# 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.

# 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

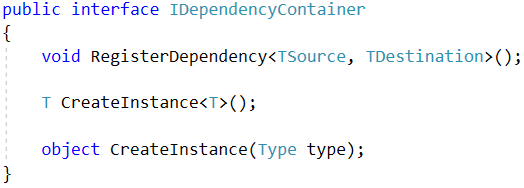
The first thing you will need to implement is the DependencyContainer, the main component of the Dependency Injection mechanism.

### 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)

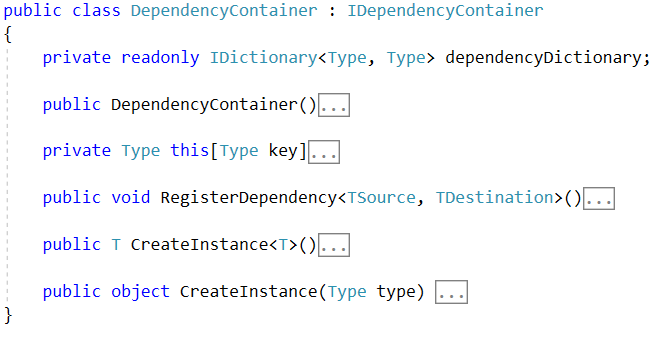
The **second overload** of the CreateInstance() method:

* 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:



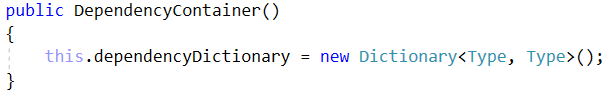
Aside from the actual methods, defined by the **interface**, we have an **overriden** [] **operator**, and a **private** **dictionary**.

* 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

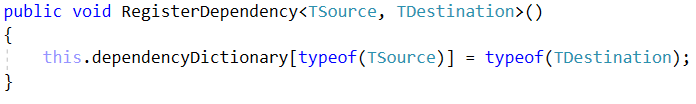
The [] **operator override** should look like this:



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

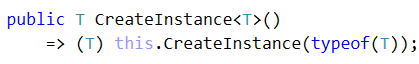
The RegisterDependency() method should look like this:



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

#### CreateInstance (first overload)

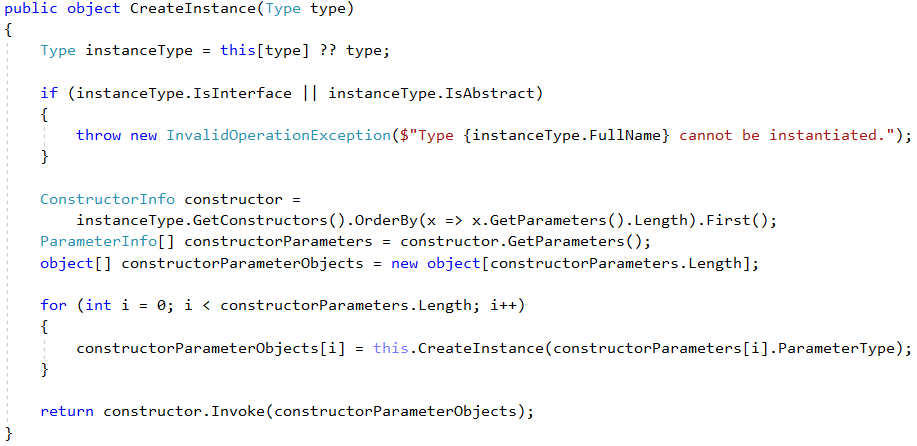
The first overload of the CreateInstance() method should look like this:



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**.