**More Than Just CRUD with .NET Core 3.1 - Part 2**

# More Than Just CRUD with .NET Core 3.1 (continued)

## **Start a Fight**

Again we start with new DTOs. First, we create a FightRequestDto with only one property, a List of int values for the CharacterIds. Do you see the possibilities here? We could not only let two RPG characters fight against each other, but we could also start a deathmatch!

using System.Collections.Generic;

namespace dotnet\_rpg.Dtos.Fight

{

public class FightRequestDto

{

public List<int> CharacterIds { get; set; }

}

}

The result of the automatic fight or deathmatch should also look a bit different. How about some kind of fighting log that will simply tell us with some sentences how the battle took place?

We create a new FightResultDto class, again with only one property, a List of string values. This will be our Log and we can already initialize this List so that we can easily add new entries right away later.

using System.Collections.Generic;

namespace dotnet\_rpg.Dtos.Fight

{

public class FightResultDto

{

public List<string> Log { get; set; } = new List<string>();

}

}

Off to the IFightService interface. We add a new method called Fight() which returns the FightResultDto and takes a FightRequestDto as request.

Task<ServiceResponse<FightResultDto>> Fight(FightRequestDto request);

Regarding the FightController we can copy another method again, replace the method name as well as the DTO and the method of the \_fightService, and we can also remove the route so that this method becomes the default POST call of this controller.

[HttpPost]

public async Task<IActionResult> Fight(FightRequestDto request)

{

return Ok(await \_fightService.Fight(request));

}

Now it’s getting interesting. Let’s create the Fight() method in the FightService. We implement the interface and add the async keyword.

Then we initialize the ServiceResponse and also initialize the Data with a new FightResultDto object.

ServiceResponse<FightResultDto> response = new ServiceResponse<FightResultDto>

{

Data = new FightResultDto()

};

Next, we add our default try/catch block as always and return a failing response in case of an error.

Inside the try-block, we want to grab all the given RPG characters. Remember, this could also be a battle with dozens of fighters - sounds great, huh?

To get the characters we access the \_context as usual, include the Weapon, the CharacterSkills and the Skill and then we use the function Where to get all the characters from the database that match the given IDs. We do that with Where(c => request.CharacterIds.Contains(c.Id)) and then turn the result into a List with ToListAsync().

List<Character> characters =

await \_context.Characters

.Include(c => c.Weapon)

.Include(c => c.CharacterSkills).ThenInclude(cs => cs.Skill)

.Where(c => request.CharacterIds.Contains(c.Id)).ToListAsync();

Alright, we’ve got our fighters. Now we define a boolean variable called defeated, set it to false and start two loops - a while loop and a foreach. The while loop stops when the first character is defeated. The idea behind the foreach is that every character will attack in order.

bool defeated = false;

while (!defeated)

{

foreach (Character attacker in characters)

{

}

}

Inside the foreach loop we grab all the opponents of the attacker first. We do that with the help of the Where() function again by simply filtering all characters that don’t have the Id of the attacker.

Then we randomly choose one opponent. We do that with new Random().Next() and then pass the opponents.Count. That way, we get a random number we can use as an index for the opponents list.

List<Character> opponents = characters.Where(c => c.Id != attacker.Id).ToList();

Character opponent = opponents[new Random().Next(opponents.Count)];

Next, we declare two variables we will use for the resulting log. The damage which is 0 and attackUsed as an empty string. You guessed it, I want to see the used weapon or skill of every single attack.

int damage = 0;

string attackUsed = string.Empty;

The next step is to decide if the attacker uses his weapon or one of his skills. To do that, we define a boolean variable useWeapon and throw a die again with new Random().Next(2). If the result is 0 we choose the weapon, if not, we choose a skill.

bool useWeapon = new Random().Next(2) == 0;

if (useWeapon)

{

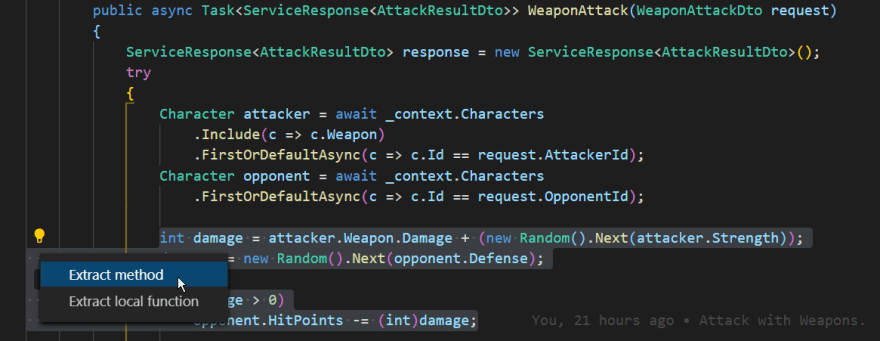
}

else

{

}

Now we can set the name of the used attack and then already calculate the damage. Since we did that already in the WeaponAttack() method, let’s extract this part and create a new method we can reuse. We can do that by selecting the code, open the quick-fix menu and choose Extract method.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--eml8xqBb--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/upuad0hmnkq2um8izm3x.png)

We can call this new method DoWeaponAttack() for instance.

private static int DoWeaponAttack(Character attacker, Character opponent)

{

int damage = attacker.Weapon.Damage + (new Random().Next(attacker.Strength));

damage -= new Random().Next(opponent.Defense);

if (damage > 0)

opponent.HitPoints -= (int)damage;

return damage;

}

So now we can use this method and pass the attacker and the opponent to get the damage value and already decrease the HitPoints of the opponent.

if (useWeapon)

{

attackUsed = attacker.Weapon.Name;

damage = DoWeaponAttack(attacker, opponent);

}

Regarding the damage for a Skill we can also extract the calculation from the SkillAttack() method and call this method DoSkillAttack(). You see that we have to pass an attacker, an opponent and a characterSkill this time.

private static int DoSkillAttack(Character attacker, Character opponent, CharacterSkill characterSkill)

{

int damage = characterSkill.Skill.Damage + (new Random().Next(attacker.Intelligence));

damage -= new Random().Next(opponent.Defense);

if (damage > 0)

opponent.HitPoints -= (int)damage;

return damage;

}

Great. So, we can use this new method for the damage, but first, we have to get a CharacterSkill.

Again we choose one randomly from the list of CharacterSkills of our attacker. We do that with new Random().Next(attacker.CharacterSkills.Count). This will be the index of our CharacterSkill we store in the variable randomSkill.

With this randomSkill variable, we can now set the name of the attackUsed by setting the Skill.Name and finally calculate the damage with the DoSkillAttack() method and give this method the attacker, the opponent and the attacker.CharacterSkill[randomSkill].

else

{

int randomSkill = new Random().Next(attacker.CharacterSkills.Count);

attackUsed = attacker.CharacterSkills[randomSkill].Skill.Name;

damage = DoSkillAttack(attacker, opponent, attacker.CharacterSkills[randomSkill]);

}

Alright, we’ve got our attacks.

That’s all the information we need to add this attack to the log. I would like to add a sentence that looks like “attacker.Name attacks opponent.Name using attackUsed with damage damage” where the damage value has to be above or equal 0. Nice.

response.Data.Log.Add($"{attacker.Name} attacks {opponent.Name} using {attackUsed} with {(damage >= 0 ? damage : 0)} damage.");

We are almost done. Now we have to decide what to do if an opponent has been defeated. So, in case the opponent.HitPoints are 0 or less we set the variable defeated to true. We increase the Victories of the attacker and the Defeats of the opponent. We can add new sentences to our log! Something like “opponent.Name has been defeated!” and “attacker.Name wins with attacker.HitPoints HP left!”. And finally, we stop the fight and leave the loop with break.

if (opponent.HitPoints <= 0)

{

defeated = true;

attacker.Victories++;

opponent.Defeats++;

response.Data.Log.Add($"{opponent.Name} has been defeated!");

response.Data.Log.Add($"{attacker.Name} wins with {attacker.HitPoints} HP left!");

break;

}

After that, we increase the Fights value of all characters in another forEach and also reset the HitPoints for the next fight.

characters.ForEach(c =>

{

c.Fights++;

c.HitPoints = 100;

});

And then we update all characters in the database with \_context.Characters.UpdateRange(characters) and save everything as usual with SaveChangesAsync().

\_context.Characters.UpdateRange(characters);

await \_context.SaveChangesAsync();

And that’s it! The automatic fight may begin!

public async Task<ServiceResponse<FightResultDto>> Fight(FightRequestDto request)

{

ServiceResponse<FightResultDto> response = new ServiceResponse<FightResultDto>

{

Data = new FightResultDto()

};

try

{

List<Character> characters =

await \_context.Characters

.Include(c => c.Weapon)

.Include(c => c.CharacterSkills).ThenInclude(cs => cs.Skill)

.Where(c => request.CharacterIds.Contains(c.Id)).ToListAsync();

bool defeated = false;

while (!defeated)

{

foreach (Character attacker in characters)

{

List<Character> opponents = characters.Where(c => c.Id != attacker.Id).ToList();

Character opponent = opponents[new Random().Next(opponents.Count)];

int damage = 0;

string attackUsed = string.Empty;

bool useWeapon = new Random().Next(2) == 0;

if (useWeapon)

{

attackUsed = attacker.Weapon.Name;

damage = DoWeaponAttack(attacker, opponent);

}

else

{

int randomSkill = new Random().Next(attacker.CharacterSkills.Count);

attackUsed = attacker.CharacterSkills[randomSkill].Skill.Name;

damage = DoSkillAttack(attacker, opponent, attacker.CharacterSkills[randomSkill]);

}

response.Data.Log.Add($"{attacker.Name} attacks {opponent.Name} using {attackUsed} with {(damage >= 0 ? damage : 0)} damage.");

if (opponent.HitPoints <= 0)

{

defeated = true;

attacker.Victories++;

opponent.Defeats++;

response.Data.Log.Add($"{opponent.Name} has been defeated!");

response.Data.Log.Add($"{attacker.Name} wins with {attacker.HitPoints} HP left!");

break;

}

}

}

characters.ForEach(c =>

{

c.Fights++;

c.HitPoints = 100;

});

\_context.Characters.UpdateRange(characters);

await \_context.SaveChangesAsync();

}

catch (Exception ex)

{

response.Success = false;

response.Message = ex.Message;

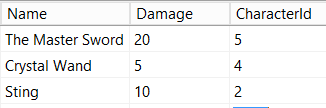
}

return response;

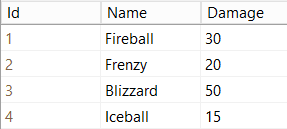
}

But before we start a fight, let’s prepare a third character, so that we actually get a deathmatch.

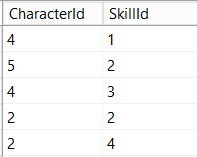
We already got Sam. In the SQL Server Management Studio, we can give him a new weapon directly. Maybe the Sting with a damage value of 10.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--oCDogLAr--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/jkciul0mw976t3214z55.png)

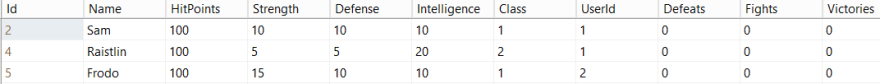
Also, we can add a new Skill, like an Iceball which makes 15 damage.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--s_K3ju58--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/vb9qt5w6v5c95ilyzvfk.png)

Then we add the skills Frenzy and Iceball to Sam in the CharacterSkills table.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--kLKciKNm--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/3rju8t20742zg9h3e1up.png)

Alright, let’s make sure that every character has 100 hit points and then we can start.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--y-SU5ZXj--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ht289yfz6n4ff2jikime.png)

The URL in Postman is http://localhost:5000/fight/, the HTTP method is POST and the body can already be an array of three characterIds.

{

"characterids" : [2,4,5]

}

Let’s fight!

{

"data": {

"log": [

"Sam attacks Frodo using Sting with 19 damage.",

"Raistlin attacks Frodo using Fireball with 33 damage.",

"Frodo attacks Raistlin using The Master Sword with 28 damage.",

"Sam attacks Raistlin using Sting with 19 damage.",

"Raistlin attacks Frodo using Crystal Wand with 7 damage.",

"Frodo attacks Sam using The Master Sword with 17 damage.",

"Sam attacks Raistlin using Frenzy with 21 damage.",

"Raistlin attacks Frodo using Crystal Wand with 6 damage.",

"Frodo attacks Raistlin using Frenzy with 19 damage.",

"Sam attacks Frodo using Iceball with 21 damage.",

"Raistlin attacks Frodo using Blizzard with 51 damage.",

"Frodo has been defeated!",

"Raistlin wins with 13 HP left!"

]

},

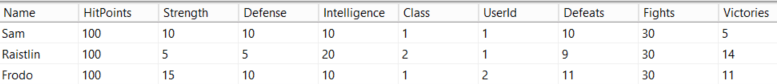
"success": true,

"message": null

}

And that’s how a deathmatch looks like. A beautiful use of different weapons and skills. You see the whole course of the fight.

Feel free to start a couple of more battles, so that the Victories and Defeats of the RPG characters change.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--6OB9uuBi--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/42xbh7r5rpu4tb1t986u.png)

We will use these values to display a highscore next.

## **Highscore: Sort & Filter Entities**

To receive a highscore or a ranking of all RPG characters that have ever participated in a fight, we need a GET method and a new DTO for the result. A request object is not necessary.

Let’s start with the DTO. In the Fight folder, we create the C# class HighscoreDTO. The properties I would like to see are the Id of the character, the Name and of course the number of Fights, Victories and Defeats.

namespace dotnet\_rpg.Dtos.Fight

{

public class HighscoreDTO

{

public int Id { get; set; }

public string Name { get; set; }

public int Fights { get; set; }

public int Victories { get; set; }

public int Defeats { get; set; }

}

}

We will use AutoMapper to map the Character information to the HighscoreDTO later, so let’s create a new map in our AutoMapperProfile class.

CreateMap<Character, HighscoreDTO>();

Okay. Next, we add a new method to the IFightService interface called GetHigscore() which doesn’t take any arguments but it returns a ServiceResponse with a List of HighscoreDTO instances.

Task<ServiceResponse<List<HighscoreDTO>>> GetHighscore();

In the FightController, we also add the new GetHighscore() method with no parameter and even no attribute, because this will be our default GET call. We simply use the method \_fightService.GetHighscore() and that’s it.

public async Task<IActionResult> GetHighscore()

{

return Ok(await \_fightService.GetHighscore());

}

Now the FightService. First, we inject the IMapper to be able to map the characters to the HighscoreDTO. We initialize the field from this parameter, add the underscore and also the AutoMapper using directive.

private readonly DataContext \_context;

private readonly IMapper \_mapper;

public FightService(DataContext context, IMapper mapper)

{

\_mapper = mapper;

\_context = context;

}

Then we can implement the interface and start writing the code for the GetHighscore() method. We start with the async keyword.

Then, we want the characters, of course. In this example we only want to see the characters that have participated in a fight, so Where the Fights value is greater than 0.

And then we want to see a ranking, so we use OrderByDescending() to order the characters by their Victories. If the number of Victories should be the same for some characters, we can then order by their Defeats ascending by using ThenBy(). In the end, we turn the result to a List.

List<Character> characters = await \_context.Characters

.Where(c => c.Fights > 0)

.OrderByDescending(c => c.Victories)

.ThenBy(c => c.Defeats)

.ToListAsync();

Now we can already create the response. If you want, you can use var for that. It’s controversial whether using var is a best practice or not. I think in the case of initializing an object of a long type name, it is okay to do so. In the end, it’s still strongly typed.

Anyways, we initialize the ServiceResponse and already set the Data by mapping all characters to a HighscoreDTO. We do that with characters.Select() and then use the \_mapper with the Map() function for every character and turn that result also into a List.

var response = new ServiceResponse<List<HighscoreDTO>>

{

Data = characters.Select(c => \_mapper.Map<HighscoreDTO>(c)).ToList()

};

And finally we return the response. Done!

public async Task<ServiceResponse<List<HighscoreDTO>>> GetHighscore()

{

List<Character> characters = await \_context.Characters

.Where(c => c.Fights > 0)

.OrderByDescending(c => c.Victories)

.ThenBy(c => c.Defeats)

.ToListAsync();

var response = new ServiceResponse<List<HighscoreDTO>>

{

Data = characters.Select(c => \_mapper.Map<HighscoreDTO>(c)).ToList()

};

return response;

}

Testing this is simple. In Postman we use the URL http://localhost:5000/fight/ and this time the HTTP method is GET. Hitting “Send” returns our honorable fighters in the right order.

{

"data": [

{

"id": 4,

"name": "Raistlin",

"fights": 30,

"victories": 14,

"defeats": 9

},

{

"id": 5,

"name": "Frodo",

"fights": 30,

"victories": 11,

"defeats": 11

},

{

"id": 2,

"name": "Sam",

"fights": 30,

"victories": 5,

"defeats": 10

}

],

"success": true,

"message": null

}

Beautiful!

## **Summary**

Well, congratulations! You have completed the whole course. You can be proud of yourself. I certainly am.

You learned how to build a Web API with all CRUD operations using the HTTP methods GET, POST, PUT and DELETE in .NET Core.

After implementing all these operations you learned how to store your entities and save your changes in a SQL Server database with Entity Framework Core. You utilized code-first migration for that.

But not everybody should have access to your entities. That’s why you added authentication to your web service. Users have to register and authenticate to get access to their data.

Additionally, you not only learned how to hash a password and verify it again, but you also implemented the use of JSON web tokens. With the help of that token, the user doesn’t have to send her credentials with every call. The token does the job for us.

Then you covered advanced relationships in Entity Framework Core and the SQL Server database. By the example of users, RPG characters, weapons, and skills, you implemented one-to-one, one-to-many and many-to-many relationships.

And finally, you got creative by letting your RPG characters fight against each other and find the best of them all.

I hope you’ve got many many insights and new skills for yourself.

Now it’s up to you. Be creative, use your new knowledge and build something amazing!

Good luck & happy coding! :)

That's it for the last part of this tutorial series. I hope it was useful for you. For more tutorials and online courses, 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.

*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/).