# 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.

# Data Binding & Data Validation

## HTML View Engine

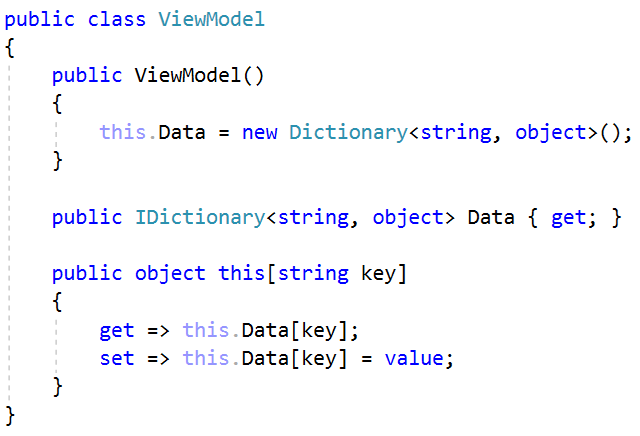
The old view engine (**View class**) was cool. We actually managed to **read** and **return HTML pages**. Which is **NOT very impressive**. What will be very impressive though, is if we introduce **templating** to our **View Engine**, so that we could **render data** in the **HTML**.

Well, let’s begin.

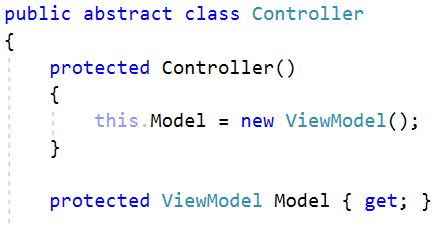
### ViewModel

Create a folder called Models in the Framework project. Create a class, called ViewModel. This class will be used by our base Controller to hold our **View Data**, which we will then **render** on the HTML pages, while reading them.

The ViewModel class holds a Dictionary, which holds **object data**, and accesses it by **string keys**. The object value will be “ToString()-ed”, in order to be **rendered** to the **HTML**. After all, we are working solely with **text**.



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



### View

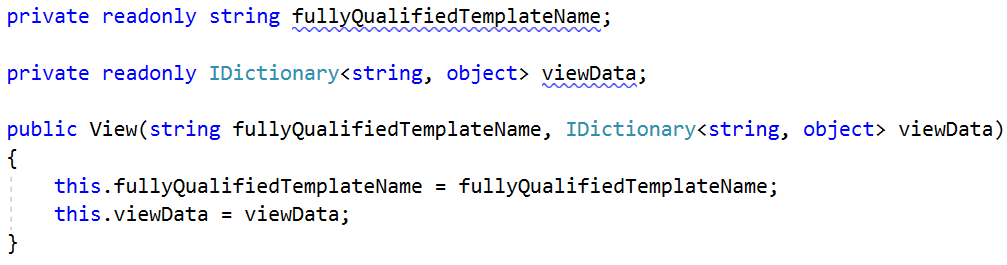
The time has come for some modifications on the View class. Now, our templating logic won’t be very cool, but it’ll do for now, we will eventually optimize it and enhance it even more. For now, it would work with {{{parameter}}} placeholder in the **HTML**.

**Example**:

1. <h1>Hi, my name is {{{name}}}</h1>
2. name = “Slim Shady”
3. <h1>Hi, my name is Slim Shady</h1>

#### Initialization

To implement the View functionality, we need to have some **view data** in the View class. Rewrite the constructor like this:

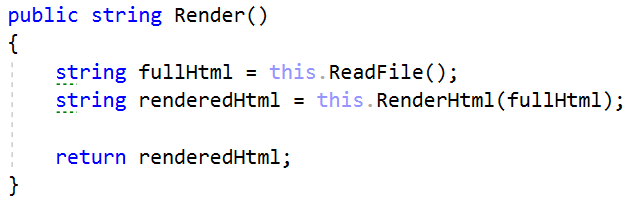


#### Functionality

But what would we do with that data? Well, here comes the real templating and data rendering.

##### Render() method

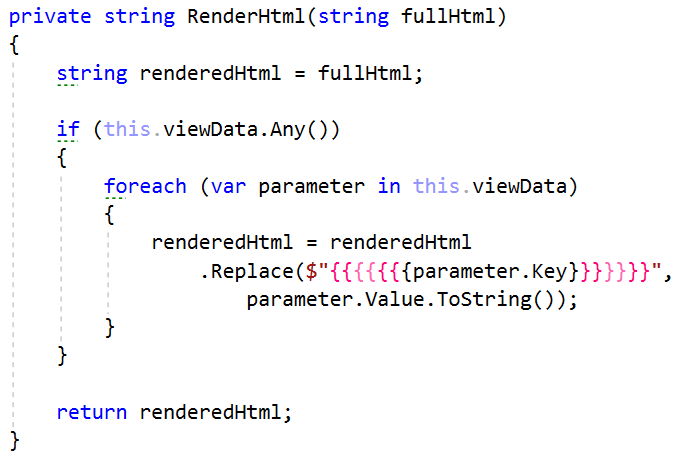
Rewrite the Render() method like this:



As you can see we use **another method**, which we still don’t have, but that is not a big problem, as we will get to that right away.

##### RenderHtml() method

Create a method called RenderHtml(). It should look like this:



It might not be the best template engine ever, but it will greatly enhance our work with Views. Now let’s get further into the Framework’s Extending.

## Data Binding

The second main change we will do is in the ControllerRouter class. This class needs to be extended quite a lot.

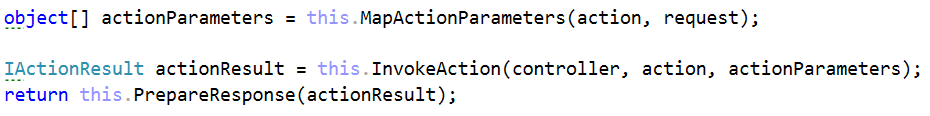
A normal **MVC Framework**, would be able to provide its consumers a user-friendly way of handling requests, wouldn’t it? Like, for example, it should support **data binding** and **data** **validation**. This is an important concept, it will help us handle the **POST Requests** more comfortably.

**Data Binding** is the process of **extracting data** from the HTTP Request’s FormData or QueryData and passing it to the corresponding Controller’s Action as a BindingModel (some **custom object**) or **primitive variable**.

The process itself is not that hard to implement so let’s do it.

### Handle() method

Modify the last part of the Handle() method, where you prepare the Response. It should look like this:



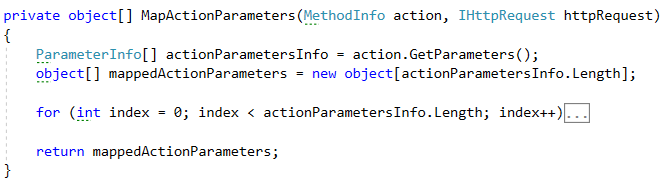
Wow, wow, wow, wait, wait, wait… There’s a lot of **new** methods and **changed** old methods here. Rest assured, we will implement them all. 😉

### MapActionParameters() method

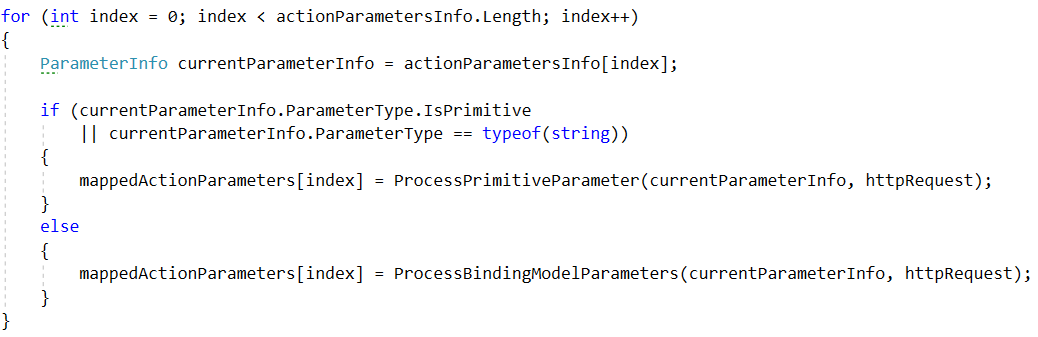
The MapActionParameters() method will extract the Controller’s Action’s **method parameters**, and will **check** if the Request’s Data holds anything that could be **mapped** to them. The steps to do that are:

1. **Extract** the Action’s Parameters.
2. **Initialize** an object[] array with the same **length** as the Action’s Parameters.
3. **Iterate** over the Action’s Parameters, checking each **parameter**’s **type**
4. If it’s a **primitive** or **string**, **check** for its **value** in the Request’s Data, and **simply** **map it**.
5. If it’s a **complex type**, then it’s a Binding Model.
6. **Instantiate** an **object** of the Parameter’s Type.
7. **Iterate** over its **properties**, **checking** for **each property**, if there is a **value** in the Request’s Data.

Essentially, the algorithm, stated above, should look like this:

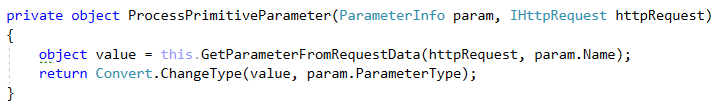


And the iteration over the actionParameters, should be like this:



Eeeh, more methods to implement. Well, high-quality code requires code element **segregation**. You will just have to deal with that. One great man once said, “Divide and Conquer”.

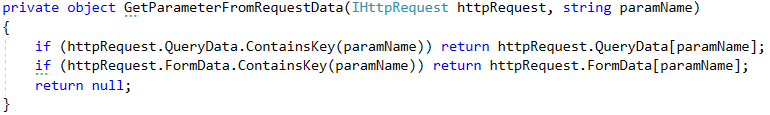
### ProcessPrimitiveParameter() method



The **value** of the **parameter** will be **extracted** using a **helpful method** – GetParameterFromRequestData(). Then it will be **converted** to the **appropriate typ**e, and **returned** as a **result**.

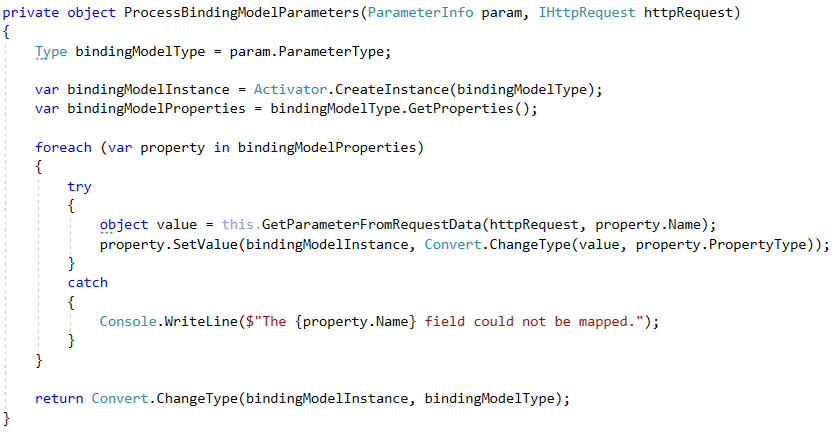
### GetParameterFromRequestdata() method

This simple method, just **checks** the Request’s Data for a **value**, and **returns** it if there is such. NULL is returned otherwise…



### ProcessBindingModelParameter() method

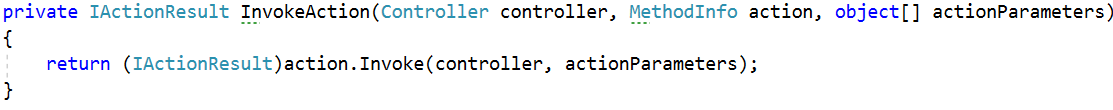
The functionality here is a little **more complex**, but it is essentially the same.



And with this we are ready with the Data Binding mechanism. Now all that is left is to mix it in the main functionality. The next method we need to implement is InvokeAction().

### InvokeAction() method

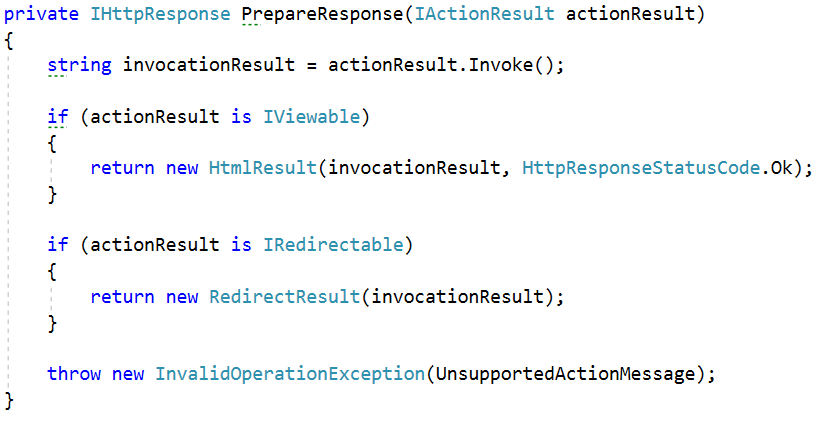
This method contains the **invocation** of the Controller’s Action. It is used solely for that purpose.



The ActionResult is returned and then passed to the PrepareResponse() method.

### PrepareResponse() method

This method is **changed** **quite a lot** since the last time you’ve seen it. It is also **quite** **refactored** now that **most of its** **functionality** is **extracted**. Its current functionality includes only extracting the **text data** from the ActionResult and dispatching it as an HttpResponse of an appropriate type:



And with this final step, we are **ready**! The simple **Data Binding mechanism** has been successfully implemented.

## Data Validation

But how do we validate a BindingModel? 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!

We will create several helpful **validation attributes**, which would be completely **extensible**, if we need more. Just like we did with the **Http Method Attributes**.

Then we will write a **mechanism** for **Data Validation**, which we will use, while **binding** the **data**. Our data binding won’t be barbaric though. We won’t do something like – One incorrect property = Brutal Application Failure = Complete Error Explosion. Naa, that would be overkill. We will just have something like a state – **Valid** / **Invalid**.

### Property Attributes

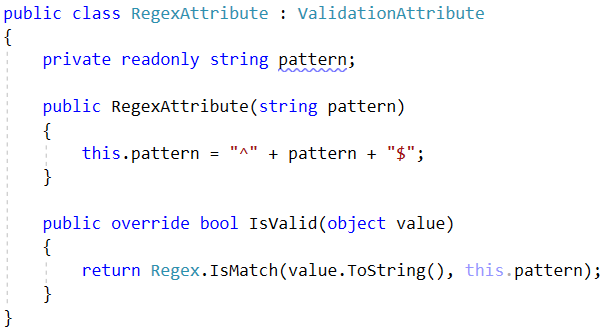
Go to the Attributes folder of the Framework project, and create a subfolder, named Property.

Now let’s create the Property Validation Attributes.

#### RegexAttribute

Create a class in the same folder, named RegexAttribute.   
It should inherit from the ValidationAttribute class.

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.

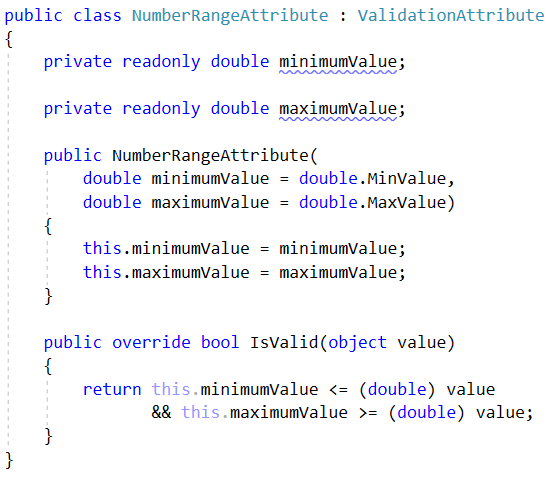


#### NumberRangeAttribute

Create another class, named NumberRangeAttribute.   
It should inherit from the ValidationAttribute class.

It should be instantiated with a double minValue and a double maxValue.

The **validation method** should **check** if the **given value** is between the minValue and maxValue.



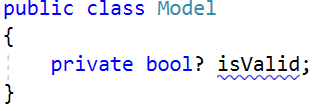
These attributes will be later used in our BindingModels as **property attributes**. Now let us write the **Data Validation mechanism**.

### Validation Functionality

One thing needs to be clear, this is only for BindingModels. You are only validating binding models. That being said, let’s begin.

#### Model

Create a class, named Model, in the Models namespace of the Framework project. It should look like this:



A **nullable boolean**…



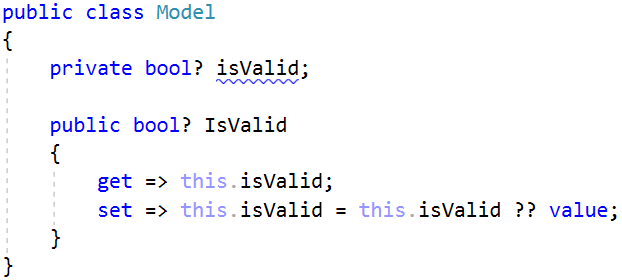
Technically, it is not practically very logically usable. But oh well, you’ll see the magic behind it soon enough.

#### Controller

Update the Controller class, adding a Model property, called ModelState, to it, in the following way:



But how are we going to update the ModelState. That’s where the nullable Boolean magic comes!   
Modify the Model class like this:

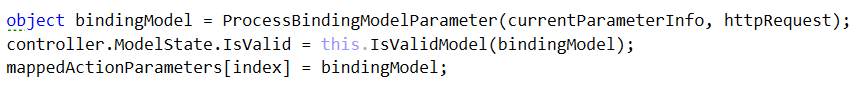


So, if the ModelState has already been set, it **won’t be modified** anymore.

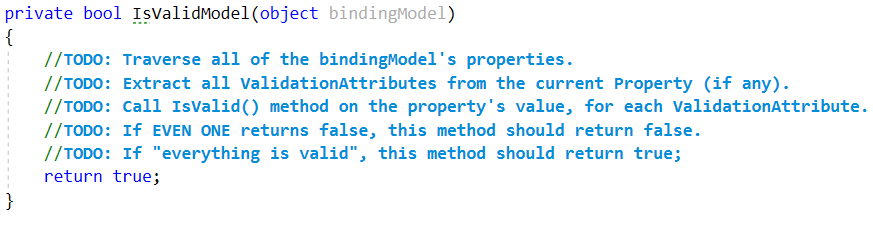
#### ControllerRouter

The last class we need to modify is the ControllerRouter. But that is where the most functionality comes.   
First, we will need to include the **Model Validation** logic to our main **Data Binding mechanism**.

Modify the MapActionParameters() method, where we set the bindingModel parameter, like this:



A **New Method**!!! **Alriiiight**! We love implementing new methods.   
Implement a **Boolean** method, named IsValidModel(). It should look like this… Nope, not going to give it to you! Instead, you’ll receive the algorithm. Implement it yourself, it is quite fun to do it.



And with this we are ready with our **Data Validation Mechanism**. On to the next part.

## 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. Currently our Server does that for us, but that is not what it should do.

Well, we must provide our Server with another Request Handler – The ResourceRouter.

Create a class, named ResourceRouter in the Routers folder in the Framework. It should implement the IHttpHandler interface.

Extract the **Resource functionality** from the ConnectionHandler to this **class**. That functionality is fully working.

Configure the Server to use this class for **Resource Handling**. That should be fairly easy, you’ve already done it once with the ControllerRouter. Just replace those **if statements** in the HandleRequest() method to use the ResourceRouter.

# Inversion of Control

A normal Web Framework can support at least a Dependency Injection mechanism, which eases its consumers’ development. Our framework will also support this type of functionality.

## Dependency Container

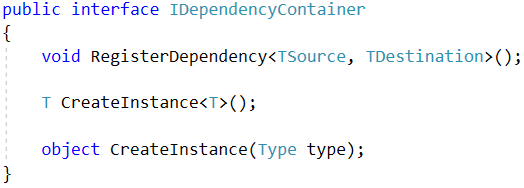
The first thing you will need to implement is the DependencyContainer, the main component of the Dependency Injection mechanism.

### Services

Create a namespace, called Services, in the SIS.Framework project. We will use this namespace for all **Framework Service functionality-oriented components**.

### IDependencyContainer

Create an interface, called IDependencyContainer, in the Services namespace. It should look like this:



Now, this interface describes a quite-**generic** behaviour. The **genericity** in this functionality will be quite useful later when we create the implementation.

#### RegisterDependency

The RegisterDependency() method, adds a **dependency** to some sort of containment. It works purely with Types (Type class). When adding a dependency you add a:

* Dependency Origin (TSource), or what will be **included** as a **parameter** in a **specific class**’s **constructor**.
* Dependency Destination (TDestination), or what will be **passed** to the **specific class**’s **constructor** as an **objects**.

#### CreateInstance()

The **first overload** of the CreateInstance() method is intended to initialize an object of a particular type. It does this, by calling the **second overload** of the method.

#### CreateInstance(Type type)

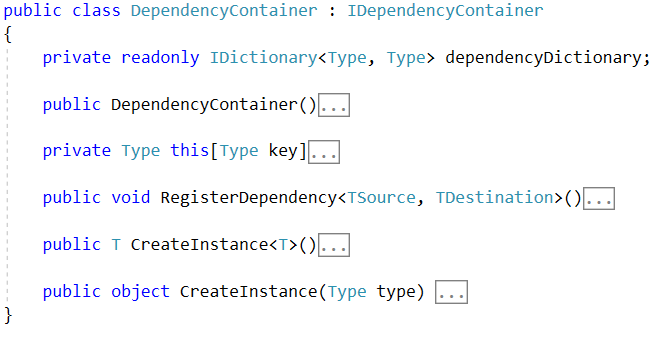
The **second overload** of the CreateInstance() method:

* Checks if there is a Destination Type of the given Type.
  + If there is, it is **extracted**, and an **object** will be **instantiated** from it.
  + If there is **no**, then the **given** Type becomes the Destination Type (the Type from which an object will be instantiated).
* **Extracts** the **constructor** of the Destination Type, and its parameters.
* For each, of its parameters, calls the CreateInstance() method again. This is a **recursive algorithm** which traverses the **dependencies** to the **deepest**, most **primitive dependency**, which **does not require** any **sub-dependencies**. This is the **main DI algorithm**.
* **Instantiates** an **object** with the **instantiated parameters** and **returns** it.

Now let’s see how those functionalities will be implemented in an actual class.

### DependencyContainer

Create a class, called DependencyContainer, in the Services namespace. It should implement the IDependencyContainer interface, and its inner implementation should look like this:



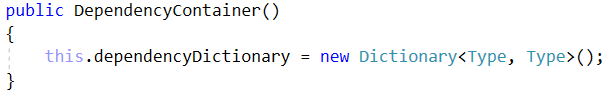
Aside from the actual methods, defined by the **interface**, we have an **overriden** [] **operator**, and a **private** **dictionary**.

* The **dictionary** is our dependency containment, that is easily deducted.
* The [] operator is only a **simplifier**, which is made to **escape** from the **exception**, while trying to access a **non-existent key** in the **dictionary**.

Now let’s see each of the methods and how it implements the **behaviour** which was **described** in the **interface section**.

#### Constructor

The constructor of the class should look like this:



It only instantiates the **Dependency Containment collection**.

#### [] Operator Override

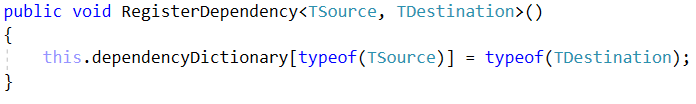
The [] **operator override** should look like this:



As you can see it returns null, if there is no such key in the dictionary. As it was stated earlier, just a simplifier.

#### RegisterDependency

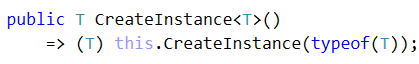
The RegisterDependency() method should look like this:



It **extracts** the Types of the **given** Origin and Destination and **adds** them as a key and value.

#### CreateInstance (first overload)

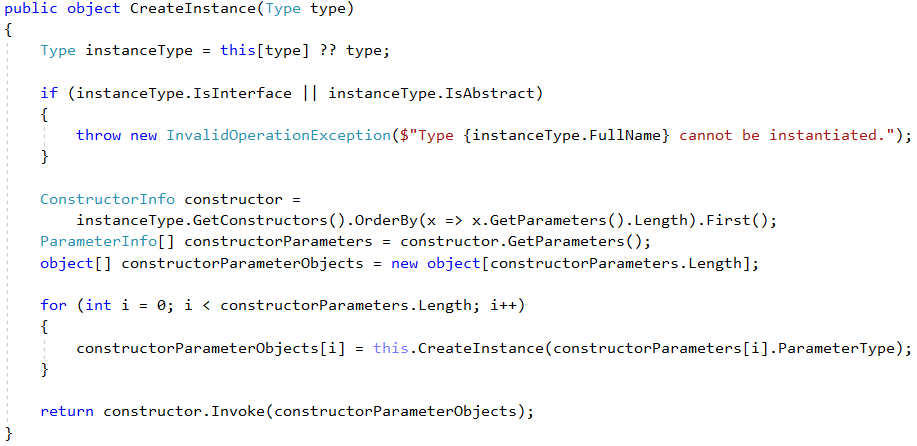
The first overload of the CreateInstance() method should look like this:



As you can see it just **calls** the **second overload** of the method and **casts** the **result** to the **given generic type**. By calling the **second overload**, the **requested object**’s **dependencies** will also be **instantiated** and **passed** as **parameters**.

#### CreateInstance (second overload)

The second overload of the CreateInstance() method contains the main functionality. It should look like this:



This method **perfectly implements** the described functionality in the **Interface section**. And with this we are ready with our **Simple Dependency Container**. This will greatly optimize our work in the **Application Development**.

## Application IoC

Try to apply the Dependency Container into the Application. For example:

* Implement some services into the application (a Service layer).
* In the ControllerRouter, where an **object** of the Controller is created, use the DependencyContainer, to **create** an **instance**.
* This will go through the **registered dependencies** (in our case – Services) and will **instantiate** them, and then **pass** them to the **s** of the Controller.
* This process will allow you to **dynamically instantiate** Controllers, without **passing** **specific parameters** to their **constructors**. Of course, if the **requested parameters** are **registered** as **dependencies**.