# Entity Framework Core Tutorial with SQLite

This repository contains a sample console application that demonstrates how to use Entity Framework Core with SQLite to perform CRUD (Create, Read, Update, Delete) operations on a database.

## Table of Contents



## Introduction

Entity Framework Core (EF Core) is a lightweight, extensible, open-source, and cross-platform version of the popular Entity Framework data access technology. It serves as an object-relational mapper (ORM), enabling .NET developers to work with a database using .NET objects and eliminating the need for most of the data-access code that typically needs to be written.

This tutorial demonstrates how to use EF Core with SQLite in a .NET console application to manage a simple book database.

## Project Structure

The project follows a clean architecture with separation of concerns:

```
ConsoleAppWithEntityFrameworkAndMySQL/
├── Program.cs                  # Application entry point
├── Model/
│   └── Book.cs                 # Entity model
├── Data/
│   ├── BookDbContext.cs        # Database context
│   └── DatabaseConfig.cs       # Database configuration
└── Service/
    ├── BookService.cs          # Business logic for books
    └── ConsoleUIService.cs     # User interface logic
```

## Setting Up Entity Framework Core

To use Entity Framework Core with SQLite, you need to add the required NuGet packages to your project:

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Relational"
Version="8.0.3" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.3"
/>
</ItemGroup>
```

## Creating Models

Models represent the entities in your database. In this project, we have a `Book` model:

```
namespace ConsoleAppWithEntityFrameworkAndMySQL.Model
{
    public class Book
    {
        public Guid Id { set; get; }
        public required string Title { set; get; }
        public required string Author { set; get; }
        public decimal Price { set; get; }
    }
}
```

Note the use of the `required` keyword for non-nullable string properties, which is a C# 11 feature that ensures these properties are initialized during object creation.

## Creating a DbContext

The `DbContext` class is the primary class that coordinates Entity Framework functionality for a data model. It represents a session with the database and can be used to query and save instances of your entities.

```
using ConsoleAppWithEntityFrameworkAndMySQL.Model;
using Microsoft.EntityFrameworkCore;

namespace ConsoleAppWithEntityFrameworkAndMySQL.Data
{
    public class BookDbContext : DbContext
    {
        public BookDbContext(DbContextOptions<BookDbContext> options) :
base(options)
        {
        }

        public DbSet<Book> Books { get; set; }
    }
}
```

The `DbSet<Book>` property represents the collection of all `Book` entities in the context, or that can be queried from the database.

## Database Configuration

It's a good practice to centralize your database configuration. In this project, we use a static `DatabaseConfig` class:

```
using Microsoft.EntityFrameworkCore;

namespace ConsoleAppWithEntityFrameworkAndMySQL.Data
{
    public static class DatabaseConfig
    {
        public static string ConnectionString => "Data Source=books.db";

        public static DbContextOptions<BookDbContext> GetDbContextOptions()
        {
            var optionsBuilder = new DbContextOptionsBuilder<BookDbContext>();
            optionsBuilder.UseSqlite(ConnectionString);
            return optionsBuilder.Options;
        }
    }
}
```

This class provides the connection string and a method to get the `DbContextOptions` needed to create a `DbContext` instance.

## Repository Pattern with Services

The repository pattern is a design pattern that isolates the data layer from the rest of the application. In this project, we implement a service layer that acts as a repository:

```
public class BookService
{
    private readonly BookDbContext _context;

    public BookService(BookDbContext context)
    {
        _context = context;
    }

    public List<Book> GetAllBooks()
    {
        return _context.Books.ToList();
    }

    public Book? GetBookById(Guid id)
    {
```

```
            return _context.Books.Find(id);
        }

        // Other methods...
    }
```

## CRUD Operations

### Create

```
public void AddBook(Book book)
{
    _context.Books.Add(book);
    _context.SaveChanges();
}
```

### Read

```
public List<Book> GetAllBooks()
{
    return _context.Books.ToList();
}

public Book? GetBookById(Guid id)
{
    return _context.Books.Find(id);
}

public List<Book> SearchBooksByTitle(string title)
{
    return _context.Books.Where(b => b.Title.Contains(title)).ToList();
}

public List<Book> SearchBooksByAuthor(string author)
{
    return _context.Books.Where(b => b.Author.Contains(author)).ToList();
}
```

### Update

```
public void UpdateBook(Book book)
{
    // If the entity is not being tracked, attach it
    if (!_context.Books.Local.Any(b => b.Id == book.Id))
    {
        _context.Books.Update(book);
```

```
        }
        // Otherwise, EF Core is already tracking it and will detect changes

        _context.SaveChanges();
    }
```

Entity Framework Core has a change tracking system that automatically detects changes to entities that were retrieved from the database. When you call `SaveChanges()`, EF Core will generate the appropriate SQL UPDATE statements for any modified entities.

There are two common scenarios for updates:

1. **Tracked entities**: When you retrieve an entity using methods like `Find()`, `FirstOrDefault()`, etc., EF Core automatically tracks changes to that entity. You can modify its properties and then call `SaveChanges()` to persist those changes.
2. **Untracked entities**: If you create a new entity instance or receive one from an external source (like a web API request), EF Core doesn't know about it. In this case, you need to explicitly tell EF Core to track it using methods like `Update()` or `Attach()`.

Our implementation handles both scenarios.

### Delete

```
public void DeleteBook(Book book)
{
    _context.Books.Remove(book);
    _context.SaveChanges();
}
```

## Database Initialization

When starting the application, we need to ensure the database exists and is properly seeded with initial data:

```
// Ensure database is created
bookService.EnsureDatabaseCreated();

// Seed initial data if needed
if (!bookService.HasAnyBooks())
{
    Console.WriteLine("Seeding initial data...");
    bookService.SeedInitialData();
    Console.WriteLine("Initial data seeded successfully!");
}
```

The `EnsureDatabaseCreated()` method calls `_context.Database.EnsureCreated()`, which creates the database if it doesn't exist:

```
public void EnsureDatabaseCreated()
{
    _context.Database.EnsureCreated();
}
```

And the `SeedInitialData()` method adds some initial data to the database:

```
public void SeedInitialData()
{
    var books = new List<Book>
    {
        new Book { Id = Guid.NewGuid(), Title = "Let us C#", Author = "Vinod",
Price = 3999.0m },
        new Book { Id = Guid.NewGuid(), Title = "Let us Python", Author = "Vinod",
Price = 3599.0m },
        new Book { Id = Guid.NewGuid(), Title = "Clean Code", Author = "Robert C.
Martin", Price = 4599.0m },
        new Book { Id = Guid.NewGuid(), Title = "Design Patterns", Author = "Erich
Gamma", Price = 5299.0m }
    };

    _context.Books.AddRange(books);
    _context.SaveChanges();
}
```

## Running the Application

The application entry point is in `Program.cs`:

```
using ConsoleAppWithEntityFrameworkAndMySQL.Data;
using ConsoleAppWithEntityFrameworkAndMySQL.Service;

// Setup database context
var dbOptions = DatabaseConfig.GetDbContextOptions();
using var dbContext = new BookDbContext(dbOptions);

// Setup services
var bookService = new BookService(dbContext);

// Ensure database is created
bookService.EnsureDatabaseCreated();

// Seed initial data if needed
if (!bookService.HasAnyBooks())
{
    Console.WriteLine("Seeding initial data...");
    bookService.SeedInitialData();
    Console.WriteLine("Initial data seeded successfully!");
```

```
}

// Run the application
var uiService = new ConsoleUIService(bookService);
uiService.RunMenu();
```

## Best Practices

1. **Separation of Concerns**: Keep your code organized by separating it into different layers (models, data access, services, UI).
2. **Repository Pattern**: Use a service or repository layer to abstract database operations from the rest of your application.
3. **Dependency Injection**: Inject dependencies like DbContext into your services rather than creating them directly.
4. **Centralized Configuration**: Keep database configuration in a central place for easy maintenance.
5. **Error Handling**: Always handle potential database errors gracefully.
6. **Use Async Methods**: For better performance in real-world applications, use the async versions of EF Core methods (`ToListAsync()`, `SaveChangesAsync()`, etc.).
7. **Migrations**: For production applications, use EF Core migrations to manage database schema changes.
8. **Understand Entity Tracking**: Be aware of how EF Core tracks entities and handle both tracked and untracked entities appropriately.

## Advanced Topics (Not Covered in This Example)

- **Migrations**: Managing database schema changes
- **Complex Relationships**: One-to-many, many-to-many relationships
- **Lazy Loading vs. Eager Loading**: Different strategies for loading related data
- **Concurrency Handling**: Dealing with concurrent database updates
- **Transactions**: Ensuring data consistency across multiple operations

## Conclusion

Entity Framework Core provides a powerful and flexible way to interact with databases in .NET applications. By following the patterns and practices demonstrated in this tutorial, you can build maintainable and efficient data access layers for your applications.

For more information, refer to the official Entity Framework Core documentation.