**All CRUD Operations with Entity Framework Core (GET, POST, PUT & DELETE)**

# Entity Framework Core (continued)

## **GET Implementations**

The very first thing we have to do to be able to access our RPG characters is to get access to the DataContext. We make use of dependency injection for that and add the context to the constructor of our CharacterService. Again, we can initialize this field from the parameter and change the name a little bit.

private readonly DataContext \_context;

public CharacterService(IMapper mapper, DataContext context)

{

\_context = context;

\_mapper = mapper;

}

With that, the \_context is available everywhere in our CharacterService.

In the GetAllCharacters() method we add a new line to get the characters from the database.

List<Character> dbCharacters = await \_context.Characters.ToListAsync();

Then, instead of selecting the static characters list, we use our new dbCharacters.

public async Task<ServiceResponse<List<GetCharacterDto>>> GetAllCharacters()

{

ServiceResponse<List<GetCharacterDto>> serviceResponse = new ServiceResponse<List<GetCharacterDto>>();

List<Character> dbCharacters = await \_context.Characters.ToListAsync();

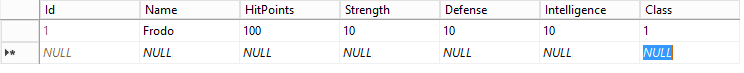
serviceResponse.Data = dbCharacters.Select(c => \_mapper.Map<GetCharacterDto>(c)).ToList();

return serviceResponse;

}

These are all the necessary changes. When we test that in Postman - URL is http://localhost:5000/Character/GetAll - we get no characters back, of course, because we haven’t added any characters in the database yet. So let’s add one in the SQL Server Management Studio.

We don’t have to enter an Id, just the name, hitpoints, and so on, and confirm the new entry with enter.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--SKlKiQrt--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/vq8nb7gb5jizelphhaem.png)

Let’s run another test and there is our Frodo.

{

"data": [

{

"id": 1,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

}

],

"success": true,

"message": null

}

Receiving a single RPG character is quite similar.

Let’s get the database character first. We do that with Character dbCharacter = await \_context.Characters.FirstOrDefaultAsync(c => c.Id == id);.

Then we simply map that dbCharacter.

public async Task<ServiceResponse<GetCharacterDto>> GetCharacterById(int id)

{

ServiceResponse<GetCharacterDto> serviceResponse = new ServiceResponse<GetCharacterDto>();

Character dbCharacter = await \_context.Characters.FirstOrDefaultAsync(c => c.Id == id);

serviceResponse.Data = \_mapper.Map<GetCharacterDto>(dbCharacter);

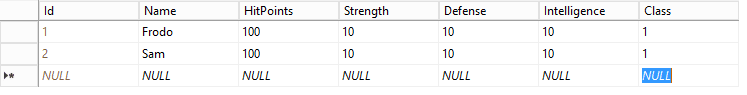
return serviceResponse;

}

By the way, you see that the yellow lines have disappeared because we’re finally using asynchronous methods. That’s cool, isn’t it?

Anyways, to test the new GetCharacterById() method, let’s add a second character to the database manually.

I’d like to call this guy Sam again.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--PWL4l48W--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/c0cxi7yi5dlnvhh26u8q.png)

Again, entering an Id is not necessary, SQL Server is doing this for us.

Alright. Back to Postman with the URL http://localhost:5000/Character/2 and we’re getting Sam back.

{

"data": {

"id": 2,

"name": "Sam",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

},

"success": true,

"message": null

}

Let’s get all RPG characters again.

{

"data": [

{

"id": 1,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

},

{

"id": 2,

"name": "Sam",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

}

],

"success": true,

"message": null

}

There are our mighty knights.

## **POST Implementations**

When we want to add an RPG character to the database, the first thing that stands out is that we don’t need to increase the Id manually. SQL Server will do this for us.

Apart from that, we won’t add the character to the static characters list anymore.

Instead, we use the AddAsync() method to add the RPG character to the database.

await \_context.Characters.AddAsync(character);

The next step is very important.

Since we made a change to the database, we have to save these changes explicitly. We do that with await \_context.SaveChangesAsync().

Last but not least, we return all RPG characters from the database.

public async Task<ServiceResponse<List<GetCharacterDto>>> AddCharacter(AddCharacterDto newCharacter)

{

ServiceResponse<List<GetCharacterDto>> serviceResponse = new ServiceResponse<List<GetCharacterDto>>();

Character character = \_mapper.Map<Character>(newCharacter);

await \_context.Characters.AddAsync(character);

await \_context.SaveChangesAsync();

serviceResponse.Data = (\_context.Characters.Select(c => \_mapper.Map<GetCharacterDto>(c))).ToList();

return serviceResponse;

}

That’s it!

Let’s test that in Postman. The URL is http://localhost:5000/Character/, the HTTP method is POST, and we send a JSON in the body with just the name, hitpoints and another class for starters.

{

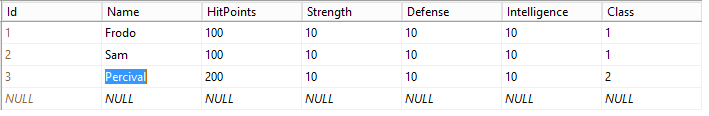
"name" : "Percival",

"hitpoints" : 200,

"class": 2

}

Again, we don’t have to send an Id and we don’t have to set it manually in the service. When we run this, we get all RPG characters back, and we can see the new entry in the database.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--VheJ-7lG--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/s2jzz1og45utm9ed6eyg.png)

## **PUT Implementations**

To update an RPG character, we first have to find a particular character in the database.

Instead of searching for the character with the given Id in the static characters list, we access the characters of the data context and search the RPG characters asynchronously.

Character character = await \_context.Characters.FirstOrDefaultAsync(c => c.Id == updatedCharacter.Id);

Overriding the properties stays the same.

But after that, we have to access the data context again and call the Update() method on that particular character.

Since we made a change to the character table, we have to save these changes again with await \_context.SaveChangesAsync();.

\_context.Characters.Update(character);

await \_context.SaveChangesAsync();

That’s it already!

Receiving the particular RPG character from the database, change the values, update the character in the database and save the changes again.

Again, it’s time to test that in Postman.

Let's receive a single character first. With the help of the result, we already have the proper JSON object to use for the update call. Let’s get Frodo with Id 1, for instance.

{

"id": 1,

"name": "Frodo",

"hitPoints": 100,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1

}

Now, we can update Frodo’s properties with the URL http://localhost:5000/Character/, the HTTP method PUT and this JSON object here.

{

"id": 1,

"name": "Frodo",

"hitPoints": 200,

"strength": 20,

"defense": 10,

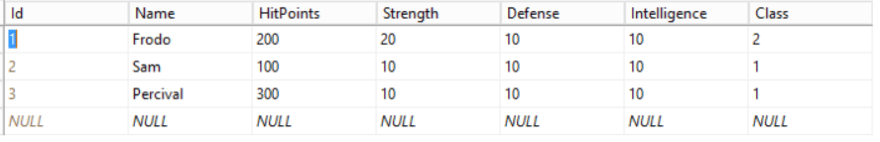
"intelligence": 10,

"class": 2

}

You can change anything you want except the Id, of course, because that’s how we find the correct RPG character.

When we run this call, we get the correct result and we can also see the changes in the SQL Server Management Studio.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--hwnSGZGY--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/gv68q5ofwqt4cuaszkod.png)

Remember, all properties of the characters will be changed in our update method. So make sure to get the correct values of the character first or implement methods to update single properties. Otherwise, you would overwrite the properties of a character with default values.

Just keep that in mind.

## **DELETE Implementations**

Alright, only deleting a character is left and then you know how to implement all the CRUD operations with Entity Framework Core.

First, let’s remove the static characters list because we won’t need it anymore.

Of course, this results in some errors in our DeleteCharacter() method, because this is the only method left using the characters list.

But don’t worry, we’ll fix that right now. Maybe you already know how to fix this.

In essence, we have to replace the old characters list with \_context.Characters.

Character character = \_context.Characters.First(c => c.Id == id);

\_context.Characters.Remove(character);

serviceResponse.Data = (\_context.Characters.Select(c => \_mapper.Map<GetCharacterDto>(c))).ToList();

After that, we call the asynchronous FirstAsync() method instead of First(). Don’t forget to add the await keyword.

Character character = await \_context.Characters.FirstAsync(c => c.Id == id);

Now, only one thing is missing. We have to save the changes to the database. So let’s call await \_context.SaveChangesAsync(); and we’re done.

Character character = await \_context.Characters.FirstAsync(c => c.Id == id);

\_context.Characters.Remove(character);

await \_context.SaveChangesAsync();

serviceResponse.Data = (\_context.Characters.Select(c => \_mapper.Map<GetCharacterDto>(c))).ToList();

To test this in Postman, we can remove the RPG character with Id 3, for instance. The URL is http://localhost:5000/Character/3 and the HTTP method is DELETE. Hit ‘Send’, and the character is gone. You can also see that in the database table, of course.

## **Summary**

Great! You successfully implemented all CRUD operations, meaning create, read, update and delete, with Entity Framework Core.

All your changes are now stored persistently in the database.

You now know what object-relational mapping is, how code-first migration works and how to use it with a SQL Server database. Additionally, you looked at your database with SQL Server Management Studio.

Regarding the code, you learned what the DbContext is and how to provide a connection string to access the database with all its entities.

Alright, it’s time to add the first relation to the RPG characters.

In the next section, we will add users who can own several characters. But before they can do that, they have to register for an account. So, in the next section, we will add a relation and also implement authentication.

That's it for the 6th part of this tutorial series. I hope it was useful for 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: Basic Authentication with a .NET Core Web API

*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/).