**Many-To-Many Relationship with Entity Framework Core**

# Advanced Relationships with Entity Framework Core (continued)

## **Many-To-Many Relation with Skills**

Implementing many-to-many relations with Entity Framework Core looks a bit different than implementing the other ones.

In our role-playing game example, we add a bunch of skills, that will be available to all characters. This means, there is no upgrading of one specific skill for a character. There’s just a pool of skills that every character can choose from. So, in general, even a knight could throw a fireball and a mage can smash his opponent in a frenzy.

The first thing to do is adding the Skill model, of course.

We create a new C# class and add the properties Id, Name and Damage.

public class Skill

{

public int Id { get; set; }

public string Name { get; set; }

public int Damage { get; set; }

}

Notice, that we don’t add a list of type Character here. We would do that if we wanted to implement a one-to-many relation, but for a many-to-many relationship, we need a special implementation - and that would be a joining table.

Entity Framework Core is currently not able to create a joining table by itself. So we have to add one manually and tell Entity Framework how to join the two entities Skill and Character.

Let’s add the model for this entity first. We create a new C# class and call it CharacterSkill. To join skills and characters now, we have to add them as properties. So, we add a Character and a Skill.

Additionally, we need a primary key for this entity. This will be a composite key of the Skill and the Character. To be able to do that, by convention we add a property CharacterId for the Character, and a property SkillId for the Skill.

public class CharacterSkill

{

public int CharacterId { get; set; }

public Character Character { get; set; }

public int SkillId { get; set; }

public Skill Skill { get; set; }

}

But that’s not the whole magic. We still have to tell Entity Framework Core that we want to use these two Ids as a composite primary key. We do that with the help of the Fluent API.

We’re switching our focus to the DataContext class. First, we add the new DbSet properties Skills and CharacterSkills.

public DbSet<Skill> Skills { get; set; }

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

After that we have to add something new. We override the method OnModelCreating(). This method takes a ModelBuilder argument, which “defines the shape of your entities, the relationships between them and how they map to the database”. Exactly what we need.

The only thing we have to configure here is the composite key of the CharacterSkill entity which consists of the CharacterId and the SkillId. We do that with modelBuilder.Entity<CharacterSkill>().HasKey(cs => new { cs.CharacterId, cs.SkillId });.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<CharacterSkill>()

.HasKey(cs => new { cs.CharacterId, cs.SkillId });

}

That’s it. Thanks to using the naming conventions for the CharacterId and the SkillId we don’t have to configure anything else. Otherwise, we would have to use the Fluent API to configure the relationship between characters and skills with methods like HasOne() and WithMany(). But Entity Framework Core will get this and we can see the correct implementation in a minute in the migration files.

There’s one last thing we have to do and that is adding the CharacterSkill list to the Character and the Skill models.

So, in both C# classes, we add a new property CharacterSkills of type List<CharacterSkill>.

public class Skill

{

public int Id { get; set; }

public string Name { get; set; }

public int Damage { get; set; }

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

}

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; }

}

Alright. When everything is saved, we’re ready to run the migration.

First, we add the new migration with dotnet ef migrations add Skill.

In the created migration file you can see that two new tables will be generated for us, Skills and CharacterSkills.

In the migration design file, a bit further down, you can now see the configuration of the relationship between the joining entity CharacterSkill and the entities Character and Skill.

modelBuilder.Entity("dotnet\_rpg.Models.CharacterSkill", b =>

{

b.HasOne("dotnet\_rpg.Models.Character", "Character")

.WithMany("CharacterSkills")

.HasForeignKey("CharacterId")

.OnDelete(DeleteBehavior.Cascade)

.IsRequired();

b.HasOne("dotnet\_rpg.Models.Skill", "Skill")

.WithMany("CharacterSkills")

.HasForeignKey("SkillId")

.OnDelete(DeleteBehavior.Cascade)

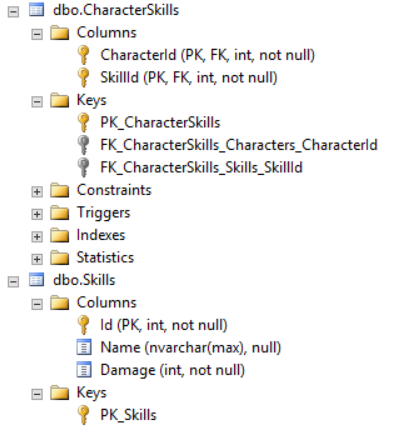
.IsRequired();

});

Again, thanks to using the naming conventions for the Id properties, we don’t have to do this manually.

It’s time to add this migration to the database with dotnet ef database update.

As soon as the update is done, you can refresh the database in SQL Server Management Studio and see the new tables Skills and CharacterSkills with the proper keys.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ff_i1QZM--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/8b8ykfvdsj00s0mm8twu.png)

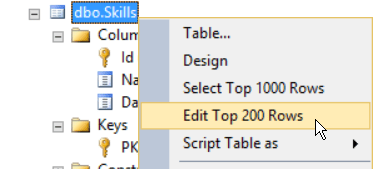
Great! It’s time to fill these tables with some content.

## **Add Skills to RPG Characters**

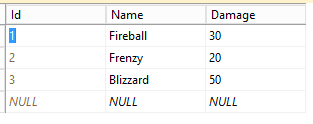
Adding new skills to the pool of skills in the database would work pretty straight forward. We need a service, an interface, the controller and so on. I’d say, we focus more on adding the relation between RPG characters and those skills.

So, instead of adding the service for the skills, let’s add some skills manually in the database with SQL Server Management Studio.

Simply right-click the Skills table and select “Edit Top 200 Rows”.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--_E0mCk9z--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/pjewmq43iqpozfbye590.png)

Now we can add some skills like Fireball, Frenzy or Blizzard.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ieYNOw-2--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/4707cmwkw4u5cartfske.png)

Great. Now we can concentrate on the relations. I can already spoiler, that we will need some DTOs. Let’s create a new folder CharacterSkill and add the new C# class AddCharacterSkillDto with the properties CharacterId and SkillId.

namespace dotnet\_rpg.Dtos.CharacterSkill

{

public class AddCharacterSkillDto

{

public int CharacterId { get; set; }

public int SkillId { get; set; }

}

}

Next, we create the folder Skill and create the DTO GetSkillDto, because we only need that one to display the skills of a character. The properties we need are Name and Damage.

namespace dotnet\_rpg.Dtos.Skill

{

public class GetSkillDto

{

public string Name { get; set; }

public int Damage { get; set; }

}

}

After that, we add one more property to the GetCharacterDto and that would be the Skills of type List<GetSkillsDto>.

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; }

}

Notice that we already access the skills directly, without using the joining entity CharacterSkill first. You’ll see how we realize that in a minute.

Okay, the DTOs are ready, now we can move on to the service and controller files.

We create a new folder called CharacterSkillService and add a new interface called ICharacterSkillService.

We add only one method that will return a ServiceResponse with a GetCharacterDto because similar to the WeaponService we can see the added skills then. We call the method AddCharacterSkill() and give it an AddCharacterSkillDto as a parameter. Of course, while we’re doing that, we have to add some using directives.

using System.Threading.Tasks;

using dotnet\_rpg.Dtos.Character;

using dotnet\_rpg.Dtos.CharacterSkill;

using dotnet\_rpg.Models;

namespace dotnet\_rpg.Services.CharacterSkillService

{

public interface ICharacterSkillService

{

Task<ServiceResponse<GetCharacterDto>> AddCharacterSkill(AddCharacterSkillDto newCharacterSkill);

}

}

Now we can already create the CharacterSkillService. This service will look pretty similar to the WeaponService. We start by implementing the ICharacterSkillService interface and add the AddCharacterSkill() method automatically and add the async keyword already.

public class CharacterSkillService : ICharacterSkillService

{

public async Task<ServiceResponse<GetCharacterDto>> AddCharacterSkill(AddCharacterSkillDto newCharacterSkill)

{

throw new NotImplementedException();

}

}

Before we write the actual code of this method, we add the constructor. Similar to the WeaponService we inject the DataContext, the IHttpContextAccessor and the IMapper. We add the using directives, initialize all fields from the parameters and if you want, add the underscore in front of every field.

private readonly DataContext \_context;

private readonly IHttpContextAccessor \_httpContextAccessor;

private readonly IMapper \_mapper;

public CharacterSkillService(DataContext context, IHttpContextAccessor httpContextAccessor, IMapper mapper)

{

\_mapper = mapper;

\_httpContextAccessor = httpContextAccessor;

\_context = context;

}

Now to the AddCharacterSkill() method.

First, we initialize the returning ServiceResponse and build an empty try/catch block.

In case of an exception, we can already set the Success state of the response to false and set the Message to the exception message.

public async Task<ServiceResponse<GetCharacterDto>> AddCharacterSkill(AddCharacterSkillDto newCharacterSkil

{

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

try

{

}

catch (Exception ex)

{

response.Success = false;

response.Message = ex.Message;

}

return response;

}

Next would be to receive the correct Character from the database that was given by the CharacterId through the AddCharacterSkillDto.

Again, it’s pretty similar to the WeaponService.

First, we access the Characters from the \_context and filter them with the method FirstOrDefaultAsync() by the newCharacterSkill.CharacterId and additionally by the authenticated User. You remember this long line to receive the user id from the claims, right?

Character character = await \_context.Characters

.FirstOrDefaultAsync(c => c.Id == newCharacterSkill.CharacterId &&

c.User.Id == int.Parse(\_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)));

But that’s not all. To receive all skills and also the related Weapon of the user, we have to include them.

We can start with the Weapon. After \_context.Characters we add .Include(c => c.Weapon). The skills are getting a bit more interesting. Again we add .Include(), but first we access the CharacterSkills and after that we access the child property Skill of the CharacterSkills with .ThenInclude().

That way, we get every property from the character that is stored in the database.

Character character = await \_context.Characters

.Include(c => c.Weapon)

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

.FirstOrDefaultAsync(c => c.Id == newCharacterSkill.CharacterId &&

c.User.Id == int.Parse(\_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier)));

With that out of the way, we add the usual null-check. So, if the character is null we set the Success state and the Message and return the response.

Character character = await \_context.Characters

.Include(c => c.Weapon)

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

.FirstOrDefaultAsync(c => c.Id == newCharacterSkill.CharacterId &&

c.User.Id == int.Parse(\_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier))

if (character == null)

{

response.Success = false;

response.Message = "Character not found.";

return response;

}

Next is the Skill. With the given SkillId from the newCharacterSkill parameter, we grab the skill from the database.

Skill skill = await \_context.Skills

.FirstOrDefaultAsync(s => s.Id == newCharacterSkill.SkillId);

Similar to the character, if we cannot find the skill with the given SkillId, we set the ServiceResponse and return it.

if (skill == null)

{

response.Success = false;

response.Message = "Skill not found.";

return response;

}

Now we have everything we need to create a new CharacterSkill.

We initialize a new characterSkill object and set the Character and Skill properties of that object to the character and the skill we got from the database before.

CharacterSkill characterSkill = new CharacterSkill

{

Character = character,

Skill = skill

};

After that, we add this new CharacterSkill to the database with AddAsync(characterSkill), save all changes to the database and finally set the response.Data to the mapped character.

await \_context.CharacterSkills.AddAsync(characterSkill);

await \_context.SaveChangesAsync();

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

And that’s the whole AddCharacterSkill() method.

public async Task<ServiceResponse<GetCharacterDto>> AddCharacterSkill(AddCharacterSkillDto newCharacterSkill)

{

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

try

{

Character character = await \_context.Characters

.Include(c => c.Weapon)

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

.FirstOrDefaultAsync(c => c.Id == newCharacterSkill.CharacterId &&

c.User.Id == int.Parse(\_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier))

if (character == null)

{

response.Success = false;

response.Message = "Character not found.";

return response;

}

Skill skill = await \_context.Skills

.FirstOrDefaultAsync(s => s.Id == newCharacterSkill.SkillId);

if (skill == null)

{

response.Success = false;

response.Message = "Skill not found.";

return response;

}

CharacterSkill characterSkill = new CharacterSkill

{

Character = character,

Skill = skill

};

await \_context.CharacterSkills.AddAsync(characterSkill);

await \_context.SaveChangesAsync();

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

}

catch (Exception ex)

{

response.Success = false;

response.Message = ex.Message;

}

return response;

}

To be able to call the service, we need the CharacterSkillController, so let’s create this new C# file.

As always, we derive from ControllerBase and add the attributes [Route(“[controller]”)], [ApiController] and [Authorize]. We need the user information, hence this controller should only be accessed by authenticated users.

[Authorize]

[ApiController]

[Route("[controller]")]

public class CharacterSkillController : ControllerBase

Then we need a constructor that only injects the ICharacterSkillService.

private readonly ICharacterSkillService \_characterSkillService;

public CharacterSkillController(ICharacterSkillService characterSkillService)

{

\_characterSkillService = characterSkillService;

}

And finally we add the public async POST method AddCharacterSkill() with an AddCharacterSkillDto as parameter which is passed to the AddCharacterSkill() method of the \_characterSkillService.

[HttpPost]

public async Task<IActionResult> AddCharacterSkill(AddCharacterSkillDto newCharacterSkill)

{

return Ok(await \_characterSkillService.AddCharacterSkill(newCharacterSkill));

}

So far the controller.

Now we register the new service in the Startup.cs file. As almost always, we use services.AddScoped() for that in the ConfigureServices() method.

services.AddScoped<ICharacterSkillService, CharacterSkillService>();

The last thing is a change to the AutoMapperProfile.

The easy part is a new map for the GetSkillDto.

CreateMap<Skill, GetSkillDto>();

Now it’s getting more interesting. I already told you, that we want to access the skills of a character directly, without displaying the joining entity CharacterSkill. We can do that with the help of AutoMapper and the help of the Select() function.

First we utilize the ForMember() function for the <Character, GetCharacterDto>-Map. With this function, we can define a special mapping for a specific member of the mapped type.

In our case, we properly want to set the Skills of the DTO.

To do that, we access the Character object and from that object - hence the function MapFrom() - we grab the CharacterSkills and select the Skill from every CharacterSkill.

CreateMap<Character, GetCharacterDto>()

.ForMember(dto => dto.Skills, c => c.MapFrom(c => c.CharacterSkills.Select(cs => cs.Skill)));

That’s how we make the jump to the skills directly.

Great! It’s time to test this.

Make sure to have your user logged in and the correct token in place. Then we can use the URL http://localhost:5000/characterskill with the HTTP method POST. The body of the call consists of the characterId and the skillId.

{

"characterid" : 5,

"skillid" : 1

}

Executing this call, we get the complete RPG character back with its weapon and the new skill.

{

"data": {

"id": 5,

"name": "Frodo",

"hitPoints": 200,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1,

"weapon": {

"name": "The Master Sword",

"damage": 10

},

"skills": [

{

"name": "Fireball",

"damage": 30

}

]

},

"success": true,

"message": null

}

When we add another skill, we see the complete array of skills.

{

"data": {

"id": 5,

"name": "Frodo",

"hitPoints": 200,

"strength": 10,

"defense": 10,

"intelligence": 10,

"class": 1,

"weapon": {

"name": "The Master Sword",

"damage": 10

},

"skills": [

{

"name": "Fireball",

"damage": 30

},

{

"name": "Frenzy",

"damage": 20

}

]

},

"success": true,

"message": null

}

In the database, you can also see that the joining table is filled with the new IDs.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--lN9mhiyi--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qwg1lb20sa8j214pjrkh.png)

Perfect! The RPG character is equipped with a weapon and skills.

Feel free to play around with this.

If you want to use the GetCharacterById() method in the CharacterService to see the equipment of any character, make sure to add the Include() method as shown before, meaning include the Weapon as well as the Skills of the CharacterSkills.

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

{

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

Character dbCharacter =

await \_context.Characters

.Include(c => c.Weapon)

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

.FirstOrDefaultAsync(c => c.Id == id && c.User.Id == GetUserId());

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

return serviceResponse;

}

When all your RPG characters are set, I guess it’s time to fight!

## **Summary**

Congrats! You implemented all types of relationships into your application.

But that’s not all.

In this chapter, first, you learned how to grab the authenticated user from a web service call and receive proper data based on that user from the database. That way you were able to show every user her own characters.

After that, we covered a one-to-one relationship. An RPG character can now be equipped with one weapon and only that one single weapon.

Regarding the many-to-many relationship, we added skills to our characters together with the necessary joining entity or table CharacterSkills. Characters are allowed to have several skills and skills can have several characters.

Apart from that you created all the necessary services and controllers to add weapons and skills and you learned how to include deeper nested entities and how to define custom mappings with AutoMapper.

In the next chapter, we will go one step further and implement functions to let the RPG characters fight against each other.

That's it for the 11th 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

*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/).