**SQLite & Data Seeding with Entity Framework Core**

# SQLite & Data Seeding with Entity Framework Core

## **Introduction**

If you don’t want to use SQL Server for your web application, you might want to choose SQLite.

Many students want to use a database that is cross-platform, lightweight, and usable in many different scenarios such as smartphones, for instance. In that case, SQLite is a popular choice.

So, it was time to add this chapter with example implementations for SQLite.

We will configure the application for SQLite, start the migrations from scratch and use the DB Browser for SQLite to have a look at the database.

Additionally, you will learn how to seed the database. This means, you don’t have to add users, characters, skills, and so on manually. You will learn how to do this with code. Very convenient and time-saving.

Let’s start with SQLite first.

## **New Package, ConnectionString & Configuration**

The very first thing we have to do to be able to use a SQLite database is adding a new package, which would be Microsoft.EntityFrameworkCore.Sqlite.

Let’s do that in the terminal with the command dotnet add package Microsoft.EntityFrameworkCore.Sqlite.

After the installation, you should see the change in your project file.

<PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="3.1.8" />

After that, we have to change or add the connection string for the SQLite database. Since this database is a file, the connection string is quite simple.

In the appsettings.json file we add a new string, like SqliteConnection, and set it to Data Source=dotnetrpg.db. Done.

"ConnectionStrings": {

"DefaultConnection": "Server=localhost\\SQLEXPRESS; Database=dotnet-rpg; Trusted\_Connection=true;",

"SqliteConnection" : "Data Source=dotnetrpg.db"

},

As soon as we add our migrations and update the database for the first time, Entity Framework will create this exact file.

Finally, we make a change to the ConfigureServices() method in the Startup.cs. Instead of SQL Server, we’re now using SQLite and the new connection string.

services.AddDbContext<DataContext>(x => x.UseSqlite(Configuration.GetConnectionString("SqliteConnection")));

Alright. We’re ready to create the SQLite database.

## **Create the SQLite Database**

Since the DataContext is now configured to use a SQLite database, the existing migration files won’t work.

The easiest way to create the SQLite database with code-first migration is to delete the migrations folder and add a new initial migration.

As soon as you have removed the folder with all migration files, add a migration again with dotnet ef migrations add Initial.

The generated files should look familiar. Tables will be created, keys will be set, and so on.

We create the database file with dotnet ef database update.

Now we need a tool to open the file and browse through the database.

## **Use the DB Browser for SQLite**

Probably the most used tool to create, design, and edit database files compatible with SQLite would be the DB Browser for SQLite. You can download it on [sqlitebrowser.org](https://sqlitebrowser.org/) and it is available for Windows, macOS, and Linux. You see, we really are cross-platform here.

So please download the DB Browser and install it. There is also a portable version available.

After the installation, please open the database file of our web application which can be found in the root directory of the project.

Please make yourself a bit familiar with the DB Browser.

You find the database structure with all the tables in the first tab. To see the content of a table you can select the second tab “Browse Data” and select the table you want to see.

When you add, edit or delete a record, make sure to click the “Write Changes” button in the top menu to make your changes persistent.

That’s it already. Any web service call works as before but now stores the data in the SQLite database - thanks to Entity Framework.

But before you start adding all the data manually in the DB Browser, let’s have a look at how to seed the database programmatically.

## **Data Seeding Preparations**

The great thing about data seeding is that you configure your seed or test data once and - in a perfect world - never have to worry about adding data to your database before testing any features.

In our case, we want to seed users with a character, and these RPG characters should already have skills and a weapon.

All that is done in our DataContext class. But before we can start, we have to make some small changes to our code.

First, we go to the Character class and add another property, the UserId. This is necessary because we have to define this foreign key when seeding the data. Setting a User object is not necessary and wouldn’t work. We have to set the Id of the User.

The next thing to modify is the CreatePasswordHash() method of the AuthRepository. To be more specific, we move this method to a static Utility class.

The reason for that is that we want to seed users, but can’t inject the AuthRepository into the DataContext to call the Register() method, for instance.

When seeding users though, we have to set every property. This means even the Id, although it is automatically incremented by the database when inserting a new user, and also the password hash.

And because I don’t want to set a byte array manually, I would like to use the CreatePasswordHash() method. You will see how to seed users in a minute. For now, let’s just make the change.

We can cut the CreatePasswordHash() method from the AuthRepository, create a static Utility class in the Data folder, paste the method and also make it public static.

namespace dotnet\_rpg.Data

{

public static class Utility

{

public static void CreatePasswordHash(string password, out byte[] passwordHash, out byte[] passwordSalt)

{

using (var hmac = new System.Security.Cryptography.HMACSHA512())

{

passwordSalt = hmac.Key;

passwordHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));

}

}

}

}

In the Register() method of the AuthRepository we now have to use this new class and that’s it.

Utility.CreatePasswordHash(password, out byte[] passwordHash, out byte[] passwordSalt);

These are all the preparations. Next, we start the data seeding with skills.

## **Seeding Skills**

As mentioned before, the whole data seeding happens in the DataContext. To be more specific, we’re using the OnModelCreating() method. We have already defined a key and a default value here. Next is the seed data.

In this case, we also configure a certain entity and then call the method HasData() and give it some, well... data.

Let’s start with skills, because they have no relations, hence no other entities need to be seeded before we can add skills.

modelBuilder.Entity<Skill>().HasData(

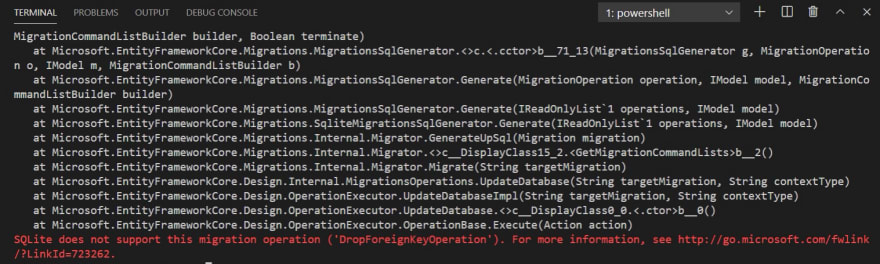
new Skill { Id = 1, Name = "Fireball", Damage = 30 },

new Skill { Id = 2, Name = "Frenzy", Damage = 20 },

new Skill { Id = 3, Name = "Blizzard", Damage = 50});

To test that, we add a new migration with dotnet ef migrations add SkillSeeding and then update the database with dotnet ef database update, but I have to warn you, this will most likely not work.

You’re probably getting the following error when you want to update the database.

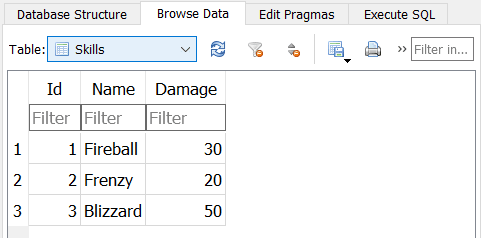
[](https://res.cloudinary.com/practicaldev/image/fetch/s--GYUaXHfD--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/hh9n2ubj6u7gv93estax.png)

You see, using SQLite with Entity Framework Core is different than using SQL Server. The SQLite provider has some migrations limitations. Most of these limitations are a result of limitations in the underlying SQLite database engine and are not specific to Entity Framework. That’s what the official documentation says.

So, the DropForeignKey operation does not work with .NET Core 3.1.

Since we created the database from scratch anyways, we can delete it again, delete the migrations folder as well and then add a new initial migration. This time, the database update should work.

After that, we can see the skills in the Skills table with the DB Browser.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--TQtcFl_y--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/kz198rvq23hdcdothx27.png)

Beautiful.

## **Seeding Entities with Relations**

Now comes the rest. We create Users, Characters, Weapons, and CharacterSkills. We don’t have to configure these entities in a specific order but we have to pay attention to the foreign keys. Characters are related to Users, Weapons and CharacterSkills are related to Characters (and Skills).

Let’s start with the Users. The code is similar to the seeding of the Skills and thanks to our preparations, we can use the CreatePasswordHash() method first to get the proper hash and salt values. And yes, since this is test data, we can use the same hash and salt values for several users.

Utility.CreatePasswordHash("123456", out byte[] passwordHash, out byte[] passwordSalt);

modelBuilder.Entity<User>().HasData(

new User { Id = 1, PasswordHash = passwordHash, PasswordSalt = passwordSalt, Username = "User1" },

new User { Id = 2, PasswordHash = passwordHash, PasswordSalt = passwordSalt, Username = "User2" }

);

Now we can seed Characters and set the proper values.

modelBuilder.Entity<Character>().HasData

new Character {

Id = 1,

Name = "Frodo",

Class = RpgClass.Knight,

HitPoints = 100,

Strength = 15,

Defense = 10,

Intelligence = 10,

UserId = 1 },

new Character {

Id = 2,

Name = "Raistlin",

Class = RpgClass.Mage,

HitPoints = 100,

Strength = 5,

Defense = 5,

Intelligence = 20,

UserId = 2 }

);

And now we can do the same for Weapons and CharacterSkills.

modelBuilder.Entity<Weapon>().HasData(

new Weapon { Id = 1, Name = "The Master Sword", Damage = 20, CharacterId = 1 },

new Weapon { Id = 2, Name = "Crystal Wand", Damage = 5, CharacterId = 2 }

);

modelBuilder.Entity<CharacterSkill>().HasData(

new CharacterSkill { CharacterId = 1, SkillId = 2 },

new CharacterSkill { CharacterId = 2, SkillId = 1 },

new CharacterSkill { CharacterId = 2, SkillId = 3 }

);

The last step is a new migration again. So we enter dotnet ef migrations add FinalSeeding and then update the database with dotnet ef database update.

After the update, we can admire our seed data in the DB Browser. Go for it!

And we can already test our data with a fight. Shall we?

Start the app with dotnet watch run, fire up Postman, and use the URL http://localhost:5000/fight for a POST request.

The body should be an array with the ids 1 and 2.

{

"characterIds" : [1,2]

}

Hit “Send” and we get a beautiful fighting log.

{

"data": {

"log": [

"Frodo attacks Raistlin using The Master Sword with 27 damage.",

"Raistlin attacks Frodo using Fireball with 37 damage.",

"Frodo attacks Raistlin using Frenzy with 20 damage.",

"Raistlin attacks Frodo using Crystal Wand with 0 damage.",

"Frodo attacks Raistlin using Frenzy with 29 damage.",

"Raistlin attacks Frodo using Blizzard with 59 damage.",

"Frodo attacks Raistlin using The Master Sword with 23 damage.",

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

"Frodo has been defeated!",

"Raistlin wins with 1 HP left!"

]

},

"success": true,

"message": null

}

I hope this was helpful. If you have any questions or more feature requests, feel free to contact me. And as always, the code is available on [GitHub](https://github.com/patrickgod/dotnet-rpg).

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/).