

Table of Contents

Introduction.....	2
Install NuGet packages.....	3
DbContext class and access via Controller.....	3
Register the DbContext service.....	6
Working with db file using DB Browser for SQLite.....	8

Demo: Shop Database, Dbcontext

Author: Baifan Zhou

Some explanations are created with the help of ChatGPT.

Introduction

This demo guides you through integrating an SQLite database into your ASP.NET Core MVC application using Entity Framework Core. You will learn how to install necessary NuGet packages, set up a `DbContext` class for database access, register the `DbContext` service in your application, and interact with the database using a popular SQLite tool.

Key Steps Covered:

- **Install NuGet Packages:**
 - We start by installing `Microsoft.EntityFrameworkCore.Sqlite` to enable SQLite database support in your project. This package ensures compatibility cross Windows, Mac, and Linux, and provides the necessary libraries to work with SQLite databases.
- **DbContext Class and Access via Controller:**
 - Create and configure the `ItemDbContext` class to manage database operations. This class inherits from `DbContext` and sets up the database schema. You'll modify the `ItemController` to use this context for data operations, replacing any mock database methods with real database queries.
- **Register the DbContext Service:**
 - Register the `DbContext` service in `Program.cs` to make it available throughout the application. This includes configuring the connection string in `appsettings.json`, which allows you to change the database connection without altering your application code.
- **Working with the Database File Using DB Browser for SQLite:**
 - Use DB Browser for SQLite to manually inspect and modify the database. Learn how to add data to your database and view its structure. This tool helps verify that your database setup is working as expected and lets you interact with your data directly.

By the end of this demo, you will have your ASP.NET Core MVC application connected to an SQLite database. Happy coding!

Install NuGet packages

In VSCode, open your project folder. Then open a terminal, and run the command

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version="8.*"
```

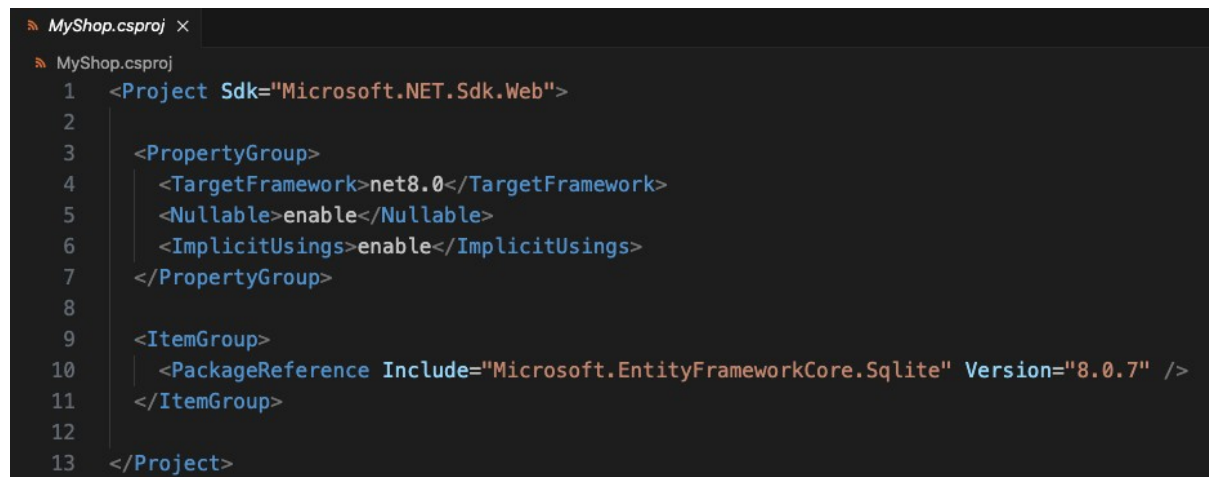
(I use Sqlite to ensure cross-platform compatibility. Windows can also use local db server.)

Note: the command needs to be run in the project folder. In my case, it is the MyShop folder. The `--version="8.*"` is to ensure version compatibility with .net 8.0. You can also specify more concrete versions.

```
baifanz@Baifans-MacBook-Pro MyShop % pwd
/Users/baifanz/ProgrammingProjects/2024ITPE3200Programming/ITPE3200-24H/3-EntityFramework/Demo-EntityFramework-1/MyShop
```

Note: Installing packages (dependencies) with `dotnet add package` for .NET projects and using `npm install` for Node.js projects are different due to the distinct ecosystems and tools they serve. .NET packages are managed by NuGet (nuget.org), using `*.csproj`. Node.js packages are managed by npm, using `package.json` and `package-lock.json`.

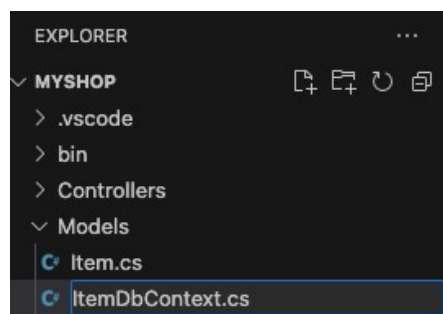
If you open `MyShop.csproj`, you will see that the new package is registered (Line 10).



```
1 <Project Sdk="Microsoft.NET.Sdk.Web">
2
3   <PropertyGroup>
4     <TargetFramework>net8.0</TargetFramework>
5     <Nullable>enable</Nullable>
6     <ImplicitUsings>enable</ImplicitUsings>
7   </PropertyGroup>
8
9   <ItemGroup>
10    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.7" />
11  </ItemGroup>
12
13 </Project>
```

DbContext class and access via Controller

Add new class named `ItemDbContext.cs` under Models



Note: in large web applications, you may have multiple databases for managing different data. In our demos, we only have shop item management, including shop items, orders, users, etc. They are all managed via `ItemDbContext.cs`

Modify the code of `Models/ItemDbContext.cs` to this:

```

1  using Microsoft.EntityFrameworkCore;
2
3  namespace MyShop.Models;
4
5  2 references
6  public class ItemDbContext : DbContext
7  {
8      0 references
9      public ItemDbContext(DbContextOptions<ItemDbContext> options) : base(options)
10     {
11         Database.EnsureCreated();
12     }
13     0 references
14     public DbSet<Item> Items { get; set; }
15 }

```

Line6 defines that the class ItemDbContext inherits from DbContext.

Line 6 – Line14 is the Constructor.


Line10 creates an empty database in case it does not exist.

Detailed explanations:

- Namespace and Imports:
 - `using Microsoft.EntityFrameworkCore;` imports Entity Framework Core functionality needed for database operations.
 - `namespace MyShop.Models;` defines the namespace for the class, organizing it within the `MyShop.Models` directory. In programming, a **namespace** is a container that holds a logical grouping of related classes, interfaces, structs, enums, and delegates. It helps to organize code, avoid name conflicts, and improve code maintainability.
- Class Definition:
 - `public class ItemDbContext : DbContext` defines the `ItemDbContext` class, which inherits from `DbContext`. This class represents the database context used by Entity Framework Core. A `DbContext` is a class that represents a session (a temporary, interactive information interchange) with the database. It is used to query and save data to the database and is typically derived from the `DbContext` base class provided by Entity Framework.
- Constructor:
 - `public ItemDbContext(DbContextOptions<ItemDbContext> options) : base(options);`
 - This constructor accepts `DbContextOptions<ItemDbContext>` as a parameter and passes it to the base class `DbContext`. This is used to configure the context, such as the database connection string.
 - `Database.EnsureCreated();` ensures that the database is created.
 - It checks whether the database associated with the current `DbContext` already exists. If it does not exist, it creates the database along with its schema (tables, indexes, etc.) based on the current model defined in the `DbContext`.
 - This method is typically used during the initial setup of an application to ensure that the database schema is created without applying migrations. It's a straightforward way to set up a database schema if the application is in a simple development stage.
- DbSet Property:
 - `public DbSet<Item> Items { get; set; };`

- Defines a `DbSet<Item>` property named `Items`. This property represents the collection of `Item` entities in the database and provides methods for querying and saving instances of `Item`.

To access the database, we need to modify the `ItemController.cs` by adding Line9 – Line 15, and modify the `IActionResults` of `Table()`, `Grid()`, and `Details()` so that they now access the data from the database.



```

1  using Microsoft.AspNetCore.Mvc;
2  using MyShop.Models;
3  using MyShop.ViewModels;
4
5  namespace MyShop.Controllers;
6
7  1 reference
8  public class ItemController : Controller
9  {
10     4 references
11     private readonly ItemDbContext _itemDbContext;
12
13     0 references
14     public ItemController(ItemDbContext itemDbContext)
15     {
16         _itemDbContext = itemDbContext;
17     }
18
19     0 references
20     public IActionResult Table()
21     {
22         List<Item> items = _itemDbContext.Items.ToList();
23         var itemsViewModel = new ItemsViewModel(items, "Table");
24         return View(itemsViewModel);
25     }
26
27     0 references
28     public IActionResult Grid()
29     {
30         List<Item> items = _itemDbContext.Items.ToList();
31         var itemsViewModel = new ItemsViewModel(items, "Grid");
32         return View(itemsViewModel);
33     }
34
35     0 references
36     public IActionResult Details(int id)
37     {
38         List<Item> items = _itemDbContext.Items.ToList();
39         var item = items.FirstOrDefault(i => i.ItemId == id);
40         if (item == null)
41             return NotFound();
42         return View(item);
43     }
44 }

```

Detailed explanations:

- `private readonly ItemDbContext _itemDbContext;`
 - Declares a private read-only field for storing an instance of `ItemDbContext`.
- `public ItemController(ItemDbContext itemDbContext)`
 - Constructor that takes an `ItemDbContext` instance as a parameter and assigns it to the `_itemDbContext` field.
 - This is an example of dependency injection, where the `DbContext` is provided to the controller by the ASP.NET Core framework.
 - Line11 – Line14 is the Constructor of the `ItemController`. It is called when an instance of `ItemController` is created, typically during the handling of an incoming HTTP request (when the Views controlled by `ItemController` are called, namely the `Table`, `Grid`, or `Details`).
- `List<Item> items = _itemDbContext.Items.ToList();`
 - Retrieves all `Item` records from the `Items` table in the database and converts them into a list.

- You can notice that the GetItems() function that served as a mock database is no longer needed and thus commented out.
- Dependency Injection (DI) is a design pattern and technique used in software development to manage dependencies between components in a scalable and maintainable way. It is commonly used in frameworks like ASP.NET Core to achieve a more modular and testable codebase.
 - **Dependency:** An object or service that a class or component needs to function.
 - **Injection:** The process of providing these dependencies to a class or component, rather than having the class create or manage them itself.

Register the DbContext service

To make the DbContext work, we need to register a new service in Program.cs (Line8 – Line11), note that we also need to import EntityFrameworkCore (Line1)

```

Program.cs
1  using Microsoft.EntityFrameworkCore;
2  using MyShop.Models;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  builder.Services.AddControllersWithViews();
7
8  builder.Services.AddDbContext<ItemDbContext>(options => {
9      options.UseSqlite(
10         builder.Configuration["ConnectionStrings:ItemDbContextConnection"]);
11 });
12
13 var app = builder.Build();
14
15 if (app.Environment.IsDevelopment())
16 {
17     app.UseDeveloperExceptionPage();
18 }
19
20 app.UseStaticFiles();
21
22 app.MapDefaultControllerRoute();
23
24 // app.MapControllerRoute(
25 //     name: "default",
26 //     pattern: "{controller=Home}/{action=Index}/{id?}");
27
28 app.Run();
  
```

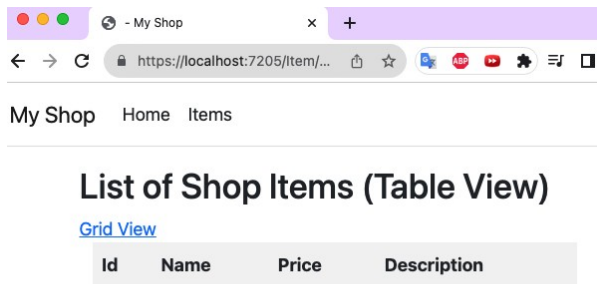
To indicate which DbContext to connect, we add the ConnectionStrings into the appsettings.json (Line9 – Line10):

```

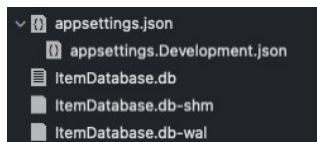
appsettings.json
1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          }
7      },
8      "AllowedHosts": "*",
9      "ConnectionStrings": {
10         "ItemDbContextConnection": "Data Source=ItemDatabase.db"
11     }
12 }
  
```

This practice separates the configuration from the application code. It means you can change the database connection without modifying the code. It also takes advantage of built-in configuration protection features to secure sensitive data (e.g., database name).

Now run the project, and go to the Items page (Table view or Grid view) you should see empty items now, since the data content in ItemDatabase.db does not exist yet.

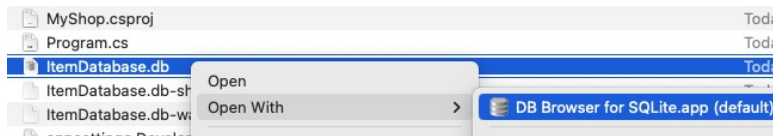


After running the project, the database is created now:



Working with db file using DB Browser for SQLite

Now let's manually add some data into the database. Using the **DB Browser for SQLite** tool, which can be downloaded from <https://sqlitebrowser.org/> for free.

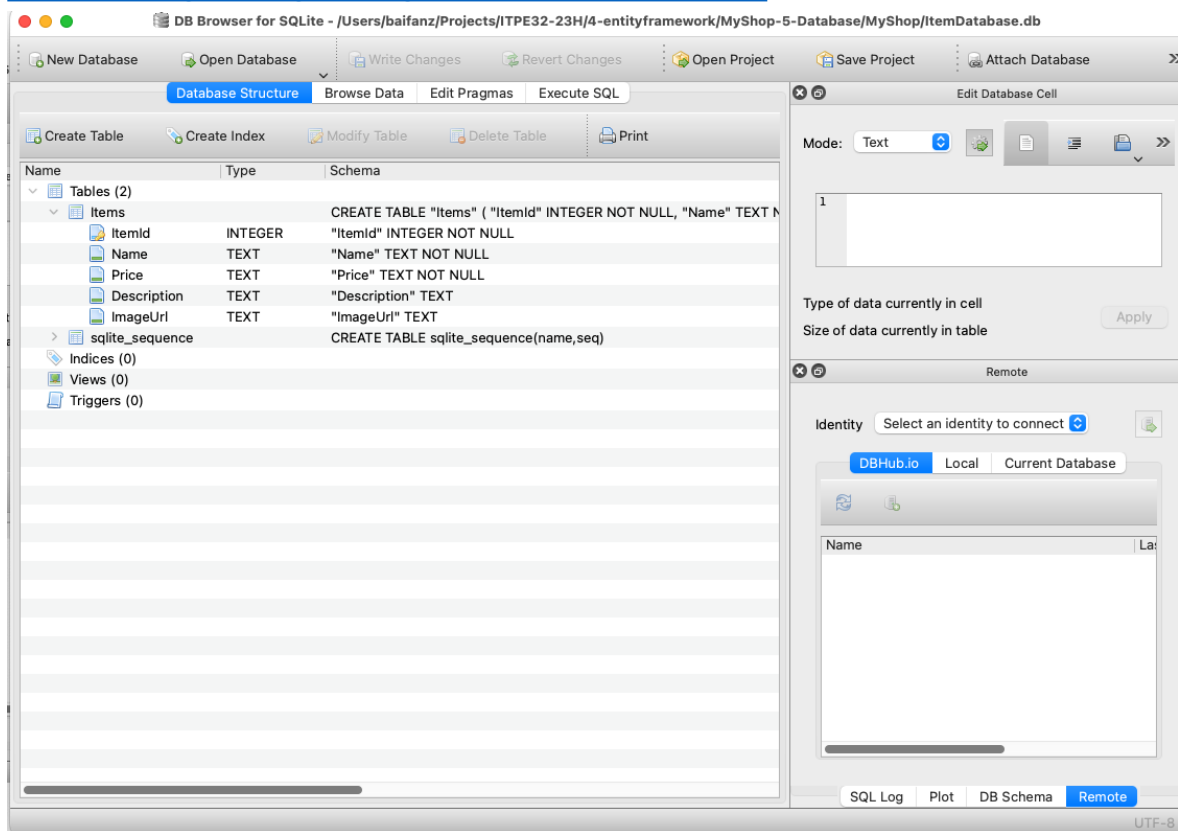


Open the ItemDatabase.db with DB Browser for SQLite, you can see that the Database Structure contains the item class that we have defined. A Database Structure is sometimes also called Data Schema, or Data Model.

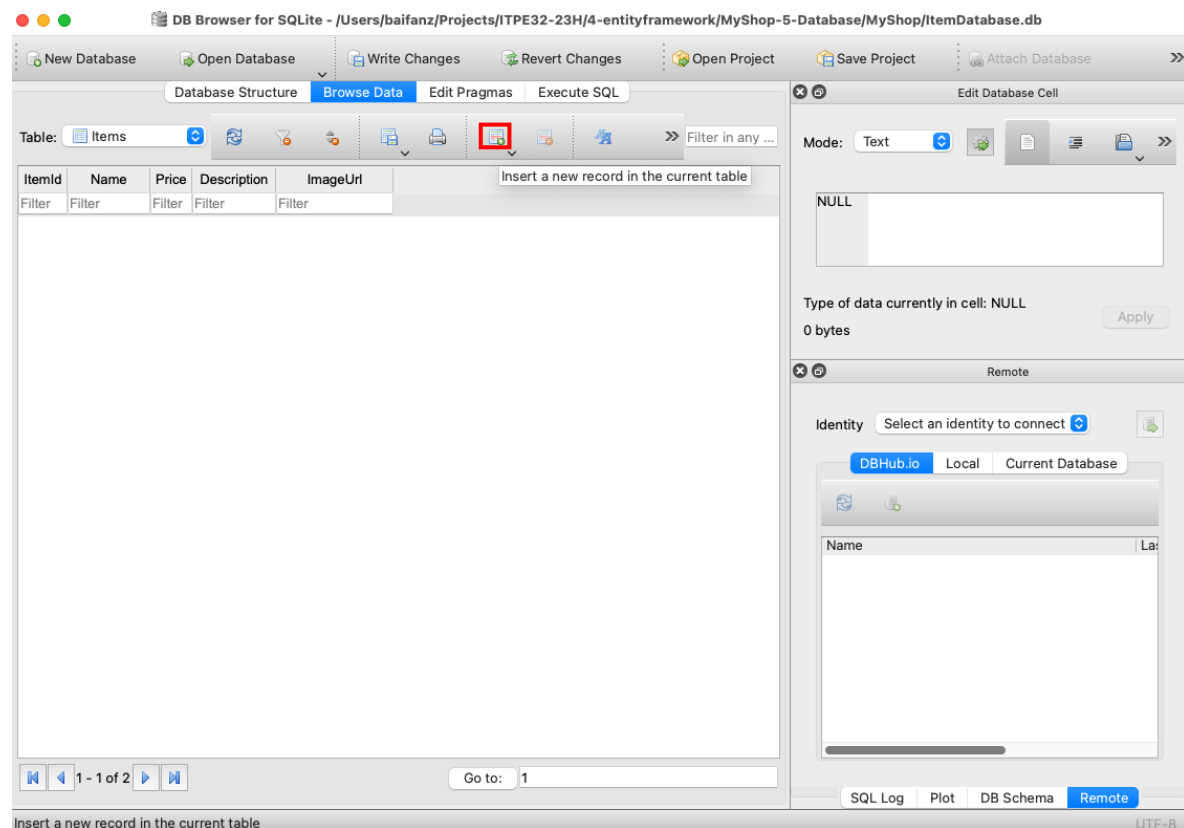
You can read more on the internet, e.g.,:

<https://planetscale.com/blog/schema-design-101-relational-databases>

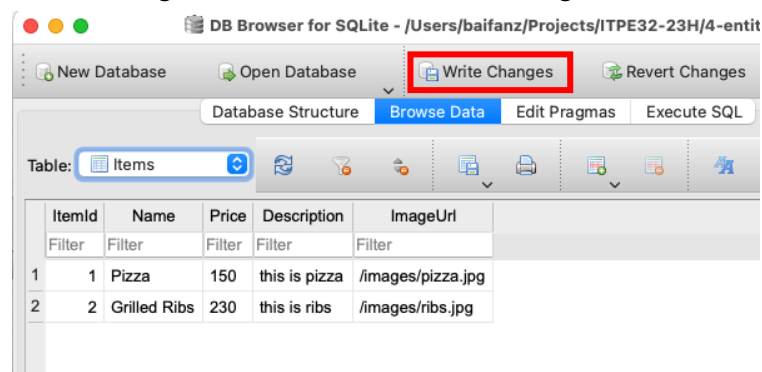
<https://www.geeksforgeeks.org/relation-schema-in-dbms/>



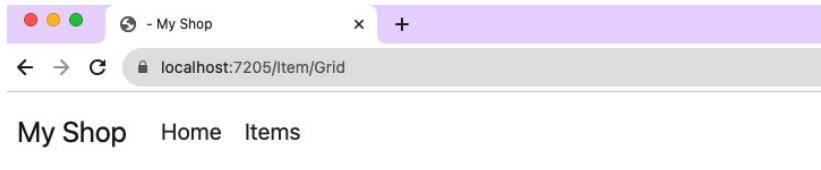
Now we go to Browse Data, which is empty. You can click the button in the red rectangle to add new entries.



After adding the entries, click "Write Changes" to save the data.



Now start debugging the project, you will see the data retrieved from the database.



List of Shop Items (Grid View)

[Table View](#)



[Pizza](#)

150.00 NOK



[Grilled Ribs](#)

230.00 NOK

Congratulations! Now you have your first program with database connection! 😊