**.NET Core 3.1 Web API & Entity Framework Jumpstart - Part 1**

his tutorial series is now also available as an online video course. You can [watch the first hour on YouTube](https://youtu.be/H4qg9HJX_SE) or [get the complete course on Udemy](https://www.udemy.com/course/net-core-31-web-api-entity-framework-core-jumpstart/?referralCode=CA390CA392FF8B003518). Or you just keep on reading. Enjoy! :)

# Introduction

The .NET Core framework is getting better and better and more important in the web development world nowadays.

Almost every request I get for new web development projects is **asking for knowledge in .NET Core**, including Web API and Entity Framework Core.

So knowing the fundamentals of back end web development with .NET Core can be highly beneficial to your career. And that’s where this tutorial series comes in.

In a short period, you will **learn how to set up a Web API**, make calls to this Web API and also **save data persistently with Entity Framework Core** and the help of **Code First Migration**.

We will get right to the point, you will see every single step of writing the necessary code and by the end of this tutorial series, you will have what it takes to say ‘yes’ to all the .NET Core project requests from any recruiter.

The only tools you need in the beginning are **Visual Studio Code** and **Postman** - both are available for free.

We will use Visual Studio Code for our implementations and Postman to make calls to the Web API. Both tools are available for Windows, Mac OS, and Linux. And since .NET Core is cross-platform, you can follow this tutorial series on any of these operating systems. (I know, Microsoft and cross-platform, it still surprises me, too.)

The back end application we’re going to build is a small **text-based role-playing game** where different users can register (we’re going to use **JSON web tokens for authentication**) and create their own characters like a mage or a knight, update attributes of these characters, set the skills and also let the characters fight against each other to see who’s better.

So, I hope you’re ready for your new skills and your new projects. Let's start!

## **Tools**

The only tools we need for now are Visual Studio Code and Postman. Additionally to that, you have to download and install the .NET Core 3.1 SDK.

VS Code can be found on <https://code.visualstudio.com/>.  
Postman is available on <https://www.getpostman.com/>.  
And the SDK can be downloaded on <https://dotnet.microsoft.com/download>. Make sure to download .NET Core **3.1** for your operating system.

So please download and install everything and then continue with the next chapter.

## **Create a new Web API**

As soon as the .NET Core SDK, Visual Studio Code and Postman is installed, we can already create our first .NET Core application which will be a Web API right away.

To start, I created a new folder called "dotnet-rpg" - for "dotnet role-playing game". Open the folder in VS Code.

I assume you’re already a bit familiar with Visual Studio Code, if not, feel free to have a look around.

While you're doing that, it might also be a good idea to install certain extensions.

First "[C# for Visual Studio Code](https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp)" by Microsoft itself. This extension will also be suggested by VS Code as soon as you create your first C# application. It includes editing support, syntax highlighting, IntelliSense, Go to Definition, Find all references, just have a look, pretty useful stuff.

Next is "[C# Extensions](https://marketplace.visualstudio.com/items?itemName=jchannon.csharpextensions)" by jchannon. As the description says, it might speed up the development workflow by adding some entries to the context menu, like Adding a new C# class or interface.

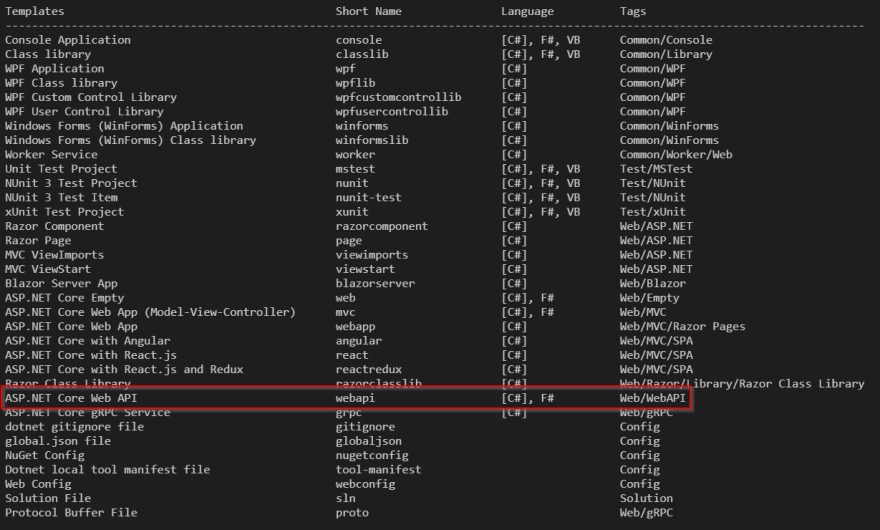
And the last one already is one of my personal favorites, the "[Material Icon Theme](https://marketplace.visualstudio.com/items?itemName=PKief.material-icon-theme)". This one simply provides lots and lots of cute icons.

Alright, but now let’s create our Web API! We open a new terminal window and then let's have a look at what the dotnet command provides.

With adding a -h you see all the available commands. The one that’s interesting for us right now is the new command, which creates a new .NET project.

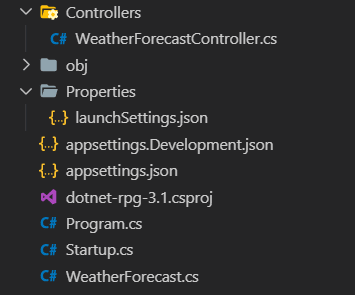
But we also got the run command to, well, run our application and also the watch command which can be used together with run to restart the application as soon as we make changes to any file. Quite useful, if you don’t want to stop and start the project by yourself every single time you make any changes.

With dotnet new -h we see all the available templates. There are a lot. For instance the plain old console application, and further down we finally got the Web API. So let’s use it!

[](https://res.cloudinary.com/practicaldev/image/fetch/s--soB58li7--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/gg7rwvn0iwyhrud7hubx.png)

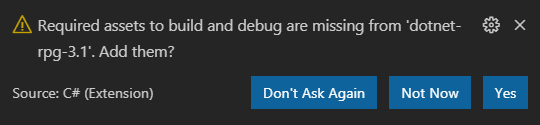
We type dotnet new webapi and hit return.

Now we see some files that have been generated for us in the explorer. Let’s go through them real quick.

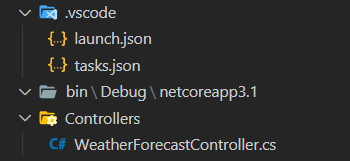
[](https://res.cloudinary.com/practicaldev/image/fetch/s--1HcfNOzo--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/obdyv6fnisxu94iaaclo.png)

At the bottom, we see the WeatherForecast class. This is just part of the default Web API project. We don’t really need it, but let’s use this example in a minute.

In the meantime, we get a little popup telling us that we should add some files. Of course, we want to add them.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--lrQTnSnW--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/1us41luvlie12c9fjzkb.png)

You should see now, that we got the .vscode folder with the launch.json and the tasks.json. Both are configuration files used for debugging, source code formatters, bundlers, and so on, but not very interesting for us at this moment.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--eO1b-QSN--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/vlv38val4k3n9snwn3m1.png)

So let’s have a look at the Startup class.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

Here we find the ConfigureServices and the Configure method. The ConfigureServices configures the app’s services, so a reusable component that provides app functionality. We will register services in the future in this method, so they can be consumed in our web service via dependency injection for instance.

Please don’t mind all these buzzwords right now...

The Configure method creates the app’s request processing pipeline, meaning the method is used to specify how the app responds to HTTP requests. As you can see we’re using HttpRedirection, Routing, and so on. With all these Use... extension methods, we’re adding middleware components to the request pipeline. For instance UseHttpRedirection adds middleware for redirecting HTTP requests to HTTPS.

To make things a bit easier for us in the beginning, let's remove the UseHttpRedirection line or at least comment it out.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

//app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

The Startup class is specified when the app’s host is built. You see that in the Program class in the CreateHostBuilder() method. Here the Startup class is specified by calling the UseStartup() method.

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

In the .csproject file we see the SDK, the target framework, in our case .NET Core 3.1 and the root namespace. Later on, we will find additional packages like Entity Framework Core in this file.

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp3.1</TargetFramework>

<RootNamespace>dotnet\_rpg</RootNamespace>

</PropertyGroup>

</Project>

Regarding the appsettings.json files we only need to know that we can add and modify some configurations here.

More interesting right now is the launchSettings.json file where the current environment is configured and also the application URL. With this URL, we will find our running web service.

"dotnet\_rpg\_3.\_1": {

"commandName": "Project",

"launchBrowser": true,

"launchUrl": "weatherforecast",

"applicationUrl": "https://localhost:5001;http://localhost:5000",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

The obj and bin folders can be ignored for now. We find temporary object- and final binary files here.

Very interesting and often used throughout this tutorial series is the Controllers folder. The first controller you see here is the generated WeatherForecast demo controller. We’ll get to the details of controllers later. For now, it’s only important to know, that we can already call the Get() method here.

[HttpGet]

public IEnumerable<WeatherForecast> Get()

{

var rng = new Random();

return Enumerable.Range(1, 5).Select(index => new WeatherForecast

{

Date = DateTime.Now.AddDays(index),

TemperatureC = rng.Next(-20, 55),

Summary = Summaries[rng.Next(Summaries.Length)]

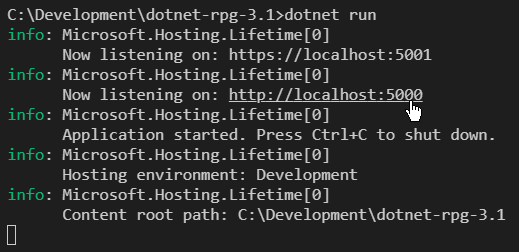
})

.ToArray();

}

## **First API Call**

In the terminal, we enter dotnet run. You see, here’s already the URL we’ve seen in the launchSettings.json. So let’s open Chrome and go to http://localhost:5000.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--eJ9UBUk6--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/2cz1vmhzllepv2fqmdan.png)

Well, the result of this URL doesn’t look very nice. That's because we have to access the WeatherForecast controller. So when we go back to VS Code, we see the name of the controller (WeatherForecast - without Controller). We also see the routing attribute ([Route"[controller]"]) to define how to access this controller - we’ll discuss how routes work in a future chapter.

[ApiController]

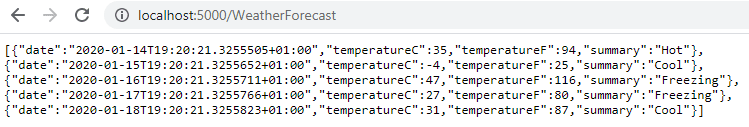
[Route("[controller]")]

public class WeatherForecastController : ControllerBase

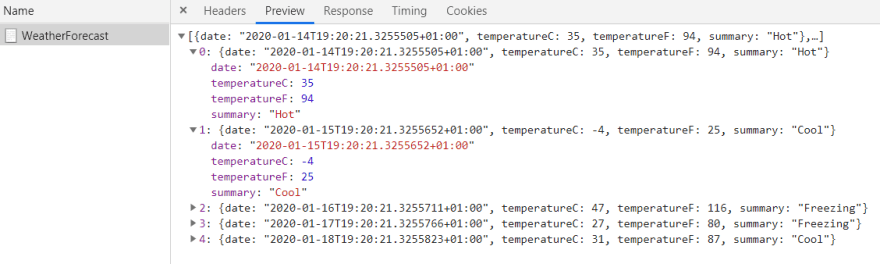
{

// ...

So we just copy the name - WeatherForecast - go back to Chrome, enter the correct route and finally, we get the results.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--4kPsIiGG--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/fcvjveohacco849m13u4.png)

We can also see the results in the console.

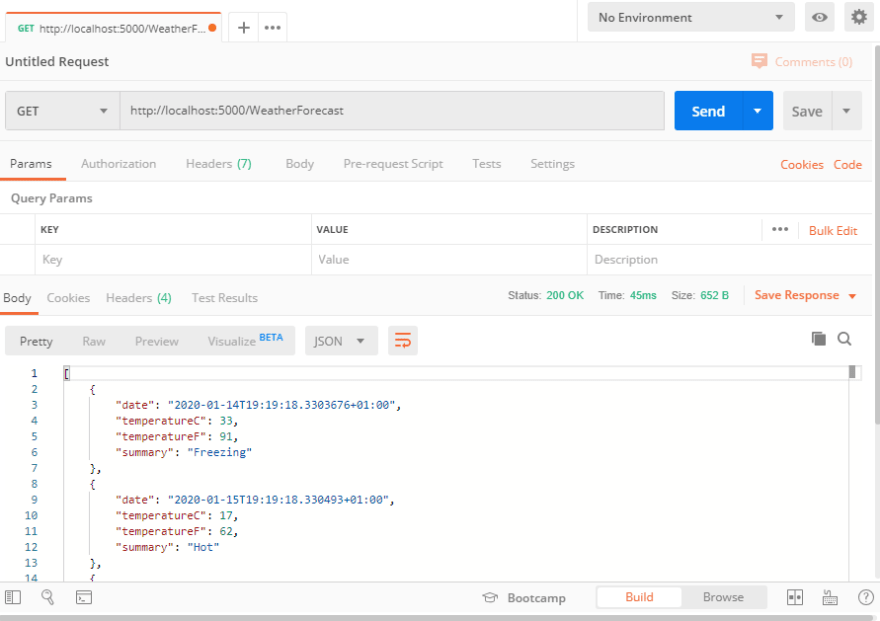
[](https://res.cloudinary.com/practicaldev/image/fetch/s--QRuCCYA1--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/ueb5oeteqw0dwk1mclln.png)

Now let's do this with Postman because Postman will be the tool we will use to test our REST calls of the Web API.

If you haven’t already, make yourself a bit familiar with Postman.

The essential part is in the middle. We can choose the HTTP Request Method - in this particular case, it is GET - then enter the URL and hit "Send".

The styling of the result should look similar to the console.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--rfptrStX--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/5mhtcvt5d8a7yc4f26y7.png)

Great! So this works. Now let’s move on and build our own web service.

# Web API Core

So far you learned how to create a Web API project in .NET Core from scratch and how to make your first API call with Postman.

In the upcoming chapters, we will create a new controller and models for our RPG (role-playing game) characters.

Additionally, we will turn our synchronous calls into asynchronous calls, make use of Data-Transfer-Objects (DTOs) and change the structure of our Web API so that it meets best practices.

But first, let’s have a look at the Model-View-Controller (MVC) pattern, which is the foundation of all this.

## **The Model-View-Controller (MVC) Pattern**

Model-View-Controller or short MVC is a software design pattern that divides the related program logic into three interconnected elements. Let me explain, what every single one of these three elements stands for and how they collaborate.

We start with the model. You could also say, the data. A character in our role-playing game is a model, for instance. It can have an Id, a name, hitpoints, attributes, skills and so on.

public class Character

{

public int Id { get; set; } = 0;

public string Name { get; set; } = "Frodo";

public int HitPoints { get; set; } = 100;

public int Strength { get; set; } = 10;

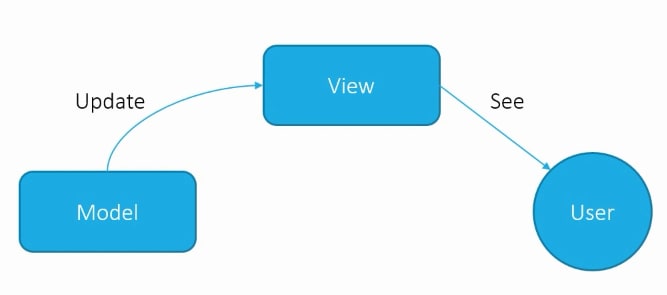
public int Defense { get; set; } = 10;

public int Intelligence { get; set; } = 10;

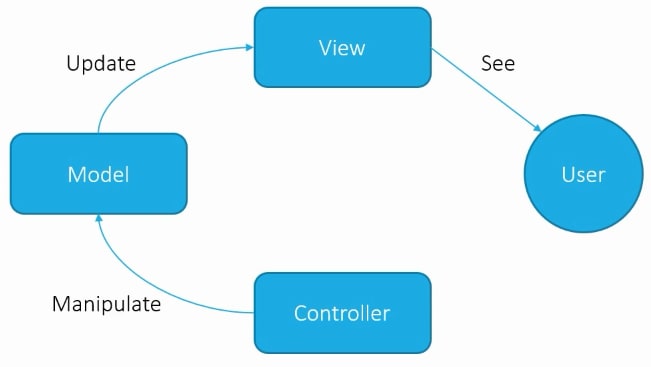
}

You as the developer know the code of your model. But the user won’t see your code. That’s where the view comes in. The user probably wants to see a representation of the character in HTML, plain text or amazing 3D graphics - depending on your game. In other words, the view is the (graphical) user interface or (G)UI.

To sum these two up, the model updates the view and the user sees the view. If the model changes, let’s say our character gets another skill or its hitpoints decreased, the view will change, too. That’s why the model always updates the view.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--EmymXTca--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/4ulcvtym5l1g16jaammh.png)

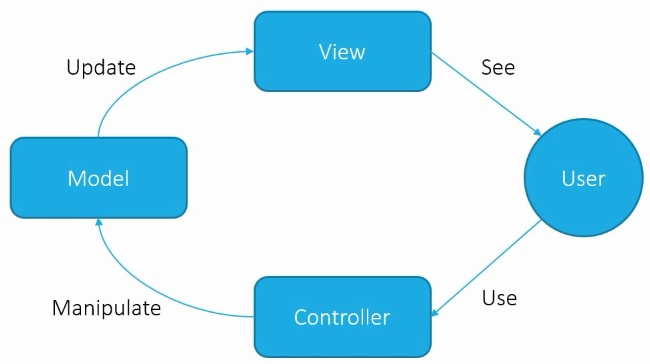
Now, what’s up with the controller? The controller does the actual work. There you will find most of your code because it manipulates your data or model. In our case, it’s the Web API that will create, update and delete your data.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--vKAY_8K2--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/00yc3pfq5ljk4uckhxvi.png)

Since we won’t have a view except the results of our calls in Postman, we’re going to build our application in the following order: First the model and then the controller. And we will always jump back and forth between those two.

With the help of the view though, the user can manipulate the data, hence properties of the RPG character with buttons, text fields and so on. In a browser game that might be JavaScript code in essence - maybe with the help of frameworks like [Angular](https://www.udemy.com/course/angular-real-world-webapp-development-w-nodejs-mariadb/?referralCode=36F1E760976712BD7B5B), React or VueJS.

This JavaScript code, in turn, uses the controller to do the manipulation and save these changes persistently in the database. The manipulated model will update the view, which is then again seen by the user and the circle starts all over again.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--1jMSJvid--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/4wh53q5iff665lc5g2zt.png)

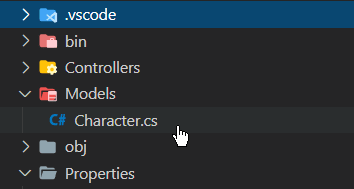
Well, that sums up the MVC pattern. Now we’re going to build our first model.

## **New Models**

The first things we need are new models. We need a model for the RPG character itself and also a model for the type of RPG character, i.e. a character class like Barbarian, Monk, Necromancer and so on.

First, we create a "Models" folder. For the character model, we will create a new class in this Models folder. If you have the “C# Extensions” installed, you can add a new C# class with a right-click, otherwise, you just create a new file.

So right-click the Models folder, then click “New C# Class” and call this class Character.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--gmhodJUt--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/8f8fpiln9e3pg6phrjzj.png)

Now let’s add some properties.

public class Character

{

public int Id { get; set; }

public string Name { get; set; } = "Frodo";

public int HitPoints { get; set; } = 100;

public int Strength { get; set; } = 10;

public int Defense { get; set; } = 10;

public int Intelligence { get; set; } = 10;

}

We will also add an RpgClass property, i.e. the type of the character. But first, we have to create a new enum for that.

So let’s add a new C# class called RpgClass and then replace class with enum.

Feel free to add any kind of role-playing class you want to add here. In this example, I use Knight, Mage, and Cleric. The most basic characters you would need I guess. Some melee action, some magic and of course never forget the healer.

public enum RpgClass

{

Knight = 1,

Mage = 2,

Cleric = 3

}

Now when we have the RpgClass enum ready, we can finally add it to the Character model.

public class Character

{

public int Id { get; set; }

public string Name { get; set; } = "Frodo";

public int HitPoints { get; set; } = 100;

public int Strength { get; set; } = 10;

public int Defense { get; set; } = 10;

public int Intelligence { get; set; } = 10;

public RpgClass Class { get; set; } = RpgClass.Knight;

}

I set the default to the Knight, but again, that’s totally up to you.

Alright, the first models are ready. Let’s add a new controller now and make a GET call to receive our first role-playing game character.

## **New Controller & GET a New Character**

To add a new controller, we create a new C# class in the Controllers folder. Let’s call this class CharacterController.

Before we can start implementing any logic, we have to make this thing a proper controller. To do that, we first derive from ControllerBase. This is a base class for an MVC controller without view support. Since we’re building an API here, we don’t need view support. If, however, we would want to add support for views, we could derive from Controller. But in our case, just make sure to add ControllerBase.

public class CharacterController : ControllerBase

After that, we have to add some attributes. The first one is the ApiController attribute. This attribute indicates that a type (and also all derived types) is used to serve HTTP API responses. Additionally, when we add this attribute to the controller, it enables several API-specific features like attribute routing and automatic HTTP 400 responses if something is wrong with the model. We’ll get to the details when we make use of these features.

[ApiController]

public class CharacterController : ControllerBase

Regarding attribute routing, that’s already the next thing we have to add. Below the ApiController attribute, we add the Route attribute. That’s how we’re able to find this specific controller when we want to make a web service call. The string we add to the Route attribute is [controller]. This means that this controller can be accessed by its name, in our case Character - so that part of the name of the C# class that comes before Controller.

[ApiController]

[Route("[controller]")]

public class CharacterController : ControllerBase

Don't forget to also add the reference Microsoft.AspNetCore.Mvc on top of the file.

using Microsoft.AspNetCore.Mvc;

Alright, let’s get into the body of our C# class. The first thing I’d like to add is a static mock character that we can return to the client. For that, you also have to add the dotnet\_rpg.Models reference.

using Microsoft.AspNetCore.Mvc;

using dotnet\_rpg.Models;

namespace dotnet\_rpg.Controllers

{

[ApiController]

[Route("[controller]")]

public class CharacterController : ControllerBase

{

private static Character knight = new Character();

}

}

Next, we finally implement the Get() method to receive our game character.

[ApiController]

[Route("[controller]")]

public class CharacterController : ControllerBase

{

private static Character knight = new Character();

public IActionResult Get()

{

return Ok(knight);

}

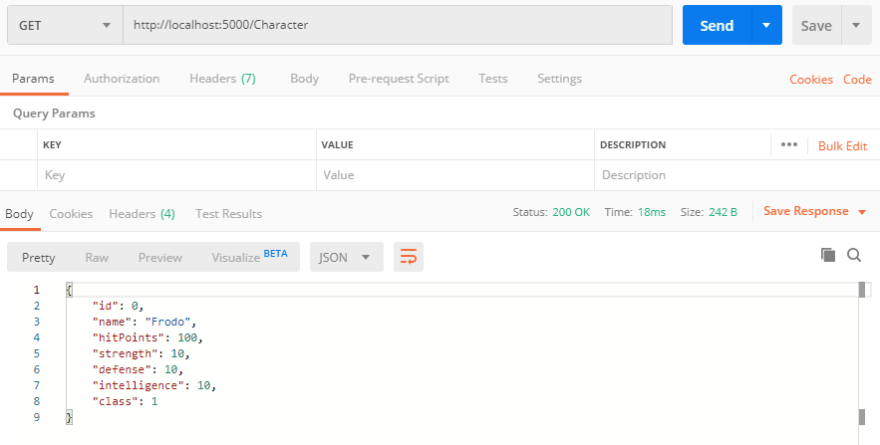
}

We return an IActionResult because this enables us to send specific HTTP status codes back to the client together with the actual data that was requested. In this method, with Ok(knight) we send the status code 200 OK and our mock character back. Other options would be a BadRequest 400 status code or a 404 NotFound if a requested character was not found.

Alright, the code is implemented. Let’s test this now with Postman.

It’s pretty straight forward now. The HTTP method is GET again, the URL is http://localhost:5000/character and nothing else has to be configured.

Hit "Send", and there is our knight!

[](https://res.cloudinary.com/practicaldev/image/fetch/s--j5Sgleye--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/xrz0ampbsyipx6jr8yvs.png)

Now pay attention to the attribute that was added to our Get() method. Exactly, there is none. When we compare this to the WeatherForecastController, we could have added an [HttpGet] attribute. But it’s not necessary for the CharacterController because the Web API supports naming conventions and if the name of the method starts with Get...(), the API assumes that the used HTTP method is also GET. Apart from that we only have one Get() method in our controller so far, so the web service knows exactly what method is requested.

However, in the next chapter, we’ll have a deeper look at these attributes.

That's it for the first part of this tutorial series. I hope it already was useful to you. To get notified for the next part, simply follow me here on [dev.to](https://dev.to/_patrickgod) or [subscribe to my newsletter](https://mailchi.mp/96620175570f/dotnetcore). You'll be the first to know.

See you next time!

Take care.

Next up: Attribute routing, HTTP methods, add a new character with POST, asynchronous calls, and more!

*Image created by cornecoba on*[*freepik.com*](https://www.freepik.com/free-vector/flying-rocket-background_1130797.htm)*.*