1. Create the Domain (Tables) using MS Excel spreadsheet
2. Create the GitHub repositories and then clone it to your development machine
3. Open Visual Studio 2022 Community Edition and create an API .NET Core project .NET8.0
4. Create the Domains on your API project by
   1. New folder named Models
   2. New folder inside the Models folder named Domain
   3. Create each domain (tables) inside the Domain folder
5. Adding Entity Framework Core Packages to connect to a SQL Server Database
   1. Right click on Dependencies and go to Manage NuGet Packages
   2. Search and browse for the following:
      1. Microsoft.EntityFrameworkCore.SqlServer
      2. Microsoft.EntityFrameworkCore.Tools
6. Creating the DBContext Class
   * Maintaining Connection to Db
   * Track Changes
   * Perform CRUD operations
   * Bridge between domain models and the database
7. Create a folder for the DBContext
   1. Create a folder named Data on the project
8. Inside the Data folder, create a new Class file for the DBContext
   1. For example, PersonalExpensesDbContext.cs
   2. Inherit DbContext to the class file, DbContext class from Microsoft.EntityFrameworkCore
   3. Create a constructor for this class (PersonalExpensesDbContext.cs)
   4. After the constructor we want to create a dbSets. A DbSet is a property of DbContext class that represents a set of entities of the database.
9. Adding ConnectionString to the Database in AppSetting.Json
10. Understanding Dependency Injection and Injecting DbContext into our application.
    1. Inject a service to Program.cs, inject the DbContext class
11. Create the database to the SQL Server using the Entity Framework Migrations
    1. Tools > NuGet Package Manager > Package Manager Console
    2. Create 2 commands:
       1. Add-Migration “Name of Migration”
       2. Update-Database
12. Create Controller and Actions: GET/POST/PUT/DELETE
13. Let’s create a controller or an endpoint for our Categories resource
14. Right click on the Controllers folder, go to the Add menu and click on Controller. Go to the API from the left hand side and choose API Controller – Empty, then click the Add button to add a controller
15. Now we have to specify the name of the controller and as you know, it has to be suffixed with the keyword controller so that the application can identify this class as a controller class. Example: CategoriesController.cs
16. Using Dependency Injection (DI), now we can use the DbContext inside the controller thru constructor
17. We start using the private file on the Action method

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using PersonalExpenses.API.Data;

using PersonalExpenses.API.Models.Domain;

namespace PersonalExpenses.API.Controllers

{

// https://localhost:portaNumber/api/categories

[Route("api/[controller]")]

[ApiController]

public class CategoriesController : ControllerBase

{

private readonly PersonalExpensesDbContext dbContext;

public CategoriesController(PersonalExpensesDbContext dbContext)

{

this.dbContext = dbContext;

}

// GET ALL CATEGORIES

// GET: https://locahost:portNumber/api/categories

[HttpGet]

public IActionResult GetAll()

{

var categories = dbContext.Categories.ToList();

return Ok(categories);

}

}

}

1. Get Region By Id Action Method

// GET SINGLE CATEGORY (Get Category By ID)

// GET: https://localhost:portnumber/api/categories/{id}

[HttpGet]

[Route("{id:Guid}")]

public IActionResult GetById([FromRoute] Guid id)

{

//var category = dbContext.Categories.Find(id);

var category = dbContext.Categories.FirstOrDefault(x => x.Id == id);

if (category == null)

{

return NotFound();

}

return Ok(category);

}

1. Change methods to us DTO – Data Transfer Object
   1. Client <-> DTO |API| Domain <-> Database
   2. Advantages of DTO:
      1. Separation of concerns
      2. Performance
      3. Security
      4. Versioning

Asynchronous Programming

* Traditional Synchronous Programming – program execution is blocked
* Poor performance (Synchronous programming)
* Async/wait keywords
* More requests

// GET ALL CATEGORIES

// GET: https://locahost:portNumber/api/categories

[HttpGet]

public async Task<IActionResult> GetAll()

{

// Get data from database - Domain Models

var categoriesDomain = await dbContext.Categories.ToListAsync();

// Map Domain Models to DTO before sending back to the client

// Map this Domain Models to Data Object Transfer (DTO)

var categoriesDto = new List<CategoryDto>();

foreach (var categoryDomain in categoriesDomain)

{

categoriesDto.Add(new CategoryDto()

{

Id = categoryDomain.Id,

Abbr = categoryDomain.Abbr,

Name = categoryDomain.Name,

CategoyImageUrl = categoryDomain.CategoyImageUrl

});

}

// Return DTO

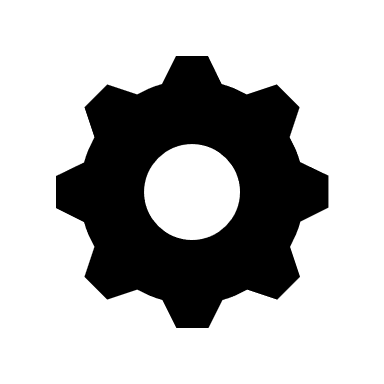
return Ok(categoriesDto);

}

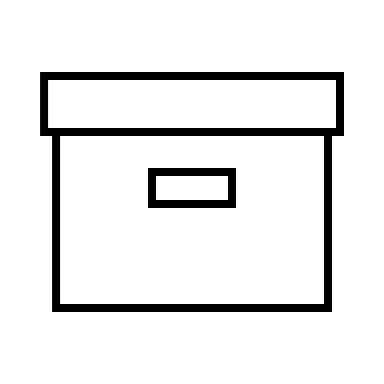
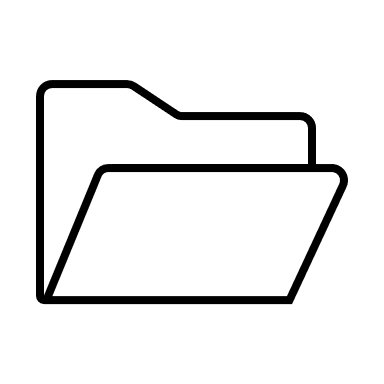
* Three things that we did above, we made the function asynchronous. We wrapped the result (IActionResult) inside a Task and we awaited on the individual calls that we wanted to be asynchronous.

**Repository Pattern and Benefits**

Repository pattern in ASP.NET Core is a design pattern that helps to separate the data access layer from the rest of the application. It provides a standard interface, which is an interface class for accessing and managing data in a data store such as a database without exposing the underlying implementation detail to the rest of the application.



DbContext



Database

Repository

Controller

Repository pattern typically involves creating an abstraction layer between the application and the data store, which is implemented by a concrete repository class. The Repository class is responsible for performing CRUD operations. That is, create, read, update and delete on the data store and it exposes a set of methods that the application can use to interact with the data.

If you remember your application now we have the Dbcontext class injected inside the controller and it is the controller who is directly talking to the database using that dbcontext. That is also a wrong practice and using the repository design pattern, we can eliminate that by adding an abstraction layer in between.

Repositories can be added in between the controller and the database so that all the operations on the database is then handled by the repository. The Dbcontext class is injected inside the repository rather than inside the controller, and it is

the repository that then injects in the controller. So the controller will use the repository instead of using the dbcontext. By doing that, the controller now has no awareness of what's being called through the DBCONTEXT, whether it's a SQL Server database or a MongoDB database, it has no idea about it.

Controller is just using the interface method exposed by the interface repository and the implementation is hidden behind the implementation repository. Using that, you can switch the logic and the data stores behind the implementation repository. For example, you can use entity framework core to store your changes in a database, or you can just use an in-memory database by creating another implementation repository. All of those changes are happening behind the repository and the controller has no knowledge about the data stores at all.

By using repository pattern in ASP.NET Core, we developers can achieve several benefits:

* Decoupling
* Consistency
* Performance
* Multiple data sources (switching)

That is decoupling the data access layer from the rest of the application, which makes it easier to maintain and test the application. Providing a standard interface for accessing data which improves the consistency and readability of the code.

Now every connection to the database goes through the repository. We can also improve the performance of the application by using caching, batching or other optimization techniques supporting multiple data sources, which allows the application to switch between different data sources without affecting the application logic.

Overall, the repository pattern is a powerful tool for managing data in ASP.NET Core applications, and we will introduce repositories in our application as well.

**Automapper**

* Object-to-object mapping
* Simplification
* Map between DTOs and Domain Models and vice-versa
* Quite powerful apart from just simple object-to-object mapping

In this lecture, we will learn more about Auto Mapper and will later implement automapper in our project.

Auto Mapper is a popular object to object mapping library for Dotnet applications, including ASP.NET Core.

It allows us to simplify the mapping process between two objects with different structures by defining mappings between their properties.

With the help of auto mapper, you can create mappings between the source and destination objects in a centralized location which can be used throughout your application.

This reduces the need for manually copying values between objects, and you can help to make your code more maintainable and less error prone.

In ASP.NET Core, Automapper is commonly used to map between domain models and view models or DTOs.

It can also be easily be integrated into your application using NuGet packages and configured using its fluent API.

Auto Mapper provides a lot of functional flexibility and functionality in how mappings can be defined, including support for custom resolvers, value formatters and mapping conventions.

This makes it a powerful tool for managing object to object mapping in your ASP.NET Core application. In the next lecture, let's go on and see how we can implement automapper in our application.

Install packages for Automapper:

1. Right-click on Dependencies
2. Go to Manage NuGet Packages
3. Browse for a package called Automapper by Jimmy Boggard – A convention-based object-object mapper.
4. Click on Install
5. Once the package is installed, close the package window
6. Start creating the mapping profiles

**Functionality for Expenses – Create, Read, Update, and Delete for Expenses**

* Seed Data Using Entity Framework Core (EF Core)
* Frequencies Controller and Action Methods
* Making Repositories Generic

**Navigation Properties**

* Allow to navigate from one Entity to another
* Expenses Domain Model will have Frequency and Category Property