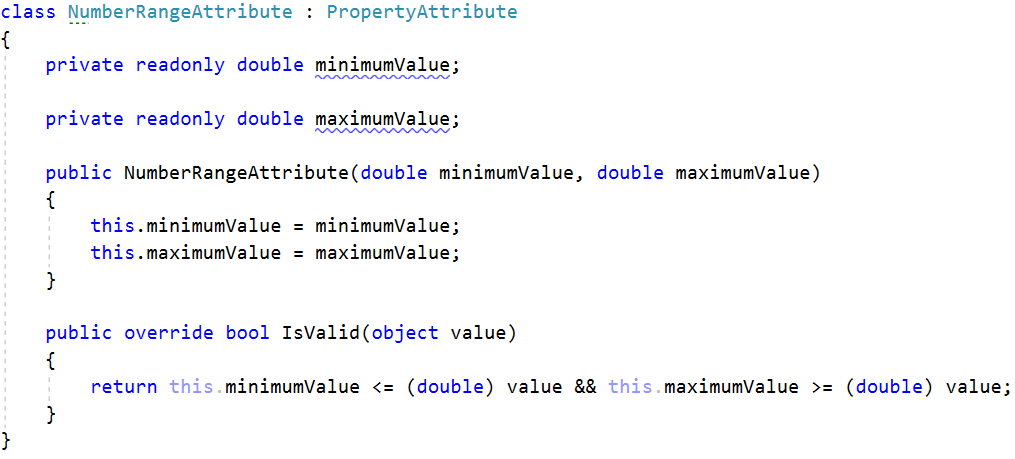
It should have a string pattern field, with which it is instantiated, and its method should check if the value matches that pattern as Regex.



#### NumberRangeAttribute

Create another class called NumberRangeAttribute which also inherits the abstract PropertyAttribute.

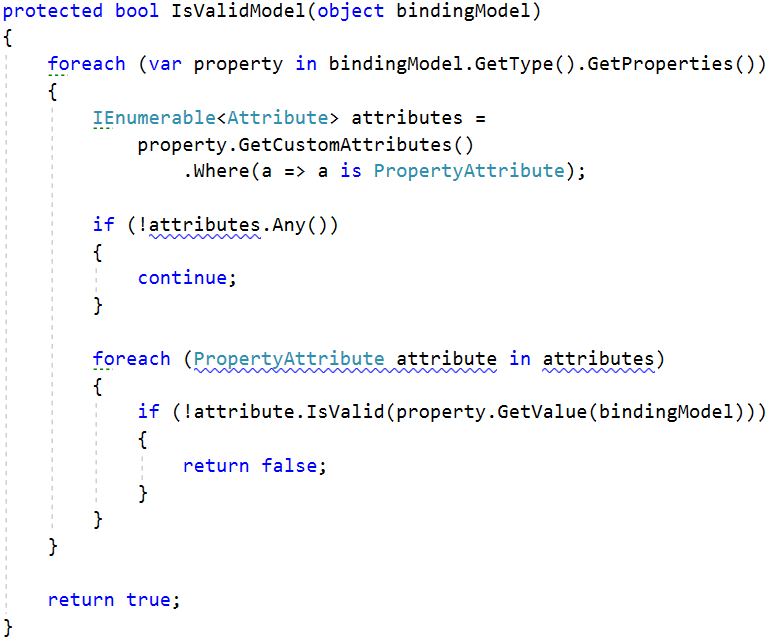
It should have a double minValue and a double maxValue, and its method should check if the **given value** is between the given values.



These attributes will be later used in our BindingModels as property attributes. Now let us write the functionality for validation.

### IsValidModel() method

This method will validate a given BindingModel’s values, by checking the attributes on it.



The method traverses all properties of the given BindingModel, for every property, it traverses its attributes, filtering only the PropertyAttributes, and validates the property’s value with every attribute.

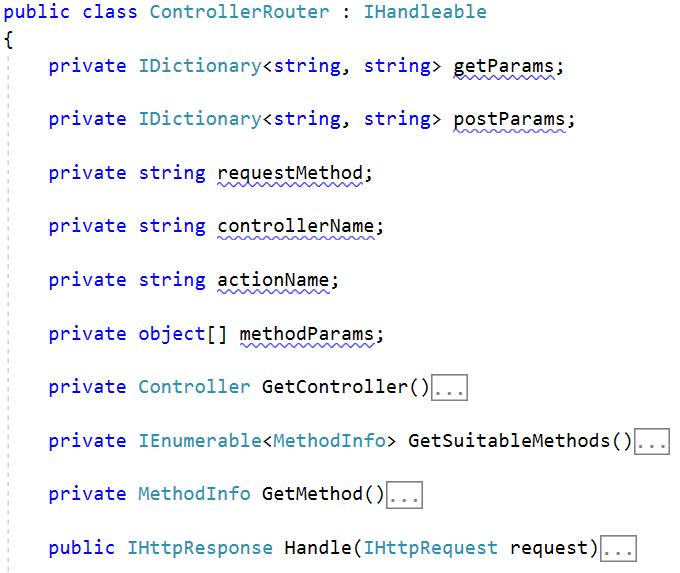
And with that we have finished most of the work on our base Controller.

For now we will leave it as it is, and move on to the next element, which is a very important element of our Framework.

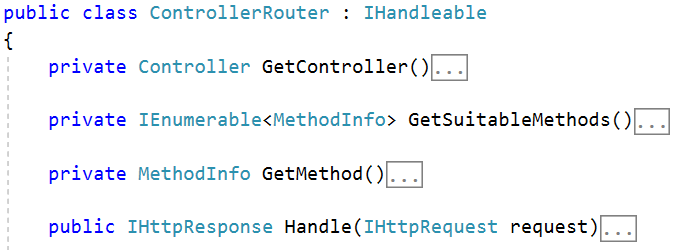
## ControllerRouter

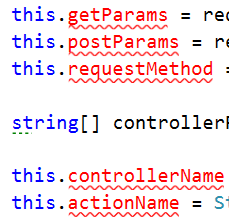
The ControllerRouter class is the Handler of the requests in our MVC Framework. As such it can be considered as the core component of it. And as such, it will undergo a lot of changes, most of which – purely **refactoring**.

So, if you haven’t already done the refactoring, the ControllerRouter should look like this:

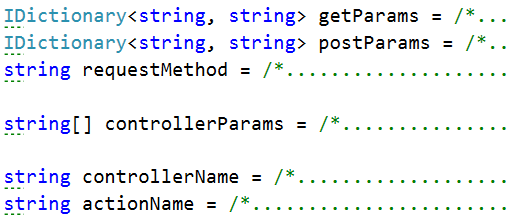


The first thing that we must see is that it saves up too much data. This class is only instantiated once, then why does it store up information for the request it processes? You can just delete the parameters of the request. Let’s leave the class purely functional - only with methods.



Ah this is better, if we need something in those methods, we’ll just pass it as a parameter. The Handle() method should be pretty broken right now:  


How do we fix that? Just make them local variables in the method, and we’ll be passing them to the other methods. The Handle() method is also pretty big… PRETTY BIG!!! So we’ll have to refactor it a lot (mainly, divide it into several other methods)

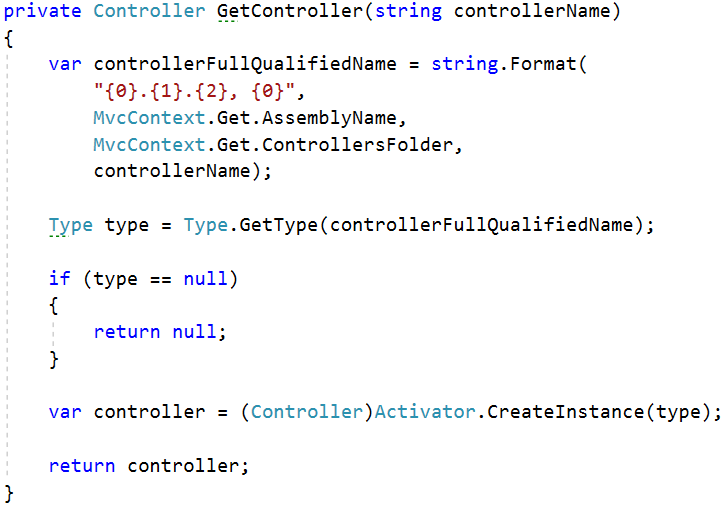


And here’s where the real fun begins. Let’s instantiate 2 variable – the controller and the method.



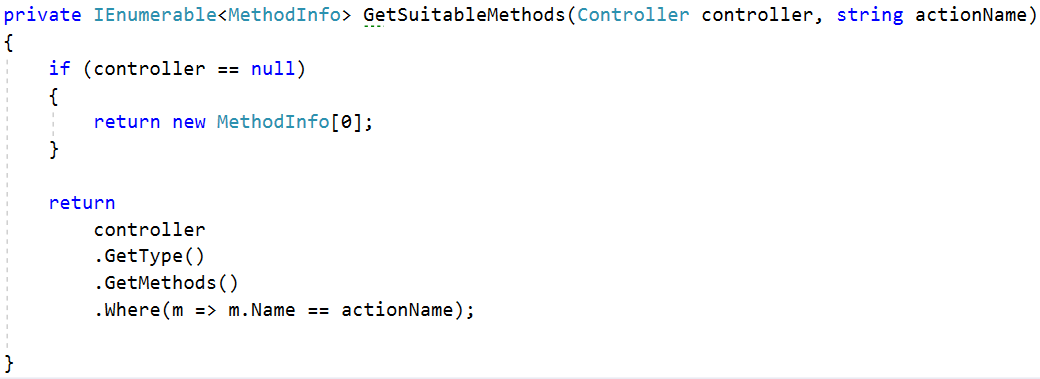
As you see we are passing new parameters to the Methods that we previously used with the class members to obtain the Controller and the Method. Let’s go fix those methods.

### GetController() method



See, not much different. We just added a new parameter, to replace the class member we removed. Now let’s do it for the other methods too.

### GetSuitableMethods() method



We just add the actionName parameter and pass it to the LINQ query down there. That’s all. Now for the last method.

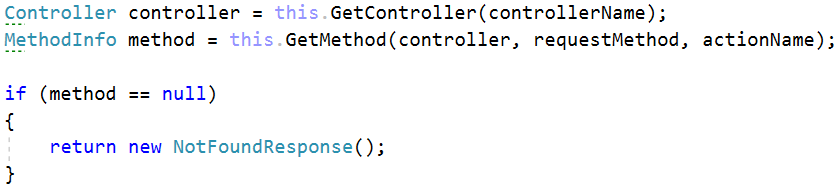
### GetMethod() method



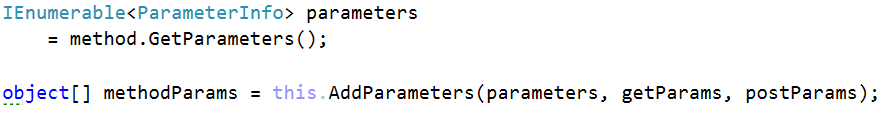
You do the changing of parameters in the method. It should be relatively easy.

Now let’s see how our Handle() method will work after we divide it to several methods.

First of course the check for the method is inevitable.



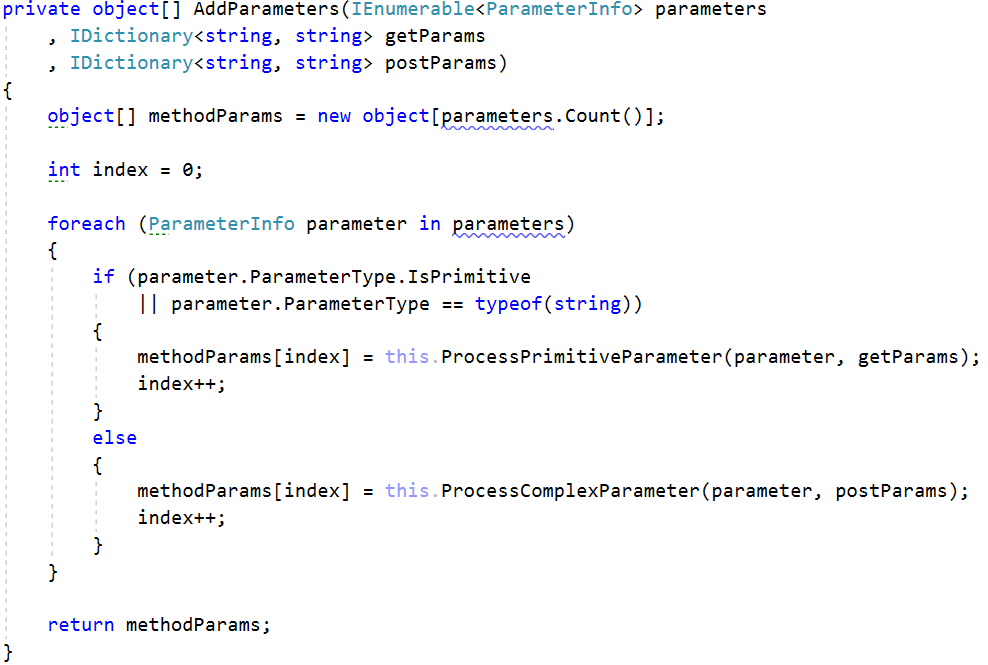
But after that is where the new stuff is.



Notice how those ugly massive foreach loops where extracted into separate methods. Notice, also, how we pass to those methods, the variables that we created. We no longer have this.getParams and this.postParams, we have them as variables. Now, let’s go implement those methods.

### AddParameters() method

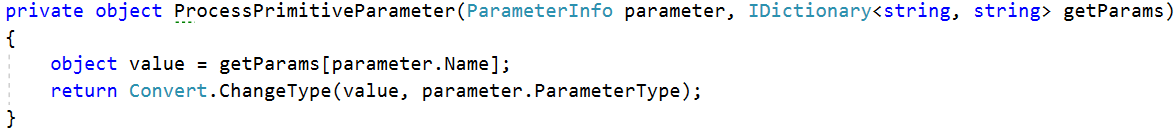
This is where we instantiate the parameters that we need to pass to the method, upon invoking it with Reflection. This is where the big foreach loop was. It is still there but it is not so big anymore, and looks better.



Now it looks a bit better, because everything looks a bit more ordered. The functionality for processing the primitive parameters and the functionality for processing the BindingModels is extracted into separate methods.

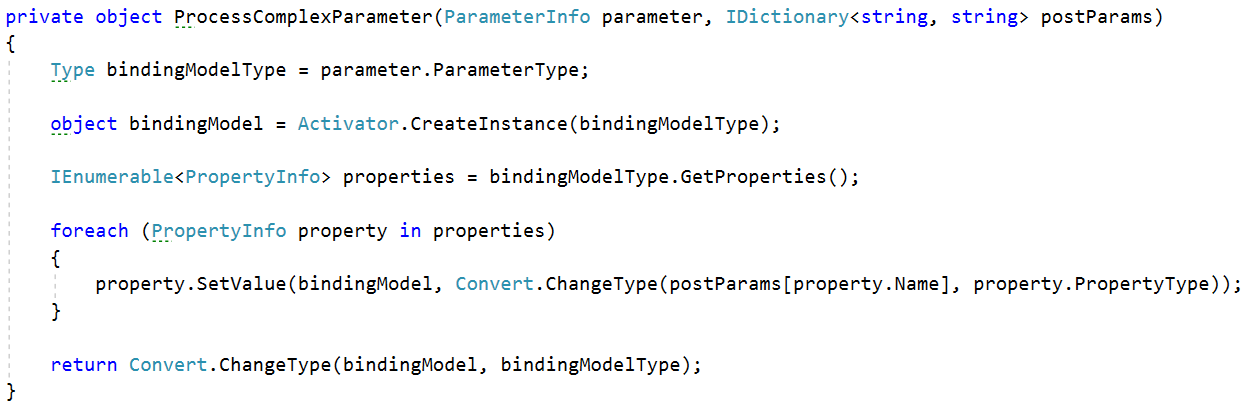
Let’s go implement those methods.

### ProcessPrimitiveParameter() method



There we go, done. Now it is indeed short, but let’s be consistent and write a bit more quality code. The other method is not so short. In fact, its longer almost longer than the method calling it.

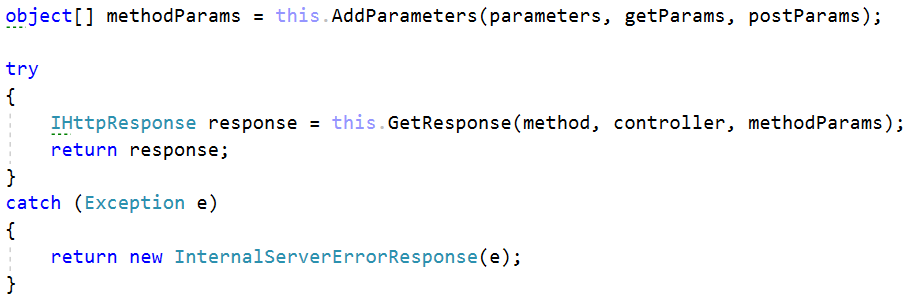
### ProcessComplexParameter() method



All these functionalities were directly placed into the Handle() method, which made it heavy, big, and hard to read and understand. Now that it is shorter and more divided, it is easier to look at, understand and ultimately DEBUG! If there is any problem with it.

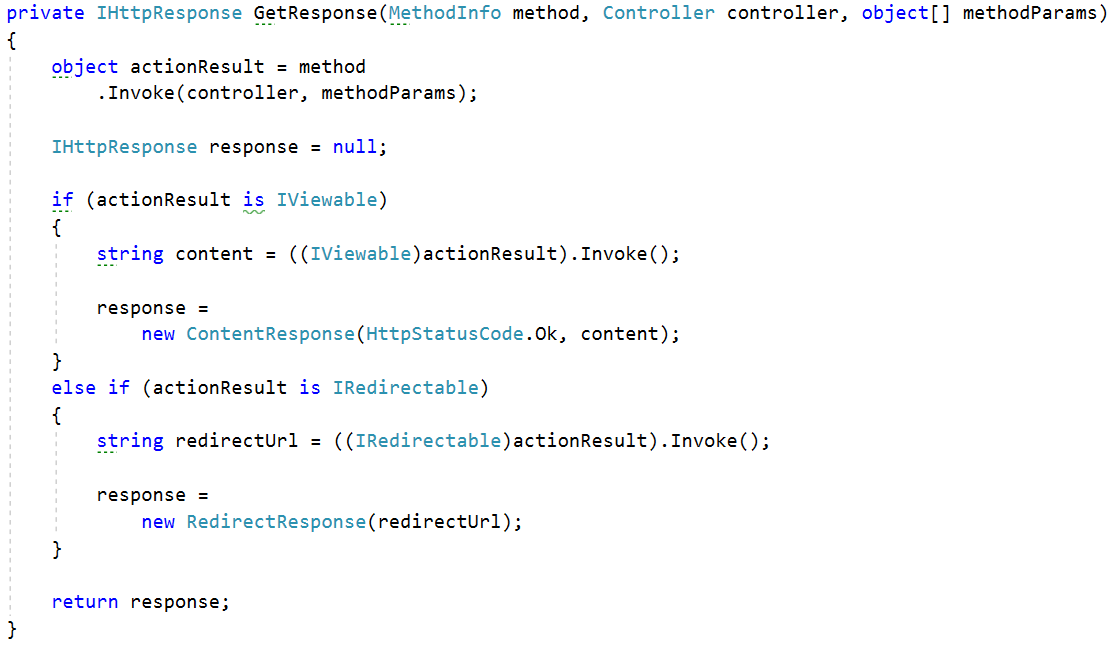
#### Divide and Conquer!

Now let’s return to the Handle() method, we did not finish there, we did not return any response.



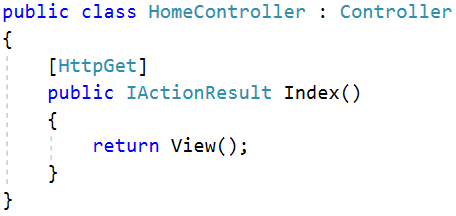
At the moment you get the method parameters, instead of doing the Response job in the Handle() method, let’s extract even that in another method. Let’s also wrap the whole result in a try catch block and return an ErrorResponse in case the method for getting the response fails. It is possible.

### GetResponse() method



Now this method looks a bit complicated, but it’s actually simple. Depending on what the method from the Controller returns, the method returns as a result a different response.

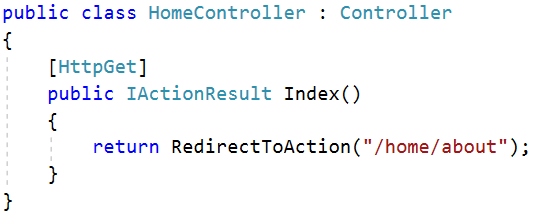
For example, if in our HomeController we say:



The method will receive an IViewable object and create a ContentResponse from it, because that’s what we did earlier in the base Controller class.

But…

For example, if in our HomeController we say:



The method will receive an IRedirectable object and create a RedirectResponse from it, which the WebServer will provide with a Location Header, and that will change the Client’s URL.

We also created that in the base Controller class.

In future if we add more ActionResult types, that would mean more If-else-if-else statements, which might be something we need to think about of a cleverer way to do it, but for now let’s leave it like this. It suits our needs.

And we are done with the ControllerRouter class. Our framework is up and running. From here on, its only extensions. But valuable extensions!

## ResourceRouter

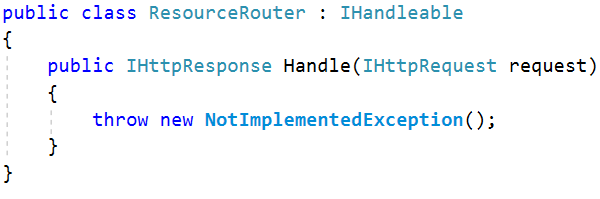
So, we have a ControllerRouter, which routes between Controller actions and returns Views. But what if we want our application to have resources? Bootstrap, images, etc. Well, we must provide our server with another Request Handler – The ResourceRouter.

### Important

This ResourceRouter will make our Framework place a design rule, which states that any application using it, must place its resources in a Resources folder. And the resources must be distributed in separate sub-folders, each sub-folder named with the file extension of the resource.

### Handle() method

Now it might sound complicated, but its actually pretty simple. Go to the Routers folder in the Framework project and create a class called ResourceRouter. Make it implement the IHandleable interface.

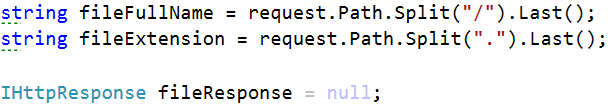


Now let’s add functionality to it.

First thing we need to do is extract the filename with extension and separately the extension of the resource. How do we do that? With the **path** of the **request** of course.

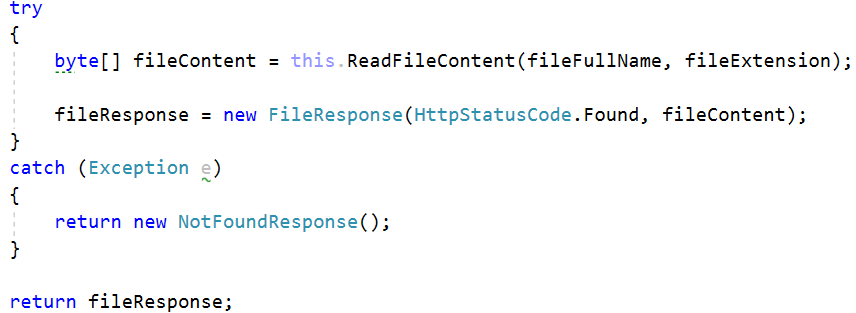
**Example request paths**: “/home/index”, “/users/profile?id=3” “/favicon.ico”.

The **Path** of the **request** is what comes after the **domain**. And if the client is requesting a file, it will be common sense for that file to be at the end of the **path**. So let’s do this.



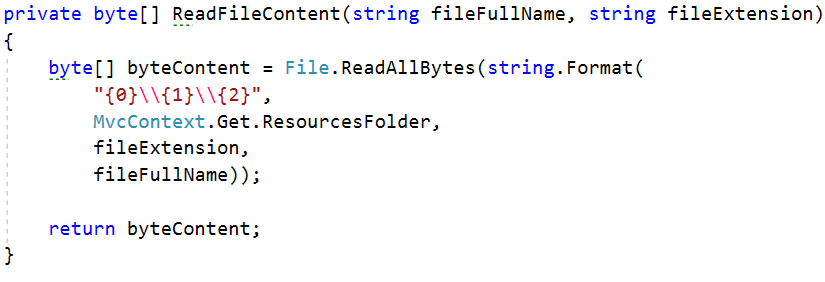
This code will extract 2 variables – the full name of the resource (Example: “favicon.ico”) and only the extension of the resource (Example: “ico”). The extension is needed to search for it in the Resources folder of the application.

And now we need to read the file as byte[] array and return it as Response. This will required some changes on the WebServer also.



Now the try catch block is inevitable, it is our Framework’s only error handling. But we see some new stuff here, like the FileResponse. This is a new response which we will implement on the WebServer.  
First, let’s implement the method we used to extract the file content.

### ReadFileContent() method



Relatively simple… Instead of File.ReadAllText we use File.ReadAllBytes. Naturally, if we are sending files over the network we cannot just turn them into text, which is where all the bytes thing comes from.

One more thing we see is the MvcContext.ResourceFolder. This will be for you to do, it is simple. 😊 Like we said, by design that folder is called “Resources”. Just create a property in the MvcContext class and initialize it from the MvcEngine.

And with this we are done with the ResourceRouter. Now there are a series of changes we must do to our WebServer in order for this work.

## WebServer changes

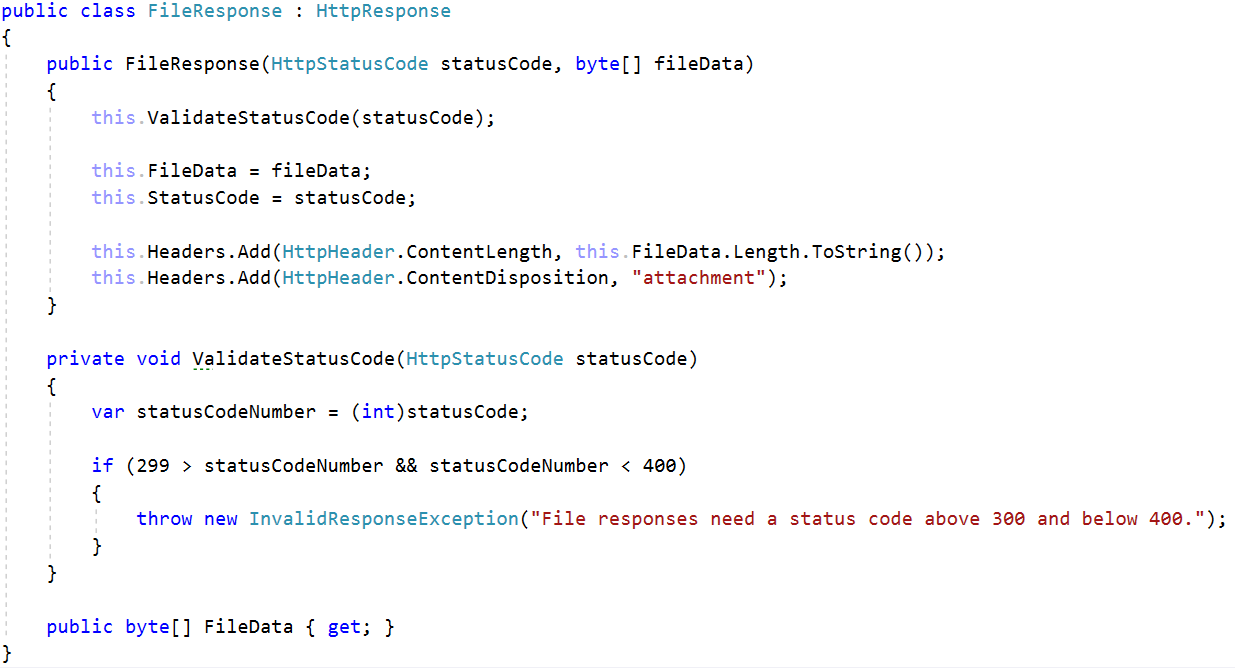
Let’s perform the needed changes on the WebServer, possibly the final changes to it.

### FileResponse

Go to the Http\Response folder, and create a class named FileResponse. Make it implement the IHttpResponse interface.

It should be initialized with a fileData byte[] array, and also have a property where to store that data. And also a status code.

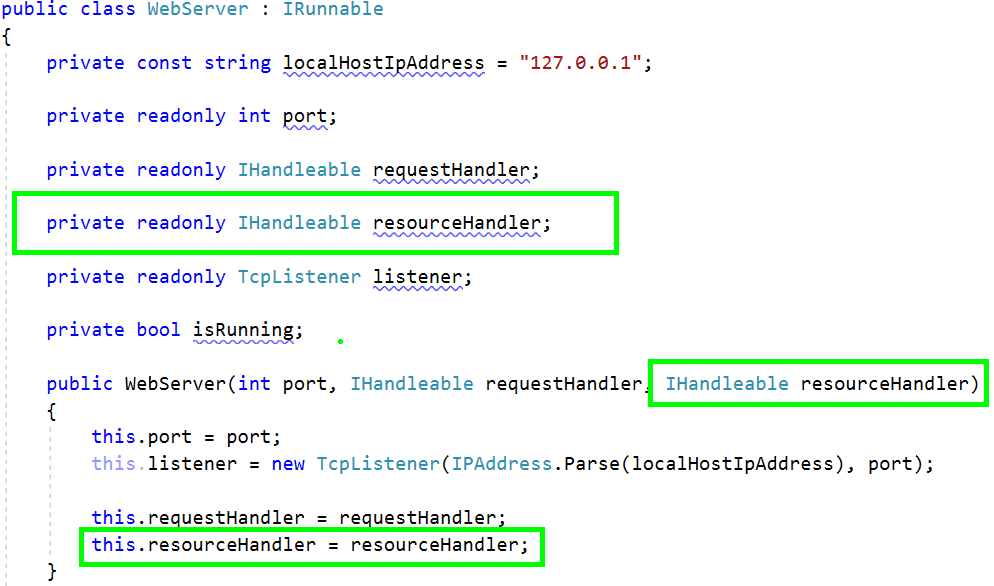
In the constructor, it should set 2 special headers for the file data we will be sending trough the network.



There is nothing more to do on the Responses side. Our next stop is the WebServer class itself.

### WebServer

Go to the WebServer class, and there we will add another IHandleable handler, where we will store the ResourceRouter.



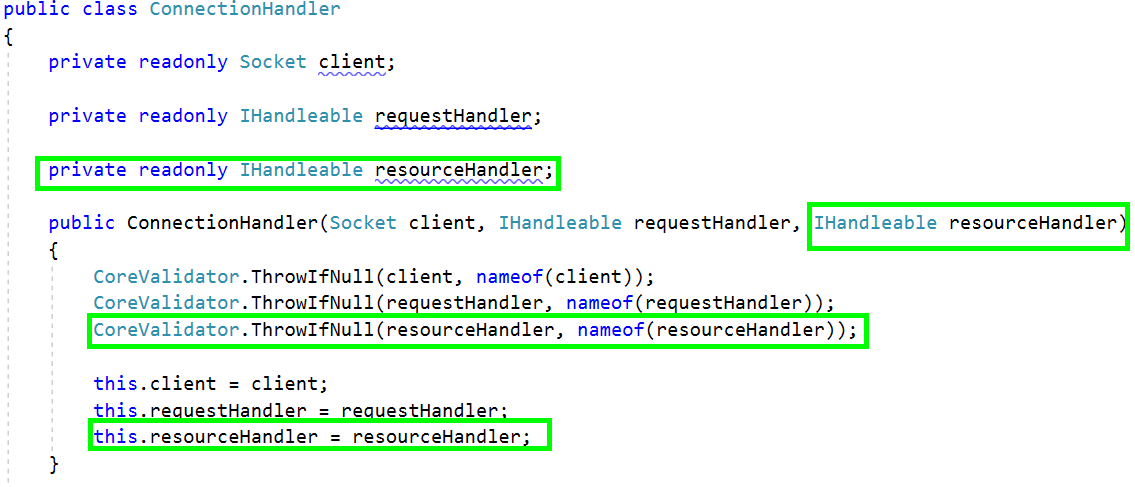
Now, this would naturally mean that we’ll have to add it to the ConnectionHandler class too. In the ListenLoop() method of the WebServer class, change the connectionHandler implementation to be so:



Now let’s go change the ConnectionHandler class.

### ConnectionHandler

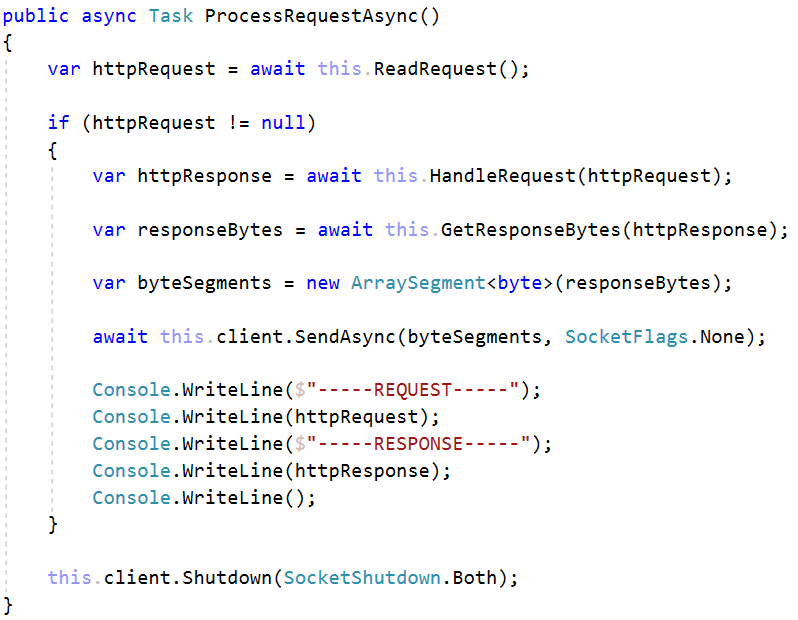
Go to the ConnectionHandler class and add a field for the ResourceRouter.



Now we will have some changes to the class’s methods. But those changes will be again dividing into methods, so they will be simple.

Let’s see how the ProcessRequestAsync() method looks.

#### ProcessRequestAsync() method



Depending on the type of request, we must use a different handler, which is why we have the HandleRequest() method.

Depending on the type of response, we must get different response bytes to send, which is why we have the GetResponseBytes() method.

Let’s go implement them.

#### HandleRequest() method



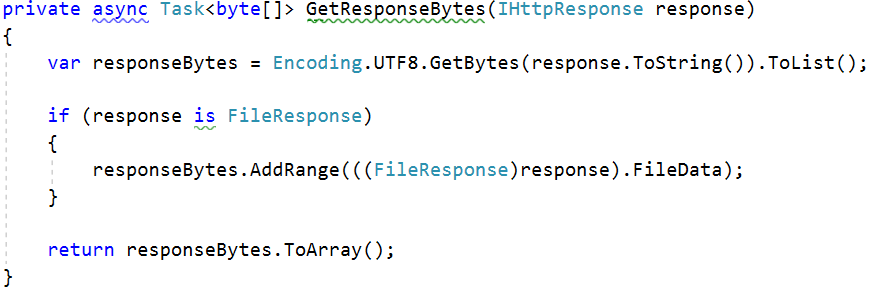
Now this is actually very clever. A valid request path contains a dot (“.”) only if it requires a resource. We will use that hack to distinct our requests.

So, if the path contains a dot, we use the resource handler.

If it does not, we set the sessionId using the server’s method, we handle the request with the request handler and we set the response session using the server’s method. Those 2 method come from the skeleton. They’ll be important for the Session Management, which we will do later.

#### GetResponseBytes() method

And now for the GetResponseBytes() method.



The usual behavior is to only “**byteify**” the response as a string, but in the case of a FileResponse, we must also attach the FileData. This is what this method is used for.

And with that, we finish with the changes on the WebServer.

## Session Management

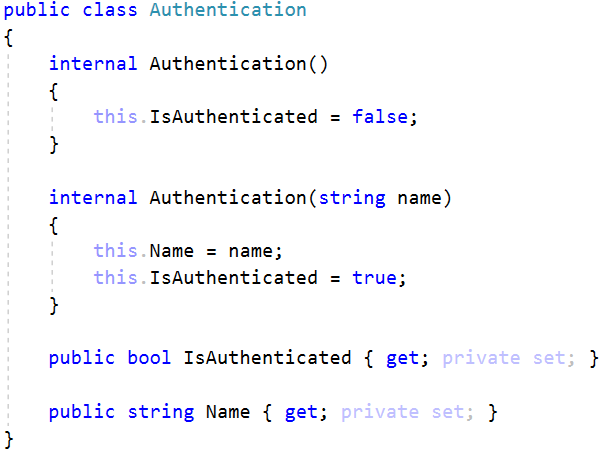
And thus, we reach the final part of our MVC Framework functionality. The Session Sanagement.

The Session Management will introducte one new class, and some changes to the base Controller class. The session management will help the applications using our Framework, check if the user is logged in an authenticated.

Let’s start with it.

### Authentication

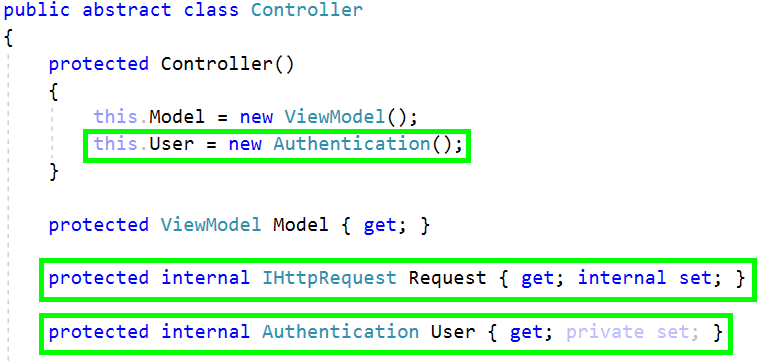
In the Framework project, create a folder Security, and in it, create a class Authentication.



This class will store our User’s session.

### Controller

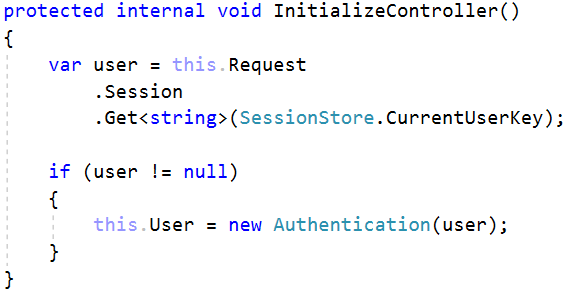
Now let’s go change the Controller class. Go into the class, and add 2 new properties. One of them should be an IHttpRequest, it will be needed for the Session. And the other should be an Authentication property called User.



Now, these things need to be initialized, each time a request is sent, so we will add a method for that.

#### InitializeController() method

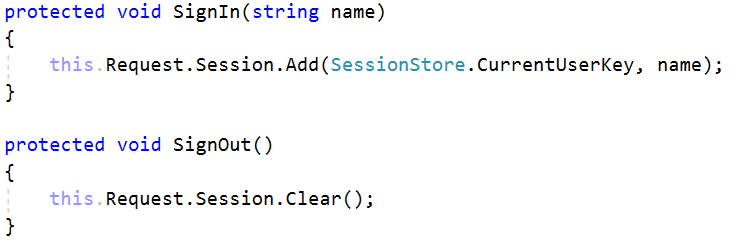
This method will initialize the Session in the controller, each time the Controller is initialized. Or in other words, each time a request is sent.



By extracting the current user from the session it sets it to the Authentication property. Which will help the current Request Session authenticate itself in the Controller.

Now, these methods and properties are quite Internal, and that will prevent the Applications from using them. Let’s provide some kind of logic for the Applications to Login a user and logout him respectively.

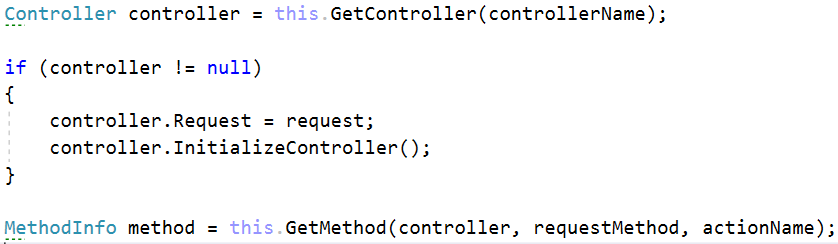
#### SignIn() and SignOut() methods



This is done with these two methods. The first one adds to the Session a username, and the other one clears the Session. Now, remember those methods on the server that were setting the Request and the Response sessions. If we set a session here, the client will receive it as a Response. If we clear it, the Client will receive nothing. If the Client receives something from the Server, he’s bound to return it back on the next request, which will be used, to Authenticate himself.

### ControllerRouter

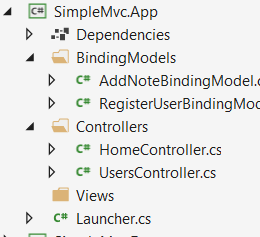
Last thing to do is add this If check to the ControllerRouter class, to ensure the Controller is Initialized.



And with that we are done, we are officially done with the framework. Our MVC Framework is ready to go, Extended and easy to use. Now let’s rebuild the Notes Application and check it’s usage.

## Notes Application

All the old Views and ViewModels were unneeded which is why we deleted them. The application should look like this at the moment.



Well let’s fill it up. First let’s create the Views, since they will be HTML pages.

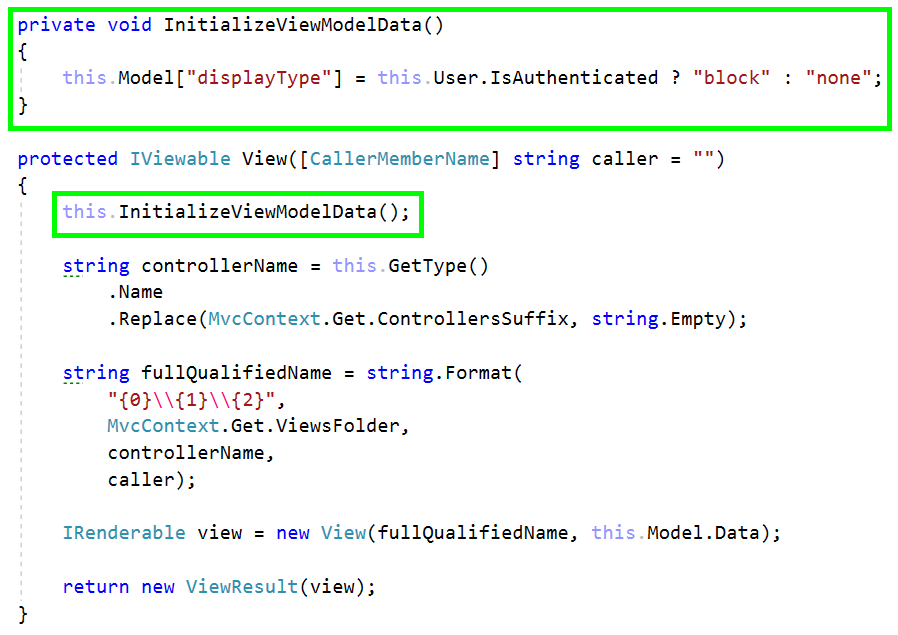
### Layout

In the Views folder, create a Layout.html page, which will be our **Layout**.



Notice how we have an image that requests for a resource, and some placeholders in this view. One of them is obvious – the {{{content}}} we already know that, that is where our View will render.

The other one though is interesting. Now this is the Logout block, and it should be only visible if we have logged in. Now go in the Controller class of the Framework project and do the following 2 things and do the following.

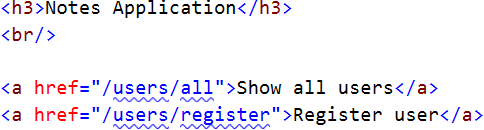


This will ensure that every time we View something the Logout button will not appear unless we are **Authenticated**.

The next step is to create a folder Resources. Then put the image head.jpg in it. You’ll find the image in this Exercise’s Resources. Now, remember! The image goes in a separate sub-folder depending on its extension.

### Index

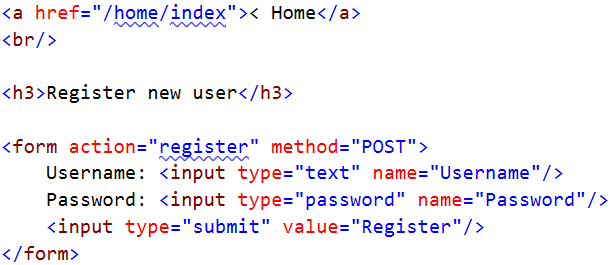
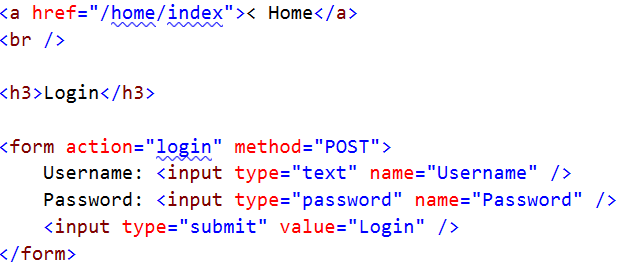
Next stop, the Index.html in the Home subfolder.



It’s the old view, but in pure **HTML**.

### Login & Register

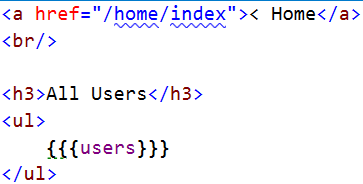
Create the subfolder Users in the Views folder, and add the casual 2 Login.html and Register.html.



They are quite authentic.

### All

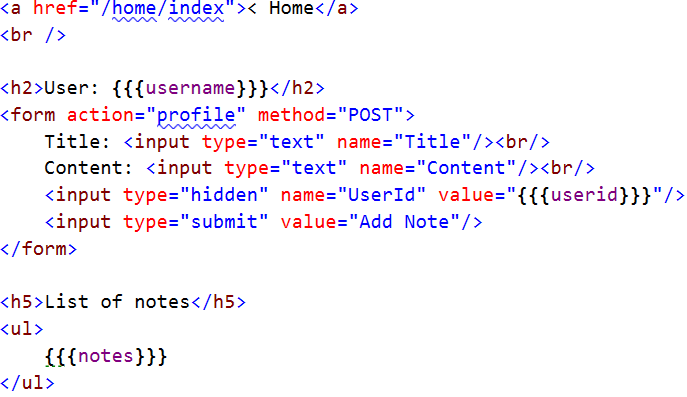
Next one is the All.html view, which views all users.



Another placeholder we have here. Something will be rendered in the UsersController it seems.

### Profile

And last but not least, the Profile.html. Which also contains information about the notes.

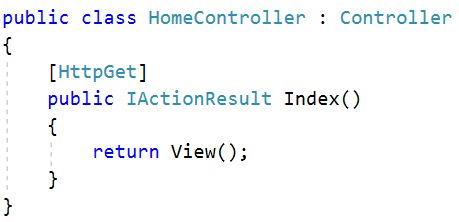


And these are all the Views.

Now, these Views are cool and all, and they will Render perfectly. But someone must write the functionality for this.

### HomeController

This controller remains the same, it has no additional logic.



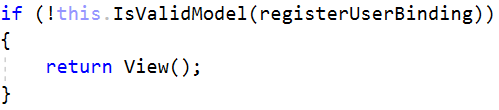
### UsersController

This controller however, has a lot of work in it.

#### Register Post

First stop, the Register post.

The only thing we have to do here is validate the BindingModel. We already know how.



#### Login

Now this is something new we are adding to our UsersController. The Login functionality. Upon login, the Controller should check if everything is valid:

* If the given user is existent
* If the given password matches the user’s password

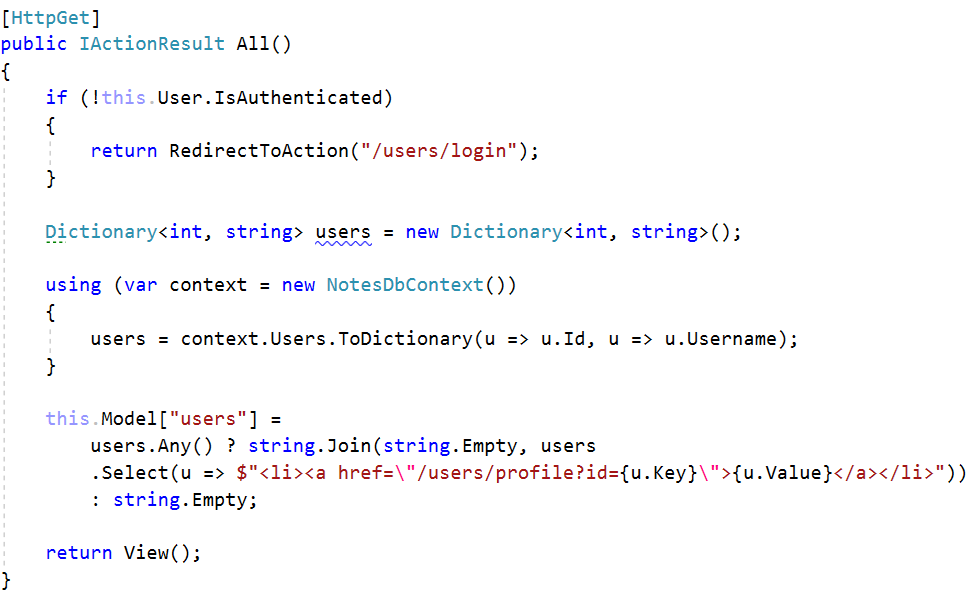
If the checks fail, the Controller should return the same View.

If everything passes, the Controller should SingIn() the user, and Redirect to Index.



#### All

The next action is All. Now here we will have some real rendering, and we will work with the ViewModel we implemented.



You see how we check first if the User is Authenticated and if its not we redirect him to the **login**. This is what the **Session Management** was necessary for.

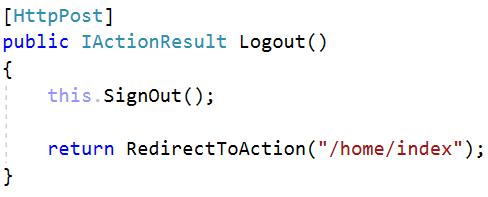
The rendering is obvious, we just set the value of the placeholder we put earlier in the View, to the data we want to render.

#### Profile

The Profile GET action also has changes in the rendering. But they are for you to do. The Profile POST action remains unchanged. It only added the Note to the database.

#### Logout

Last action! We have Login, but what about Logout? Well, this is where we implement it.



This is all that is needed. Just the SignOut() method call, and a redirect. And with this. The application should work just fine.

## Refactoring

You should have pretty much a lot of knowledge in C# Development by now. Assuming you know OOP, you should know its principles and the rules of **High Quality Architecture**. As a bonus task, optimize and refactor anything you think can be fixed and ensure maximum quality on the Framework and Server.