.NET Microservices – Azure DevOps and AKS

Section 2: Users Microservice - Notes

Clean Architecture

In the context of **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.

1. DTOs (Data Transfer Objects)

Conceptual Understanding:

- **DTO (Data Transfer Object)**: DTOs are simple objects used to transfer data between layers or services in a system. They don't contain any business logic and are mainly used to carry data.
- **Purpose**: By using DTOs, you decouple your internal models from external clients or services, allowing for more flexible code and easier adaptation to changes. It also helps avoid exposing internal data models directly.
- **In Clean Architecture**: DTOs live in the **Core** layer because they are shared between different application layers (e.g., Application Layer, Web Layer). They facilitate communication between components without coupling them.

Now, let's break down the code file by file.

File 1: AuthenticationResponse Record

```
namespace eCommerce.Core.DTO;


public record AuthenticationResponse(
  Guid UserID,
  string? Email,
  string? PersonName,
  string? Gender,
  string? Token,
  bool Success
)
```

```
{
  // Parameterless constructor
  public AuthenticationResponse() : this(default, default, default, default, default, default)
  {
  }
}
```

Conceptual Breakdown

- **Namespace (eCommerce.Core.DTO)**: The namespace defines the logical grouping of your code. Here, the Core.DTO namespace represents that the file belongs to the core layer of your eCommerce application and contains DTOs.
- **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.
- **Properties**:
    - Guid UserID: Stores the user's unique identifier.
    - string? Email: The user's email. The ? denotes it is a nullable string.
    - string? PersonName: The user's name, also nullable.
    - string? Gender: Represents the gender of the user, nullable string.
    - string? Token: Stores the authentication token.
    - bool Success: A flag to indicate whether the authentication was successful.
- **Parameterless Constructor**: The empty constructor is required for some deserialization frameworks or APIs that need to create objects without parameters.
    - The this(default, default, ...) call assigns default values (e.g., null for reference types, Guid.Empty for Guid, false for bool) to each parameter when no arguments are provided.

File 2: GenderOptions Enum

```
namespace eCommerce.Core.DTO;


public enum GenderOptions
{
  Male, Female, Others
}
```

Conceptual Breakdown

- **Enum (enum)**: An enum is a value type that defines a set of named constants. In this case, the GenderOptions enum provides specific, pre-defined options for the gender of a user.
- **Values**:
    - Male, Female, Others: These are the predefined gender options available. Using an enum ensures that only these values are used throughout the system, enforcing data consistency.
- **Benefits of Enums**:
    - **Type Safety**: When you use enums, you prevent invalid values (e.g., typos in strings) from being assigned.
    - **Readability**: Makes code more readable by giving meaningful names to constants.

File 3: LoginRequest Record

```
namespace eCommerce.Core.DTO;


public record LoginRequest(
  string? Email,
  string? Password);
```

Conceptual Breakdown

- **LoginRequest DTO**: This record serves as a simple DTO to encapsulate the data required for a login operation.
- **Properties**:
    - string? Email: The user's email for login. Nullable because it may not always be required at all layers.
    - string? Password: The password for authentication. Nullable as well, to support flexibility (though in practice, passwords are usually required).

- **Why use a DTO for Login?**:
    - 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.

File 4: RegisterRequest Record

```
enamespace eCommerce.Core.DTO;


public record RegisterRequest(
 string? Email,
 string? Password,
 string? PersonName,
 GenderOptions Gender);
```

Conceptual Breakdown

- **RegisterRequest DTO**: This DTO is used to capture the data needed for registering a new user in the system.
- **Properties**:
    - string? Email: User's email, typically required for registration.
    - string? Password: User's password, required for authentication.
    - string? PersonName: The user's name.
    - GenderOptions Gender: The GenderOptions enum ensures the gender is one of the predefined options (Male, Female, Others).
- **Why use a RegisterRequest DTO?**:
    - **Encapsulation**: It isolates the data required for registration from other parts of the system, focusing only on what is necessary for this operation.
    - **Input Validation**: Easier to validate the incoming registration request, as all necessary data is packaged in one object.

Summary of Concepts and Usage in Code:

1. **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.
2. **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.
3. **Enums**:
    - Enums provide a type-safe way of working with fixed sets of values. Using enums like GenderOptions ensures that your application only works with valid gender types, reducing the chance of errors.
4. **Parameterless Constructor**:
    - Required for scenarios where deserialization frameworks need to create objects without passing in parameters.
5. **Separation of Concerns**:
    - DTOs help to cleanly separate the data model that interacts with external systems from your internal business logic, ensuring flexibility and maintainability.

Clean Architecture - Core Layer and 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.

Entity: ApplicationUser

Let's break down the ApplicationUser entity:

```
namespace eCommerce.Core.Entities;


/// <summary>
/// Define the ApplicationUser class which acts as entity model class to store user details in data store
/// </summary>
public class ApplicationUser
{
  public Guid UserID { get; set; }
  public string? Email { get; set; }
  public string? Password { get; set; }
  public string? PersonName { get; set; }
  public string? Gender { get; set; }
}
```

Conceptual Breakdown

- **Namespace (eCommerce.Core.Entities)**:
    - The **Entities** folder under the **Core** layer contains classes that represent the core business objects. These objects usually map directly to concepts in your business domain and might be stored in a data store (e.g., a database). The eCommerce.Core.Entities namespace logically organizes these models within the core of the application.

Understanding Each Part of the Code

1. Class Definition: ApplicationUser

- **public class ApplicationUser**:
    - The ApplicationUser class defines the structure of a user entity within the system. This class is primarily concerned with the core information related to the user (ID, email, password, etc.).
    - In **Clean Architecture**, entities are at the heart of the domain and are usually pure data models. These models define the essential characteristics and behavior of the domain objects.

2. Properties

- **Guid UserID { get; set; }**:
    - This is the **primary identifier** for each user in the system. The Guid (Globally Unique Identifier) ensures that each user has a unique and non-sequential ID.
    - **Guid** is preferred over sequential integers because it reduces the likelihood of collision in a distributed system and adds a layer of security by making it harder to guess other IDs.
    - This is a property with both a **getter** and a **setter**, meaning that the value can be read and modified.
- **string? Email { get; set; }**:
    - Stores the **email address** of the user. The string? type indicates that this field is nullable, meaning it can store either a valid email string or null.
    - In the context of validation, it's important to ensure that emails follow a valid format, though this will usually be handled in the application or presentation layer.
- **string? Password { get; set; }**:
    - Stores the **password** of the user.
- **string? PersonName { get; set; }**:
    - Represents the **name** of the user (e.g., full name, first name, or display name).
- **string? Gender { get; set; }**:
    - Stores the **gender** of the user. It's represented as a string, and it's nullable, which means users might not provide this information.

Conceptual Understanding of Core Entities

- **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).

- **Why Use Nullable Strings?**:
    - Nullable strings (like string?) allow flexibility in scenarios where certain fields (such as PersonName or Gender) may not be mandatory at every point in time.
    - In terms of validation, nullable fields require checks to ensure they meet application-specific constraints, but those checks should happen outside the entity (usually in the application layer).
- **GUID for IDs**:
    - Using **GUIDs** as identifiers allows for distributed systems to generate unique IDs without coordination between different parts of the system. This is useful in systems that may be horizontally scaled or that need to prevent ID collisions across distributed databases.

Best Practices and Common Considerations:

- **Entity Design**:
    - The ApplicationUser entity should contain only fields and properties that are directly relevant to the user. Other aspects (e.g., authentication methods, user preferences) could be represented by additional entities or value objects.
- **Separation of Concerns**:
    - The entity does not handle validation, business rules, or persistence logic. These responsibilities are delegated to other layers (e.g., Application Layer for validation, Infrastructure Layer for persistence).

Clean Architecture - Repository Contracts in Core Layer

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 Contract: IUsersRepository

Let's break down the code:

```
using eCommerce.Core.Entities;


namespace eCommerce.Core.RepositoryContracts;


/// <summary>
/// Contract to be implemented by UsersRepository that contains data access logic of Users data store
/// </summary>
public interface IUsersRepository
{
  /// <summary>
  /// Method to add a user to the data store and return the added user
  /// </summary>
  /// <param name="user"></param>
  /// <returns></returns>
  Task<ApplicationUser?> AddUser(ApplicationUser user);



  /// <summary>
  /// Method to retrieve existing user by their email and password
  /// </summary>
```

```
 /// <param name="email"></param>
 /// <param name="password"></param>
 /// <returns></returns>
 Task<ApplicationUser?> GetUserByEmailAndPassword(string? email, string? password);
}
```

Conceptual Breakdown

1. Interface Definition: IUsersRepository

- **public interface IUsersRepository**:
    - This is the **repository contract** that defines how the application will interact with the user data. It ensures that any repository implementation (e.g., for a database or in-memory storage) will follow this contract.
    - **Purpose**: The interface is used to define the methods that the actual repository (which deals with database interaction) must implement. By depending on interfaces (abstractions) rather than concrete classes, we can easily swap out implementations.
- **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).

2. AddUser Method

```
Task<ApplicationUser?> AddUser(ApplicationUser user);
```

- **Purpose**:
    - This method is used to **add a new user** to the data store (e.g., a database) and return the added user after it's stored.
- **Parameters**:
    - ApplicationUser user: This represents the **user entity** to be added to the data store. It’s an instance of the ApplicationUser class, which contains information such as UserID, Email, Password, PersonName, and Gender.

- **Return Type**:
    - Task<ApplicationUser?>:
        - The method is **asynchronous** (hence the Task<> wrapper) because data access operations (like saving to a database) often involve I/O operations, which are typically time-consuming.
        - The method returns an ApplicationUser?, meaning it returns the user that was successfully added, or null if the operation failed (e.g., if the data store rejected the operation).
        - Using asynchronous methods improves application performance by not blocking the main thread during I/O-bound operations.

3. GetUserByEmailAndPassword Method

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

- **Purpose**:
    - This method retrieves an existing user from the data store based on the provided **email** and **password**. It is commonly used in authentication processes.
- **Parameters**:
    - string? email: The email of the user to be retrieved. The ? indicates it's nullable, meaning this method can handle cases where the email is not provided.
    - string? password: The password of the user, also nullable. This is expected to be stored securely (usually hashed) in the actual data store.
- **Return Type**:
    - Task<ApplicationUser?>:
        - The method is asynchronous and returns an ApplicationUser?. If the user is found with the provided email and password, it returns that user. If no user matches the criteria, it returns null.
        - The use of asynchronous programming (Task<>) ensures that the method is non-blocking, which is crucial for applications that need to scale.

Summary of Repository Contract Code and Concepts

1. **Repository Contract (IUsersRepository)**:
    - This contract defines two essential methods for managing user data: adding a user and retrieving a user by their email and password.
    - The contract helps decouple the business logic from the actual data store, as any implementation (SQL database, NoSQL, etc.) must conform to this interface.

2. **Asynchronous Methods**:
    - Both methods are asynchronous (Task<>) to ensure that the application can handle time-consuming I/O operations, such as database access, without blocking the main execution thread.
3. **Method Responsibilities**:
    - AddUser(ApplicationUser user): Adds a user to the data store and returns the added user or null if the operation fails.
    - GetUserByEmailAndPassword(string? email, string? password): Retrieves a user based on email and password for scenarios like authentication, returning null if no match is found.

In this way, the **Core** layer stays **completely independent of infrastructure concerns** (like databases). By using **repository contracts**, you ensure that the business logic depends only on abstractions and not on concrete implementations of data access. This makes the application flexible and easier to test or extend.

Clean Architecture - Service Contracts in Core Layer

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.

Service Contract: IUsersService

Let's break down the code:

```
using eCommerce.Core.DTO;


namespace eCommerce.Core.ServiceContracts;


/// <summary>
/// Contract for users service that contains use cases for users
/// </summary>
public interface IUsersService
{
  /// <summary>
  /// Method to handle user login use case and returns an AuthenticationResponse object that contains status of login
  /// </summary>
  /// <param name="loginRequest"></param>
  /// <returns></returns>
  Task<AuthenticationResponse?> Login(LoginRequest loginRequest);



  /// <summary>
  /// Method to handle user registration use case and returns an object of AuthenticationResponse type that represents status of user registration
  /// </summary>
  /// <param name="registerRequest"></param>
  /// <returns></returns>
  Task<AuthenticationResponse?> Register(RegisterRequest registerRequest);
}
```

Conceptual Breakdown

1. Interface Definition: IUsersService

- **public interface IUsersService**:
    - The IUsersService interface defines the **service contract** for handling **user-related business operations** such as login and registration.
    - It defines two essential methods (Login and Register) that represent **use cases** in the application. These methods are typically implemented in the **Application Layer** or an external service, but they are defined here in the **Core** layer as abstract contracts.
    - By defining this interface, we ensure that the business logic is decoupled from any framework-specific or implementation-specific details. The actual login and registration logic can be implemented in any way, and you can easily swap the implementation without changing the rest of the application.

2. The Login Method

Task<AuthenticationResponse?> Login(LoginRequest loginRequest);

- **Purpose**:
    - The Login method handles the **user login use case**. It takes a LoginRequest object as input and returns an AuthenticationResponse object that contains the status of the login attempt (e.g., success or failure, along with other details like a token).
- **Parameters**:
    - LoginRequest loginRequest:
        - The loginRequest is a **Data Transfer Object (DTO)** that carries the **email** and **password** for the login attempt.
        - Using DTOs ensures that the data being passed between layers or services is well-defined and encapsulates only the necessary information (in this case, the credentials).
- **Return Type**:
    - Task<AuthenticationResponse?>:
        - The method is **asynchronous** (Task<>) because operations like verifying credentials and generating authentication tokens often involve external systems like databases or identity servers. These I/O-bound operations should not block the execution thread.
        - It returns an AuthenticationResponse?, which means the method will return an AuthenticationResponse object (with details such as the login status, token, and user details) or null if the login fails.
- **AuthenticationResponse**:
    - This is a **DTO** that encapsulates the result of the login attempt, including the user's ID, email, token, and a boolean indicating whether the login was successful.

3. The Register Method

Task<AuthenticationResponse?> Register(RegisterRequest registerRequest);

- **Purpose**:
    - The Register method handles the **user registration use case**. It allows users to create a new account by providing the necessary details such as email, password, name, and gender, and it returns an AuthenticationResponse that represents the status of the registration.
- **Parameters**:
    - RegisterRequest registerRequest:
        - The registerRequest is a **DTO** that encapsulates the user's **email**, **password**, **name**, and **gender**. This DTO ensures that only the relevant data required for user registration is passed to the service layer.
- **Return Type**:
    - Task<AuthenticationResponse?>:
        - Like the Login method, the Register method is asynchronous. Registering a user typically involves I/O operations such as storing the user in the database and sending confirmation emails.
        - The return type is an AuthenticationResponse?, which indicates whether the registration was successful. If successful, it will return a response with the user's details and a token; otherwise, it will return null.
- **AuthenticationResponse**:
    - The AuthenticationResponse DTO also applies here, representing the status of the registration attempt and containing data such as the newly created user's ID and authentication token.

DTOs in Service Contracts

In both methods, the parameters and return values are **DTOs** (Data Transfer Objects). DTOs are essential in **Clean Architecture** because they isolate the business layer from the external world. The business logic operates on these DTOs rather than interacting directly with HTTP requests or database models, keeping it clean and decoupled.

- **LoginRequest**:
    - A DTO that encapsulates the **email** and **password** fields required for the login process.
- **RegisterRequest**:
    - A DTO that contains the information required for user registration: **email**, **password**, **name**, and **gender**.

- **AuthenticationResponse**:
    - A DTO that represents the outcome of both the login and registration processes. It contains the user's ID, email, name, token, and a boolean indicating whether the operation was successful.

Summary of Service ContractCode and Concepts

1. **Service Contract (IUsersService)**:
    - The IUsersService defines two essential business use cases: user login and user registration.
    - These methods operate on **DTOs** (like LoginRequest and RegisterRequest) and return an AuthenticationResponse, which represents the outcome of the operation.
2. **Asynchronous Programming**:
    - Both methods (Login and Register) are **asynchronous** because they likely involve I/O-bound tasks such as interacting with databases or external services (like token generation). Using asynchronous methods ensures that the application remains responsive.
3. **Business Logic Encapsulation**:
    - The service contract abstracts away the **business logic**. While the actual implementation resides in other layers (Application or Infrastructure), the core layer defines the contract that must be adhered to by the implementation.
4. **DTOs**:
    - The use of **DTOs** (like LoginRequest, RegisterRequest, and AuthenticationResponse) ensures that the business logic is separated from external concerns like HTTP requests. DTOs provide a clear and consistent way to pass data between layers.

Clean Architecture - Validators in Core Layer

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.

LoginRequestValidator

The first validator is for the **LoginRequest** DTO, which ensures that the login data (email and password) is valid.

```
using eCommerce.Core.DTO;

using FluentValidation;


namespace eCommerce.Core.Validators;


public class LoginRequestValidator : AbstractValidator<LoginRequest>

{

 public LoginRequestValidator()

 {

  //Email

  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");

 }

}
```

Conceptual Breakdown of LoginRequestValidator

1. AbstractValidator<LoginRequest>

- **Purpose**:
    - This class inherits from AbstractValidator<T>, where T is the type of the object to be validated. In this case, it's validating a LoginRequest object.
    - It provides a base class for creating validation rules for DTOs.

2. Validating the Email Property

```
RuleFor(temp => temp.Email)
 .NotEmpty().WithMessage("Email is required")
 .EmailAddress().WithMessage("Invalid email address format");
```

- **RuleFor(temp => temp.Email)**:
    - This defines a validation rule for the Email property of LoginRequest.
- **NotEmpty()**:
    - This rule ensures that the Email field is not empty. If it is empty, the specified message **"Email is required"** is returned.
- **EmailAddress()**:
    - This rule ensures that the Email field contains a valid email address. If the email format is invalid, the message **"Invalid email address format"** is returned.

3. Validating the Password Property

```
RuleFor(temp => temp.Password)
 .NotEmpty().WithMessage("Password is required");
```

- **RuleFor(temp => temp.Password)**:
    - This defines a validation rule for the Password property of LoginRequest.
- **NotEmpty()**:
    - This rule ensures that the Password field is not empty. If it is empty, the message **"Password is required"** is returned.

RegisterRequestValidator

The second validator is for the **RegisterRequest** DTO, which validates registration data such as email, password, name, and gender.

```
using eCommerce.Core.DTO;

using FluentValidation;


namespace eCommerce.Core.Validators;


public class RegisterRequestValidator : AbstractValidator<RegisterRequest>

{

  public RegisterRequestValidator()

  {

    //Email

    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");


    // Validate the PersonName property.

    RuleFor(request => request.PersonName)

      .NotEmpty().WithMessage("PersonName can't be blank")

      .Length(1, 50).WithMessage("Person Name should be 1 to 50 characters long");


    // Validate the Gender property.

    RuleFor(request => request.Gender)

      .IsInEnum().WithMessage("Invalid gender option");

  }

}
```

Conceptual Breakdown of RegisterRequestValidator

1. AbstractValidator<RegisterRequest>

- This class validates the RegisterRequest DTO, and it defines rules for properties such as email, password, name, and gender.

2. Validating the Email Property

```
RuleFor(temp => temp.Email)
 .NotEmpty().WithMessage("Email is required")
 .EmailAddress().WithMessage("Invalid email address format");
```

- The rules here are the same as in LoginRequestValidator. It ensures that the email is not empty and that it follows a valid email address format.

3. Validating the Password Property

```
RuleFor(temp => temp.Password)
 .NotEmpty().WithMessage("Password is required");
```

- Like the email, this ensures that the password is not empty. This is a basic validation that could be expanded upon to include additional rules like length and complexity (e.g., requiring numbers and special characters).

4. Validating the PersonName Property

```
RuleFor(request => request.PersonName)
 .NotEmpty().WithMessage("PersonName can't be blank")
 .Length(1, 50).WithMessage("Person Name should be 1 to 50 characters long");
```

- **Purpose**:
  - This rule validates the PersonName field of the RegisterRequest.
- **NotEmpty()**:
  - Ensures that the PersonName is not blank. If it is, the message **"PersonName can't be blank"** is returned.
- **Length(1, 50)**:
  - This ensures that the name must be between **1** and **50** characters in length. If it's shorter or longer than this range, the message **"Person Name should be 1 to 50 characters long"** is returned.

5. Validating the Gender Property

```
RuleFor(request => request.Gender)
 .IsInEnum().WithMessage("Invalid gender option");
```

- **Purpose**:
    - The Gender property must have a valid enum value from GenderOptions (i.e., Male, Female, or Others). If the gender is not part of the enum, an **"Invalid gender option"** message is returned.
- **IsInEnum()**:
    - This ensures that the provided gender value matches one of the allowed enum values. This validation is especially useful to prevent invalid or unexpected values from being passed.

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.

3. Asynchronous Validation

- While these validators are synchronous, FluentValidation supports **asynchronous validation** as well, which is useful when rules depend on external systems (like checking if an email already exists in the database).

Summary of Validations Code and Concepts

1. **Validators in Clean Architecture**:
    - Validators, defined in the **Core** layer, ensure that data entering the system is valid and follows business rules. By defining validation rules here, you decouple validation logic from the rest of the application, promoting reusability and testability.
2. **FluentValidation**:
    - FluentValidation provides a **fluent interface** for defining validation rules, making the code more readable and expressive. It allows for complex validation scenarios with ease.
3. **LoginRequestValidator**:
    - Ensures that the email and password in a login attempt are present and valid.
4. **RegisterRequestValidator**:
    - Validates user registration data, including checks for a valid email format, non-empty password, valid name length, and gender matching one of the allowed enum values.
5. **Best Practices**:
    - It's essential to place validation in the **Core** layer because this keeps business logic clean and ensures that the domain rules are applied consistently across the application.

Mappers in Core Layer (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.

What is AutoMapper?

AutoMapper is an object-object mapping library for .NET, used to automatically map properties between objects of different types. It eliminates the need for manually assigning properties from one object to another, especially when both objects have similar properties but are used for different purposes (e.g., DTOs vs Entities).

AutoMapper relies on configuration classes called **Profiles** to define how the mappings between source and destination types should occur.

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.

Mapper: ApplicationUser to AuthenticationResponse

The first mapper is for converting the **ApplicationUser** entity to an **AuthenticationResponse** DTO. This mapping is useful when, after user authentication, you want to send a response to the client with limited but relevant user data.

Code Example:

```
using AutoMapper;

using eCommerce.Core.DTO;

using eCommerce.Core.Entities;


namespace eCommerce.Core.Mappers;


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
```

```
    ;
 }
}
```

Conceptual Breakdown:

1. **CreateMap<ApplicationUser, AuthenticationResponse>**:
    - This line defines a mapping from the ApplicationUser entity to the AuthenticationResponse DTO. AutoMapper will automatically try to match properties with the same names between the source (ApplicationUser) and destination (AuthenticationResponse).
2. **ForMember Usage**:
    - **ForMember(dest => dest.UserID, opt => opt.MapFrom(src => src.UserID))**:
        - This explicitly defines that the UserID property of AuthenticationResponse will be mapped from the UserID property of ApplicationUser. Even though AutoMapper can infer this automatically for same-named properties, specifying it manually provides clarity.
    - **ForMember(dest => dest.Email, opt => opt.MapFrom(src => src.Email))**:
        - Similarly, the Email property from ApplicationUser is mapped to AuthenticationResponse.
    - **ForMember(dest => dest.Success, opt => opt.Ignore())**:
        - The Success field in AuthenticationResponse is not mapped from any property in ApplicationUser. This field will likely be set manually during the authentication process.
    - **ForMember(dest => dest.Token, opt => opt.Ignore())**:
        - The Token property is ignored, as it will be generated after authentication and is not part of the ApplicationUser entity.

Use Case:

- This mapping is commonly used after a user successfully logs in. The application retrieves the user details from the database (as an ApplicationUser object), maps it to AuthenticationResponse, and adds any other necessary information (like Success and Token) before sending it as a response.

Mapper: RegisterRequest to ApplicationUser

The second mapper is for converting a **RegisterRequest** DTO to an **ApplicationUser** entity. This is necessary when registering a new user, as the request data must be converted into an entity that can be stored in the database.

Code Example:

```
using AutoMapper;

using eCommerce.Core.DTO;

using eCommerce.Core.Entities;


namespace eCommerce.Core.Mappers;


public class RegisterRequestMappingProfile : Profile

{

  public RegisterRequestMappingProfile()

  {

    CreateMap<RegisterRequest, ApplicationUser>()

      .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.ToString()))  // Convert enum to string

      .ForMember(dest => dest.Password, opt => opt.MapFrom(src => src.Password));

  }

}
```

Conceptual Breakdown:

1. **CreateMap<RegisterRequest, ApplicationUser>**:
    - This line defines the mapping from the RegisterRequest DTO to the ApplicationUser entity. This is useful for converting registration data into an entity that can be saved to the database.
2. **ForMember Usage**:
    - **ForMember(dest => dest.Email, opt => opt.MapFrom(src => src.Email))**:
        - Maps the Email field of the RegisterRequest to the Email field of the ApplicationUser. This is straightforward since both properties have the same name and type.

- **ForMember(dest => dest.PersonName, opt => opt.MapFrom(src => src.PersonName))**:
    - Maps the PersonName field of the RegisterRequest to the PersonName field of the ApplicationUser.
- **ForMember(dest => dest.Gender, opt => opt.MapFrom(src => src.Gender.ToString()))**:
    - The Gender property in RegisterRequest is of type GenderOptions (an enum), but in ApplicationUser, it's stored as a string. This rule uses ToString() to convert the enum to its string representation.
- **ForMember(dest => dest.Password, opt => opt.MapFrom(src => src.Password))**:
    - The password is directly mapped from the DTO to the entity. In a real-world scenario, passwords should be hashed before storing them, but that logic would typically happen elsewhere, not in the mapping.

Use Case:

- When a user registers, the data submitted in a RegisterRequest is converted to an ApplicationUser entity, which is then saved to the database. This mapping ensures the DTO is properly translated into a format suitable for persistence.

Summary of AutoMappers Code and Concepts

1. **AutoMapper in Clean Architecture**:
    - AutoMapper simplifies the mapping process between different layers (DTOs, entities). It's especially useful when DTOs have properties with the same names as the entities they map to, allowing automatic mapping.
2. **Profiles in AutoMapper**:
    - **Profiles** are a way to organize mappings. Each Profile defines rules for converting between different types (e.g., ApplicationUser to AuthenticationResponse).
3. **Custom Mapping with ForMember**:
    - The ForMember() method is used to explicitly define how a property should be mapped, offering flexibility. Properties that don't need to be mapped (like computed properties) can be ignored using Ignore().
4. **ApplicationUser to AuthenticationResponse**:
    - The mapper converts ApplicationUser to AuthenticationResponse, ensuring that only necessary fields are returned to the client after a successful login.
5. **RegisterRequest to ApplicationUser**:
    - Converts RegisterRequest to ApplicationUser, preparing the data for storage during user registration.

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.

Code Walkthrough

Here's the complete code for UsersService, followed by a breakdown of each part.

```
using AutoMapper;

using eCommerce.Core.DTO;

using eCommerce.Core.Entities;

using eCommerce.Core.RepositoryContracts;

using eCommerce.Core.ServiceContracts;


namespace eCommerce.Core.Services;


internal class UsersService : IUsersService

{

 private readonly IUsersRepository _usersRepository;

 private readonly IMapper _mapper;


 public UsersService(IUsersRepository usersRepository, IMapper mapper)

 {

  _usersRepository = usersRepository;

  _mapper = mapper;

 }
```

```
  public async Task<AuthenticationResponse?> Login(LoginRequest loginRequest)
  {
    ApplicationUser? user = await _usersRepository.GetUserByEmailAndPassword(loginRequest.Email,
loginRequest.Password);

    if (user == null)
    {
      return null;
    }

    // Return AuthenticationResponse with success and token
    return _mapper.Map<AuthenticationResponse>(user) with { Success = true, Token = "token" };
  }

  public async Task<AuthenticationResponse?> Register(RegisterRequest registerRequest)
  {
    // Create ApplicationUser object from RegisterRequest using AutoMapper
    ApplicationUser user = _mapper.Map<ApplicationUser>(registerRequest);
    ApplicationUser? registeredUser = await _usersRepository.AddUser(user);

    if (registeredUser == null)
    {
      return null;
    }

    // Return AuthenticationResponse with success and token
    return _mapper.Map<AuthenticationResponse>(registeredUser) with { Success = true, Token =
"token" };
  }
}
```

Conceptual Breakdown

1. Constructor Injection

```
public UsersService(IUsersRepository usersRepository, IMapper mapper)
{
  _usersRepository = usersRepository;
  _mapper = mapper;
}
```

- **Dependency Injection**: The constructor injects two dependencies: IUsersRepository and IMapper. The repository handles data access, while AutoMapper handles object mapping.
- These dependencies are essential to keeping the service isolated from low-level logic (like database queries) and focusing purely on business rules.

2. Login Method

```
public async Task<AuthenticationResponse?> Login(LoginRequest loginRequest)
{
  ApplicationUser? user = await _usersRepository.GetUserByEmailAndPassword(loginRequest.Email, loginRequest.Password);

  if (user == null)
  {
    return null;
  }

  return _mapper.Map<AuthenticationResponse>(user) with { Success = true, Token = "token" };
}
```

- **LoginRequest**: The method accepts a LoginRequest DTO that contains the user's email and password. This DTO simplifies the request payload from the client.
- **Repository Call**:
  - It uses the IUsersRepository.GetUserByEmailAndPassword() method to fetch the user from the data store based on the credentials.
  - If no user is found (the credentials are invalid), it returns null.

- **Mapping**:
    - If the user is found, it uses AutoMapper to convert the ApplicationUser entity into an AuthenticationResponse DTO.
    - The with expression modifies the returned object by setting Success = true and generating a placeholder Token.
- **Token Generation**:
    - In a real-world scenario, this is where you'd generate a JWT or some other form of authentication token. For now, the placeholder "token" is used.

3. Register Method

```
public async Task<AuthenticationResponse?> Register(RegisterRequest registerRequest)
{
 // Create ApplicationUser object from RegisterRequest using AutoMapper
 ApplicationUser user = _mapper.Map<ApplicationUser>(registerRequest);
 ApplicationUser? registeredUser = await _usersRepository.AddUser(user);

 if (registeredUser == null)
 {
  return null;
 }

 return _mapper.Map<AuthenticationResponse>(registeredUser) with { Success = true, Token = "token" };
}
```

- **RegisterRequest**: Accepts a RegisterRequest DTO, which contains details like email, password, name, and gender.
- **AutoMapper Mapping**:
    - AutoMapper converts the RegisterRequest into an ApplicationUser entity. This conversion includes mapping properties like Email, Password, and PersonName.
- **Repository Call**:
    - The repository's AddUser() method is called to add the new user to the data store. If the user is successfully added, the method returns the registered user.
    - If registration fails for any reason (e.g., a database error), the method returns null.

- **Response Mapping**:
    - Similar to the Login method, the ApplicationUser is mapped to an AuthenticationResponse DTO, and the with expression sets Success = true and generates a placeholder Token.

Summary of Key Concepts of "Sevices" in "Core" layer

1. **Service Layer**:
    - Handles business logic and is responsible for orchestrating interactions between the repository and external consumers (controllers, clients).
2. **AutoMapper**:
    - Used to map between domain models (entities) and DTOs (for API communication), reducing boilerplate code.
3. **Repository Pattern**:
    - Repositories abstract the data access logic. The service uses these repositories to interact with the data store without worrying about the specifics of how data is stored or retrieved.
4. **DTOs**:
    - Used to encapsulate the data sent or received from the client. This layer of abstraction ensures that only necessary information is exposed to the client, enhancing security.
5. **Immutable Records and with Expressions**:
    - Records in C# are immutable, meaning once created, their properties can't be changed. However, you can create a new object with modified properties using the with expression.

By implementing services this way, we ensure that our application is maintainable, testable, and adheres to clean architecture principles.

Dependency Injection File in Core Layer

In the **Core** layer of a Clean Architecture solution, dependency injection (DI) is crucial for managing service lifetimes and dependencies. The DI configuration helps in setting up the application's services and ensuring they are available for use across different parts of the application.

The DependencyInjection class in the **Core** layer provides an extension method to configure and register services with the .NET Core dependency injection container. This allows you to inject services into other parts of the application, such as controllers or other services.

Key Concepts

1. Dependency Injection (DI)

- **Dependency Injection** is a design pattern used to implement IoC (Inversion of Control), where the control of dependencies is inverted from the class itself to an external container. This promotes loose coupling and improves testability.

2. Service Lifetime Management

- **Transient**: Services are created each time they are requested. Ideal for lightweight, stateless services.
- **Scoped**: Services are created once per request. Useful for services that should be shared within a single request.
- **Singleton**: Services are created once and shared across all requests. Best for stateless services that do not need to be recreated.

3. Adding Services to the DI Container

- **Adding Services**: You register services with the dependency injection container using methods like AddTransient, AddScoped, or AddSingleton.

Code Walkthrough

Here's the complete code for the DependencyInjection class in the Core layer, followed by a detailed explanation.

```
using eCommerce.Core.ServiceContracts;

using eCommerce.Core.Services;

using eCommerce.Core.Validators;

using FluentValidation;

using Microsoft.Extensions.DependencyInjection;


namespace eCommerce.Core;
```

```
public static class DependencyInjection
{
  /// <summary>
  /// Extension method to add core services to the dependency injection container
  /// </summary>
  /// <param name="services"></param>
  /// <returns></returns>
  public static IServiceCollection AddCore(this IServiceCollection services)
  {
    // TO DO: Add services to the IoC container
    // Core services often include validation, caching and other business components.

    // Registering the UsersService as a transient service
    services.AddTransient<IUsersService, UsersService>();

    // Registering FluentValidation validators from the assembly containing the LoginRequestValidator
    services.AddValidatorsFromAssemblyContaining<LoginRequestValidator>();

    return services;
  }
}
```

Conceptual Breakdown

1. Extension Method for IServiceCollection

```
public static class DependencyInjection
{
  public static IServiceCollection AddCore(this IServiceCollection services)
  {
    // Code to add services here
    return services;
  }
}
```

- **Static Class and Method**:
    - The DependencyInjection class is a static class, and AddCore is a static extension method for IServiceCollection. This is a common pattern for configuring DI in .NET Core applications.
- **Extension Method**:
    - The AddCore method is an extension method, which means it extends the IServiceCollection interface. This allows you to call it like an instance method on IServiceCollection.

2. Registering Services

```
services.AddTransient<IUsersService, UsersService>();
```

- **Service Registration**:
    - The AddTransient method registers UsersService with the DI container. This means that a new instance of UsersService will be created each time it is requested.
- **Interface to Implementation Mapping**:
    - IUsersService is the interface that UsersService implements. By registering this mapping, you ensure that whenever IUsersService is requested, an instance of UsersService will be provided.

3. Registering Validators

services.AddValidatorsFromAssemblyContaining<LoginRequestValidator>();

- **FluentValidation Integration**:
    - The AddValidatorsFromAssemblyContaining method registers all validators in the assembly containing the specified validator (in this case, LoginRequestValidator). This is a convenient way to add all validators without needing to register each one individually.
- **Validators**:
    - FluentValidation is used for validating data transfer objects (DTOs) and ensures that incoming data adheres to specified rules. By registering validators, you enable automatic validation within the application.

Summary of Key Concepts in "Dependency Injection" file in "Core" layer:

1. **Dependency Injection (DI)**:
    - A design pattern used to manage dependencies between classes, promoting loose coupling and testability.
2. **Service Lifetimes**:
    - Services can be registered with different lifetimes (Transient, Scoped, Singleton), affecting how and when instances are created.
3. **Extension Methods for IServiceCollection**:
    - Used to configure and register services with the DI container in a clean and modular way.
4. **AutoMapper and Validators**:
    - **AutoMapper** maps between DTOs and entities, simplifying data transformation.
    - **FluentValidation** provides a clean and expressive way to validate DTOs and other objects.

By following these practices, you ensure that your application's dependencies are well-managed, and its services are easy to maintain and test.

DbContext in Infrastructure Layer

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

Key Concepts

1. Dapper and Npgsql

- **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.
- **Npgsql**:
    - A .NET data provider for PostgreSQL. It allows .NET applications to interact with PostgreSQL databases.

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.

```
using Microsoft.Extensions.Configuration;
using Npgsql;
using System.Data;

namespace eCommerce.Infrastructure.DbContext;

public class DapperDbContext
{
  private readonly IConfiguration _configuration;
  private readonly IDbConnection _connection;

  public DapperDbContext(IConfiguration configuration)
  {
    _configuration = configuration;
    string? connectionString = _configuration.GetConnectionString("PostgresConnection");

    // Create a new NpgsqlConnection with the retrieved connection string
    _connection = new NpgsqlConnection(connectionString);
  }

  public IDbConnection DbConnection => _connection;
}
```

Conceptual Breakdown

1. Constructor and Configuration

```
public DapperDbContext(IConfiguration configuration)
{
 _configuration = configuration;
 string? connectionString = _configuration.GetConnectionString("PostgresConnection");

 // Create a new NpgsqlConnection with the retrieved connection string
 _connection = new NpgsqlConnection(connectionString);
}
```

- **Constructor Injection**:
    - The constructor accepts an IConfiguration instance, which is injected via dependency injection. This instance provides access to configuration settings, including connection strings.
- **Retrieving Connection String**:
    - GetConnectionString("PostgresConnection") retrieves the connection string named "PostgresConnection" from the configuration. This string is used to establish a connection to the PostgreSQL database.
- **NpgsqlConnection**:
    - NpgsqlConnection is created using the connection string. This object manages the connection to the PostgreSQL database and is used for executing queries and commands.

2. DbConnection Property

```
public IDbConnection DbConnection => _connection;
```

- **IDbConnection Interface**:
    - The IDbConnection interface is a common abstraction for database connections in ADO.NET. By exposing _connection as IDbConnection, the class adheres to a common interface, promoting flexibility and testability.
- **Property Getter**:
    - The DbConnection property provides access to the underlying database connection. This allows other components (e.g., repositories or services) to use this connection for database operations without directly managing connection details.

Summary of Key Concepts in DbContext

1. **Dapper and Npgsql**:
    - **Dapper** is used for lightweight data access, and **Npgsql** is the data provider for PostgreSQL.
2. **DbContext Pattern**:
    - Manages database connections and provides an abstraction for data access, regardless of whether you use Entity Framework or another data access technology.
3. **IConfiguration**:
    - Used to retrieve configuration settings, such as database connection strings, from various sources.
4. **IDbConnection Interface**:
    - Provides a common abstraction for database connections, allowing for flexibility and testability.

By using the DapperDbContext class, you efficiently manage your database connections and maintain separation of concerns between data access and business logic. This approach aligns with Clean Architecture principles, promoting a modular and maintainable codebase.

Repository in Infrastructure Layer

In the **Infrastructure** layer, the UsersRepository class handles data access logic, interfacing with the database using **Dapper**. This class implements the IUsersRepository contract, defining methods for interacting with the ApplicationUser entities.

Key Concepts

1. Dapper

- **Dapper**:
    - A micro ORM (Object-Relational Mapper) that provides a way to interact with the database using SQL queries while mapping results to objects.

2. Repository Pattern

- **Repository Pattern**:
    - A design pattern that abstracts data access logic, providing a clean API for accessing data and separating the data layer from the business logic.

3. Dependency Injection

- **Dependency Injection (DI)**:
    - Used to provide the DapperDbContext instance to the repository, ensuring loose coupling and promoting testability.

Code Walkthrough

Here's the complete code for the UsersRepository class, followed by a detailed explanation.

```
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;
  }
}
```

Conceptual Breakdown

1. Constructor and Dependency Injection

```
public UsersRepository(DapperDbContext dbContext)
{
  _dbContext = dbContext;
}
```

- **Constructor Injection**:
    - The constructor accepts an instance of DapperDbContext, which is injected via DI. This allows the repository to use the database connection provided by DapperDbContext.

2. Adding a User

```
public async Task<ApplicationUser?> AddUser(ApplicationUser user)
{
  user.UserID = Guid.NewGuid(); // Generate a new unique user ID

  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;
  }
}
```

- **Generate Unique ID**:
    - user.UserID = Guid.NewGuid(); generates a new GUID for the UserID, ensuring each user has a unique identifier.

- **SQL Query**:
    - The SQL INSERT query adds a new user to the "Users" table. The @UserID, @Email, @PersonName, @Gender, and @Password parameters are replaced with the values from the user object.
- **ExecuteAsync**:
    - ExecuteAsync executes the SQL command asynchronously. The method returns the number of rows affected. If greater than 0, it indicates success, and the user object is returned; otherwise, null is returned.

3. Retrieving a User

```
public async Task<ApplicationUser?> GetUserByEmailAndPassword(string? email, string? 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;

}
```

- **SQL Query**:
    - The SQL SELECT query retrieves a user based on their Email and Password. The parameters @Email and @Password are replaced with the values from the parameters object.
- **QueryFirstOrDefaultAsync**:
    - QueryFirstOrDefaultAsync executes the query asynchronously and returns the first matching record or null if no match is found.

Summary of Key Concepts of Repository

1. **Dapper**:
    - A micro ORM that simplifies data access with SQL queries, offering high performance and ease of use.
2. **Repository Pattern**:
    - Provides a clean API for data access and abstracts data operations, promoting separation of concerns.
3. **Dependency Injection (DI)**:
    - Ensures loose coupling and testability by providing necessary dependencies to classes.
4. **SQL Operations**:
    - **Insert**: Adds new records to the database.
    - **Select**: Retrieves records based on specific criteria.

By using the UsersRepository class, you efficiently manage data access operations for user entities, keeping data-related logic separate from business logic and aligning with Clean Architecture principles.

Dependency Injection in Infrastructure Layer

In the **Infrastructure** layer, the DependencyInjection class is responsible for registering services with the dependency injection (DI) container. This setup allows various components of the application to obtain the required services through DI, promoting modularity and testability.

Key Concepts

1. Dependency Injection (DI)

- **Dependency Injection**:
    - A design pattern that allows objects to receive their dependencies from an external source rather than creating them internally. This promotes loose coupling, modularity, and ease of testing.

2. Service Registration

- **Service Registration**:
    - The process of adding services to the DI container. These services can then be injected into other components, such as controllers or services, at runtime.

3. Service Lifetimes

- **Service Lifetimes**:
    - **Transient**: Services are created each time they are requested. Used for lightweight, stateless services.
    - **Scoped**: Services are created once per request. Ideal for services that need to maintain state during a single request.
    - **Singleton**: A single instance is created and shared throughout the application's lifetime. Suitable for services that are expensive to create or need to maintain a single state.

Code Walkthrough

Here's the complete code for the DependencyInjection class in the Infrastructure layer, followed by a detailed explanation.

```
using eCommerce.Core.RepositoryContracts;

using eCommerce.Infrastructure.DbContext;

using eCommerce.Infrastructure.Repositories;

using Microsoft.Extensions.DependencyInjection;


namespace eCommerce.Infrastructure;


public static class DependencyInjection

{

 /// <summary>

 /// Extension method to add infrastructure services to the dependency injection container

 /// </summary>

 /// <param name="services"></param>

 /// <returns></returns>
```

```
 public static IServiceCollection AddInfrastructure(this IServiceCollection services)
 {
  // TO DO: Add services to the IoC container
  // Infrastructure services often include data access, caching and other low-level components.

  services.AddTransient<IUsersRepository, UsersRepository>();

  services.AddTransient<DapperDbContext>();

  return services;
 }
}
```

Conceptual Breakdown

1. Extension Method

```
public static IServiceCollection AddInfrastructure(this IServiceCollection services)
{
 // TO DO: Add services to the IoC container
 // Infrastructure services often include data access, caching and other low-level components.
}
```

- **Extension Method**:
    - This method is defined as a static method in a static class. It extends IServiceCollection, allowing the addition of infrastructure-specific services to the DI container.

2. Registering Services

services.AddTransient<IUsersRepository, UsersRepository>();

- **Service Registration**:
    - AddTransient<IUsersRepository, UsersRepository>() registers the UsersRepository class as an implementation of the IUsersRepository interface. This means whenever IUsersRepository is requested, an instance of UsersRepository will be provided.
- **Transient Lifetime**:
    - This specifies that a new instance of UsersRepository will be created each time it is requested.

services.AddTransient<DapperDbContext>();

- **DapperDbContext Registration**:
    - AddTransient<DapperDbContext>() registers DapperDbContext as a transient service. This means a new instance of DapperDbContext will be created each time it is needed.

3. Returning IServiceCollection

return services;

- **Fluent Configuration**:
    - Returning the IServiceCollection instance allows for chaining multiple service registrations and configurations within the DI setup.

Summary of Key Concepts of Dependency Injection in Infrastructure Layer

1. **Dependency Injection (DI)**:
    - A design pattern that promotes loose coupling and modularity by injecting dependencies rather than creating them internally.
2. **Service Registration**:
    - The process of adding services to the DI container to be injected where needed.
3. **Service Lifetimes**:
    - **Transient**: Services created each time requested.
    - **Scoped**: Services created once per request.
    - **Singleton**: A single shared instance throughout the application's lifetime.
4. **Extension Method**:
    - Allows the addition of services to IServiceCollection, enabling a modular and organized setup of service registrations.

By using the DependencyInjection class, you ensure that your infrastructure services, such as repositories and data contexts, are correctly registered and managed by the DI container, supporting the Clean Architecture principles and enhancing the maintainability of your application.

Program.cs File in API Layer

The Program.cs file in an ASP.NET Core application configures and sets up the application's services, middleware, and request pipeline. It is essential for initializing the application and specifying how various components interact with each other.

Key Concepts

1. WebApplication

- **WebApplication**:
    - A class that provides a simplified way to configure and build an ASP.NET Core application. It consolidates the functionalities of IHostBuilder and IWebHostBuilder into one class.

2. Service Registration

- **Service Registration**:
    - Adding services to the DI container, which are then available for dependency injection throughout the application.

3. Middleware

- **Middleware**:
    - Components that handle HTTP requests and responses in the request pipeline. Middleware can handle tasks like error handling, authentication, and logging.

4. Swagger/OpenAPI

- **Swagger/OpenAPI**:
    - Tools for generating interactive API documentation and specifications.

5. CORS

- **CORS (Cross-Origin Resource Sharing)**:
    - A security feature that controls how resources are shared between different origins (domains).

Code Walkthrough

Here's a detailed explanation of each section of the Program.cs file.

```
using eCommerce.Infrastructure;

using eCommerce.Core;

using eCommerce.API.Middlewares;

using System.Text.Json.Serialization;

using eCommerce.Core.Mappers;

using FluentValidation.AspNetCore;


var builder = WebApplication.CreateBuilder(args);
```

- **WebApplication.CreateBuilder**:
    - Initializes a new instance of the WebApplicationBuilder class, which is used to configure and build the web application. args represents command-line arguments passed to the application.

```
// Add Infrastructure services

builder.Services.AddInfrastructure();

builder.Services.AddCore();
```

- **Service Registration**:
    - AddInfrastructure and AddCore methods register services from the Infrastructure and Core layers, respectively, with the DI container.

```
// Add controllers to the service collection

builder.Services.AddControllers().AddJsonOptions(options => {

  options.JsonSerializerOptions.Converters.Add(new JsonStringEnumConverter());

});
```

- **AddControllers**:
    - Registers controllers with the DI container and sets up the MVC framework.

- **AddJsonOptions**:
    - Configures JSON serialization options for the application. JsonStringEnumConverter ensures that enums are serialized as their string representations instead of numeric values.

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

- **AddAutoMapper**:
    - Registers AutoMapper with the DI container. This configuration scans the assembly for mapping profiles, such as ApplicationUserMappingProfile, to set up object-to-object mappings.

// FluentValidations

builder.Services.AddFluentValidationAutoValidation();

- **AddFluentValidationAutoValidation**:
    - Registers FluentValidation and configures it to automatically validate models based on the validators defined in the application.

// Add API explorer services

builder.Services.AddEndpointsApiExplorer();

- **AddEndpointsApiExplorer**:
    - Adds services required for endpoint exploration, enabling API discovery and documentation.

// Add swagger generation services to create swagger specification

builder.Services.AddSwaggerGen();

- **AddSwaggerGen**:
    - Registers Swagger services to generate API documentation and interactive UI. Swagger helps in creating a visual representation of the API endpoints and their details.

```
// Add cors services
builder.Services.AddCors(options =>
{
  options.AddDefaultPolicy(builder => {
    builder.WithOrigins("http://localhost:4200")
    .AllowAnyMethod()
    .AllowAnyHeader();
  });
});
```

- **AddCors**:
    - Configures Cross-Origin Resource Sharing (CORS) policies. The default policy allows requests from http://localhost:4200 (commonly used for Angular applications) and permits any HTTP method and header.

```
// Build the web application
var app = builder.Build();
```

- **Build**:
    - Builds the WebApplication instance from the configuration defined in the WebApplicationBuilder.

```
app.UseExceptionHandlingMiddleware();
```

- **UseExceptionHandlingMiddleware**:
    - Adds custom exception handling middleware to the request pipeline. This middleware handles exceptions globally and provides appropriate responses or logs.

// Routing

app.UseRouting();

app.UseSwagger(); // Adds endpoint that can serve the swagger.json

app.UseSwaggerUI(); // Adds Swagger UI (interactive page to explore and test API endpoints)

app.UseCors();

- **UseRouting**:
    - Adds routing middleware to the request pipeline, enabling routing based on the request URL.
- **UseSwagger**:
    - Adds middleware to serve the Swagger specification (swagger.json) endpoint.
- **UseSwaggerUI**:
    - Adds middleware to serve the Swagger UI, providing an interactive interface for testing and exploring the API.
- **UseCors**:
    - Adds CORS middleware to apply the configured CORS policy to incoming requests.

// Auth

app.UseAuthentication();

app.UseAuthorization();

- **UseAuthentication**:
    - Adds authentication middleware to the pipeline, enabling user authentication.
- **UseAuthorization**:
    - Adds authorization middleware to the pipeline, enforcing authorization policies.

// Controller routes

app.MapControllers();

- **MapControllers**:
    - Maps controller endpoints to routes, enabling the application to respond to HTTP requests routed to controllers.

app.Run();

- **Run**:
    - Starts the web application and begins listening for incoming HTTP requests.

Summary of Key Concepts of Program.cs file in "API" layer

1. **WebApplication**:
    - Simplified class for building and configuring an ASP.NET Core application.
2. **Service Registration**:
    - Configuring services and middleware in the DI container.
3. **Middleware**:
    - Components that handle requests and responses in the pipeline.
4. **Swagger/OpenAPI**:
    - Tools for API documentation and interactive testing.
5. **CORS**:
    - Security feature controlling resource sharing between different origins.
6. **FluentValidation**:
    - Library for automatically validating models based on defined rules.

By configuring services, middleware, and routes in the Program.cs file, you set up a comprehensive request pipeline and ensure that the application components are correctly wired together.

appsettings.json File in API Layer

The appsettings.json file is a configuration file used in ASP.NET Core applications to manage settings and configurations for different aspects of the application. It is a JSON file where you can specify various settings like logging, connection strings, application settings, etc.

Key Concepts

1. Configuration System

- **Configuration System**:
    - ASP.NET Core uses a flexible configuration system that allows you to load settings from various sources, including JSON files, environment variables, and command-line arguments.

2. Logging Configuration

- **Logging**:
    - ASP.NET Core provides built-in logging services. You can configure log levels for different categories to control the verbosity of logging output.

3. Connection Strings

- **Connection Strings**:
    - A connection string specifies how the application connects to a database. It includes details like the server address, database name, and authentication credentials.

Code Walkthrough

Here's a detailed explanation of each section of the appsettings.json file.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings":
```

```
 {
   "PostgresConnection": "Host=localhost; Port=5432; Database=eCommerceUsers;
Username=postgres; Password=admin"
 }
}
```

1. Logging Configuration

```
"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft.AspNetCore": "Warning"
  }
}
```

- **Logging**:
  - Configures the logging system for the application.
- **LogLevel**:
  - Specifies the minimum log level for different categories of logs.
  - **Default**:
    - Sets the default log level to Information, meaning all logs of level Information and above (e.g., Warning, Error) will be recorded.
  - **Microsoft.AspNetCore**:
    - Sets the log level for the Microsoft.AspNetCore namespace to Warning. This means only warnings and errors from ASP.NET Core will be logged, reducing the verbosity compared to the default setting.

2. Allowed Hosts

```
"AllowedHosts": "*"
```

- **AllowedHosts**:
  - Specifies which hosts are allowed to access the application. The wildcard * allows any host to make requests to the application, which is suitable for development environments but should be restricted in production for security reasons.

3. Connection Strings

"ConnectionStrings":

{

"PostgresConnection": "Host=localhost; Port=5432; Database=eCommerceUsers; Username=postgres; Password=admin"

}

- **ConnectionStrings**:
    - Contains the connection string settings for various databases used by the application.
- **PostgresConnection**:
    - Defines the connection string for connecting to a PostgreSQL database.
    - **Host**:
        - Specifies the database server's address (localhost means the server is running on the same machine as the application).
    - **Port**:
        - The port number used by the PostgreSQL server (5432 is the default port for PostgreSQL).
    - **Database**:
        - The name of the database to connect to (eCommerceUsers in this case).
    - **Username**:
        - The username used to authenticate with the database (postgres).
    - **Password**:
        - The password associated with the username (admin).

Summary of Key Concepts of appsettings.json

1. **Configuration System**:
    - ASP.NET Core's flexible system for managing application settings from multiple sources.
2. **Logging**:
    - Controls the verbosity of logs based on categories and severity levels.
3. **Connection Strings**:
    - Specifies how the application connects to databases, including server details and authentication credentials.

The appsettings.json file centralizes configuration settings, making it easier to manage and adjust various aspects of the application without changing code.

ExceptionHandlingMiddleware in API Layer

The ExceptionHandlingMiddleware class is a custom middleware designed to handle exceptions that occur during the processing of HTTP requests in an ASP.NET Core application. This middleware captures exceptions, logs them, and provides a consistent error response to clients.

Key Concepts

1. Middleware

- **Middleware**:
    - Components in the request pipeline that handle requests and responses. They are used to perform operations like logging, authentication, exception handling, etc.
- **RequestDelegate**:
    - Represents a method that can process HTTP requests. It is used by middleware to call the next component in the pipeline.

2. Exception Handling

- **Exception Handling**:
    - The process of catching and managing exceptions to prevent application crashes and provide meaningful error responses.
- **Logging**:
    - Capturing and recording details of exceptions for debugging and monitoring purposes.

Code Walkthrough

ExceptionHandlingMiddleware Class

```
using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Http;
```

```
using System.Threading.Tasks;

namespace eCommerce.API.Middlewares;

public class ExceptionHandlingMiddleware
{
  private readonly RequestDelegate _next;
  private readonly ILogger<ExceptionHandlingMiddleware> _logger;

  public ExceptionHandlingMiddleware(RequestDelegate next,
ILogger<ExceptionHandlingMiddleware> logger)
  {
    _next = next;
    _logger = logger;
  }

  public async Task Invoke(HttpContext httpContext)
  {
    try
    {
      await _next(httpContext);
    }
    catch (Exception ex)
    {
      //Log the exception type and message
      _logger.LogError($"{ex.GetType().ToString()}: {ex.Message}");

      if (ex.InnerException is not null)
      {
        //Log the inner exception type and message
        _logger.LogError($"{ex.InnerException.GetType().ToString()}: {ex.InnerException.Message}");
```

```
    }


    httpContext.Response.StatusCode = 500; //Internal Server Error

    await httpContext.Response.WriteAsJsonAsync(new { Message = ex.Message, Type =
ex.GetType().ToString() });

   }

  }

}
```

- **Constructor**:

  ```
  public ExceptionHandlingMiddleware(RequestDelegate next,
  ILogger<ExceptionHandlingMiddleware> logger)

  {

   _next = next;

   _logger = logger;

  }
  ```

  - **RequestDelegate next**:
    - Represents the next delegate (middleware) in the pipeline.
  - **ILogger<ExceptionHandlingMiddleware> logger**:
    - Provides logging capabilities specific to this middleware.

- **Invoke Method**:

  ```
  public async Task Invoke(HttpContext httpContext)

  {

   try

   {

    await _next(httpContext);

   }

   catch (Exception ex)

   {

    //Log the exception type and message

    _logger.LogError($"{ex.GetType().ToString()}: {ex.Message}");
  ```

```
    if (ex.InnerException is not null)
    {
      //Log the inner exception type and message
      _logger.LogError($"{ex.InnerException.GetType().ToString()}:
{ex.InnerException.Message}");
    }

    httpContext.Response.StatusCode = 500; //Internal Server Error
    await httpContext.Response.WriteAsJsonAsync(new { Message = ex.Message, Type =
ex.GetType().ToString() });
  }
}
```

- **Try-Catch Block**:
  - **await _next(httpContext);**:
    - Calls the next middleware in the pipeline.
  - **Catch Block**:
    - Catches any exceptions thrown by the subsequent middlewares or request handlers.
    - **Logging**:
      - Logs the type and message of the exception and any inner exceptions.
    - **Response**:
      - Sets the HTTP response status code to 500 (Internal Server Error).
      - Writes a JSON response containing the exception message and type.

ExceptionHandlingMiddlewareExtensions Class

```
public static class ExceptionHandlingMiddlewareExtensions
{
  public static IApplicationBuilder UseExceptionHandlingMiddleware(this IApplicationBuilder builder)
  {
    return builder.UseMiddleware<ExceptionHandlingMiddleware>();
  }
}
```

- **Extension Method**:
    - Provides a convenient way to add the ExceptionHandlingMiddleware to the HTTP request pipeline.
    - **UseExceptionHandlingMiddleware Method**:
        - **this IApplicationBuilder builder**:
            - Allows extension methods to be called on IApplicationBuilder.
        - **builder.UseMiddleware<ExceptionHandlingMiddleware>()**:
            - Adds the ExceptionHandlingMiddleware to the pipeline.

Summary of Key Concepts of ExceptionHandlingMiddleware

1. **Middleware**:
    - Handles HTTP requests and responses, and can be used for cross-cutting concerns like exception handling.
2. **Exception Handling**:
    - Catches and manages exceptions to prevent application crashes and provide meaningful error responses.
3. **Logging**:
    - Captures details of exceptions for debugging and monitoring.
4. **Extension Methods**:
    - Provide a convenient way to configure and add middleware to the request pipeline.

The ExceptionHandlingMiddleware ensures that exceptions are logged and handled gracefully, improving the reliability and maintainability of the application.

AuthController in API Layer

The AuthController class is an ASP.NET Core MVC controller responsible for handling authentication-related API requests, such as user registration and login. It interacts with the IUsersService to process these requests and return appropriate responses.

Concepts

1. ASP.NET Core Controllers

- **Controller**:
    - A class that handles HTTP requests and returns responses. It typically interacts with services and repositories to process business logic.
- **ControllerBase**:
    - A base class for controllers that do not require views. It provides methods for creating HTTP responses.
- **ApiController Attribute**:
    - Indicates that the controller responds to web API requests. It also enables some conventions like automatic model validation.
- **Route Attribute**:
    - Specifies the URL pattern for the controller's actions. In this case, [Route("api/[controller]")] maps to api/auth.

2. Dependency Injection

- **Constructor Injection**:
    - The controller's constructor receives an instance of IUsersService, which is injected by ASP.NET Core's dependency injection system.

3. HTTP Actions

- **IActionResult**:
    - Represents the result of an action method. It allows returning different types of HTTP responses.
- **HttpPost Attribute**:
    - Specifies that the action method responds to HTTP POST requests.
- **Status Codes**:
    - BadRequest(400): Indicates that the request is invalid.
    - Unauthorized(401): Indicates that authentication is required or invalid.
    - Ok(200): Indicates that the request was successful.

Code Walkthrough

AuthController Class

```
using eCommerce.Core.DTO;
using eCommerce.Core.ServiceContracts;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace eCommerce.API.Controllers
{
 [Route("api/[controller]")] //api/auth
 [ApiController]
 public class AuthController : ControllerBase
 {
  private readonly IUsersService _usersService;

  public AuthController(IUsersService usersService)
  {
   _usersService = usersService;
  }
```

```
    //Endpoint for user registration use case
    [HttpPost("register")] //POST api/auth/register
    public async Task<IActionResult> Register(RegisterRequest registerRequest)
    {
      //Check for invalid registerRequest
      if (registerRequest == null)
      {
        return BadRequest("Invalid registration data");
      }

      //Call the UsersService to handle registration
      AuthenticationResponse? authenticationResponse = await
_usersService.Register(registerRequest);

      if (authenticationResponse == null || authenticationResponse.Success == false)
      {
        return BadRequest(authenticationResponse);
      }

      return Ok(authenticationResponse);
    }

    //Endpoint for user login use case
    [HttpPost("login")]
    public async Task<IActionResult> Login(LoginRequest loginRequest)
    {
      //Check for invalid LoginRequest
      if (loginRequest == null)
      {
        return BadRequest("Invalid login data");
```

```
    }

    AuthenticationResponse? authenticationResponse = await _usersService.Login(loginRequest);

    if (authenticationResponse == null || authenticationResponse.Success == false)
    {
      return Unauthorized(authenticationResponse);
    }

    return Ok(authenticationResponse);
   }
  }
}
```

- **Constructor**:

  ```
  public AuthController(IUsersService usersService)
  {
   _usersService = usersService;
  }
  ```

  - **IUsersService usersService**:
    - Injected service for user-related operations, such as registration and login.

- **Register Method**:

  ```
  [HttpPost("register")] //POST api/auth/register
  public async Task<IActionResult> Register(RegisterRequest registerRequest)
  {
   //Check for invalid registerRequest
   if (registerRequest == null)
   {
    return BadRequest("Invalid registration data");
   }
  ```

```
 //Call the UsersService to handle registration
 AuthenticationResponse? authenticationResponse = await
_usersService.Register(registerRequest);


 if (authenticationResponse == null || authenticationResponse.Success == false)
 {
  return BadRequest(authenticationResponse);
 }


 return Ok(authenticationResponse);
}
```

- **[HttpPost("register")]**:
  - Maps this method to handle POST requests at api/auth/register.
- **Check for null**:
  - Ensures that the registerRequest object is not null before proceeding.
- **Service Call**:
  - Calls IUsersService.Register() to process registration and receive an AuthenticationResponse.
- **Response Handling**:
  - Returns BadRequest if registration fails or the response indicates failure.
  - Returns Ok with the AuthenticationResponse if registration succeeds.

- **Login Method**:

```
[HttpPost("login")]
public async Task<IActionResult> Login(LoginRequest loginRequest)
{
 //Check for invalid LoginRequest
 if (loginRequest == null)
 {
  return BadRequest("Invalid login data");
 }
```

```
 AuthenticationResponse? authenticationResponse = await
_usersService.Login(loginRequest);


 if (authenticationResponse == null || authenticationResponse.Success == false)
 {
  return Unauthorized(authenticationResponse);
 }


 return Ok(authenticationResponse);
}
```

- **[HttpPost("login")]**:
  - Maps this method to handle POST requests at api/auth/login.
- **Check for null**:
  - Ensures that the loginRequest object is not null before proceeding.
- **Service Call**:
  - Calls IUsersService.Login() to authenticate the user and receive an AuthenticationResponse.
- **Response Handling**:
  - Returns Unauthorized if login fails or the response indicates failure.
  - Returns Ok with the AuthenticationResponse if login succeeds.

Summary of Key Concepts of AuthController

1. **Controllers**:
   - Handle HTTP requests and return responses, typically interacting with services.
2. **Dependency Injection**:
   - Allows for the injection of services into controllers, promoting loose coupling and testability.
3. **HTTP Actions**:
   - Use attributes like [HttpPost] to map HTTP requests to action methods.
   - Return IActionResult to provide various HTTP responses.

4. **Validation**:
    - Check for null or invalid data before processing requests to ensure that only valid data is handled.

The AuthController provides endpoints for user registration and login, utilizing dependency injection and service contracts to manage authentication logic and ensure consistent error handling and responses.