# Exercises: MVC Introduction

Problems for exercises and homework for the "Web Dev Basics" course from the official "Applied Programmer" curriculum. In these exercises we will create a very **simple HTTP Server**. We will extend it every time and design it to mimic Microsoft’s IIS.

## Introduce Controllers

Our first task in this exercise is to **create a controller functionality** in our server, which should **route controller** **action methods to responses**. Then, we will create **controller classes** with **actions**:

Graphical user interface, text, application, chat or text message

Description automatically generated

### Step 1: Modify Routing Table

If you remember, until now we used **pre-action methods** in the Startup **class** to **modify a response** before it is returned to the browser. We did this, as we needed the **HTTP request** and **response** but now our **controller methods** will obtain the **request** and **directly return a response**.

For this reason, we should **modify our mapping methods** to use the **request data** from the browser and **return a response** through a **function**.

To do this, **modify the methods** in the IRoutingTable **class** like this:

Graphical user interface, text, application

Description automatically generated

Note that the three methods now **accept a parameter of type** Request and **return a** Response. This is possible because of the Func<T, T> **delegate**. In this way, the **request** may be **sent to the response function** as a **lambda** **expression**: request => response.

Now **implement the methods** in the RoutingTable **class**. First, our **routes collection** should now change its **type** to **contain a response function** like this:

Text

Description automatically generated with medium confidence

As we used the new **keyword** in the **constructor**, it should **stay the same**:

Text

Description automatically generated

Then, make the Map(…) **method** of the class **reusable** by the MapGet(…) and MapPost(…) **methods**, which we will implement next. This method should **accept a method** and a **path** and **map them to a response**. Finally, it should **return** the **current** IRoutingTable **instance**. Write the method like this:

Graphical user interface, text, application

Description automatically generated

Write the methods for **mapping** "GET" and "POST" **requests**. They should accept a **path** and a **delegate function** and use the Map(…) **method** from above with **different request methods**:

A screenshot of a computer

Description automatically generated with medium confidence

At the end, **modify** the MatchRequest(Request request) **method** from before. The difference is that it should now **return not only a response**, but the **whole response function** with the **request**. Do it like this:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now we have a **routing table**, which allows us to use the **request** **data** when forming a response.

### Step 2: Create Controller Class

To create a **controller functionality**, start by **creating** the Controller **class**, which should be in a **folder** named "Controllers" in the "BasicWebServer.Server" **project**. **Define the class** as abstract, as it won't be instantiated directly:

Graphical user interface, text, application, chat or text message

Description automatically generated 

A picture containing logo

Description automatically generated

Create the **constructor** of the class and **a property** for the **HTTP request**, as our controller actions will use it:

Text

Description automatically generated

Then, create a **method**, which **returns responses** with **different response types**, which we have already created:

A picture containing graphical user interface

Description automatically generated

Now we have the **base class for all controllers**, which can return **different responses**. We will see how to use it later.

### Step 3: Create Routing Extensions

Before we create controller classes, we should think about how **URL paths** in our app will be **mapped** to the **controller methods**. We want to do the **mapping** in the Startup() **class** of the "BasicWebServer.Demo" **project** like this:

Text

Description automatically generated

To do this, we should **add mapping methods** to the **routing table** we have, which should **accept the controller**, a **path** and a **response function**.

Start with creating the RoutingTableExtensions **class**, placed in the "Controllers" **folder**:



Then, implement the MapGet<TController>(…) **method**, which should **extend** the MapGet(…) **method** from the RoutingTable **class**. We often create **extension methods**, as we want to achieve better **separation of concerns** and **single responsibility**.

The MapGet<TController>(…) **method** should accept a **generic parameter of type** Controller and return IRoutingTable:



Also, it should **accept the current routing table instance** to be **extended**, a **path** and a **function**, which **accepts a** **controller** and **returns a response**. Do it like this:

Text

Description automatically generated

On the next line, it is important to **add a generic type constraint** for TController to be of **type** Controller:



Then, write the **method implementation**. The method will use the MapGet(…) **method** of the RoutingTable **class**. It should **create a controller instance** of a **given controller type** and **pass it to the method**. For the creation of the controller, we will use a **separate method** – CreateController<TController>(Request request), which we will implement later.

For now, the MapGet<TController>(…) **method body** is the following:



The MapPost<TController>(…) **method** of the RoutingTableExtensions **class** is the **same** as the one for the "GET" **request**. The difference is that it uses the MapPost(…) **method** of the RoutingTable **class**. Write the method like this:

Text

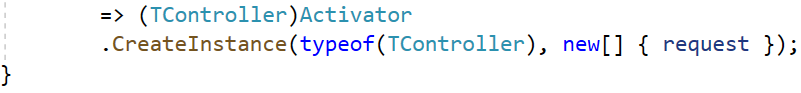
Description automatically generated

Now implement the method for **creating a controller of a given type**, which **accepts a request**:



To **create a controller**, we will use the Activator **C# class** with its CreateInstance() **method**. This method **creates an** **instance of the specified type** using the constructor that best matches the specified parameters. It accepts a **controller type** and a params **object**, which we will use to add the **request to our constructor**. When the controller class is **instantiated**, **cast** the result to TController to **set the type** of the variable.

Write the CreateController<TController>(Request request) **method** like this:



This method will use the **constructor** from the Controller **class**. It will also **set the request**, which will be used later:

A picture containing graphical user interface

Description automatically generated

Now we have the **full controller functionality** implemented. We should just **add the controller classes**.

### Step 4: Create Controllers with Actions

Now we will create our first controller– the HomeController, which will use **methods** to return **responses**.

With the **new routing**, we should **map**, for example, "/" to the Index() **method** of the HomeController **class** in the Main() **method** like this:

Graphical user interface, text

Description automatically generated

**Create** the HomeController **class** and we will see later how to **map paths** to its **actions**.

#### HomeController Class

Go to the "BasicWebServer.Demo" **project** and **create a folder** named "Controllers". In the folder, create a **new class** HomeController, which should **inherit the base** Controller **class** and **use its constructor**:

Graphical user interface, text

Description automatically generated

##### Index() Method

Then, **create** an Index() **method** – it should **return** a TextResponse by using the Text(string text) **method** of the Controller **class** like this:



Try the **controller method** in the **browser**. Don't forget to **map the path to the controller action**, as shown above, and include the "BasicWebServer.Server.Controllers" **namespace** in the Startup **class** to use the routing extensions.

The TextResponse should be **returned to the browser**:

Graphical user interface, text, application, chat or text message

Description automatically generated

Go to the HomeController **class** and **implement the missing methods**.

##### Redirect() Method

First, let's create the **mapping** for a "GET" **request** to "/Redirect" in the Startup **class**. It should **invoke** the Redirect() **method** of the HomeController **class** like this:

Text

Description automatically generated

**Implement** the Redirect() **method in the** HomeController – it should **redirect** to the <https://softuni.org/> **site**:

A picture containing text

Description automatically generated 

**Run the app** in the browser and go to "/Redirect" – you should be **redirected** to the **SoftUni's site**:

Graphical user interface, application, Teams

Description automatically generated  Graphical user interface, text, application

Description automatically generated

##### Html() Method

Create the **mapping** for a "GET" **request** to "/HTML" in the Startup **class** – it should **invoke** the Html() **method** of the HomeController **class**:

Text

Description automatically generated

Write the Html() **method**, which should only **display the form** with the **name** and **age** to be filled in. Get the **form constant field** from the Startup **class** and **place it** **in this class**. Then, **use it in the method** like this:

Graphical user interface, text, email

Description automatically generatedGraphical user interface, text

Description automatically generated

Visit "/HTML" in the browser – the **HTML form** with **name** and **age** should be displayed:

Graphical user interface, text, application

Description automatically generated

##### HtmlFormPost() Method

The next **method** we should create is HtmlFormPost(), which should **get the name and age** from the **request** **form** and **return** a TextResponse.

Clear the Startup **class** from the AddFormDataAction(…) **method** and make a "POST" request to "/HTML" to invoke the HtmlFormPost() **method**:

Text

Description automatically generated

Implement the HomeController action as shown below. In the HtmlFormPost() **method** we will **get the form data directly** from the **request**, as we have access to it.

A picture containing text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

**Test the method** in the browser. Go to "/HTML", **fill in data** in the **form** and **submit it**. The result should be the following:

Graphical user interface, text, application

Description automatically generated  Graphical user interface, text, application, chat or text message

Description automatically generated

##### Content() Method

Go on with the Content() **method**, which should only **return the form** with the [Download] **button**.

The **mapping** in the Startup **class** is the following:

Text

Description automatically generated

Get the **form field** from the Startup **class** and **use** it:

Text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Run the app** and go to "/Content" – you should see the [Download Sites Content] **button**:

Graphical user interface, text, application, chat or text message

Description automatically generated

##### DownloadContent() Method

Next, we should **map** the "POST" **request** of "/Content" to the DownloadContent() **method**:

Text

Description automatically generated

The DownloadContent() should get the **HTML content** from **sites** and **add it to a text file,** which will be **downloaded**. Until now, this was done by the DownloadSitesAsTextFile(string fileName, string[] urls) and DownloadWebSiteContent(string url) methods in the Startup **class**. **Move** them to the HomeController **class** together with the FileName **property**:

Graphical user interface, text

Description automatically generated with medium confidence Graphical user interface, text, application

Description automatically generated

Use the **methods** and the **property** in the **controller action** like this:

A picture containing text

Description automatically generatedGraphical user interface

Description automatically generated

As you see, the DownloadContent(…) method **invokes other methods** to **download the sites HTML content**. Now it should **return** a FileResponse with the **file name**:



**Try out the file downloading** in the browser. Go to "/Content", **click on the button** and the "content.txt" **file** should be **downloaded**:

Graphical user interface, text, application, chat or text message

Description automatically generated  Graphical user interface, text, application, chat or text message

Description automatically generated

##### Cookies() Method

Next, implement the Cookies() **method**, which should **set the cookies** when "/Cookies" is accessed. Add the **mapping** in the Startup **class** like this:

Text

Description automatically generated

We have already implemented the **cookie functionality** in the Startup **class** and you know that we needed to **add the cookies to the HTML response** before it is returned to the browser.

To do this, we will **modify** the Html(string html) **method** of the Controller **class** to accept a CookieCollection and **add each cookie to the response**. Then, an **HTML response** should be returned like this:

A picture containing logo

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Then, let's go back to the HomeController **class** and **implement** the Cookie() **method**. **Copy the cookie functionality** **logic** from the Startup **class** and **modify it**. The method should look like this:

A picture containing text

Description automatically generatedText

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidenceText

Description automatically generated with medium confidence

Note that we invoke the Html(…) **method** with **cookies** when we want them to be **added to the response** (when they are set for the first time or they are missing). Otherwise, we just send a **pure HTML response content**.

**Examine the cookies** in the browser by **accessing** "/Cookies" at least twice:

Graphical user interface, text, application

Description automatically generated  Graphical user interface, text, application

Description automatically generated

Note that the **session cookie is created** with other cookies because of the AddSession() **method** in the HttpServer **class**.

##### Session() Method

**Map** the "/Session" to the Session() **method** like this:

Text

Description automatically generated

Now you should **implement** the Session() **method**, which **adds a session cookie to the response** (if the session doesn't exist) and **returns a text**. Get the **session functionality** from the Startup **class** and **remove it from there**. The Session() **method** should be the following:

A picture containing text

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Look at the session data by accessing "/Session":

Graphical user interface, text, application, chat or text message

Description automatically generated Graphical user interface, text, application, chat or text message

Description automatically generated

Now we have implemented all the **actions** from the HomeController **class**, **cleaned up** the Startup **class** and **mapped paths to the controller methods**. **Run** the app, **navigate to different pages** and make sure that all functionalities of the app are **working correctly**.

#### UsersController Class

In the previous exercise we implemented the **login**, **user profile** and **logout functionalities**. Now we will create a separate **controller** for them, called UsersController, as they are connected to the user. The controller will have **actions** for the above functionalities.

Now create the UsersController **class** in the "Controllers" **folder**. It should **inherit the base** Controller **class**:

Graphical user interface, text

Description automatically generated

##### Login() Method

The Login() **method** should return the **login** **form** when a "GET" **request** is send to "/Login". Create the mapping in the Startup like this:

Text

Description automatically generated

Copy the **login form** from the Startup **class** and **write the method** to return it like this:

Text

Description automatically generated

Text

Description automatically generated

Go to "/Login" in the browser and you should see the **login form**:

Graphical user interface, text, application

Description automatically generated

##### LogInUser() Method

The LogInUser() method is invoked on a "POST" request to "/Login" and implements the login functionality:

Text

Description automatically generated

Copy the logic for the LogInUser() **method** from the LoginAction(…) method in the Startup class. You need to change it a little bit – now we won't return the login form when login is unsuccessful but will directly return a RedirectResponse to "/Login".

Do it as shown below. Don't forget to get the Username and Password **constant** **fields** from the Startup **class**, as well:

Graphical user interface, text

Description automatically generatedText

Description automatically generated with medium confidence

Note that we invoke the Html(…) **method** with a CookieCollection when we want to **add cookies to the** **response**.

Try the "**login**" **functionality**. Go to "/Login", **submit the form** with **valid** **and invalid data**. Make sure the functionality has the **correct behavior**:

Graphical user interface, text, application

Description automatically generated  Graphical user interface, text, application

Description automatically generated  Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application, chat or text message

Description automatically generated

##### Logout() Method

A "GET" request to "/Logout" should **invoke** the Logout() **method**:

Text

Description automatically generated

Text

Description automatically generated

The Logout() **method** **logic** is the same from before. Get it from the LogoutAction(…) **method** in the Startup **class** and **write the controller action** like this:

A picture containing text

Description automatically generated Text

Description automatically generated

Test the logout functionality:

Graphical user interface, text, application, chat or text message

Description automatically generated

##### GetUserData() Method

The last method of the UsersController **class** is invoked when the user accesses "/UserProfile".

The GetUserData() method is shown below. If the user is **not logged-in**, they should be **redirected** to "/Login". Otherwise, they should **see the user data**, e.g. username. Do it like this:

A picture containing text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Go to "/UserProfile" and **try out** the page with a **logged-in user**:

Graphical user interface, text, application, chat or text message

Description automatically generated

When the user is **not logged-in**, they should be **redirected** to the "Login" **page**.

Note that the access to the "User Profile" **page** depends only on the **session** – if it is **deleted**, the user may stay **logged-in the app**, but they **won't see the user data** until they log in again:

Graphical user interface, application

Description automatically generated 

Graphical user interface, application

Description automatically generated

You can **fix this by yourself**. Remember that the aim of this is to show the **user info always** when the user is **logged** in the app.

Now we have implemented the UsersController with its **actions for login**, **logout** and **user profile**. **Run** the app, **navigate to different pages** and make sure that all functionalities of the controller are **working correctly**.

### Step 5: Clear Solution

It is a good idea to **delete the pre-render action functionality** from our **server** as we don't use it anymore. To do this, first go to the HttpServer **class** and **remove the following lines** from the Start() **method**:

A picture containing text

Description automatically generatedText

Description automatically generated with low confidenceGraphical user interface, text, application

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidence

Then, **clean** the Response class:

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with medium confidence

Clean the ContentResponse **class**, too:

Graphical user interface, text, application

Description automatically generated

Now clean the HtmlResponse and TextResponse **classes**, as well:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

We also did a lot of **cleaning** of the Startup **class**. However, make sure that it looks as shown below and **doesn't have** **useless properties or methods**:

Text

Description automatically generated

Now you can go on with adding the **views functionality** to the server.

## Introduce Views

As you know, for the "/HTML" and "/Content" **paths** we returned **HTML forms**, which were just strings in our HomeController **class**. In the UsersController, we have the same case with the **login form** on "/Login". In this task, we will **move the HTML forms** to .cshtml files (views), which will be found and used by the **controller actions**.

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

### Step 1: Create View Response

To **return a view**, we will need to **create a new response class**. Create the ViewResponse **class**, which should **inherit the** ContentResponse **class**:



Add a **constant field** for the "/" **path separator**, as we are going to need it, and **create a constructor**, which should **accept** a **view name** and a **controller name** and **use the base constructor**:

Text

Description automatically generated

As you can see, we **pass an empty string** to the **base response class** – we do it, as we **do not have the content** of the **body** yet. We should **read the view file** and get it first.

To get the **HTML** from our **views**, we will need the **full path to them**. By **convention**, each **view** has the .cshtml **file** **extension** and is **accessed** on "…/Views/{controllerName}/{viewName}.cshtml".

That's why we will check whether the **view name** **contains the path separator** and, if not, **construct the name to be in format** "{controllerName}/{viewName}":

Text

Description automatically generated with low confidence

On the next step, we should **get the full path** to the **view** by **adding the path** to the "Views" **folder before** and the .cshtml **extension after** the **view name**:

Text

Description automatically generated with low confidence

Finally, we should **read the view file content** as a **text** and **add it to the response body**:

Text

Description automatically generated with low confidence

### Step 2: Modify Controller to Return View

Go to the Controller **class** of the **server project** and **add a method** to return a ViewResponse. We will use **reflection** to do this in the best way.

The View() **method** should **accept a view name**, which should be taken with the [CallerMemberName] **attribute**, which allows you to obtain the method or property name of the **caller** to the method. Then, the method should **return** a ViewResponse with the **view name** and the **controller name**:



The GetControllerName() **method** **gets the** **controller name**, without the "**Controller**" part (for example "**HomeController**"  "**Home**"):

Text

Description automatically generated with medium confidence

Now the **controllers** can **return a view**. Let's see how to use them.

### Step 3: Create Views and Modify Controller Actions

First, go to the HomeController **class** and look at the methods, which **return HTML content** – Html() and Content(). **Modify** them to **return** a ViewResponse like this:

A picture containing text

Description automatically generated Graphical user interface, text, application

Description automatically generated

Now go to the UsersController and make the Login() **method** return a **view** in the same way:

A picture containing text

Description automatically generated

Then, let's create the **views** they will use.

Start with creating a "Views" **folder** in the "BasicWebServer.Demo" **project**, where we will **store our views**. In this folder, create a **new one** with the **name of the controller** and **add** **two** .cshtml **files**, named after the **actions**, which use them.

Graphical user interface, text, application

Description automatically generated  Graphical user interface, application

Description automatically generated

The **file** **structure** should be the following for now:

Graphical user interface, text, application, chat or text message

Description automatically generated

Get the **HTML form** from the HtmlForm **field** in the HomeController **class** and **add it** to the Html.cshtml **file**. The **view** should look like this:

Graphical user interface, text

Description automatically generated

Do the same with the DownloadForm **field** and the Content.cshtml **view** **file**:

Graphical user interface, application

Description automatically generated

Do the same with the **view** for the Login() **method** of the UsersController. The **file structure** and the **view** should look as shown below. Don't forget to **remove** the LoginForm **property** from the UsersController **class**:

Graphical user interface, text, application, chat or text message

Description automatically generated Text

Description automatically generated

Now you have **all the views** you need in the right folders.

### Step 4: Modify the .csproj File

Before we try the **view functionality**, we need to **modify** the .csproj **file** of the "BasicWebServer.Demo" **project**. To **access** that file, **double-click** on the **project name** in the Solution Explorer:



To use the **views**, we need them to be **copied** to the "net5.0" **folder** of **our project**, where the .dll **files** are. To do this, **add** the following lines:

Graphical user interface, text, application

Description automatically generated

When you **run the app**, you will see that the "Views" **folder** **with** **its files** is copied to the "net5.0" **folder**, where the app searches for the views:

Graphical user interface, application

Description automatically generated Graphical user interface, application

Description automatically generated Graphical user interface, application

Description automatically generated with medium confidence

### Step 5: Try Controllers with Views

**Run the demo app** and **navigate** to "/HTML", "/Content" and "/Login" – the **HTML forms should appear**:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generatedGraphical user interface, application

Description automatically generated

Note that if you have any problems, you should check first if you have a **mistake in naming** – it happens often.

## Introduce Models

We have already implemented **controllers** and **views** in our server. To create a full **MVC** **structure** we should **implement** **models** and **use them in views and controllers**.

Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application

Description automatically generated

### Step 1: Modify ViewResponse Class

In order to **use models in the views**, we should **modify** the ViewResponse **class**. First, it should **accept a model object** in the **constructor** with a **default value** NULL and later check if we have a **model coming from the controller**:

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application

Description automatically generated

With the PopulateModel(string viewContent, object model) we will **modify the content** from the **view**. Get the **model properties** with their **name** and **values** like this:

Graphical user interface, text, application

Description automatically generated

Each **model property** in our view will be **surrounded** by **two opening curly brackets** and **two closing curly brackets** like this: "{{Name}}". That's why we need to **get each property value** and **put it on its place** in the **view content**:

Text

Description automatically generated

At the end, **return the modified view content** to the method:

A picture containing logo

Description automatically generated

### Step 2: Create Model and View

As our **views can now use model data**, we will **create a model with properties** for **name** and **age**. Create a "Models" **folder** in the "BasicWebServer.Demo" **project** and **create the** FormViewModel **model class** in it:

Graphical user interface, text

Description automatically generated with medium confidence

Next, **create a view**, which uses these **properties** to **display a message with the model data**. Name the view "HtmlFormPost.cshtml" and place it in the "/Models/Home" **folder**, as we will **invoke** it from the HtmlFormPost() **method** in the HomeController **class**:

Graphical user interface, text, application, chat or text message

Description automatically generated

The **view** should have the **model properties** **surrounded** by **curly brackets** and should look like this:



Use the **view** and the **model** in a **controller action**.

### Step 3: Modify Controller Method to Return View with Model

First, go to the Controller **class** and create a View(…) **method**, which **accepts a model** and a **view name** and returns a ViewResponse:

A picture containing logo

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Use the above method in the HtmlFormPost() **method** of the HomeController **class**. Until now, we returned a TextResponse with the **name** and **age values** from the **request**. Now we will create a FormViewModel with this **data** and **pass it to the view**:

A picture containing text

Description automatically generated Graphical user interface, text, application

Description automatically generated

### Step 4: Try Controller with View and Model

**Run** the demo app and **navigate** to "/HTML". Enter some **valid data** in the **form fields**:

Graphical user interface, text, application

Description automatically generated

**Click** on the [Save] **button** to **submit the form**. Now you should see the **returned view** with the **form data** you submitted:

Graphical user interface, text, application

Description automatically generated

Now you have a **working MVC architecture** in your server.