**Attribute Routing, HTTP Request Methods & Best Practices in .NET Core Web API**

# Web API Core (continued)

## **First Steps with Attribute Routing**

Before we start with a new method, let’s add another mock character to the controller by replacing the single character with a list of characters, because we will implement an additional method to return that list.

Let's call the second character “Sam”.

private static List<Character> characters = new List<Character> {

new Character(),

new Character { Name = "Sam"}

};

Don't forget to also add the using directive to System.Collections.Generic for the List<T> type.

using System.Collections.Generic;

using System.Linq;

using dotnet\_rpg.Models;

After that we have to make a slight modification to our existing Get() method. We now return the complete list of characters.

public IActionResult Get()

{

return Ok(characters);

}

Let’s test this modification in Postman. Same HTTP method, same URL, hit “Send”, and there are our mighty knights. Beautiful.

[

{

"id": 0,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

},

{

"id": 0,

"name": "Sam",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

}

]

Now let’s add a method to return a single character. And let’s just return the first character of the list by default.

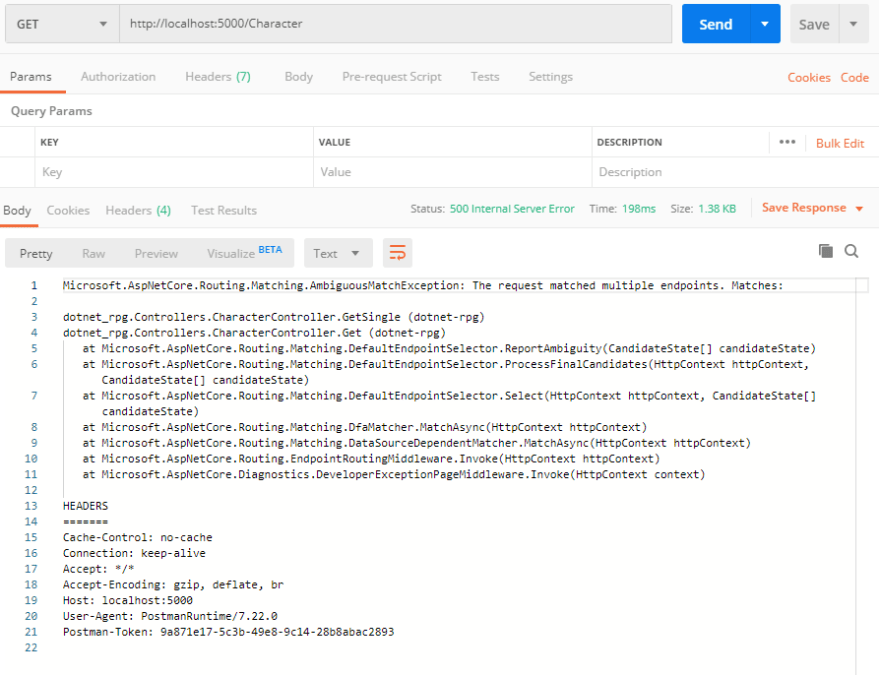
public IActionResult GetSingle()

{

return Ok(characters[0]);

}

When we try another request in Postman now, we’re getting an error!

[](https://res.cloudinary.com/practicaldev/image/fetch/s--BojsQiAH--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/axty8damt74dpp1c0g2t.png)

To be more exact, we've got an AmbiguousMatchException. This means, the Web API doesn’t know which method to use, because we have two Get() methods with no parameters. That’s where we have to add routing attributes.

So let’s add one to the first method and call the route GetAll.

[Route("GetAll")]

public IActionResult Get()

{

return Ok(characters);

}

You see, with Route and then a string we can decide how to get to a particular method. It's time to try this out in Postman again.

When we use the same URL as before (http://localhost:5000/Character) we get the first character back.

{

"id": 0,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

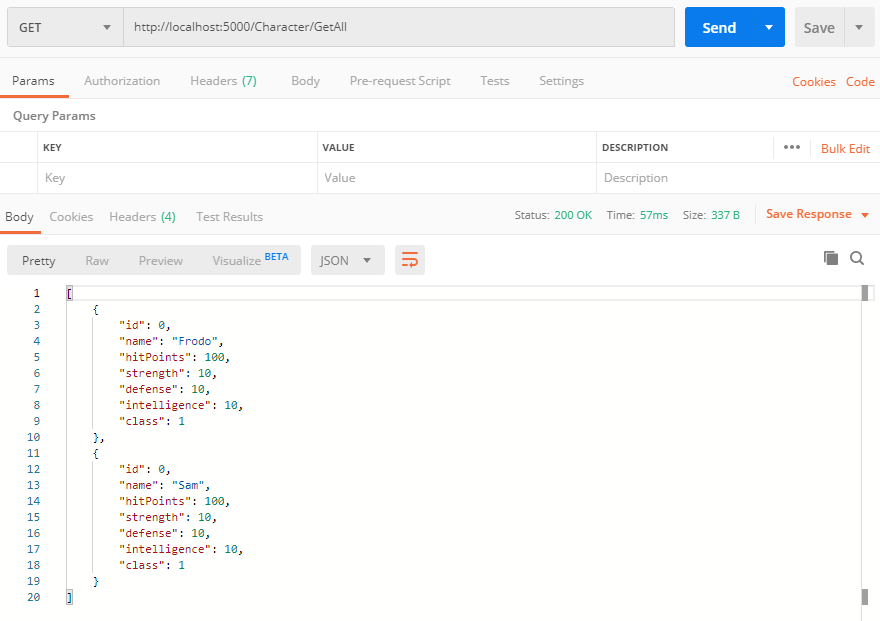
"defense": 10,

"intelligence": 10,

"class": 1

}

When we now add the string GetAll after the name of the controller and a slash (http://localhost:5000/Character/GetAll) we get all our knights back. Perfect.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--sNvgSss1--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/lwp4jzdk91pimvm25efl.png)

There’s still one more thing we can add, and that’s the HTTP request method.

In fact, we can combine the route with the HTTP method. It’s quite simple. Just replace Route with HttpGet, leave the string and you’re done.

[HttpGet("GetAll")]

public IActionResult Get()

{

return Ok(characters);

}

When we test this again in Postman, everything should work as expected.

Since receiving the first RPG character of a list can get quite boring, let’s add an argument to our request.

## **Routing with Parameters**

To get a particular RPG character, we could send the Id of this certain character to the web service.

First, we add the Id to the mock characters in the controller. The default value is 0, so we just add the Id with the value 1 to the second character.

private static List<Character> characters = new List<Character> {

new Character(),

new Character { Id = 1, Name = "Sam"}

};

Now it’s getting interesting. We add a new parameter to the GetSingle() method and use LINQ to find the RPG character with the given Id in the list of all characters. We do that with FirstOrDefault() followed by a lambda expression. This method now returns the first character where the Id of the character equals the given Id.

public IActionResult GetSingle(int id)

{

return Ok(characters.FirstOrDefault(c => c.Id == id));

}

To make this work, we also have to add the System.Linq type.

using Microsoft.AspNetCore.Mvc;

using dotnet\_rpg.Models;

using System.Collections.Generic;

using System.Linq;

The last thing would be to add an attribute with the route, the HTTP request method and the parameter. The parameter has to be added in curly brackets.

[HttpGet("{id}")]

public IActionResult GetSingle(int id)

{

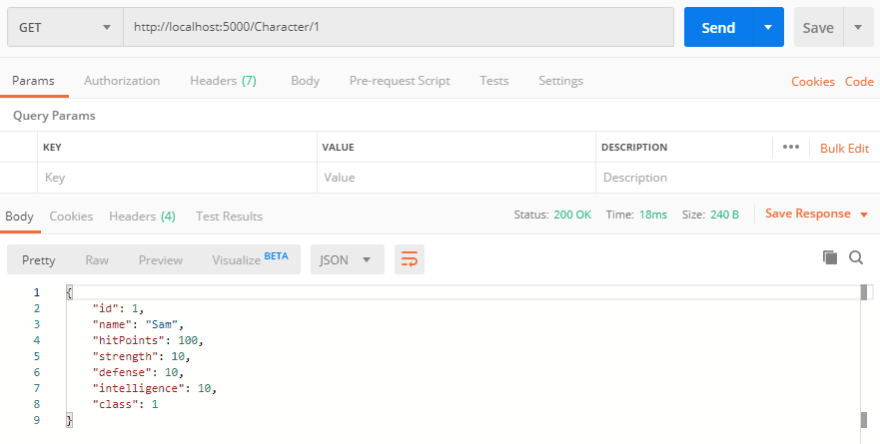
return Ok(characters.FirstOrDefault(c => c.Id == id));

}

That’s it!

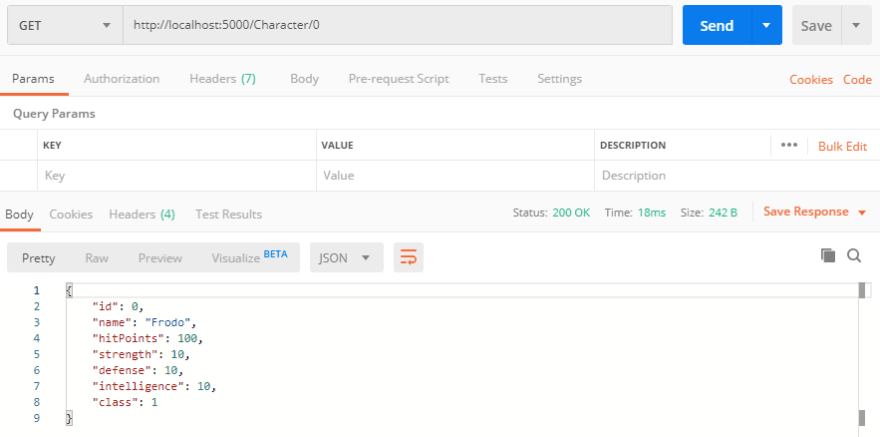
The parameter in the route has to match the parameter name in the function itself. Let’s test that with Postman.

The only thing we have to enter after the controller name is the Id, e.g. 1 (http://localhost:5000/Character/1).

[](https://res.cloudinary.com/practicaldev/image/fetch/s--dasUmqZf--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/s2f16k56xyxep8nksi3r.png)

And there is Sam.

Testing this with 0, here's Frodo.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--QxksysFM--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/0dmbwh7qhn6xmmos8fn3.png)

This works just fine. Before we implement more methods in the controller, let’s have a look at the various HTTP request methods we will cover in this tutorial series.

## **HTTP Methods Explained**

The Hypertext Transfer Protocol or in short HTTP defines - as the MDN Web Docs put it -

a set of request methods to indicate the desired action to be performed for a given resource.

To this date, we can find 9 request methods in the documentation. The most common ones are GET, POST, PUT and DELETE.

These are the ones that are covered in this tutorial series, because you almost always can do everything that needs to be done with only these four.

Let’s go over them shortly.

The GET method requests a representation of the specified resource.

If you’re using this request, you should only receive data from the web service and not send data to it. Of course, you might want to send certain characteristics to the service, like an Id or a string and the back-end then grabs the proper object from the database, but you won’t send any objects to the service. You have already seen this when you received our RPG characters.

Sending objects to the service, in turn, would be done with a POST request.

The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

This means that you might want to add a complete new RPG character to the database, for instance. POST means, to put it simply, add or create a new object, and do whatever you want to do with that object in the back-end. Usually, this new object will be saved in a database.

The PUT method replaces all current representations of the target resource with the request payload.

So this is in essence an update of the complete object. If you want to change a property of an RPG character, let’s say just the name for example, you would send the whole object to the service and it would overwrite the complete entry in the database. Of course, there can be variations of that process, but the standard way is exactly that. Send an object to the service that already exists, and update every property of this object.

And finally the DELETE request. You know it already,

The DELETE method deletes the specified resource.

So you send an Id to the service via the URL, the service usually looks up the corresponding entry in the database and erases it. If you rather want to do a soft delete, meaning, just set a flag so that this entry will not be shown on the client, it’s actually an update, where you would then use the PUT request method.

But to make one thing clear: Nothing works automatically. You have to write all the service methods by yourself and define which HTTP request method has to be used by the client. You could in fact write a service method that uses GET, but then simply delete an object with the given Id. It’s all up to you.

Now how do the CRUD (Create, Read, Update, Delete) operations match with the HTTP request methods in general? Maybe you know it already.

To create an object, you would use POST. Reading an object is done with a GET call, updating with PUT and deleting, well, with DELETE, of course.

## **Add a New Character with POST**

Now that you know the most important HTTP methods, let’s continue with the Web API by creating a controller method to add a new character.

The idea behind that is, that the client - in our case Postman - sends a JSON object to the service, and the service then creates a new character based on the JSON data.

So let’s start writing the method.

public IActionResult AddCharacter(Character newCharacter)

As parameter we use the Character type.

In the body of the function we simply add the new character to the characters list, and then we return the complete list, so that we can see that the new character is part of the complete list.

public IActionResult AddCharacter(Character newCharacter)

{

characters.Add(newCharacter);

return Ok(characters);

}

Last but not least, we add an attribute to this method, and that would be the [HttpPost] attribute, because then we are able to send data to the service.

[HttpPost]

public IActionResult AddCharacter(Character newCharacter)

{

characters.Add(newCharacter);

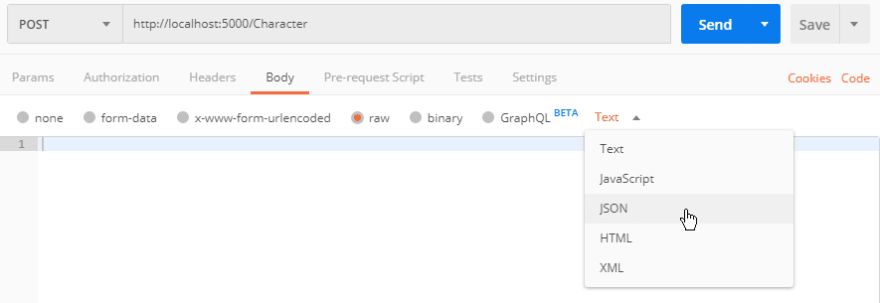
return Ok(characters);

}

It’s important to mention that the data or the JSON object, respectively, is sent via the body of this request. When we sent the Id to the service, we did it via the URL, now it’s done with the body. That’s a crucial difference.

Let’s test that now.

You can open a new tab in Postman, choose POST as HTTP method this time, the URL is again http://localhost:5000/character, but now we also add a body to the call. Choose raw and also the JSON format.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--g7tyhSUv--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/l10pmclsl8b0um21oxjd.png)

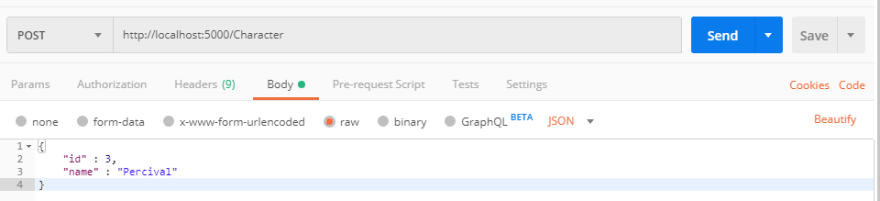
It's okay to send a new RPG character with just an Id and a name to the service.

{

"id" : 3,

"name" : "Percival"

}

[](https://res.cloudinary.com/practicaldev/image/fetch/s--klN6iw2u--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/e8r7u2dzqmbqqlrcss3g.png)

After hitting "Send" we see our complete list with the new character.

[

{

"id": 0,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

},

{

"id": 1,

"name": "Sam",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

},

{

"id": 3,

"name": "Percival",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

}

]

Just keep in mind, we don’t store the data persistently. So when we add another character, we get four in total, but as soon as we stop the web service and start it again, the list consists of the two initial mock characters and the added characters are gone.

## **Best Practice: Web API Structure**

Currently we’re doing all the logic of our web service calls in the Controller. The thing is, if an application or a web service is growing, you might want to separate the work into different classes. Or, if you need to do the same work over and over again, you don’t want to copy your code and paste it into different controllers, of course.

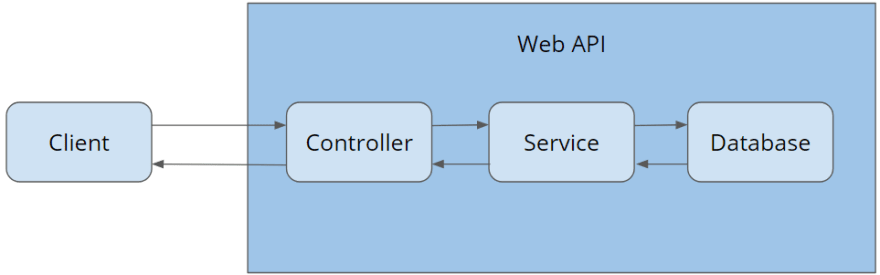
That’s where services come in.

The controllers should actually be pretty simple and just forward data to the service and return the result to the client. Nothing else.

To be able to do that, we will inject the necessary services into the controller - so we will use dependency injection. The great thing about dependency injection is, that you’re able to use the same service in several controllers and if you want to change the actual implementation of a service, you just change one service class and you don’t have to touch every single controller where you’re using this service.

We’ll come to the details when we actually implement this stuff.

So, the client sends a request to the Web API. The controller takes this request, calls the corresponding service request, the service then does all the magic (getting an RPG character out of the database, for instance) and then the result goes back to the controller and then to the client. That’s it.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--0M7YiAkx--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/zbw97x5kspanfkpqhj17.png)

Apart form that, we can also introduce the idea of Data Transfer Objects, or short DTOs. We already have Models, but it’s common to use these DTOs for the communication between client and server.

The difference is this: DTOs are objects you won’t find in the database, i.e. they won’t be mapped. Models, in turn, are a representation of a database table.

When we have a look at our RPG character model, later on we will see a table in the database that has exactly the same properties or fields.

Let’s say we add the field DateCreated or IsDeleted. That's information the user does not need to see.

In this case we want to save this information in the database but don’t want to send it back to the client. Here the DTO comes in.

We grab the model and map information of the model to the DTO. There are libraries that do this for us, like Automapper, so we don’t have to do this manually.

Apart from that we can also create DTOs that combine properties of several models. They simply give us more freedom in what we want to return to the client.

And it’s not only about returning data. You’ve already seen the example of creating a new character. In that case, we could use a DTO as well. So, an object with certain information the client sends to the web service. The service then grabs these information and maps them to the actual model.

We’ll use DTOs in future chapters so that everything should become clear.

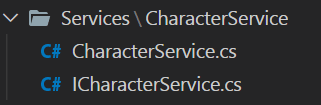
Alright, enough with the theory, let’s build the structure in our project now.

## **Character Service**

So, let’s implement a clean structure now. We start with creating new folders.

The first one is the Services folder, and in there we create the CharacterService folder.

In that folder we create the interface ICharacterService and the corresponding implementation class CharacterService.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--u3gc6AdT--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/3yu9291m5j4bhfu7svft.png)

The interface gets three methods. In essence, the methods you already know from the CharacterController.

public interface ICharacterService

{

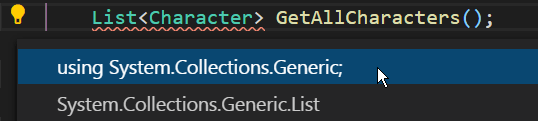
List<Character> GetAllCharacters();

Character GetCharacterById(int id);

List<Character> AddCharacter(Character newCharacter);

}

After adding these methods, we have to add using directives for the missing types List<T> and Character. Instead of adding them manually, we can click the lightbulb or press ctrl + . (control and period) and select the proper using directive there.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--AAIz7mSz--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/djom1pa2wdy9jkw510ne.png)

using System.Collections.Generic;

using dotnet\_rpg.Models;

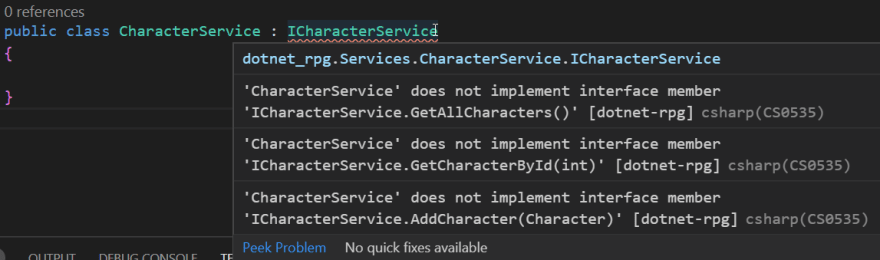
As soon as the using directives are ready, we can add the ICharacterService interface to the CharacterService class.

public class CharacterService : ICharacterService

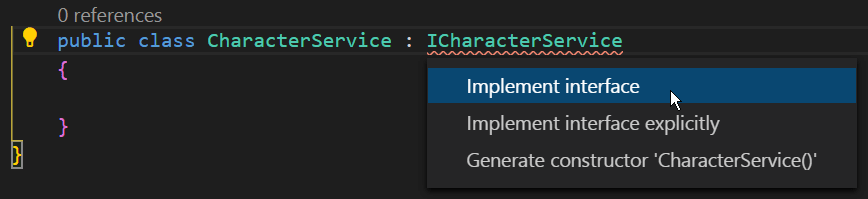
{

}

You should see an error now that says, that the interface methods are not implemented.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--P03B78_z--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/mhu8sp0730bbua17w90i.png)

Again, we can fix that real quick by clicking the yellow lightbulb on the left or pressing ctrl + . on the keyboard and choose “Implement interface”.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--HT-YY0A6--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/7blm2lhdxxf3kes98qlj.png)

As you can see, the methods have been generated for us, but we still have to implement the bodies and add the using directives again.

Regarding the bodies of the new three methods, we can actually just jump between the CharacterController and CharacterService and copy and paste the code.

public List<Character> AddCharacter(Character newCharacter)

{

characters.Add(newCharacter);

return characters;

}

public List<Character> GetAllCharacters()

{

return characters;

}

public Character GetCharacterById(int id)

{

return characters.FirstOrDefault(c => c.Id == id);

}

Additionally we have to add the characters list, of course. So let's copy this one as well.

private static List<Character> characters = new List<Character> {

new Character(),

new Character { Id = 1, Name = "Sam"}

};

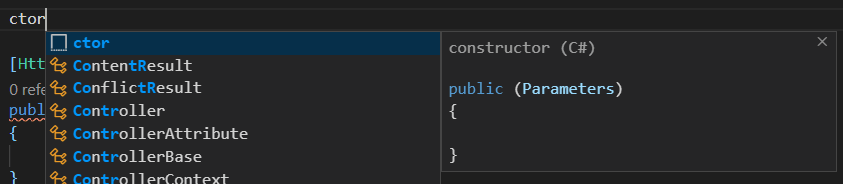
Last not least, we add the System.Linq type, so that we can use the FirstOrDefault() method.

using System.Collections.Generic;

using System.Linq;

using dotnet\_rpg.Models;

Regarding the CharacterController, we have to implement some more changes. First we need a constructor. We can add it with the snippet ctor. So, just write ctor and then hit tab.

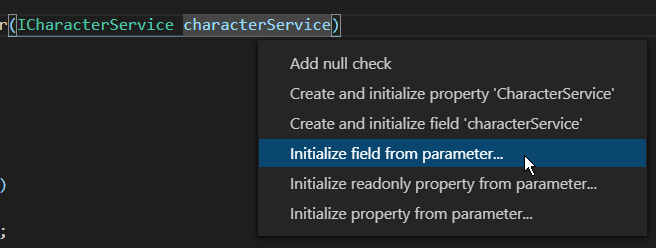
[](https://res.cloudinary.com/practicaldev/image/fetch/s--zJXPU7_X--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/za5u382076tteo0ufj3x.png)  
The parameter we want to add here is our ICharacterService interface.

public CharacterController(ICharacterService characterService)

{

}

We then add the using directive for dotnet\_rpg.Services.CharacterService and then create a new private field for the characterService. We can do that by putting the cursor over the argument, hit ctrl + . and then choose "Initialize field from parameter".

[](https://res.cloudinary.com/practicaldev/image/fetch/s--cLXPSKyf--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/blpdj42nykyzb3a9m9c9.png)

Additionally I like to add an underscore in front of the field name.

private readonly ICharacterService \_characterService;

public CharacterController(ICharacterService characterService)

{

\_characterService = characterService;

}

So now we already inject our new CharacterService into the controller. The list of characters can be removed.

Regarding the bodies of the methods, we have to call the \_characterService. So the code inside the brackets of the Ok() statement has to be replaced by the corresponding CharacterService method.

[HttpGet("GetAll")]

public IActionResult Get()

{

return Ok(\_characterService.GetAllCharacters());

}

[HttpGet("{id}")]

public IActionResult GetSingle(int id)

{

return Ok(\_characterService.GetCharacterById(id));

}

[HttpPost]

public IActionResult AddCharacter(Character newCharacter)

{

return Ok(\_characterService.AddCharacter(newCharacter));

}

Let's test the first method in Postman, i.e. http://localhost:5000/Character/GetAll.

We are getting an error!

System.InvalidOperationException: Unable to resolve service for type 'dotnet\_rpg.Services.CharacterService.ICharacterService' while attempting to activate 'dotnet\_rpg.Controllers.CharacterController'.

The Web API wants to inject the ICharacterService, but doesn't know which implementation class it should use. So we better tell it.

To do that, we open the Startup.cs, and add the line services.AddScoped<ICharacterService, CharacterService>(); to the ConfigureServices() method.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddScoped<ICharacterService, CharacterService>();

}

Now the Web API knows, that it has to use the CharacterService class whenever a controller wants to inject the ICharacterService. The beauty of that is, that whenever we want to change that and use another implementation class for instance, we just change this line and we're done. You don't have to change anything in the controllers injecting the ICharacterService.

With AddScoped() we create a new instance of the requested service for every request that comes in. There are also the methods AddTransient() and AddSingleton(). AddTransient() provides a new instance to every controller and to every service, even within the same request, and AddSingleton() creates only one instance that is used for every request.

We only need AddScoped().

When we test all the calls again in Postman, we don't get an error anymore. The GetAll call, for example, returns Frodo and Sam as expected.

[

{

"id": 0,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

},

{

"id": 1,

"name": "Sam",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

}

]

That's it for the second part of this tutorial series. Hope it was useful to you. To get notified for the next part, you can 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: Asynchronous calls, Data-Transfer-Objects (DTOs), modify a character with PUT & delete a character

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

### But wait, there’s more!

* Let’s connect on [Twitter](https://twitter.com/_PatrickGod), [YouTube](https://www.youtube.com/channel/UCq8LldVrjqe61KQttZlLW8g), [LinkedIn](https://www.linkedin.com/in/patrickgod) or here on [dev.to](https://dev.to/_patrickgod).
* Get the [5 Software Developer’s Career Hacks](http://patrickgod.com/5-software-developers-career-hacks/) for free.
* Enjoy more valuable articles for your developer life and career on [patrickgod.com](http://patrickgod.com/).