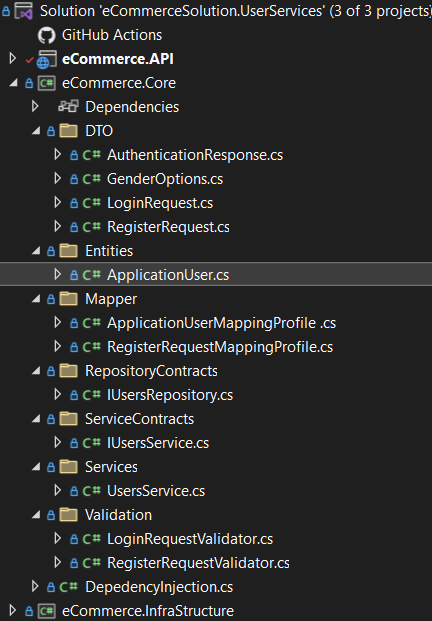
## **User Service Details**

We used clean architecture to build user service microservices.

## **CORE**

**Clean Architecture**, the **Core** layer contains business logic and entities that are completely independent of external services like databases, APIs, or UI frameworks. This layer serves as the heart of the application, containing the essential domain logic and models.



DTO instead of defining class components we used Record which is new in .net 9 it is also reference type but work differently from class as it compare type to type.

**DTOs in Clean Architecture**:

* DTOs are used to transport data between layers without involving complex domain models. They keep your core business logic independent from infrastructure concerns.

**Records**:

* Records in C# are immutable by default and support value equality. They are perfect for DTOs because they encapsulate data and make working with immutable types easier.

**record Keyword**: This defines a **record type** in C#. A record is a reference type that provides built-in functionality for **value equality**, meaning two records with the same data are considered equal, unlike classes which use reference equality.

* **Benefits of Records**: Records are immutable by default. You can define a set of properties that don’t change, making them perfect for DTOs since they only carry data. You also get automatic ToString(), Equals(), and GetHashCode() implementations.

**Why use a DTO?**

* Separation of concerns: Keeps the user model separate from the request model, avoiding unnecessary dependencies on internal details of the user entity.
* Flexibility: It allows modifications (e.g., adding fields) without impacting the user entity or other parts of the system.

A screen shot of a computer

AI-generated content may be incorrect.

After DTO for external interaction next is entities In **Clean Architecture**, the **Entities** in the **Core** layer represent the fundamental business models that encapsulate your domain logic. These entities are pure C# classes that define the structure and behavior of the objects used within your domain. They are central to the business logic and are free from any dependency on external systems like databases, APIs, or frameworks.

A screenshot of a computer

AI-generated content may be incorrect.

**Entities in Clean Architecture**:

* Entities represent **core domain objects** that are meaningful to the business. They encapsulate data and sometimes behavior that is central to the domain.
* In the **Core** layer, entities are designed to be **independent of frameworks**. This ensures that they can evolve without the constraints of infrastructure concerns such as databases or APIs.

**Data Representation**:

* The ApplicationUser entity only represents data. It doesn’t include any behavior or business rules (such as password encryption or email validation). Those responsibilities belong to higher layers of the application (e.g., Application or Infrastructure layers).

**Repository Contract**

**IUsersRepository**

In Clean Architecture, the Core layer defines the essential business logic and contracts, while the actual implementation details (e.g., how to interact with databases) are pushed to the Infrastructure layer. This approach follows the Dependency Inversion Principle, which means that high-level policies (business logic) should not depend on low-level details (data storage or retrieval).

The Repository Pattern is commonly used in this architecture to abstract data access logic. In the Core layer, we define repository contracts (interfaces), while the actual repository implementation (that interacts with the database) resides in the Infrastructure layer. This allows us to change the data source (e.g., switch databases or use an API) without changing the business logic.

**Repository Pattern**:

* The repository pattern abstracts the **data access logic**. Instead of directly interacting with the database or data source, the application interacts with a repository, which provides methods for adding, retrieving, updating, or deleting entities.
* In Clean Architecture, this pattern helps decouple business logic from infrastructure concerns (like databases or external services).

**Service Contract:**

**IUsersService**

In Clean Architecture, the Service Contracts (typically interfaces) in the Core layer define the business use cases or operations that the application can perform. These interfaces contain business logic but are abstracted from the actual implementation. The implementations of these contracts are usually placed in the Application or Infrastructure layer.

The Core layer should only know about the interfaces, which provides flexibility and decouples the business logic from the underlying infrastructure or framework dependencies.

**Validation:**

In **Clean Architecture**, the **Core** layer defines business logic, and validation is a crucial part of ensuring the integrity of data before it enters the system. Validation is typically handled by **validators** in the Core layer to ensure that the data passed to the system meets the necessary business rules.

Here, we are using **FluentValidation**, a popular .NET library, to implement validation logic in a clean, readable, and flexible way. FluentValidation allows you to define rules for different properties of your DTOs, ensuring that the input data adheres to certain conditions before the application processes it.

As it is Depreceated but we can still use older package for validation purpose.

Intall Nuget package

**FluentValidation.AspNetCore** Version 11 both in API project and Core project.

public class LoginRequestValidator : AbstractValidator<LoginRequest>

{

public LoginRequestValidator()

{

//

RuleFor(temp => temp.Email)

.NotEmpty().WithMessage("Email is required")

.EmailAddress().WithMessage("Invalid email address format");

//Password

RuleFor(temp => temp.Password)

.NotEmpty().WithMessage("Password is required");

}

}

Add reference in program.cs

// FluentValidations

builder.Services.AddFluentValidationAutoValidation();

In dependency injection class add this

// Registering FluentValidation validators from the assembly containing the LoginRequestValidator

services.AddValidatorsFromAssemblyContaining<LoginRequestValidator>();

**FluentValidation Concepts**

**1. Rule Definitions**

* **RuleFor()**: This is the key function in FluentValidation that defines a rule for a specific property.
  + Each rule targets a particular property of the DTO (e.g., Email, Password).

**2. Validation Methods**

* **NotEmpty()**: Ensures that a field has a value and is not null or whitespace.
* **WithMessage()**: Attaches a custom error message when the validation fails. This improves the clarity of error handling for end users.
* **EmailAddress()**: Ensures that a string follows a valid email format.
* **Length()**: Enforces that a string property falls within a specific length range.
* **IsInEnum()**: Ensures that a property value matches a valid enum option.

**Automapper**

In the **Core** layer of Clean Architecture, mapping is a crucial aspect. You often need to convert between various layers' models, such as mapping data transfer objects (DTOs) to entities and vice versa. To automate this conversion and avoid repetitive code, we use **AutoMapper**, a library that helps map properties between objects, particularly when working with DTOs, view models, and entities.

**Key Concepts of AutoMapper**

* **Profiles**: AutoMapper uses Profile classes to configure mappings. Each profile contains rules for converting from one type to another.
* **CreateMap**: This method inside the profile defines a map between a source type and a destination type.
* **ForMember**: This method is used to customize how specific properties are mapped. You can specify which property of the destination object should be mapped from which property of the source object.
* **Ignore**: Sometimes you don't want to map certain properties, especially when they are computed or generated after the mapping. In such cases, Ignore() is used.

Install Nuget package older version then 15 as it is now commercial so need license in latest version

Add dependency in core project

Add it with class name and in mapper class extend mapper class with profile so where ever profile class is exented project will call it using AddAutomapper

builder.Services.AddAutoMapper(typeof(ApplicationUserMappingProfile).Assembly);

public class ApplicationUserMappingProfile:Profile

{

public ApplicationUserMappingProfile()

{

CreateMap<ApplicationUser, AuthenticationResponse>()

.ForMember(dest => dest.UserID, opt => opt.MapFrom(src => src.UserID))

.ForMember(dest => dest.Email, opt => opt.MapFrom(src => src.Email))

.ForMember(dest => dest.PersonName, opt => opt.MapFrom(src => src.PersonName))

.ForMember(dest => dest.Gender, opt => opt.MapFrom(src => src.Gender))

.ForMember(dest => dest.Success, opt => opt.Ignore()) // Computed or added later

.ForMember(dest => dest.Token, opt => opt.Ignore()) // Token is not part of ApplicationUser

;

}

}

**Services in the Core Layer**

In the Core layer of a Clean Architecture solution, the service layer defines business logic for the application. It handles interactions between repositories (which deal with data persistence) and external controllers or consumers (such as APIs). This is where most of the application’s business rules are implemented, and it’s a key layer for encapsulating the domain logic.

In this case, we have a UsersService that handles user-related operations like login and registration. These methods interact with a repository that manages the persistence of user data and return an appropriate response to the client.

The service also leverages AutoMapper to map between DTOs and domain entities, ensuring that only necessary data is exposed externally.

**Key Concepts**

**1. Dependency Injection (DI)**

ASP.NET Core provides dependency injection (DI) by default, which allows you to easily inject required services into classes. In this example, the UsersService uses DI to inject the IUsersRepository and IMapper objects.

**2. AutoMapper in Services**

The service uses AutoMapper to map between DTOs and Entities:

* Mapping from ApplicationUser (entity) to AuthenticationResponse (DTO).
* Mapping from RegisterRequest (DTO) to ApplicationUser (entity).

This mapping reduces boilerplate code that would otherwise involve manually assigning each field from the entity to the DTO and vice versa.

**3. Repository Pattern**

The service interacts with a repository that abstracts data access logic. This repository is responsible for fetching and saving user data, and by abstracting it, the service does not need to know the details of how data is stored (database, file, etc.). This separation of concerns leads to cleaner, more maintainable code.

**4. DTOs for API Responses**

DTOs (Data Transfer Objects) are used to send and receive data. For login and registration, the service returns an AuthenticationResponse, which contains information like UserID, Email, and Token. Using DTOs ensures that only necessary data is shared, keeping sensitive information (e.g., password) hidden.

**5. with Expression for Immutable Objects**

C# record types, like AuthenticationResponse, are immutable by default, meaning you can't modify their properties once set. However, with the with expression, you can create a new object by copying an existing one, while modifying some of its properties. This is useful for returning a new AuthenticationResponse object with updated Success and Token fields.

**Infrastructure Layer** :

**DbContext in Infrastructure Layer**

In the **Infrastructure** layer of a Clean Architecture solution, the DbContext class is responsible for managing database connections and providing abstraction for data access. In this specific case, the DapperDbContext class uses **Dapper** (a micro ORM) for database operations and **SqlServer**

**Key Concepts**

**1. Dapper and SQLServer**

* **Dapper**:
  + A lightweight Object-Relational Mapper (ORM) for .NET. It simplifies data access and is known for its high performance compared to other ORMs like Entity Framework

**2. DbContext Pattern**

* **DbContext**:
  + Typically refers to Entity Framework Core’s context class, but here it's used more generically. It provides methods to manage database connections and perform operations.

**3. IConfiguration Interface**

* **IConfiguration**:
  + Used to read configuration settings from various sources (e.g., appsettings.json, environment variables). In this case, it retrieves the database connection string.

**Code Walkthrough**

Here’s the complete code for the DapperDbContext class, followed by a detailed explanation.

public class DapperDbContext

{

private readonly IConfiguration \_configuration;

private readonly IDbConnection \_connection;

public DapperDbContext(IConfiguration configuration)

{

\_configuration = configuration;

string connectionString = \_configuration.GetConnectionString("DefaultConnection");

\_connection = new SqlConnection(connectionString);

}

public IDbConnection DbConnection => \_connection;

}

using Dapper;

using eCommerce.Core.DTO;

using eCommerce.Core.Entities;

using eCommerce.Core.RepositoryContracts;

using eCommerce.Infrastructure.DbContext;

namespace eCommerce.Infrastructure.Repositories;

internal class UsersRepository : IUsersRepository

{

private readonly DapperDbContext \_dbContext;

public UsersRepository(DapperDbContext dbContext)

{

\_dbContext = dbContext;

}

public async Task<ApplicationUser?> AddUser(ApplicationUser user)

{

// Generate a new unique user ID for the user

user.UserID = Guid.NewGuid();

// SQL Query to insert user data into the "Users" table

string query = "INSERT INTO public.\"Users\"(\"UserID\", \"Email\", \"PersonName\", \"Gender\", \"Password\") VALUES(@UserID, @Email, @PersonName, @Gender, @Password)";

int rowCountAffected = await \_dbContext.DbConnection.ExecuteAsync(query, user);

if (rowCountAffected > 0 )

{

return user;

}

else

{

return null;

}

}

public async Task<ApplicationUser?> GetUserByEmailAndPassword(string? email, string? password)

{

// SQL query to select a user by Email and Password

string query = "SELECT \* FROM public.\"Users\" WHERE \"Email\"=@Email AND \"Password\"=@Password";

var parameters = new { Email = email, Password = password };

ApplicationUser? user = await \_dbContext.DbConnection.QueryFirstOrDefaultAsync<ApplicationUser>(query, parameters);

return user;

}

}

So Dapper is mini ORM but fast where we have more control on sql queries .

We have used middleware service as well for exception handling and swagger ui.

Rest is same for controllers.

In controllers injecting service class and calling business methods