**Update & Remove Entities in .NET Core 3.1 Web API with PUT & DELETE**

# Web API Core (continued)

## **Modify a Character with PUT**

To modify or update an RPG character, we have to add a new method to the ICharacterService interface, the CharacterService class and the CharacterController. Let’s start with the interface and add the method UpdateCharacter. The return type is a GetCharacterDto so that we can see the changes directly, and the parameter is a new DTO, UpdateCharacterDto.

public interface ICharacterService

{

Task<ServiceResponse<List<GetCharacterDto>>> GetAllCharacters();

Task<ServiceResponse<GetCharacterDto>> GetCharacterById(int id);

Task<ServiceResponse<List<GetCharacterDto>>> AddCharacter(AddCharacterDto newCharacter);

Task<ServiceResponse<GetCharacterDto>> UpdateCharacter(UpdateCharacterDto updatedCharacter);

}

Let’s add the new DTO real quick. We create the new C# class UpdateCharacterDto and actually can copy and paste all properties from the GetCharacterDto.

public class UpdateCharacterDto

{

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;

}

Now we can implement the method in the CharacterService. We implement the interface automatically and then start with the ServiceResponse. Then we try to find the RPG character with the given id of the updatedCharacter in the characters list. After that, we overwrite almost every property of this RPG character one by one, i.e. the Name, Class, Defense, Hitpoints, Intelligence and the Strength. And finally, we map the Character to the GetCharacterDto and return the ServiceResponse. One last thing is missing, we have to add the async keyword.

public async Task<ServiceResponse<GetCharacterDto>> UpdateCharacter(UpdateCharacterDto updatedCharacter)

{

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

Character character = characters.FirstOrDefault(c => c.Id == updatedCharacter.Id);

character.Name = updatedCharacter.Name;

character.Class = updatedCharacter.Class;

character.Defense = updatedCharacter.Defense;

character.HitPoints = updatedCharacter.HitPoints;

character.Intelligence = updatedCharacter.Intelligence;

character.Strength = updatedCharacter.Strength;

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

return serviceResponse;

}

Okay, off to the CharacterController. In essence, we can copy the AddCharacter() method, change the name, the service method, and the parameter, and also the HTTP method attribute, which is [HttpPut].

[HttpPut]

public async Task<IActionResult> UpdateCharacter(UpdateCharacterDto updatedCharacter)

{

return Ok(await \_characterService.UpdateCharacter(updatedCharacter));

}

It’s time to test that in Postman. First, let’s get all RPG characters in one tab for reference.

{

"data": [

{

"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

}

],

"success": true,

"message": null

}

In another tab, we can do the update call. The HTTP method is PUT, the URL is http://localhost:5000/Character. In the body, we have to enter an id and then the properties we want to change. Maybe give the character a different name, more hitpoints and also turn the character into a mage.

{

"id" : 1,

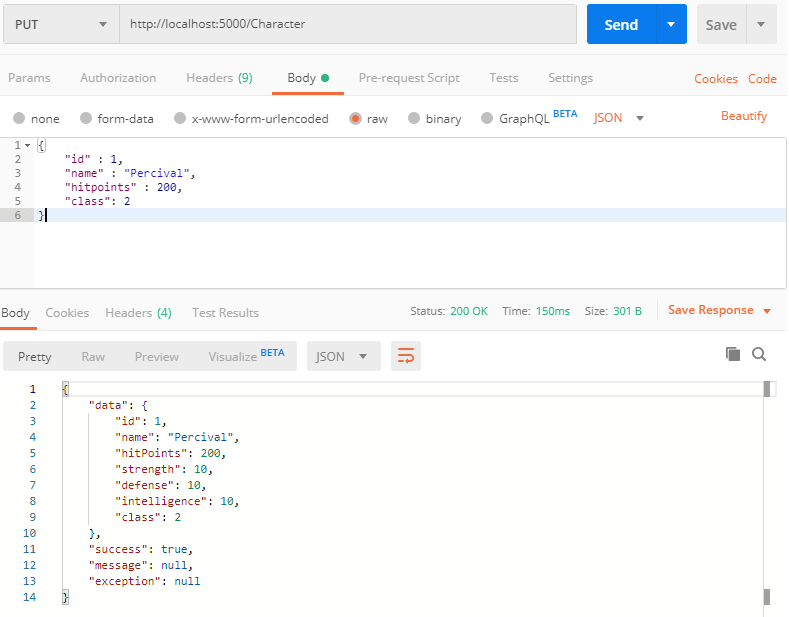
"name" : "Percival",

"hitpoints" : 200,

"class": 2

}

Hit “Send”, and there’s the updated character.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--OdVLvU_u--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/5f2lnvmb70tb2814533m.png)

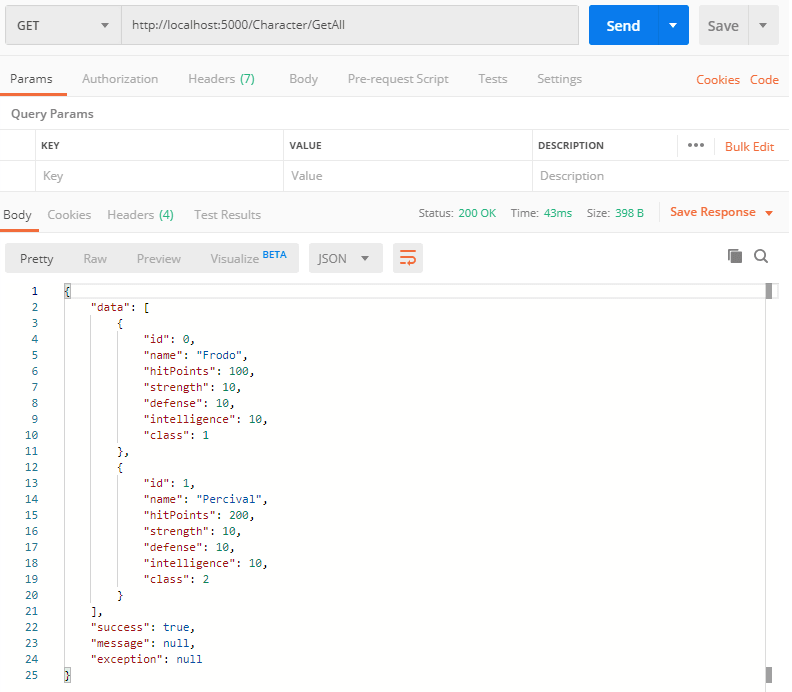
Keep in mind, that every single property has been updated. This means, that the Strength, Defense, and so on will be overridden, even though we did not send a value for these properties with the body.

If you would run another update and leave the RPG class, Percival would be a knight again, because that's the default value.

So, you have to pay attention to how you design your front end in this case. Do you want to update single properties or all at once by receiving the current values of the character and save them back again, even though they did not change?

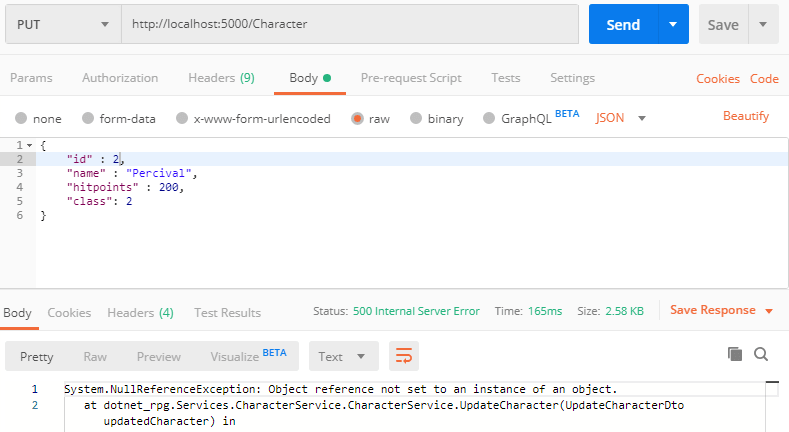
Alright, that was just a side note.

Now, what about the whole character list?

[](https://res.cloudinary.com/practicaldev/image/fetch/s--4jJzPzKQ--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/e2buoibfgbybiecm9zbz.png)

Works. Perfect.

Now, let’s try to update a character that doesn’t exist. Just change the id to 2 and see what happens.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--5wU_bRP5--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/hnvvxvagxzda0yycgeia.png)

We’re getting a NullReferenceException. We have two options now. Either we catch that exception with a try/catch block or we just check if we find a character in the characters list.

Or we do both.

Let’s start with the try/catch block. In the catch block, we just set serviceResponse.Success to false and the serviceResponse.Message to the actual exception message.

public async Task<ServiceResponse<GetCharacterDto>> UpdateCharacter(UpdateCharacterDto updatedCharacter)

{

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

try

{

Character character = characters.FirstOrDefault(c => c.Id == updatedCharacter.Id);

character.Name = updatedCharacter.Name;

character.Class = updatedCharacter.Class;

character.Defense = updatedCharacter.Defense;

character.HitPoints = updatedCharacter.HitPoints;

character.Intelligence = updatedCharacter.Intelligence;

character.Strength = updatedCharacter.Strength;

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

}

catch (Exception ex)

{

serviceResponse.Success = false;

serviceResponse.Message = ex.Message;

}

return serviceResponse;

}

That’s it already. Let’s test that again in Postman.

{

"data": null,

"success": false,

"message": "Object reference not set to an instance of an object."

}

A possible front end can work with that. Maybe another message would be more suitable for the user.

Alternatively, we can add a slight modification to the CharacterController. You see, that we’re still getting a status code 200 OK. Well, a character wasn’t found, so maybe we can also return a 404 Not Found response.

So in the CharacterController we first get the ServiceResponse. Then we check if the Data is null. If so, we return NotFound(), else we return Ok() as usual.

[HttpPut]

public async Task<IActionResult> UpdateCharacter(UpdateCharacterDto updatedCharacter)

{

ServiceResponse<GetCharacterDto> response = await \_characterService.UpdateCharacter(updatedCharacter);

if (response.Data == null)

{

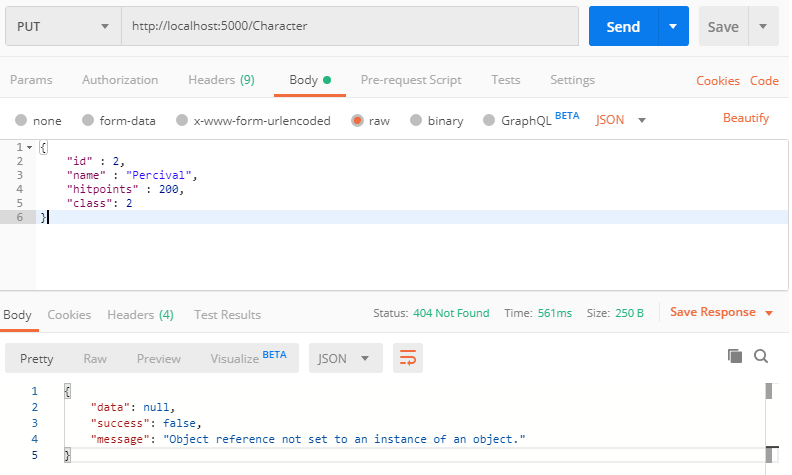
return NotFound(response);

}

return Ok(response);

}

When we test that now in Postman, we’re still getting the whole response back, but the status code is 404 Not Found.

[](https://res.cloudinary.com/practicaldev/image/fetch/s---uLUewRi--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/a1h6v2u2r7v5uesorfvz.png)

Feel free to play around with that. For instance, you do not have to use the Message of the ServiceResponse only in case of an error. What about a success message like “You’re character has been saved.”?

Anyways, let’s move on and remove an RPG character.

## **Delete a Character**

To delete an RPG character, again, we have to make modifications to the ICharacterService interface, the CharacterService and the CharacterController.

Let’s call the method in the interface DeleteCharacter() with id as an argument. And let’s return the remaining RPG characters.

Task<ServiceResponse<List<GetCharacterDto>>> DeleteCharacter(int id);

Regarding the CharacterService, we can implement the interface automatically again and then copy the code of the UpdateCharacter() method and just make some changes.

First, we add the async keyword. Then we fix the ServiceResponse, because we return a list of characters. Instead of FirstOrDefault() we could use First(). The difference is, that FirstOrDefault() will return null if no matching entity was found, and First() throws an exception. So let's take advantage of that.

If, however, a character was found, we remove it from the list with characters.Remove(character).

In the end, we return a mapped list of the remaining characters, similar to the expression in the GetAllCharacters() or AddCharacters() method.

public async Task<ServiceResponse<List<GetCharacterDto>>> DeleteCharacter(int id)

{

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

try

{

Character character = characters.First(c => c.Id == id);

characters.Remove(character);

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

}

catch (Exception ex)

{

serviceResponse.Success = false;

serviceResponse.Message = ex.Message;

}

return serviceResponse;

}

The controller method is a combination of the GetSingle() and UpdateCharacter() method.

You can copy the GetSingle() method. First, change the method name to Delete() and also use the attribute HttpDelete. The body is almost the same as the body of the UpdateCharacter() method. Only the ServiceResponse is different and the call to the CharacterService. It is DeleteCharacter() with the given id, of course.

[HttpDelete("{id}")]

public async Task<IActionResult> Delete(int id)

{

ServiceResponse<List<GetCharacterDto>> response = await \_characterService.DeleteCharacter(id);

if (response.Data == null)

{

return NotFound(response);

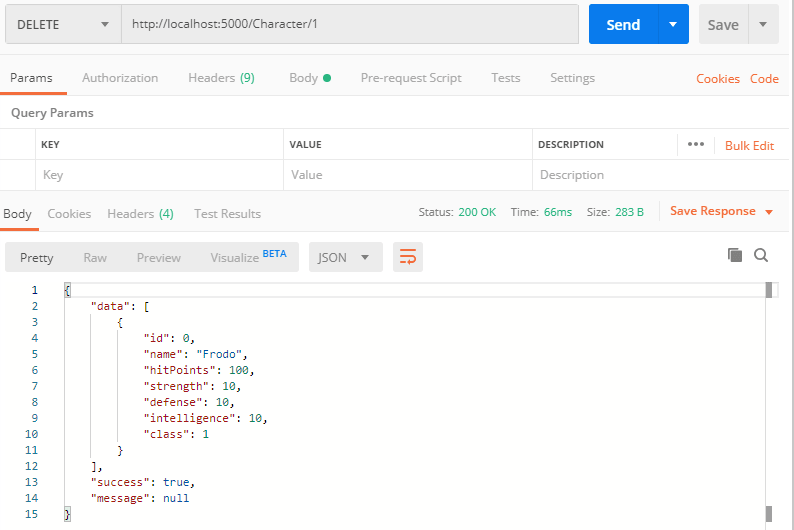
}

return Ok(response);

}

Let’s test that.

The HTTP request method is DELETE, URL is http://localhost:5000/Character/1 to remove the character with Id 1.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--PiVzt3DW--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/qkbfp9er70bdgm48np6m.png)

Perfect! This works just fine.

Our Web API with all CRUD operations is done. It’s time to save the data in a database with Entity Framework Core and SQL Server.

## **Summary**

Congratulations! A lot has happened in the previous lectures.

You created your first Web API with .NET Core 3.1 with all CRUD operations, meaning Create, Read, Update and Delete.

You learned what HTTP Request Methods are and used GET, POST, PUT and DELETE for your web service calls.

Additionally, you now know what the Model-View-Controller or MVC design pattern is and how it works.

And you learned some concepts and practices used in large real-world applications like a clean structure of your Web API with dependency injection, asynchronous calls, a proper service response, and data transfer objects.

You’re definitely ready for the Entity Framework Core section.

That's it for the 4th part of this tutorial series. Hope it 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: Entity Framework Core

*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/).