# 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: MVC Framework – Advanced

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

We will now extend the Framework, so that we can build dynamic and functional MVC Web Applications which will be hosted on the Handmade HTTP Server.

**NOTE**: Some functionalities will get removed, and new ones will be added on their place. This process is essential in development... Things get deprecated over time.

# View Engine & Security

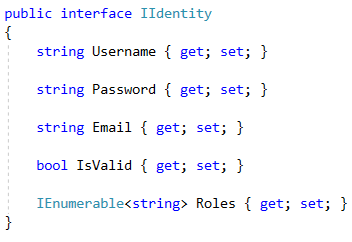
A normal Web Framework supports a Security mechanism, which ensures comfort in developing applications, which require Authentication & Authorization. Our framework will also support this type of functionality. Our framework will also need a beautiful View Engine.

## Security

The first thing with which we will extend our framework is the Security functionality. Let’s implement that then.

### Identity

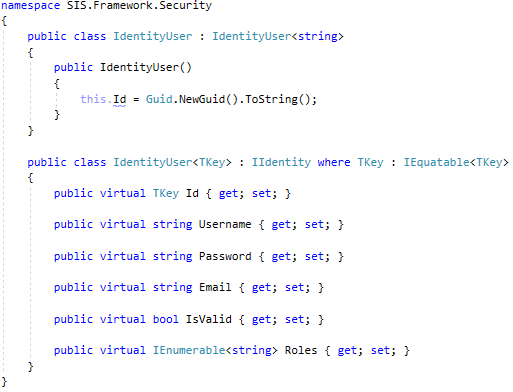
Create a namespace, called Security, and in it, create an interface named IIdentity. The interface should look like this:



Just a normal Identity contract, which will be used in our Security mechanism. And now you might be thinking:  
“Username – string, okey...   
Password – string, okey...  
Email – string, okey...  
but why are the Roles – strings. When normally we persist them in the database”.

Don’t worry. With the implementation of the interface, you’ll see how you can modify that easily in your application.

Create a class, in the same namespace, called IdentityUser. Its implementation would be quite interesting:



As you can see the Framework provides its consumers with a **generic** IdentityUser, which can **store** its **key** in any way possible. The default behavior is a string – **GUID**.

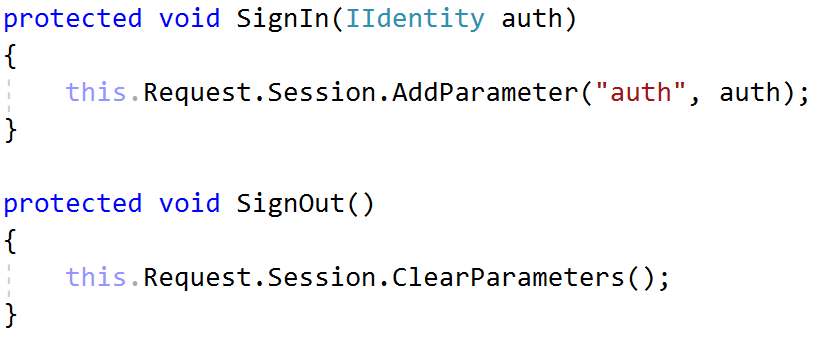
As you can see the IdentityUser has all of its properties – **virtual**, so that they can be easily overridden, if needed.

This should be enough for now. Now let’s go include these magical components in the main flow.

### Controller

Next thing we need to do, is provide our consumers with the mechanism of **Authenticating** clients. This should be done in the Controller, as it is the **main** **component** that **consumers consume directly** from the Framework.

Extend the Controller, by adding the following 2 methods to it:



By storing a **particular object** in the SessionStorage, we are creating a state for the current client.   
By storing the Identity of the current user, we are creating an **Authentication**.

This is the **Authentication algorithm** we used in the **beginning** when we had just a little portion of what has now become our **MVC Framework**.

SignIn() – stores the **User data** in the SessionStorage, so that it can be easily accessed.

SignOut() – clears the Session **data**.

This is all there is to it, you now actually have an **authentication algorithm**, with which you can **check** if **clients** are **authenticated**. But you should also have a convenient way of accessing the current Identity.

Add the following property to the Controller class:

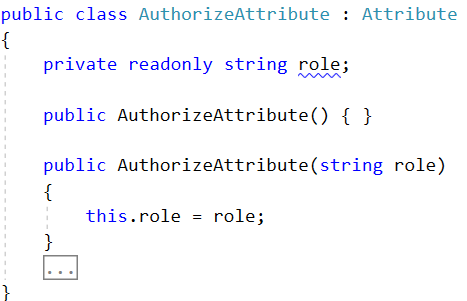


This will help us easily **extract** the **current Identity**, if there is any. If there is not, it should return **null**, according to the GetParameter() method behavior. (of course, if you’ve followed the document … :)).

And with this we should be finished with the Controller class, but we are not ready with the Security yet. Apart from Authentication, Security includes another practice – Authorization. So we need to do that too.

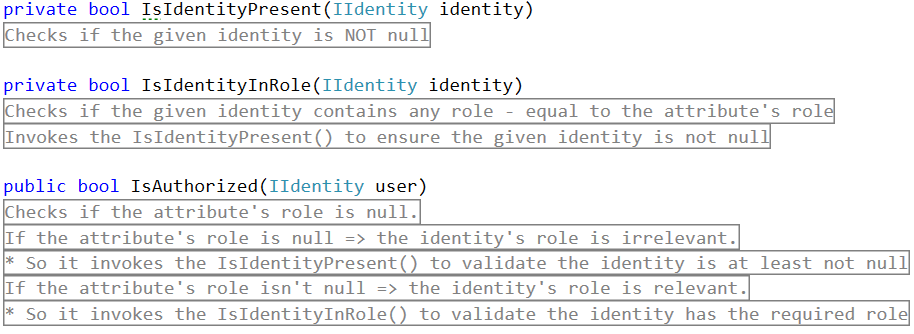
### AuthorizeAttribute

In the Attributes namespace, create a **sub-namespace**, called Action. In that **sub-namespace**, create a class called AuthorizeAttribute. It should look like this:



This Attribute will be put on Controller Actions to indicate that the Client accessing that functionality must be Authenticated and / or must have a particular Role.

This will be indicated though several methods, which are in that little “…” region you see on the screenshot. The methods will be given to you with explanation of their behavior, but the implementation itself is yours to do:

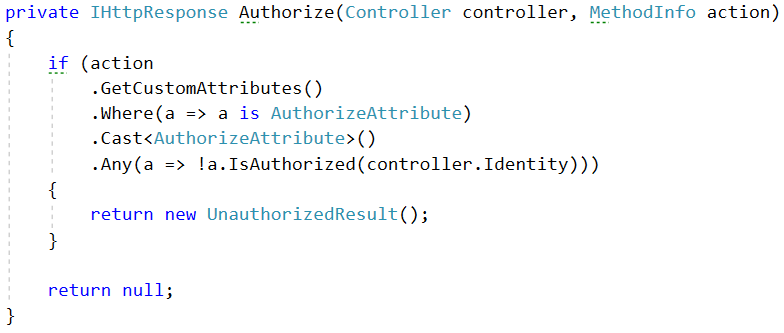


That will be enough for this attribute, now let’s do the thing we do with all newly added components – include them in the main flow.

### ControllerRouter

To implement Authorization in our Framework, we must modify its main component – the ControllerRouter.

Add the following method to the class:

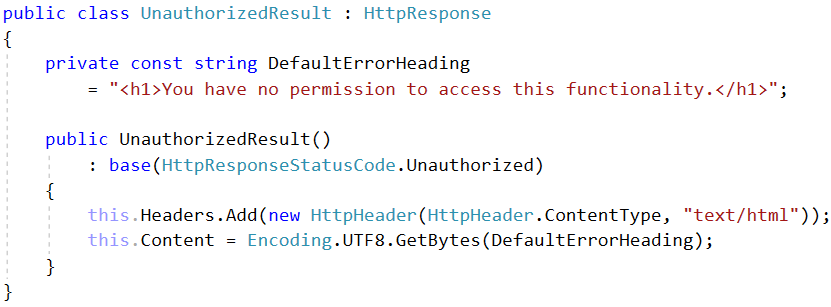


This method just checks if the given action contains an AuthorizeAttribute, and if there is, invokes the IsAuthorized() method to the **controller**’s current Identity.

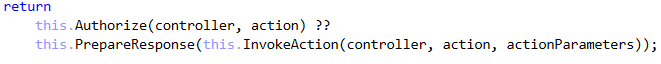
The result is:

* NULL if the current Identity satisfies all Authorization requirements.
* UnauthorizedResult if the current Identity is not Authenticated or does **NOT** contain the required Role.

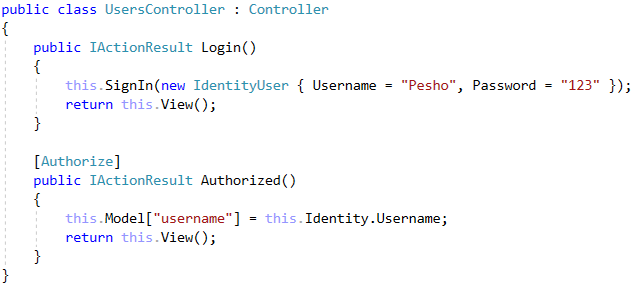
**Side** **NOTE**: The UnauthorizedResult is a simple class in the WebServer project’s Results namespace. It looks like this:



Now that we have the Authorize() method like that (returning HttpResponse), we could just modify the return functionality of ControllerRouter’s Handle() method to look like this:



And with this we are finished with the Authorization functionality and with the Security of our framework. Go test it out:



If you’ve **implemented everything correctly**, you shouldn’t be able to access the Authorized() action without accessing the Login() action first.

## Tazer View Engine

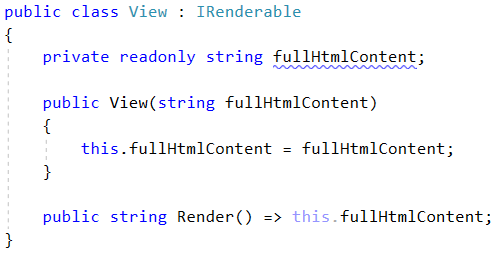
The next big thing is the View Engine extending. It will be quite fun and dynamic to do so get ready. Introducing… The Tazer View Engine.



Okay, now serious. We need to get going.

### View class

The first thing we will modify is the View class. We will cut a lot from it, mainly because the functionality’s place is not there. It should look like this:



Pretty clear, yup. That’s because this class is only for Viewing the content.

### ViewEngine class

Create a class in the same namespace, called ViewEngine. This class will do the main job of rendering Views. The functionality we need to implement is Error View, Layout View, and Dynamic Rendering View. We also need to implement one thing called Display Templates – **HTML documents** describing how a **complex** **objects** should be rending. We will also implement **Collection** rendering functionality.

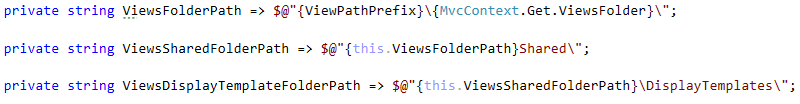
First, we will start with the static things, then we will get to the dynamic ones.

Implement the following Constants:

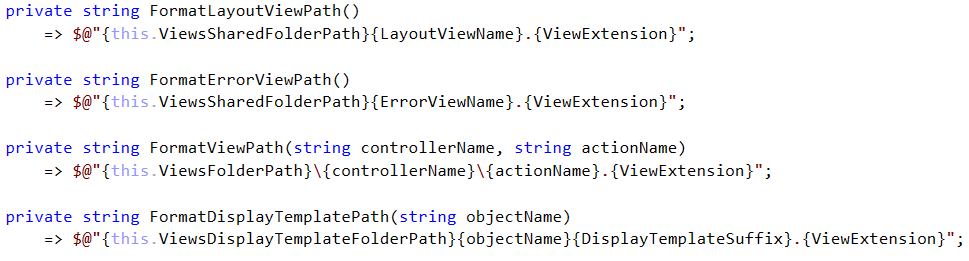


Some of these **constants** are taken from the previous View class. Most of them are new though, not to mention that there is a **Regex pattern**. We will need the Regex Pattern to match the **collection** **placeholder**, you’ll see that further in the document.

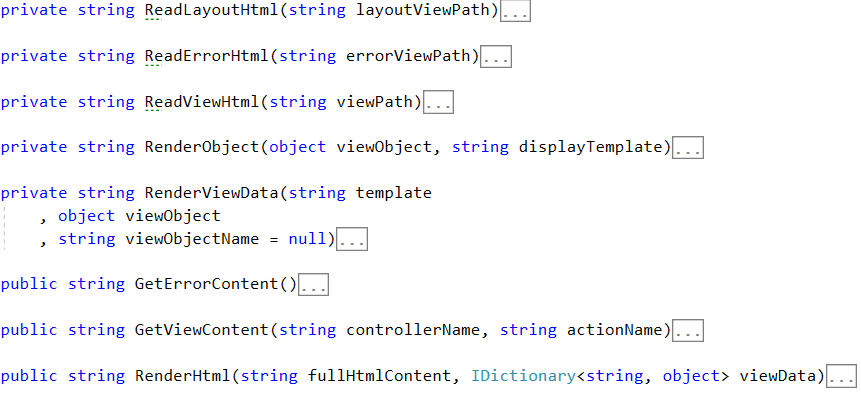
The next thing we will implement is the “**constant**” values which are **evaluated compile-time**:



And now for some **helper methods**, which will combine the constants above:

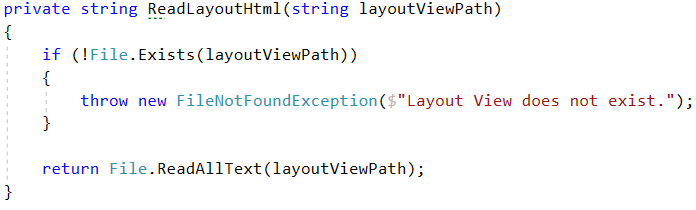


All these **properties** and **methods** we just implemented, will be needed in the next few methods, which describe the main **View Engine Rendering** mechanism. Create the following methods, and we will describe them one by one:



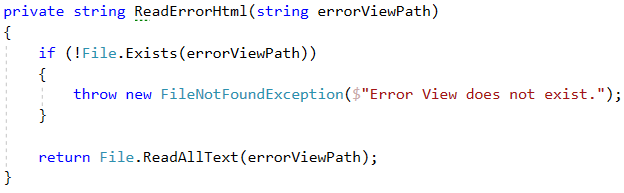
#### RenderLayoutHtml() method

Let’s start with the RenderLayoutHtml() method. Now you must have already noticed the problem that you need to import the **CSS**, **JS**, and set the **title**, set the **footer**, and **boilerplate** a lot of the **HTML**, for **every single view**. This is dumb. That is why we will create a Layout **HTML** which will act as the base template, and every sub-template will hold a portion HTML which will be rendered to it.

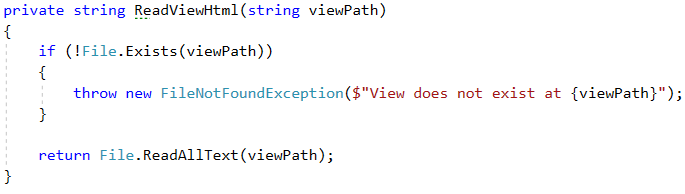


This is all there is to it. The next 2 methods are quite analogical, so we will just state them below.

#### RenderErrorHtml() method

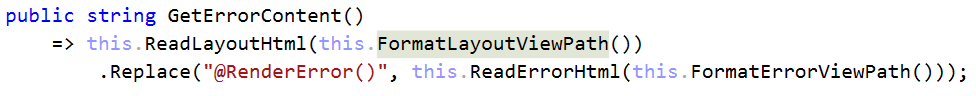


#### RenderViewHtml() method



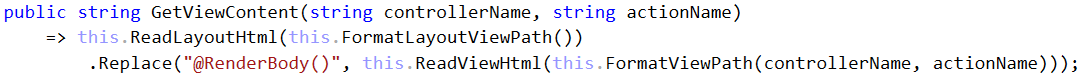
Now let’s go see the public methods. We are skipping the RenderViewData() and RenderObject() methods, even though they are the next in order, but that’s because they are quite complicated. That’s where the DisplayTemplate rendering mechanism is. We will show them last.

#### GetErrorContent() method



This method will be used to generate the **full content** for Errors, in order to render Server-Side errors in a more appropriate way. It uses the **Layout HTML** too, which means your **Error** view will also be **partial**.

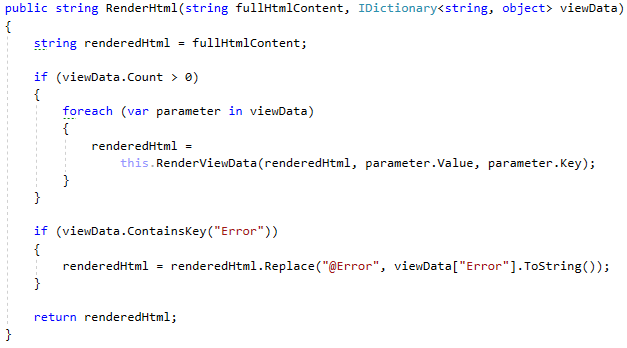
#### GetViewContent() method



This method will be used to generate common Views. It uses the **Layout HTML**.

#### RenderHtml() method

When the **HTML** (View or Error) is **generated**, you should use the RenderHtml() method to render data to it. The mechanism is different than before, as the **Rendering** is done **post-factum**. But this way it’s better as we can **unify** the **errors** and **views**.



And here’s where the main thing comes. The rendering of the View Data.

#### RenderViewData() method

Before we show you what this abominable creation of some wicked mind, you should at least understand how the algorithm works:

The method receives 3 main parameters – template, viewObject, viewObjectName.

The template is the **HTML** onto which the **given object** will be rendered**.** The **name** of the **object** defines the placeholder onto which the **given object** should be **rendered**. The placeholder is @Model.{viewObjectName}.

**Example**: <h1>@Model.Username</h1> = <h1>Pesho</h1>

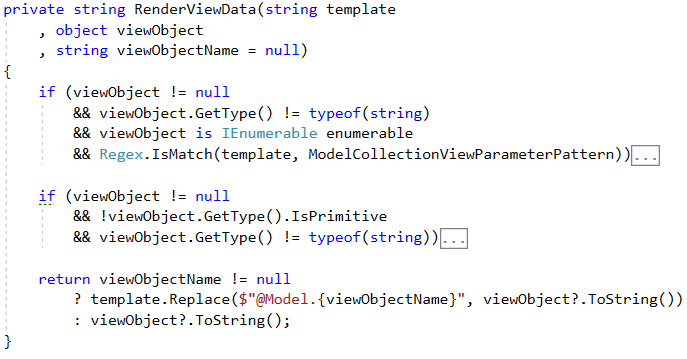
The **replacement** of **placeholder** and **value** is done on the template parameter. When the replacement is **finished**, the template is **returned**, with its **replaced state**.

**IMPORTANT**: In case the viewObjectName is **NULL**, that means that this is **NOT** an **object** which **should be replaced** over the template. In that case, the viewObject is returned with its **string** **representation**, **instead** of the **template**.

The viewObject itself is an **object**, which is taken from the ViewData dictionary.   
There are **3 cases** that the Tazer View Engine handles.

* The viewObject is a **collection** of some sort.
* The viewObject is a **non-primitive object** of some sort.
* The viewObject is a **primitive** of some sort.

Now let’s see how the method handles those 3 states:



##### Collection Rendering Mechanism

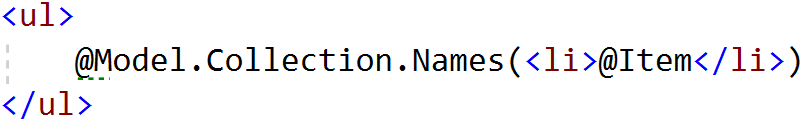
The first **if statement** checks if the viewObject is a **collection**. If it is, then the collection must follow a particular pattern on the template.

That pattern is @Model.Collection.{collectionName}({itemFormat}).

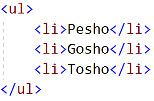
The collectionName is directly linked to the viewObjectName, in order to distinct when **multiple collections** are **rendered** over a **single template**.

The itemFormat is how the items should be formatted. It must contain @Item. That’s the item placeholder, where items will be rendered.

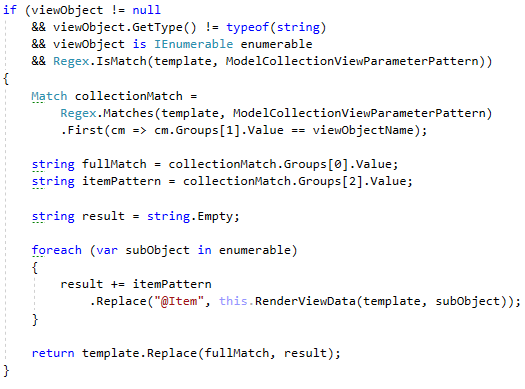
Here’s an example of a **collection placeholder** and its result. Say we have a collection of names: [Pesho, Gosho, Tosho]. Our HTML would look like this:



And the **result** of the **rendering** would be:



Not that complex, right? And it’s very dynamic and flexible. Now let’s see how that functionality is implemented.



As you can see first we extract all **Matches** that **match** the **Regex pattern** we **implemented** in the **constants**, to **capture** all **Collection placeholders** in the **template**. Then we **extract** that which matches our viewObjectName. Our **Regex pattern** is configured with **groups** to extract the collectionName, and the itemPatternFormat.

Then we just create a string (yes, yes, it can be optimized with a StringBuilder), we **iterate** **over** the **collection**, and we replace in the itemPattern, the @Item placeholder with the rendered state of the **current element** of the **collection**. This is a direct recursion, but why so? Well, you’ll understand soon enough.

Then we replace the fullMatch in the template. In other words, that **collection placeholder**, that we are currently working with. And then we **return** the replaced template.

This is the **collection rendering mechanism**, now let’s see how a **non-primitive object** is **rendered**.

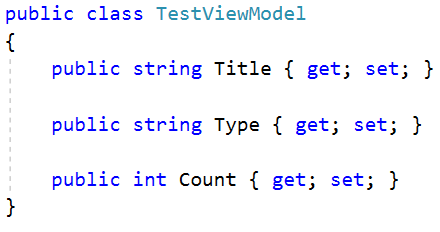
##### Object Rendering Mechanism

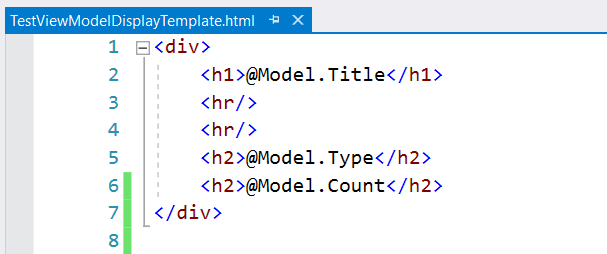
The second **if statement** checks if the viewObject is a **non-primitive object**. If it is, then it must have a **display template**, or an **HTML document** which specifies how that **object** should be **rendered**.

That **display template** must be **located** in {ViewsFolder}\Shared\DisplayTemplates.

That **display template** must be **named** {ObjectName}DisplayTemplate.html.

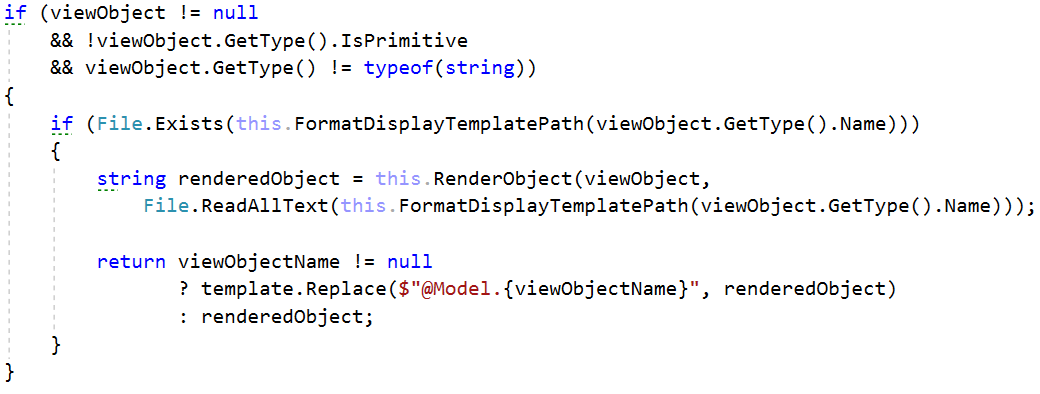
**Example**:





If the **required circumstances** are **not satisfied**, there should be **no** **return statement**, and the **method** should continue to the **default return statement**.

Now let’s see how that functionality is implemented:



By using the **helper methods** from the beginning, we are **formulating** the **full Display Template Path**, and we are **checking** if it exists. If it doesn’t the behavior is straight, but if it does exist, we **render** the **Object**, using the RenderObject() method (which we will see later) and the **display template**.

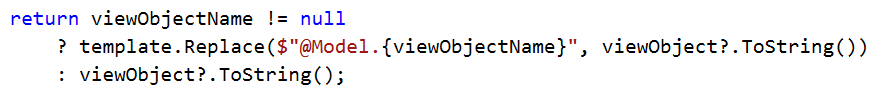
Finally, we check if the viewObjectName is **NULL**:

* If it is, we **return** the **object’s string representation**.
* If it isn’t we **replace** a **placeholder** in the **given** template.

This is all there is to the **Object Rendering Mechanism**. Now let’s see the **default rendering mechanism**.

##### Primitive (Default) Rendering Mechanism

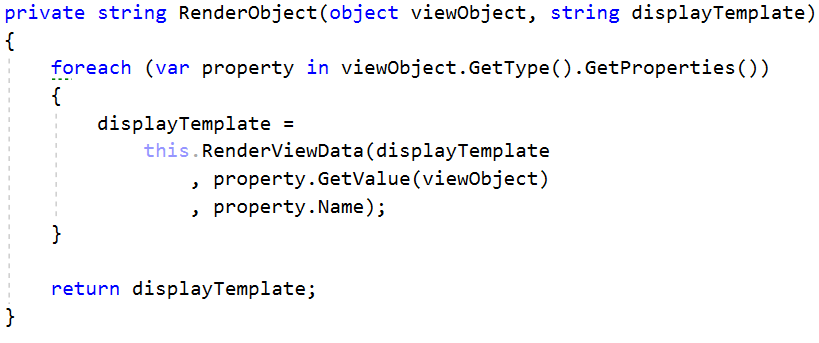
This mechanism is used for primitive data types, and is the default rendering mechanism, it just replaces the placeholder in the template with the **given object’s** **string representation** or **returns** the **string representation** if the given viewObjectName is **NULL**.



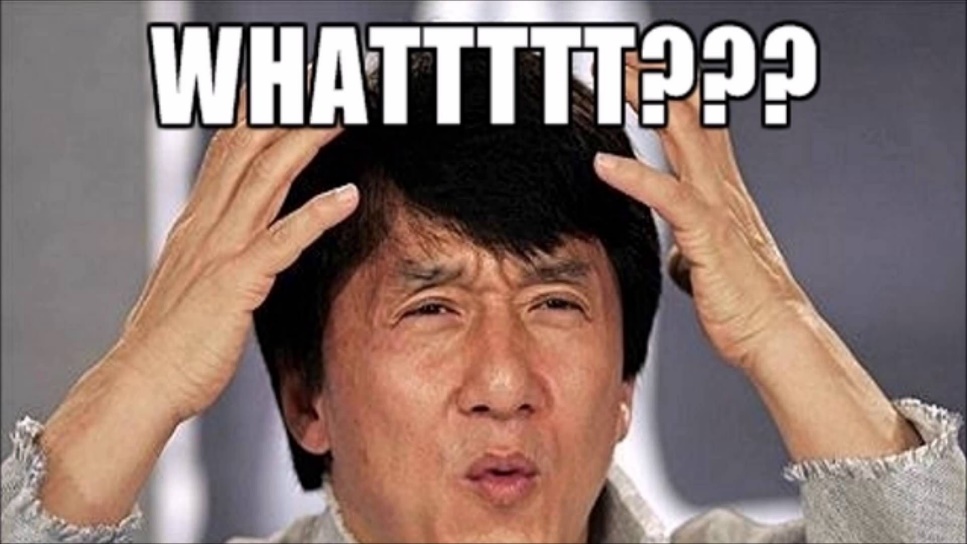
And with that we finalize the **recursive method**, time to see the RenderObject() method, and get an idea of what this mechanism is about.

#### RenderObject() method

This method is used to **render non-primitive objects**. It iterates over the **object’s** **properties** and **renders** them over the given **display template**.



But wait, this is an **indirect recursion**?!? And we are passing the **display template** as the **placeholding template** to it? Whaaat?



Well, imagine the **current case**. You have a ViewModel, which has **properties**, one of which is a **Collection**, of ViewModels, which have **properties**, one of which is another ViewModel.



You will have to go **really deep** and have **different templates**, for the **inner** ViewModels, that’s why you just create an **inner recursion** which **works** with **another template** as the **placeholder template**. The **initial recursion** initiates with the **View**, but the **deeper** the **recursion goes**, the **deeper** the **templates** go.

At one point everything will **return** to the **initial View** and will become one **big formatted template** which will be returned to the **user**.

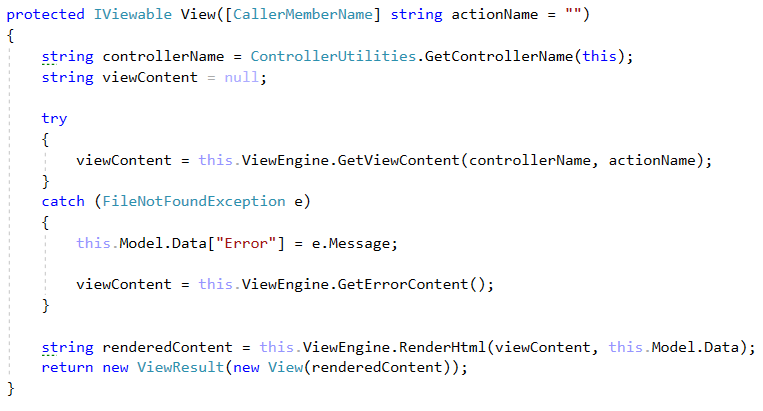
This is how our Tazer View Engine works. Now let’s go include it in the **main flow**, and have you play with it.

### Controller class

Modify the Controller class by adding the following property to it:



Then modify the View() method to look like this:



This way the View (or Error, in case something breaks) will be rendered and returned to the **User** – well-formatted. Easy-peasy. Piece of cake. Everything should now work, exactly as we want it.

Go test it out. Implement some ViewModels, some DisplayTemplates, play with the ViewEngine. Get familiar with it and its nature.