**More Than Just CRUD with .NET Core 3.1**

# More Than Just CRUD with .NET Core 3.1

## **Introduction**

It is time to let our RPG characters fight.

In the upcoming lectures, we will build a service with functions to let our characters attack another character with a weapon or with one of their skills.

Additionally, we implement a function to let characters fight by themselves until one of them, well, has no hit points anymore.

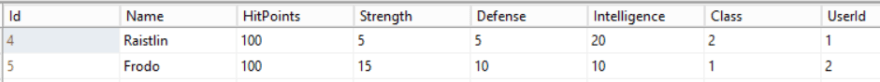
This will result in victories and defeats which is some kind of highscore we can use to display our RPG characters in a certain order.

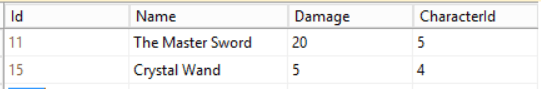
So, with no further ado, let the games begin!

## **Prepare to Fight!**

While you weren’t watching, I added some relations between characters, weapons, and skills so that we have characters available to fight against each other. It’s not necessary, but maybe prepare one or two characters yourself so that you can use them for the upcoming implementations.

For now, we’ve got Frodo with the Master Sword and the skill Frenzy, and Raistlin with the Crystal Wand and the skills Fireball and Blizzard. I’m really curious about how they will compete.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Tk6k_1-K--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/r3ba6omlsyez97a7wejf.png)

[](https://res.cloudinary.com/practicaldev/image/fetch/s--faxxP1Y---/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qj2wt578lgln53g8msbx.png)

[](https://res.cloudinary.com/practicaldev/image/fetch/s--YpzvviW7--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/n5325kp3dmqxmqk3v6sy.png)

Anyways, to count the upcoming fights, victories, and defeats, let’s add exactly these three properties as int 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;

public User User { get; set; }

public Weapon Weapon { get; set; }

public List<CharacterSkill> CharacterSkills { get; set; }

public int Fights { get; set; }

public int Victories { get; set; }

public int Defeats { get; set; }

}

We can also add them to the GetCharacterDto.

public class GetCharacterDto

{

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;

public GetWeaponDto Weapon { get; set; }

public List<GetSkillDto> Skills { get; set; }

public int Fights { get; set; }

public int Victories { get; set; }

public int Defeats { get; set; }

}

Now let’s add these properties to the database. First, we add a new migration with dotnet ef migrations add FightProperties.

You see that the migration will add three new columns to the Characters table with 0 as the default value. Exactly what we need.

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.AddColumn<int>(

name: "Defeats",

table: "Characters",

nullable: false,

defaultValue: 0);

migrationBuilder.AddColumn<int>(

name: "Fights",

table: "Characters",

nullable: false,

defaultValue: 0);

migrationBuilder.AddColumn<int>(

name: "Victories",

table: "Characters",

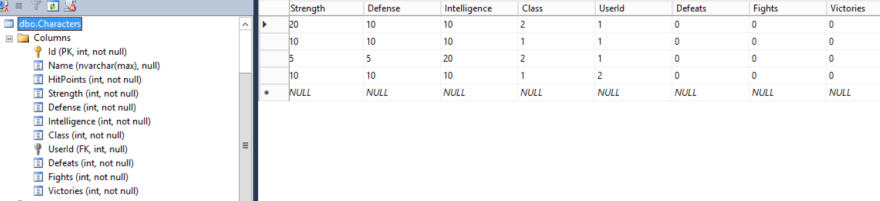
nullable: false,

defaultValue: 0);

}

Let’s update the database with dotnet ef database update.

After the update, we can see the new columns in the SQL Server Management Studio.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--IwsQjshw--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/fqkpnj5hwxnemswkhbw2.png)

Next would already be a FightService with the corresponding interface and controller.

We create a new folder FightService and add the C# classes IFightService and FightService which implements the interface.

namespace dotnet\_rpg.Services.FightService

{

public class FightService : IFightService

{

}

}

After that, we create the FightController for the web service calls.

We add the ControllerBase class and the [Route()] and [ApiController] attribute but leave out authentication. Of course, it would make sense to add authentication here, but I think you know how to implement it now and for this example, it’s sufficient to just let the RPG characters fight without the permission of their respective owners.

using Microsoft.AspNetCore.Mvc;

namespace dotnet\_rpg.Controllers

{

[ApiController]

[Route("[controller]")]

public class FightController : ControllerBase

{

}

}

We can also already add a constructor to the FightController which injects the FightService.

public class FightController : ControllerBase

{

private readonly IFightService \_fightService;

public FightController(IFightService fightService)

{

\_fightService = fightService;

}

}

Alright, and finally we register the FightService in the Startup.cs file with services.AddScoped().

services.AddScoped<IFightService, FightService>();

Great! So far the preparations. Coming up next, we implement a way to attack a character with a weapon.

## **Attack with Weapons**

The idea behind attacking other characters with a weapon is that we define an attacker and an opponent and use the weapon of the attacker (remember there can only be one weapon) to fight against the opponent. Pretty simple actually.

To start, we need some DTOs for the request and the result. So, let’s create a Fight folder and the C# class WeaponAttackDto which only consists of the AttackerId and the OpponentId.

namespace dotnet\_rpg.Dtos.Fight

{

public class WeaponAttackDto

{

public int AttackerId { get; set; }

public int OpponentId { get; set; }

}

}

Since the attacker can only have one weapon, we only need to know who the attacker is. No need for a weaponId.

For the response, we create a new C# class called AttackResultDto. We can put anything we want in there. For me, it’s interesting to know the names of the attacker and the opponent, their resulting hit points and the damage that was taken.

namespace dotnet\_rpg.Dtos.Fight

{

public class AttackResultDto

{

public string Attacker { get; set; }

public string Opponent { get; set; }

public int AttackerHP { get; set; }

public int OpponentHP { get; set; }

public int Damage { get; set; }

}

}

Alright, that’s it for the DTOs. Next, we add a constructor to the FightService class where we only inject the DataContext to access the Characters of the database.

A short side note about that. In a minute we will get certain characters by their Id. We have already implemented that in the CharacterService. So we could actually inject the ICharacterService and use the method GetCharacterById() here. But we would have to use an authenticated user for that which would take a bit more effort during the tests and since this example is supposed to be a bit smaller, we just use the DataContext directly again.

But, don’t get me wrong, please do it that way, if you want to. It’s totally fine if you want to extend this example. I would recommend doing so, because it’s a great exercise, hence a great way to learn.

Anyways, let’s move on with the IFightService interface.

We only need one method. This method returns a ServiceResponse with an AttackResultDto, we call it WeaponAttack() and pass a WeaponAttackDto as request.

using System.Threading.Tasks;

using dotnet\_rpg.Dtos.Fight;

using dotnet\_rpg.Models;

namespace dotnet\_rpg.Services.FightService

{

public interface IFightService

{

Task<ServiceResponse<AttackResultDto>> WeaponAttack(WeaponAttackDto request);

}

}

Alright, let’s use the automatic implementation of the interface in the FightService real quick, add the async keyword and then we jump to the FightController before we bring the WeaponAttack() method to life.

public async Task<ServiceResponse<AttackResultDto>> WeaponAttack(WeaponAttackDto request)

{

throw new NotImplementedException();

}

The implementation of the FightController is straightforward.

Similar to the other controllers we have a public async method returning a Task with an IActionResult. The method is called WeaponAttack() and receives a WeaponAttackDto as request. In the body, we return the result of the awaited \_fightService.WeaponAttack() method. Last but not least, don’t forget the [HttpPost] attribute with ”Weapon” for the route. Done.

[HttpPost("Weapon")]

public async Task<IActionResult> WeaponAttack(WeaponAttackDto request)

{

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

}

Now we get to the implementation of the WeaponAttack() method of the FightService.

We already did it before, we start with initializing the ServiceResponse, create a try/catch block and send a failing response back in case of an exception.

public async Task<ServiceResponse<AttackResultDto>> WeaponAttack(WeaponAttackDto request)

{

ServiceResponse<AttackResultDto> response = new ServiceResponse<AttackResultDto>();

try

{

}

catch (Exception ex)

{

response.Success = false;

response.Message = ex.Message;

}

return response;

}

Now, we need the Characters. That’s what I was talking about when I mentioned the sidenote. You could use the CharacterService to receive the attacker or access the \_context.Characters directly, only include the Weapon and, of course, find the one Character whose Id matches the request.AttackerId.

Regarding the opponent, we don’t have to include anything, we just need the one Character with the matching request.OpponentId. Be careful, if you want to use the CharacterService here. If the opponent is not a character of the authenticated user, it won’t work, because the GetCharacterById() function only gives you the RPG characters that are related to the user. So, you would have to write another method that doesn’t care about that relation.

Character attacker = await \_context.Characters

.Include(c => c.Weapon)

.FirstOrDefaultAsync(c => c.Id == request.AttackerId);

Character opponent = await \_context.Characters

.FirstOrDefaultAsync(c => c.Id == request.OpponentId);

We’ve got the fighting characters, great. Now comes the fun part, we calculate the damage. Again, feel free to be creative and use your own formula.

My ingenious formula takes the Damage of the Weapon and adds a random value between 0 and the Strength of the attacker. After that, a random value between 0 and the Defense of the opponent will be subtracted from the damage value. Crazy, huh?

If the damage is above 0 - yes, it could be below if the attacker is not very strong or the opponent has a tremendous armor - we subtract the damage from the HitPoints of the opponent.

And if the HitPoints are below or equal 0, we could return a Message like $"{opponent.Name} has been defeated!";.

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

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

if (damage > 0)

opponent.HitPoints -= (int)damage;

if (opponent.HitPoints <= 0)

response.Message = $"{opponent.Name} has been defeated!";

After that, we have to make sure to Update() the opponent in the database and also save these changes with SaveChangesAsync().

\_context.Characters.Update(opponent);

await \_context.SaveChangesAsync();

Finally, we write the response.Data. We simply set the Name and the HitPoints of the attacker and the opponent as well as the damage.

response.Data = new AttackResultDto

{

Attacker = attacker.Name,

AttackerHP = attacker.HitPoints,

Opponent = opponent.Name,

OpponentHP = opponent.HitPoints,

Damage = damage

};

That’s it.

public async Task<ServiceResponse<AttackResultDto>> WeaponAttack(WeaponAttackDto request)

{

ServiceResponse<AttackResultDto> response = new ServiceResponse<AttackResultDto>();

try

{

Character attacker = await \_context.Characters

.Include(c => c.Weapon)

.FirstOrDefaultAsync(c => c.Id == request.AttackerId);

Character opponent = await \_context.Characters

.FirstOrDefaultAsync(c => c.Id == request.OpponentId);

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

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

if (damage > 0)

opponent.HitPoints -= (int)damage;

if (opponent.HitPoints <= 0)

response.Message = $"{opponent.Name} has been defeated!";

\_context.Characters.Update(opponent);

await \_context.SaveChangesAsync();

response.Data = new AttackResultDto

{

Attacker = attacker.Name,

AttackerHP = attacker.HitPoints,

Opponent = opponent.Name,

OpponentHP = opponent.HitPoints,

Damage = damage

};

}

catch (Exception ex)

{

response.Success = false;

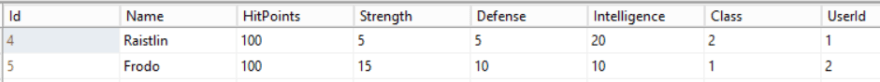
response.Message = ex.Message;

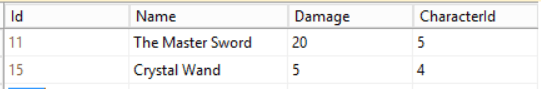
}

return response;

}

To test this, we’ve got Frodo and Raistlin using the Master Sword and a Crystal Wand.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Tk6k_1-K--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/r3ba6omlsyez97a7wejf.png)

[](https://res.cloudinary.com/practicaldev/image/fetch/s--faxxP1Y---/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qj2wt578lgln53g8msbx.png)

You see, Frodo has better values, but maybe Raistlin has better chances to win throwing some fireballs later.

In Postman we use the URL http://localhost:5000/fight/weapon with POST as the HTTP method. The body consists of the attackerId and the opponentId.

{

"attackerid" : 5,

"opponentid" : 4

}

Make sure you have started the application with dotnet watch run and then let’s attack!

{

"data": {

"attacker": "Frodo",

"opponent": "Raistlin",

"attackerHP": 100,

"opponentHP": 71,

"damage": 29

},

"success": true,

"message": null

}

It’s pretty obvious, Frodo is stronger.

If we let Raistlin attack by switching the Ids, we see that the Crystal Wand can’t do much.

{

"data": {

"attacker": "Raistlin",

"opponent": "Frodo",

"attackerHP": 71,

"opponentHP": 97,

"damage": 3

},

"success": true,

"message": null

}

If we let Frodo attack a couple more times, we see that Raistlin is defeated.

{

"data": {

"attacker": "Frodo",

"opponent": "Raistlin",

"attackerHP": 97,

"opponentHP": -18,

"damage": 31

},

"success": true,

"message": "Raistlin has been defeated!"

}

Alright, that was fun! Let’s implement “The Return of the Mage” now by using some skill attacks.

## **Attack with Skills**

First, we add a new DTO called SkillAttackDto with three int members, the AttackerId, the OpponentId and this time also the SkillId. Since a Character can have multiple skills, we want to specify which skill should be used for the attack.

namespace dotnet\_rpg.Dtos.Fight

{

public class SkillAttackDto

{

public int AttackerId { get; set; }

public int OpponentId { get; set; }

public int SkillId { get; set; }

}

}

Then we need a new method, of course. In the IFightService interface we add the method SkillAttack() with the SkillAttackDto as request this time.

Task<ServiceResponse<AttackResultDto>> SkillAttack(SkillAttackDto request);

We’ll cover the FightService in a second. Let’s deal with the FightController instead real quick. You already know it. We can just copy the WeaponAttack() method and make a few little changes.

The method is called SkillAttack(), of course, again with the SkillAttackDto as request, we call the corresponding method of the \_fightService and change the route to ”Skill”.

[HttpPost("Skill")]

public async Task<IActionResult> SkillAttack(SkillAttackDto request)

{

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

}

Regarding the FightService we can copy the WeaponAttack() method as well and make some changes to the new method. First the name and request type again, of course.

Receiving the attacker looks a bit different now. Instead of including the Weapon, we include the CharacterSkills and then the Skill of each CharacterSkill. Again, here’s an opportunity to use the CharacterService. Receiving the opponent stays the same.

Character attacker = await \_context.Characters

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

.FirstOrDefaultAsync(c => c.Id == request.AttackerId);

Character opponent = await \_context.Characters

.FirstOrDefaultAsync(c => c.Id == request.OpponentId);

Now comes something new, we have to get the correct Skill. Since we included the skills of the RPG character, we don’t access the skills via the \_context, we just look if the attacker really has that specific skill.

We initialize a new CharacterSkill object and look through the attacker.CharacterSkills to find the one where the Skill.Id of the CharacterSkill equals the request.SkillId.

If we don’t find one, meaning the characterSkill object is null, because the user requested a wrong SkillId, we return a failing response with a message like {attacker.Name} doesn't know that skill..

CharacterSkill characterSkill =

attacker.CharacterSkills.FirstOrDefault(cs => cs.Skill.Id == request.SkillId);

if (characterSkill == null)

{

response.Success = false;

response.Message = $"{attacker.Name} doesn't know that skill.";

return response;

}

Regarding the damage we go really crazy. We take the same formula but use characterSkill.Skill.Damage, of course, and replace the Strength with the Intelligence of the attacker.

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

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

if (damage > 0)

opponent.HitPoints -= (int)damage;

if (opponent.HitPoints <= 0)

response.Message = $"{opponent.Name} has been defeated!";

And that’s already it.

public async Task<ServiceResponse<AttackResultDto>> SkillAttack(SkillAttackDto request)

{

ServiceResponse<AttackResultDto> response = new ServiceResponse<AttackResultDto>();

try

{

Character attacker = await \_context.Characters

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

.FirstOrDefaultAsync(c => c.Id == request.AttackerId);

Character opponent = await \_context.Characters

.FirstOrDefaultAsync(c => c.Id == request.OpponentId);

CharacterSkill characterSkill =

attacker.CharacterSkills.FirstOrDefault(cs => cs.Skill.Id == request.SkillId);

if (characterSkill == null)

{

response.Success = false;

response.Message = $"{attacker.Name} doesn't know that skill.";

return response;

}

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

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

if (damage > 0)

opponent.HitPoints -= (int)damage;

if (opponent.HitPoints <= 0)

response.Message = $"{opponent.Name} has been defeated!";

\_context.Characters.Update(opponent);

await \_context.SaveChangesAsync();

response.Data = new AttackResultDto

{

Attacker = attacker.Name,

AttackerHP = attacker.HitPoints,

Opponent = opponent.Name,

OpponentHP = opponent.HitPoints,

Damage = damage

};

}

catch (Exception ex)

{

response.Success = false;

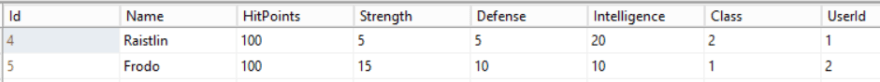
response.Message = ex.Message;

}

return response;

}

Before we test this, make sure to reset the hit points of our opponents.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Tk6k_1-K--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/r3ba6omlsyez97a7wejf.png)

In Postman we use the URL http://localhost:5000/fight/skill this time and add the skillId to the request.

{

"attackerid" : 5,

"opponentid" : 4,

"skillid" : 1

}

When we use a skill the character does not have, we should get our proper error message like Frodo doesn’t know that skill.

{

"data": null,

"success": false,

"message": "Frodo doesn't know that skill."

}

But what happens if Frodo uses the skill Frenzy?

{

"attackerid" : 5,

"opponentid" : 4,

"skillid" : 2

}

He still makes good damage!

{

"data": {

"attacker": "Frodo",

"opponent": "Raistlin",

"attackerHP": 100,

"opponentHP": 77,

"damage": 23

},

"success": true,

"message": null

}

Let’s fight back with Raistlin and his fireballs or even with the mighty Blizzard!

{

"attackerid" : 4,

"opponentid" : 5,

"skillid" : 3

}

{

"data": {

"attacker": "Raistlin",

"opponent": "Frodo",

"attackerHP": 77,

"opponentHP": -16,

"damage": 51

},

"success": true,

"message": "Frodo has been defeated!"

}

That’s what I call a fight! The Blizzard makes a lot of damage and Frodo has been defeated.

Okay, but instead of doing these attacks manually, let’s implement an automatic fight next.

That's it for the 12th 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: More Than Just CRUD with .NET Core 3.1 - Part 2

*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/).