

Thesis Dissertation

**A Generic Architecture & Base Software for Application
Development:
Employee Management System as a Proof of Concept**

Stylianos Adamou

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

May 2024

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

**A Generic Architecture & Base Software for Application
Development**

Stylianos Adamou

Supervisor

Dr. Elpida Keravnou-Papailiou

Thesis was submitted for partial fulfilment of the requirements for the award with the degree
of Bachelor in Computer Science at University of Cyprus

Μάιος 2024

Ευχαριστίες

I would like to express my sincere thanks and gratitude to all those who helped me in the preparation of my thesis. Especially Dr. Elpida Keravnu-Papailiou for her patience and cooperation during the writing of my thesis.

Περίληψη

Building Modern Software Applications is a complex procedure that requires a lot of background in the Computer Science Field. Nowadays, Web Applications require at least a basic knowledge in Software Development, Object-Oriented Programming, Security, Databases, Data Structures and Web Technologies. This diploma thesis investigates a software development architecture having a specific set of requirements that can help to extend it and thus create a generic architecture for all future applications development. Building a generic architecture will force developers inside a team to follow specific code rules and make the development process easier with guidelines on how to do a task and where to place each part of the code. Following this generic architecture, we will focus on the backend technologies to build a base software that integrates multiple of the latest and most modern technologies recommended by Microsoft and create a reusable software for future applications. An Employee Management System will then use this base software as the starting point of the development. Features related with the Employee Management System will then be added to this base software again by aligning with the principles of the generic architecture. Overall, this research contributes to produce a starting point for software developers that want to use some of the latest and Microsoft recommended technologies, frameworks, patterns, and libraries but also to force developers to work under a specific generic architecture for better maintenance of the applications and write cleaner code under rules.

Table of Contents

Chapter 1	Introduction.....	7
	1.1 Introduction	7
	1.2 Problem Definition	7
	1.3 Purpose of the research	8
	1.4 Research questions	9
	1.5 Chapters Summary	9
Chapter 2	Architecture	10
	2.1 Clean Architecture	10
	2.1.1 The Principles of Clean Architecture	12
	2.1.2 Our Clean Architecture Layers	13
	2.2 SOLID Principles	15
	2.3 Design Patterns	22
	2.3.1 Mediator Pattern	22
	2.3.2 CQRS Pattern	23
	2.4. SOLID Principles & Our Clean Architecture	26
Chapter 3	Frameworks and Libraries	31
	3.1 C#	31
	3.2 Angular	31
	3.3 SQL Server	32
	3.4 ASP.NET Core Framework	32
	3.4.1 Minimal APIs	34
	3.4.2 Middlewares	35
	3.4.3 Filters	39
	3.4.4 Authentication and Authorization	39
	3.4.5 Net Libraries	43
	3.4.5.1 MediatR	43
	3.4.5.2 Fluent Validation	46
	3.4.5.3 Dapper	52

Chapter 4	Application Design.....	54
	4.1 Database Design	54
	4.1.1 Schemas	54
	4.1.2 Tables	55
	4.1.3 Database Diagrams	59
	4.2 Clean Architecture in .Net Projects Structure	61
	4.2.1 Infrastructure Layer	61
	4.2.2 Domain Layer	62
	4.2.3 Application Layer	64
	4.2.4 Presentation Layer	66
	4.3 Code Decisions	68
	4.3.1 Domain Layer	68
	4.3.2 Infrastructure layer	69
	4.3.3 Presentation Layer	71
 Chapter 5	 Employee Management System User Interface.....	 77
	5.1 User Interface	77
 Chapter 6	 Conclusion.....	 89
	6.1 Evaluation	89
	6.2 Future Work	90
	6.3 Feedback	90
	6.4 Conclusion	91
 References		 92

Chapter 1

Introduction

1.1 Introduction	7
1.2 Problem Definition	7
1.3 Purpose of the research	8
1.4 Research Questions	9

1.1 Introduction

In today's fast-paced digital world, technology plays a significant role in simplifying our lives and transforming the way we work, communicate, and manage daily tasks. Modern software applications are indispensable tools in both personal and professional lives. Generally, applications aim to address the inefficiencies of manual processes and provide a scalable and user-friendly solution for various problems. For example, using applications for managing resources, projects, personnel can help businesses operate smoothly and efficiently. Technology progress offers a variety of tools and frameworks to create solutions solving different business problems.

1.2 Problem Definition

The primary objective of this research is to address the current inefficiencies faced by most of the business companies. To start with, absence of a generic architecture that proposes a set of rules and steps for developers to follow when implementing features related to a Web API app was raising issues in maintainability and extension of a system. Another problem is the lack of a reusable base software project to start an application for faster development initialization. Implementing the same common functionalities for different projects such as authentication, authorization, logging, error-handling, validation for each future project slows down the beginning of new application not to mention the fact that a developer may forget how to he/she did it before a long period. Integration of the same technologies, libraries, frameworks used

from a team to build a Web API can also be time-consuming extending this way for each new project the start of application-specific features implementation. Therefore, we can conclude that without having a generic architecture based on which we can build a reusable base software containing common features for different future projects delays the development initialization without letting teams to focus only on the business logic of the new project. Last, team leaders and HR departments struggle with managing employees and their roles in different projects manually through Excel files, which is both time-consuming and prone to errors.

1.3 Purpose of the research

This diploma thesis targets to face these issues by investigating for an architecture that can be extended to create a generic one while also creating a reusable base software following the principles of the proposed generic architecture. Then, the reusable base software will be used as the starting point of the development for each new project with different business logic. That is how Employee Management System will be used as a proof of concept that architecture is generic but also that base software can be reused in new projects to initiate development. Therefore, having this base software we will implement Employee Management System application-specific business features following again the principles of the generic Architecture. It's very important to follow the rules proposed from the generic architecture in order that technical choices made on the base software can be used without affecting the business rules of a different project separating this way technologies from business.

This diploma thesis will start the research trying to detect for an architecture that can be extended to create our generic architecture. This requires from the architecture to provide concepts like the separation of application-specific business logic from the technologies, libraries, frameworks used in an application and easy integration, or replacement of technical details used. Moreover, is crucial for the desired architecture to provide code rules and steps for developer to follow on how to implement a task through object-oriented design. Then, the research will concentrate to create a base software saved in an internal company repository so different team building Web APIs can clone it for each new project to start development ensuring consistency across all teams within a company. This research aims to develop modern software that integrates essential frameworks like ASP.NET Core, SQL Server, libraries such as FluentValidation, Swashbuckle, JwtBearer, MediatR, Dapper while adhering design patterns like Mediator and CQRS. Note that all the previous technologies are recommended by Microsoft. As we said before, this research will be validated using the Employee Management

System. This research is significant because there is no existing base software repository that combines these specific technologies and common features for different projects as explained before into a single ASP.NET Core application.

1.4 Research Questions

The following are the research questions answered in this thesis:

1. Can we combine all those frameworks, libraries and common features into a base software without violating the generic architecture and the design patterns?
2. Can we create a base software that can be used for all future projects with different business logic?

1.5 Chapters Summary

In Chapter 2, the thesis tries to find an architecture that fit our requirements and after that it creates a new generic architecture that is based on the architecture selected.

In Chapter 3, the research concentrates on building the base software system by integrating frameworks libraries and common features between different projects like authentication, authorization, validation, logging, exception, handling and more other. While integrating those technologies, we ensure that we do not violate our generic architecture as proposed in Chapter 2.

In Chapter 4, we use Employee Management System as proof of concept that our architecture is generic, and that base software can be used along with the application features without business rules getting affected. It proves that our architecture is generic by providing a set of steps that each new project can follow to implement the features and shows that Employee Management System uses the principles proposed from our architecture.

In Chapter 5, we demonstrate the User Interface of Employee Management System.

Chapter 2

Architecture

2.1 Clean Architecture	10
2.1.1 The Principles of Clean Architecture	12
2.1.2 Our Clean Architecture Layers	13
2.2 SOLID Principles	15
2.3 Design Patterns	22
2.3.1 Mediator Pattern	22
2.3.2 CQRS Pattern	23
2.4 SOLID Principles & Our Clean Architecture	26

2.1 Clean Architecture

In software development, there are many architectures to build a project and it's important to focus on what truly matters in good design. An architecture independent from any framework, library, and generally from any technical implementation (UI, Database, External Services) that allows easily both the integration of new technologies, and the replacement of existing ones sum up to what makes up a powerful architecture. Clean, Understandable, Maintainable, Extensible, and Adaptive code if encouraged through architecture code writing rules will help developers under the same Team to interact efficiently with the software without issues promoting reusability of code and making the scale and change of existing features or addition of features easier. Separating the different areas of software is a crucial requirement that can help Team Leaders to split work in a way that each developer can focus on his part and specifically to the area that he/she is more experienced while also promotes the separation between application business logic and technologies used. If we find an architecture that is based on this separation then we can achieve our target to extend this architecture based on which we will build a base software with a set of technologies and common features and reuse this software as a starting point in multiple projects with different business. Business rules should not depend on technical implementations allowing us to focus in each new project on business rules. In our case, the focus is to search for an architecture that promotes those

concepts so it can be used from different Teams that build Web APIs among the different projects they handle.

Clean Architecture is one popular approach that stands out because it emphasizes making the software easy to maintain, scale, and adapt. It was first introduced by Robert C. Martin (aka "Uncle Bob") in his book "Clean Architecture: A Craftsman's Guide to Software Structure and Design". Clean Architecture revolves around a simple idea. Following this architecture, we organize and structure the code in the form of layers, with each layer having its specific purpose (Single Responsibility Principle). The actual representation of it, as proposed by "Uncle Bob" includes 4 concentric circles each one representing a layer that defines a different area of software while arrows represent dependencies between the layers (see Figure 2.1.1). Nobody forces you to have only 4 layers. The number of layers depends on the scale and the complexity resulting in fewer or more layers in our application. The only constraint that needs to be satisfied is the dependency rule as explained in Clean Architecture principles. [20] Uncle Bob states that once you go further in the higher abstraction level the software becomes. Outer circles are technical details like Frameworks, database, external while inner circles are the business rules like entities and use cases. An entity encapsulates the most general and high-level rules, which is the business logic. You don't expect those entities object to be affected by changes like API calls, navigation, or any other application related stuff.

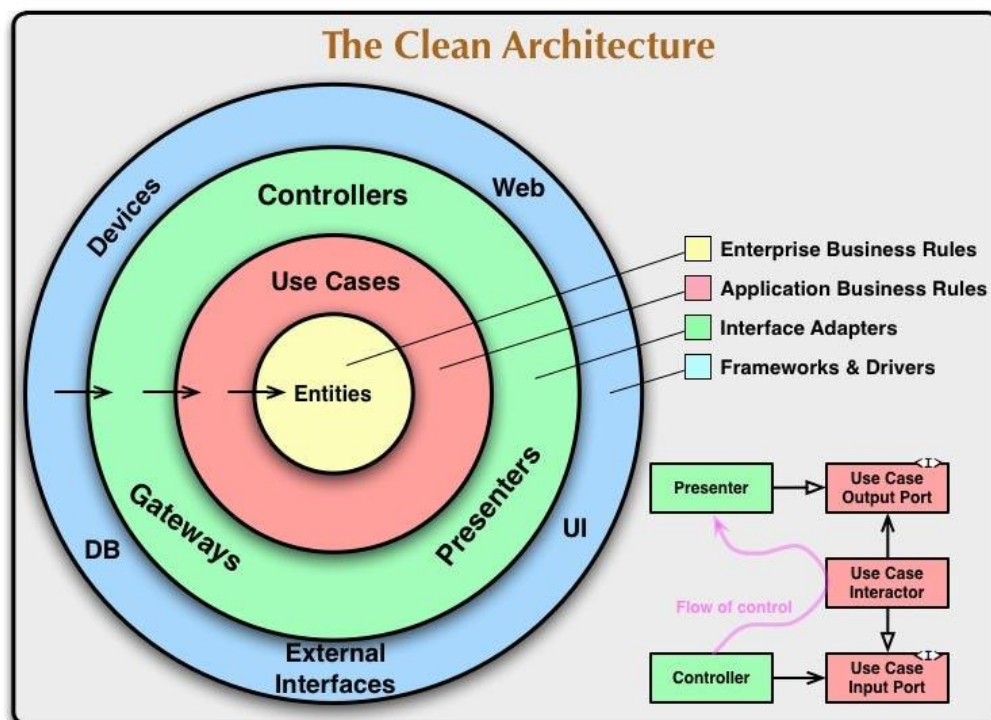


Figure 2.1.1: Uncle Bob's Clean Architecture

2.1.1 The Principles of Clean Architecture

Clean Code Architecture comes with a set of principles. These principles are:

- **Dependency Rule:** In Clean Architecture, dependencies can only flow inward toward the core business logic. This means that the **outer layers are dependent on the inner layers**, but the **inner layers do not depend on any of the outer layers**. Nothing in an inner circle can know anything at all about something in an outer circle. Since inner layers do not depend on them, they can be easily replaced. [20] This also means that you can use whatever Framework, Database, UI, External Service you want as long as it satisfies the Application's Core business rules enforced with inward dependencies.
- **Separation of Concerns:** This separation is achieved by dividing the software into layers, each one with a distinct responsibility and through the Dependency Rule. Separating different areas of software into layers enables splitting the work to assign each developer his layer where he/she is more experienced. This separation guarantees an easy way both to integrate new technologies or replace existing technologies without affecting different layers and with minimal code changes. Clean Architecture enforces a clear separation between the business rules (inner layers) and technical details (outer layers). This means that business rules and logic cannot depend on a specific library or technical implementation in the application. [21] For example, User Interface, Database, any external service of the app can be easily swapped out with a different one without business rules getting affected. This idea allows developers to focus on the implementation of use cases/business rules decoupled from technologies. Last, it offers loose coupling between layers.
- **Independence of Framework:** This architecture does not depend on the existence of a specific library or framework. This allows you to use whatever framework, language and tool without any constraint.
- **Use of SOLID Principles:** SOLID Principles are the fundamental guidelines related to object-oriented design aiming to create clean, understandable, reusable, extensible, and maintainable code. In the SOLID Principles section, we will explain them and in SOLID Principles & Clean Architecture section we will discuss how to apply those principles in our application. [21]
- **Testability:** Clean Architecture can help with writing unit tests for the business logic independently from external dependencies since Application Core does not depend on Infrastructure Layer. [1]

In this stage, we need to mention that Dependency Inversion Principle as will be explained detailed in SOLID Principles section helps to apply Dependency Rule and achieve the separations of Concern. A possible Dependency Rule violation is when a use case located in Application Layer wants to communicate with a repository located in Infrastructure Layer to get data. This call must not be direct because an inner circle cannot mention an outer circle. Dependency Inversion states that source code of high-level modules should not mention the source code of low-level modules at compile-time and suggest the following solution. To invert the dependency so the application core (Application and Domain Layer) which contains the business logic not to depend on data access and external details but Infrastructure and Presentation layer to depend on Application Core. This is achieved by defining interfaces in Application Core that the use case will call, and the infrastructure layer will implement it. Basically, we take advantage of dynamic polymorphism to generate compile time code dependencies to lower levels even though at runtime high-level modules depend on low level modules. Last, to achieve this dynamic polymorphism we need a framework where implementations can be wired up to interfaces via dependency injection and that is reason why we choose ASP.NET Core since it supports this mechanism.

2.1.2 Our Clean Architecture Layers

In that part, I want to mention that in our application we will apply something similar but not the exact same to what “Uncle Bob” proposed. This happens because it contains an extra layer called Interface Adapters indicated with green color in Figure 2.1.1 that will not be used in our architecture. This layer propose to follow MVC Architecture with presenters, views, and Controllers. It also force Web Framework and Database to depend on this MVC architecture adding extra overhead which restrict us. However, we preserve the ideas of Uncle bob like Dependency Rule, Separation of concerns, Solid Principles, Independence from Framework, Independence from Technical Details and Testability. The next figure introduces our Clean Architecture approach, and we will look at each layer and discuss what it could contain (see Figure 2.1.2.1).

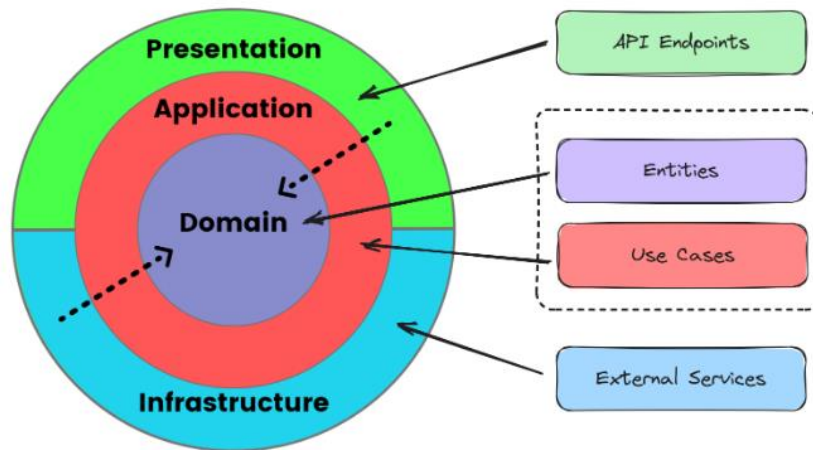


Figure 2.1.2.1: Our Clean Architecture

1. **Domain Layer:** The innermost cycle represents the domain layer containing the **enterprise business rules** and logic. The most highly abstracted and stable layer is Domain since it never gets affected from any use case or technology. This layer should be independent of any other layers. All other layers should depend on this. This layer can contain entities, repository interfaces, enums and most of the abstractions. [22]
2. **Application Layer:** The Application layer contains **application business rules**. Those are the features of your app known as use cases in terms of software development. It acts as an orchestrator, that manages the flow of data to and from entities to implement use case. This layer is responsible for taking the request data from the presentation layer, interacting with the database through infrastructure layer, ensuring business logic is implemented and passing a response back to the presentation layer. Changes in external details like Database, UI, Framework does not affect this layer. In this layer we can have the interfaces of application-specific and external services, the implementation of application-specific services, use cases, validators, DTOs like request and responses, exceptions, handlers. [20,22]
3. **Presentation Layer:** The presentation layer is the entry point of the application and plays the role of traffic cop since is responsible for managing the incoming requests from clients and sending them to the corresponding use case fast and efficiently. All the layers of the architecture are gathered from this layer to compile the monolithic app as a single assembly. Here, the Web API framework of the application should be placed and the API Routes. [23]
4. **Infrastructure Layer:** The infrastructure layer contains everything related to external services and concerns such as database, email providers, authentication provider, files storage and more. This layer is the librarian of our app making sure that use cases requested information can be accessed through repositories and external services

interfaces. This layer defines the implementation of the abstraction defined in the Application layer. Database, repository implementation, table entities, external services should be placed here. [22]

In Figure 2.1.1, the outermost circle contains UI but in our highly representation of Clean Architecture (see Figure 2.1.2.1) we didn't refer to it since is a different project. In this paper, we are focusing on the backend technologies and that is why we split frontend to another angular project that is not analyzed in diploma. The presentation layer is still able to get requests from UI or any other App. [20] Removing interface adapters layer can help us make the Web API implementation independent from MVC architecture and use controllers or any other approach we want. This give us the possibility to use a powerful feature of ASP.NET Core such as Minimal APIs for Web Framework instead of using slow Web APIs approach with controllers or even worse using MVC architecture. Minimal APIs will be explained in the Technologies Section. To conclude, removing interface adapters layer along with UI removal was a set of necessary changes so we can adjust Clean Architecture as proposed by Uncle bob to our requirements for an architecture that technical details are free to the developer.

Before start to explain SOLID Principles and how to apply them in our application it's important to mention that **we have found the desired architecture, and we extend it creating our generic architecture. Having it we can focus now to build the base software that will follow the principles of our generic architecture.**

2.2 SOLID Principles

The SOLID Principles are the fundamental guidelines related to object-oriented design. SOLID is an acronym that encapsulates 5 design principles which are SRP, OCP, LSP, ISP and DIP. Those principles serve as guiding light for developers and software systems. By following these principles, you ensure code modularity, extensibility, smooth integration to changes in existing requirements and the minimization of bugs. [1] Now, let's explore each detailed each one of these principles along with code examples.

Single Responsibility Principle (SRP): The SRP states that a class can be changed only for one reason. That means that a **class should have a single responsibility**. A maintainable and understandable code is ensured by following this principle. [1]

```
// Bad example violating SRP
public class Customer
{
    public void AddCustomer()
    {
        // Code to add a customer to the database
    }

    public void SendEmail()
    {
        // Code to send an email to the customer
    }
}
```

Figure 2.2.1: SRP Violation [2]

```
// Good example following SRP
public class CustomerService
{
    public void AddCustomer()
    {
        // Code to add a customer to the database
    }
}

public class EmailService
{
    public void SendEmail()
    {
        // Code to send an email to the customer
    }
}
```

Figure 2.2.2: SRP [2]

In the above figure 2.2.1, a class named Customer is shown, which violates the Single Responsibility Principle. The Customer Class is responsible for two different functionalities like adding a customer and sending an email to the customer. Any modifications to the database or email sending logic would lead to changes to the customer class. This violates the SRP since principle states that a class should have only one reason to change but in the given example the class has multiple reasons to change. To rectify this violation and follow what SRP states, class need to separate the responsibilities into two classes. In the refactored code (see Figure 2.2.2), the CustomerService class takes on the responsibility of adding a customer to the database with AddCustomer() method encapsulating the database-related operations within the class. The EmailService class is associated now with sending emails to customers through SendEmail() method encapsulating the email sending logic. Any changes to email sending logic will be handled from EmailService and not Customer class and vice-versa. This ensures that modifications to the database operations or email sending logic will only impact the respective class, minimizing the effect on different parts of code. Therefore, by splitting functionalities, we effectively isolate email sending logic from database operations, aligning with the SRP. Also, we achieve better separation of concerns and create a code that is easier to maintain and extend. [3]

Open-Closed Principle (OCP): The Open-Closed Principle says that you should be able to add new features to software entities without changing how it already works. Classes, Functions, Interfaces, should be **open for extension but closed for modification**. This principle suggests using abstraction and inheritance.


```
// Bad example violating OCP
public enum VehicleType
{
    Car,
    Motorcycle
}

public class Vehicle
{
    public VehicleType Type { get; set; }

    public double CalculateInsurancePremium()
    {
        double basePremium = 1000;

        // Calculate insurance premium based on vehicle type
        if (Type == VehicleType.Car)
        {
            return basePremium * 1.5;
        }
        else if (Type == VehicleType.Motorcycle)
        {
            return basePremium * 2;
        }
        return basePremium;
    }
}
```

Figure 2.2.3: OCP Violation [2]

```
// Good example following OCP
public abstract class Vehicle
{
    public abstract double CalculateInsurancePremium();
}

public class Car : Vehicle
{
    public override double CalculateInsurancePremium()
    {
        double basePremium = 1000;
        return basePremium * 1.5;
    }
}

public class Motorcycle : Vehicle
{
    public override double CalculateInsurancePremium()
    {
        double basePremium = 1000;
        return basePremium * 2;
    }
}
```

Figure 2.2.4: OCP [2]

In the bad example (see Figure 2.2.3), the Vehicle class contains a property for VehicleType and CalculateInsurancePremium() method which calculates insurance premiums based on vehicle type. However, this violates the OCP because adding a new vehicle type requires modifying the existing class, which should remain closed for modification. By introducing an abstract Vehicle Class (act like an interface), we can create two classes that inherit from this class (see Figure 2.2.4). Therefore, derived classes can override CalculateInsurancePremium() method and implement their own insurance premium calculation logic. This approach ensures code extensibility. If we want to add a new vehicle type, we can create a new class that inherits from Vehicle without changing the existing code of the abstraction and aligning with the principles of the OCP. Usually, abstraction is the interface or the abstract class. [3]

Liskov Substitution Principle (LSP): The Liskov Substitution Principle (LSP) states that objects of superclass should be able to be replaced with objects of subclass without affecting the correctness of the program and that objects of subclass should be able to access all the methods and properties of the superclass. [4] The application of LSP guarantees that **derived classes can be used interchangeably with their base classes**. [3]

```
// Bad example violating LSP
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public virtual void SetWidth(int width)
    {
        Width = width;
    }

    public virtual void SetHeight(int height)
    {
        Height = height;
    }

    public int CalculateArea()
    {
        return Width * Height;
    }
}
```

```
public class Square : Rectangle
{
    public override void SetWidth(int width)
    {
        Width = width;
        Height = width;
    }

    public override void SetHeight(int height)
    {
        Width = height;
        Height = height;
    }
}
```

Figure 2.2.5: LSP Violation [2]

In Figure 2.2.5, the Square class inherits from Rectangle class indicating that square is a special case of a rectangle. However, this violates the LSP principle since behaviour of Square class is not substitutable for the behaviour defined by the Rectangle class which can lead to unexpected behaviour. Rectangle class has separate methods to set width and height independently and since Square class inherit from Rectangle class and overrides these methods to ensure that both width and height are always equal.

```
Rectangle rectangle = new Square();
rectangle.SetWidth(5);
rectangle.SetHeight(3);

int area = rectangle.CalculateArea();
```

Figure 2.2.6: LSP Possible Mistake [2]

In Figure 2.2.6, Rectangle is a super class of Square. When a new object of Square is assigned in object of Super class Rectangle, SetWidth() and SetHeight() of Rectangle will be replaced by method of Square based to inheritance rules. Therefore, instead of 5*3 that we were expecting the result is 3*3 affecting the correctness of the program which violates the LSP.

```
// Good example following LSP
public abstract class Shape
{
    public abstract int CalculateArea();
}

public class Rectangle : Shape
{
    public int Width { get; set; }
    public int Height { get; set; }

    public override int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Shape
{
    public int SideLength { get; set; }

    public override int CalculateArea()
    {
        return SideLength * SideLength;
    }
}
```

Figure 2.2.7: LSP [2]

A refactored code that comes closer to LSP principle should break the inheritance relationship between Rectangle and Square. [2] Instead, we introduce an abstract base class for different shapes with an abstract method CalculateArea() (see Figure 2.2.7). This enforces subclasses to implement this method, so each one has their own logic on how to calculate the area based on the shape type. Square class extends Shape by introducing property SideLength and implement calculation of area specific to square while, Rectangle class extends Shape by introducing properties width and height to calculate area specific to rectangles. [24]

Compare LSP & OCP

LSP intends to ensure that interaction between superclass and subclass does not affect the correctness while OCP ensures code extensibility by inheriting an abstraction which will be closed for modification. Both principles use abstractions with interfaces or abstract classes through inheritance. In LSP, the abstraction is the superclass that act both like a common implementation or a contract. Another difference is that OCP states that the abstraction should not modified but only used through inheritance, while LSP can use a base implementation of an abstract class that can be changed in the subclass resulting to different logic.

Interfaces vs Abstract Classes

Both examples, force subclasses to implement functions but the only difference is that interface act like a contract and declares method signatures while abstract serves as a base class and can have abstract methods (no implementation) and non-abstract methods (can have implementation and will be inherited by derived classes). With interfaces we define the

behaviour that can be implemented by multiple unrelated classes. On the other side, abstract classes offers a hierarchical structure and provide common implementation for derived classes to reuse reducing code duplication. [24]

```
public interface Bird{
    public void fly();
    public void walk();
}

public class Parrot implements Bird{
    public void fly(){ // to do}
    public void walk(){ // to do }
} // ok

public class Penguin implements Bird{
    public void fly(){ // to do }
    public void walk(){ // to do }
} // it's break the principle of LSP.
```

Figure 2.2.8: LSP Violation Example 2, [6]

```
public interface Bird{
    // to do;
}

public interface FlyingBird extends Bird{
    public void fly(){}
}

public interface WalkingBird extends Bird{
    public void walk(){}
}

public class Parrot implements FlyingBird, WalkingBird {
    public void fly(){ // to do}
    public void walk(){ // to do }
}

public class Penguin implements WalkingBird{
    public void walk(){ // to do }
}
```

Figure 2.2.9: LSP Example 2 [6]

An introduction for Figure 2.2.8 is that Penguins belong to the group of birds that cannot fly. In the 2.2.8, flying and walking are what a bird can do (Bird Interface). The issue is that Penguin inherit bird functionalities even though it can only walk and not fly. Since child class Penguin can use the fly method LSP is violated. Solution is provided 2.2.9 by breaking Bird interface to FlyingBird and WalkingBird interfaces since Penguin now only implements WalkingBird interface. Now, Penguin object cannot access the fly method.

Interface Segregation Principle (ISP): The Interface Segregation Principle advises that breaking down large interfaces into smaller more focused ones, ensure that clients only need to depend on the interfaces that they use and not to those they do not use. [2]

```
// Bad example violating ISP
public interface IWorker
{
    void Work();
    void Eat();
    void Sleep();
}

public class Robot : IWorker
{
    public void Work()
    {
        // Code for performing work
    }

    public void Eat()
    {
        // Robots don't eat!
        throw new NotSupportedException();
    }

    public void Sleep()
    {
        // Robots don't sleep!
        throw new NotSupportedException();
    }
}
```

Figure 2.2.10: ISP Violation [2]

```
// Good example following ISP
public interface IWorker
{
    void Work();
}

public interface IEater
{
    void Eat();
}

public interface ISleeper
{
    void Sleep();
}

public class Human : IWorker, IEater, ISleeper
{
    public void Work()
    {
        // Code for performing work
    }

    public void Eat()
    {
        // Code for eating
    }

    public void Sleep()
    {
        // Code for sleeping
    }
}

public class Robot : IWorker
{
    public void Work()
    {
        // Code for performing work
    }
}
```

Figure 2.2.11: ISP [2]

In Figure 2.2.10, Robot implements interface IWorker, and it implement eat() method and throws exception for eat() and sleep() method. This violates the ISP, because it forces Robot to give implementation for methods that it doesn't need since robot don't eat or sleep. To solve this, we break down the monolithic IWorker interface into smaller, specific interfaces to align to the ISP, ensuring that only depend on the methods they need (see Figure 2.2.11). By separate interfaces, robot class can only implement the IWorker interface and do not implement unnecessary methods. Following ISP by splitting the interfaces based on specific behaviour, we minimize empty methods or exceptions since classes implements only related functionalities. This promotes better separation of concerns.

Dependency Inversion Principle (DIP): The Dependency Inversion Principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions. Abstraction should not depend on details and details should depend on abstractions. High-level modules should depend on abstractions and not concrete implementations. [9]

```
// Bad example violating DIP
public class DataAccess
{
    public void SaveData(string data)
    {
        // Code to save data to a specific database
    }
}

public class UserService
{
    private readonly DataAccess _dataAccess;

    public UserService()
    {
        _dataAccess = new DataAccess();
    }

    public void CreateUser(string username, string password)
    {
        // Code to create a user

        _dataAccess.SaveData("User created: " + username);
    }
}
```

Figure 2.2.12: DIP Violation [2]

```
// Good example following DIP
public interface IDataAccess
{
    void SaveData(string data);
}

public class DataAccess : IDataAccess
{
    public void SaveData(string data)
    {
        // Code to save data to a specific database
    }
}

public class UserService
{
    private readonly IDataAccess _dataAccess;

    public UserService(IDataAccess dataAccess)
    {
        _dataAccess = dataAccess;
    }

    public void CreateUser(string username, string password)
    {
        // Code to create a user

        _dataAccess.SaveData("User created: " + username);
    }
}
```

Figure 2.2.13: DIP [2]

In Figure 2.2.12, UserService is tightly coupled to a specific implementation of DataAccess class, making it difficult to change the data access class. In Figure 2.2.13, UserService depends on IDataAccess abstraction instead of the specific implementation in DataAccess, aligning with Dependency Inversion Principle. This promotes loose coupling between classes, making it easier to switch to a different data access logic or extend existing one without need to modify UserService class. If now we switch to a different database UserService class would remain same since we will call SaveData() method from IDataAccess interface. What will change is that we will need to add a different class that implements IDataAccess layer like PostgresDataAccess and at the configuration we need to change something so program build with PostgresDataAccess class instead of SqlDataAccess.

2.3 Design Pattern

2.3.1 Mediator Pattern

Mediator is a behavioral design pattern. This pattern is suggested to be used when there is a lot of communication and dependency between classes. In this design pattern, communication is achieved through a single channel and classes know what to communicate for, not how to communicate. The basic idea is that mediator pattern defines a new object that encapsulates how objects communicate with each other. [10] Therefore, when object A wants to communicate with an object B “mediator” is responsible to communicate to the other object B and **response back to A**. This centralizes the dependencies to the mediator object without direct references between objects that want to communicate. With “mediator” we reduce

dependencies and achieve loose coupling between classes making the application easy to maintain. [11]

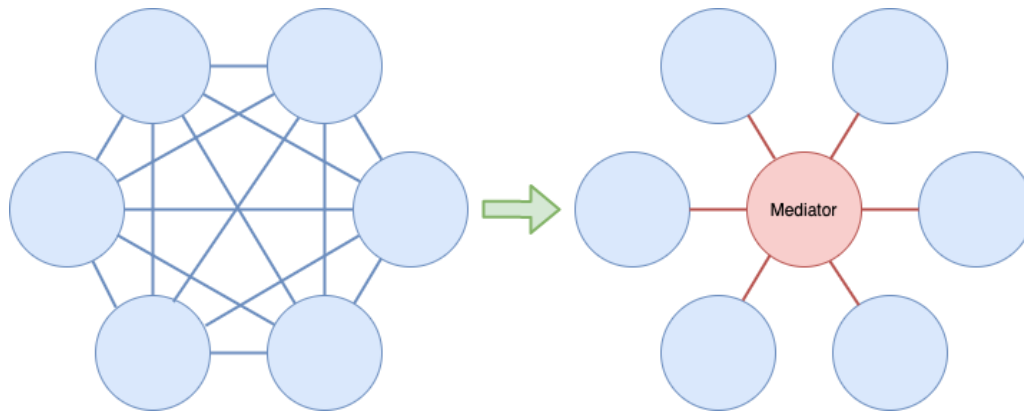


Figure 2.3.1.1: Mediator

Mediator Pattern is mostly explained using the example of control tower for airplanes. Airplanes must know the location of other airplanes during takeoff and landing. When the number of airplanes is small like 2-3 planes then communicating directly with the other pilots is not an issue. The issue arises when airplanes are much more. This requires pilots to contact the control tower. You can think that airplanes are the classes that want to communicate to understand that flight tower is playing the role of the Mediator class. [10]

Mediator Pattern & Clean Architecture

Using Mediator Pattern, we reduce the dependencies and communication between the different layers of clean architecture. For example, when an endpoint located in the Presentation layer of clean architecture wants to communicate with a handler located in Application layer then using Mediator pattern, we are reducing the communication and loose coupling between the 2 layers.

2.3.2 CQRS Pattern

Command Query Responsibility Segregation is a software architecture design pattern. CQRS is based on the Single Responsibility Principle of object-oriented Programming. CQRS pattern advice to separate command from queries, so each Command or Query Handler has a single responsibility. Command Handlers are responsible only for Write Operation to the database like Create, Update, Delete while Query Handlers are responsible for Read operations from the database. [10] **Handler is the class responsible for implementing use cases and processing**

all Command or Query Requests that wants to interact with the database to get or manage data.

Single or Multiple Databases?

A Crucial decision when implementing CQRS Database is whether we will split the database into 2 different database schemas for querying and updating data. The most common approach is using the same schema for reading and managing data because of the simplicity it provides with both operations without need extra management at database level. However, when application grows or has increased complexity, it becomes difficult to maintain. Another disadvantage is SQL best practices such as normalization optimizes the write schema and indexes optimizes the read schema but when using both of those in the same schema neutralize each other. Indisputably, having separate schemas for commands and queries allows you to scale them independently. Obviously, this decision depends on the requirements and whether the benefits of splitting to multiple database schemas gives such a big performance that outweighs the research and the implementation of managing multiple databases.

In case multiple databases are chosen, we provide a high-level overview of CQRS system (see Figure 2.3.2.1). Notice that both application and database level separate commands from queries and that is why we need to synchronize the updates with the read database. This idea introduces eventual consistency and fault tolerance strategies in CQRS systems which increase the complexity of the application. However, separating databases allows you to choose the best database for your requirements like SQL Database, Event Sourcing on the write database while using NoSQL, RavenDB, MongoDB for read operations. Using the same database for both is possible. Note that using the same database you can define 2 virtual databases that synchronizes into one database. [12]

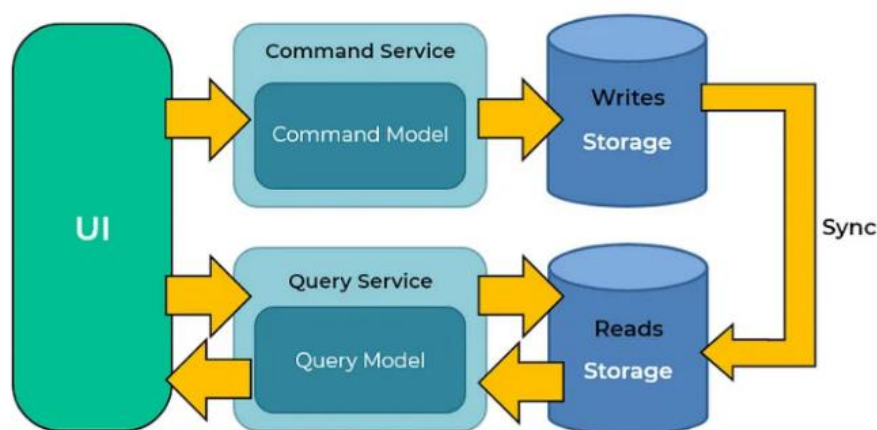


Figure 2.3.2.1: CQRS System

Advantages of CQRS

The main advantage is scalability by managing read and write operation through different Handler objects allows those operation to be independent. This independence idea when project grows, helps the scale of each operation, and enables developers to optimize each operation based on its unique demands. [13] CQRS pattern promotes efficient management of increasing read and write loads by splitting those operations on two different databases and adding more resources where needed. Notice that by having 2 different databases we can avoid errors occurring when a crash happens. If user wants to read data, he/she will be able to get them even though write operation database crashed, and vice-versa. [10] Another advantage is the maintainability through clear separation of concerns. The distinct handling of commands and queries simplifies the system's behaviour offering a structured project with cleaner and more maintainable code. [13] The CQRS pattern offers significant advantages in terms of performance and system responsiveness by segregating the responsibilities of read and write operations. Now, systems can finely tune read models for efficient querying and reporting, while prioritizing data consistency and integrity in write operations. [13]

Disadvantages of CQRS:

CQRS adds a layer of complexity to the system. Managing and synchronizing different databases and handling eventual consistency can be challenging for developers. A key issue with CQRS is eventual consistency, where updates from commands may not immediately show up in read models. This can be problematic for real-time applications. To address this, more complexity might be introduced with event-driven architecture. Maintaining separate read and write models in CQRS requires syncing them. Introducing CQRS to a team unfamiliar with the pattern can slow the learning and implementation curve. [13]

CQRS Implementation Level In Our System

Based on the analysis, we decided that CQRS software architecture patterns offer a lot of benefits and is an approach that can lead to designing scalable, maintainable, and high-performance systems. However, to avoid unnecessary overhead in the learning curve we will use a single database and only splitting command and query handlers only as a high-level of abstraction in application layer. This way, we will have the potential for performance optimization in a scenario where our application becomes large. The fact that clean architecture is split in layers will give us the chance to make any changes to the infrastructure level that contains the database implementation without needing to change other layers of the architecture. The code implementation of this design pattern is going to be analyzed in the Mediator section.

2.4. SOLID Principles & Our Clean Architecture

SRP Architecture examples:

Consider having a common abstraction and repository implementation for managing both user authentication and user management. This code design violates SRP, so we need to create separate interfaces, repositories for handling authentication and user management ensuring that each class has a single responsibility. [9]

With CQRS pattern we separate command from queries, so each Command or Query Handler has a single responsibility. Commands are responsible only for Write Operation while Queries are responsible for read operations. [9]

Open Closed Principle Through Email Example:

In Clean Architecture, OCP is a principle that encourage the use of abstractions and inheritance. In our project, we have used interfaces for both repositories and services. This idea aligns with OCP because we use interfaces to define abstractions that can be extended without modifying existing code. What do we mean by this? Extensibility can be achieved by defining an IEmailService interface (see Figure 2.4.1) that can be implemented for various email server providers. Now, for each email provider we will have an EmailService class where we implement the IEmailService interface and extend the class without modifying the existing code of IEmailService.

```
namespace Core.Domain.Abstractions.Email;
1 reference
public interface IEmailService
{
    1 reference
    Task SendEmail(List<string> to, string subject, string content, string emailFrom = "aaa");

    1 reference
    Task SendEmail(List<string> to, MessageTemplateEnum templateType, string emailFrom = "aaa");
}
```

Figure 2.4.1: IEmailService interface

How can we send Email inside Our Clean Architecture?

Let's first explore how we can achieve to send an email. In Figure 2.4.5, the selection of which email settings to use is chosen in AddEmail() method of Extension Class. We can use either default email settings (see Figure 2.4.5 line 13) or any other email settings we want (see Figure 2.4.5 comment line 12). The method AddSimpleEmailing() (see Figure 2.4.4) is the one responsible to create a transient IEmailSender instance (see figure 2.4.4 line 56) which means that a different instance of a resource is given every time it's requested. IEmailSender serves as a base implementation since it encapsulates the email sending logic independently from SMTP provider. We can easily conclude that choosing the SMTP server is programmer

responsibility by passing the appropriate SMTP settings to AddSimpleEmailing() method to initialize a transient instance of IEmailSender along with SMTP settings (see figure 2.4.4 line 57) so we can use the functionality of sending an email through SendEmail() method of IEmailSender instance (see figure 2.4.2) in any command or query handler (see Figure 2.4.3). This IEmailSender instance is created at the beginning of the program from AddEmail() method of Extension class since when app start running program.cs located in presentation layer search for all services per layer to register them. AddEmail() method located in Infrastructure layer will automatically be called when scanning happens. Since our application has only one email provider, we directly use IEmailSender instance without needing to create various EmailService classes.

```

10 namespace TechLink.Core.Emailing.Services;
11
12 public interface IEmailSender
13 {
14     Task SendEmail(EmailMessage message, Sender sender);
15 }
16 #if false // Decompile log

```

Figure 2.4.2: IEmailSender interface

```

//email
List<string> to = new() { request.Email };
string subject = messageTemplate.Subject;
string content = messageTemplate.Message
    .Replace("{BASE_URL}", baseUrl)
    .Replace("{FIRST_NAME}", user.Name)
    .Replace("{REDIRECT_URL}", redirectUrl);
await _emailSender.SendEmail(new EmailMessage(to, subject, content), null);

//resp
return new ResetOrForgotPasswordResponse { Value = true };

```

Figure 2.4.3: handle method of ResetOrForgotPasswordHandler

```

49 public static IServiceCollection AddSimpleEmailing(this IServiceCollection serviceCollection, SmtConfiguration smtpConfiguration)
50 {
51     if (smtpConfiguration == null)
52     {
53         throw new EmailingException("Could not register Emailing module configuration");
54     }
55
56     serviceCollection.AddTransient<IEmailSender, EmailSender>();
57     serviceCollection.AddSingleton(smtpConfiguration);
58     return serviceCollection;
59 }

```

Figure 2.4.4: AddSimpleEmailing() method

```

7 namespace Core.Infrastructure.Common.Email;
8 public static class Extension
9 {
10     public static void AddEmail(this IServiceCollection services, IConfiguration configuration)
11     {
12         //var emailConfigs = configuration.GetSection(nameof(EmailConfigurationMOA)).Get<EmailConfigurationMOA>();
13         var emailConfigs = configuration.GetSection(nameof(EmailConfigurationDefault)).Get<EmailConfigurationDefault>();
14         services.AddSimpleEmailing(new TechLink.Core.Emailing.Configuration.SmtConfiguration
15         {
16             SmtServer = emailConfigs.SmtServer,
17             From = emailConfigs.From,
18             Port = emailConfigs.Port,
19             UserName = emailConfigs.UserName,
20             Password = emailConfigs.Password,
21             Socket = Enum.Parse<SecureSocketOptions>(emailConfigs.Socket),
22             UseSSL = emailConfigs.UseSSL
23         });
24     }
25 }
26

```

Figure 2.4.5: Extension class

How OCP is violated and applied by encouraging abstractions?

OCP main idea is the usage of abstractions. One benefit, it provides us the change to have different implementations of the same abstraction if needed. Until now, we didn't apply the

OCP practically in our code since we didn't create any EmailService that can extend IEmailService or create various type of EmailServices. Even though we don't need to modify existing code to send an email (Use SendEmail() method (see Figure 2.4.3) the issue is that code is not extensible. When you send an email, you maybe want to add some extra logic like saving to database and returning that EmailMessageId (see Figure 2.4.6) which is not contained in the encapsulated email sending logic in IEmailSender interface. Therefore, aligning with OCP is crucial by defining a higher abstraction of send email (see Figure 2.4.1). OCP is applied when our concrete implementation (see Figure 2.4.6) implements that higher abstraction and contains that extra logic (or any other extra logic) combined with email sending logic.

```

9 public class EmailDefaultService : IEmailService
10 {
11     private readonly IEmailSender _emailSender;
12     private readonly IDataAccessLayerContext _db;
13
14     0 references
15     public EmailDefaultService(IEmailSender emailSender, IDataAccessLayerContext db)
16     {
17         _emailSender = emailSender;
18         _db = db;
19     }
20
21     2 references
22     public async Task<EmailResponse> SendEmail(List<string> to, string subject, string content)
23     {
24         try
25         {
26             await _emailSender.SendEmail(new EmailMessage(to, subject, content), null); // Send Email
27             // Insert Email Sent In Email History Table
28             long insertedId = await _db.QueryFirstOrDefaultAsync<long>($"[com].[EmailMessage_Insert]", new
29             {
30                 Subject = subject, Message = content, ToEmail = to[0]
31             });
32             return new EmailResponse
33             {
34                 IsSuccessful = true,
35                 EmailMessageId = new List<long> { insertedId },
36             };
37         }
38         catch (Exception ex)
39         {
40             return new EmailResponse
41             {
42                 IsSuccessful = false,
43                 ErrorDescription = ex.Message,
44             };
45         }
46     }
47 }

```

Figure 2.4.6: EmailDefaultService class

How other SOLID Principles applied in Emailing?

Defining a focused interface specific only how to send an email either by building a custom EmailMessage in Handler or using a pre-defined Template from database ensures that ISP is also satisfied (see Figure 2.4.1). SRP is applied in our code because EmailDefaultService class has only a single responsibility which is to save that the email was sent in the database.

DIP from Clean Architecture View

In Clean architecture, high-level modules are typically the inner layers that are closer to the business logic. Those modules contain application-specific logic, business rules, **entities** and **use cases**. Examples of **high-level modules** are the **Application** and the **Domain** layer. On the other hand, **low-level details** refer to concrete implementations of external services, database interactions, file system operations, APIs Endpoints, controllers, exception handling,

authentication, authorization. Typically, represents the outer layers of the architecture like **Presentation** and **Infrastructure** layer.

Example with DIP definition & Clean Architecture from Project

It is important now to understand in detail from the aspect of Clean Architecture the definition of Dependency Inversion Principle. “High-level policies should not depend on low-level details” means that **application or domain layer** should **not depend** on the specific **implementation** details of low-level layers like **presentation or infrastructure layer**. In our example, we can think **UserSaveHandler** as the **High-level policy** since it is our use case and **belongs to the Application Layer**. **UserRepository** can be mapped as a **low-level detail** that **belongs to Infrastructure layer** and the conclusion from this is that **UserSaveHandler** does not depend on the specific implementation details of **UserRepository**. Applying the dependency inversion principle starts by **introducing an abstraction between the high-level policy and the low-level details to remove the direct dependency between them**. That is why **UserSaveHandler** depends on **IUserRepository** interface and not to **UserRepository** (see Figure 2.4.7). [19] Therefore, the idea that High-level policies should not depend on low-level details is satisfied. From, “Abstractions should not depend on details and details should depend on abstractions” we can conclude that abstractions like interfaces or contracts that defines how the high-level policies interact with lower-level components should not depend on the concrete implementation, but the implementation should depend on the abstractions defined by the inner layers. We can **consider IUserRepository as the abstraction** and the **UserRepository** as the **low-level detail that depends on abstraction**. **IUserRepository** which is the abstraction also does not depend on details which is **UserRepository** so we can state that the idea that abstractions should not depend on details and details should depend on abstractions is also satisfied. **Removing the ‘uses’ relationship between high and low-level policy and adding an implements relationship of low-level policy on the abstraction makes what we call Dependency Inversion.** [19]

```
public record UsersSaveResponse : BooleanDto;
4 references
public record UsersSaveCommand : UsersSaveRequestDto, IRequest<UsersSaveResponse>;
1 reference
public class UsersSaveHandler : IRequestHandler<UsersSaveCommand, UsersSaveResponse>
{
    private readonly IUserRepository _userRepository;
    private readonly IRelationsRepository relationsRepo;

    0 references
    public UsersSaveHandler(IUserRepository userRepository, IRelationsRepository relationsRepo)
    {
        _userRepository = userRepository;
        this.relationsRepo = relationsRepo;
    }
}
```

Figure 2.4.7: UserSaveHandler class

How DIP ensures Dependency Rule?

In Clean Architecture, the Dependency Rule is one of the core ideas and the reason that this architecture works is a result of Dependency Inversion. Without DIP, the Dependency Rule idea is violated since dependencies do not flow inward. But why does this happen? Simply because `UserSaveHandler` located at Application Layer need `UserRepository` instance that belongs to Infrastructure Layer. Since, Infrastructure layer is the outer layer of Application layer this violates Dependency Rule idea that inner layers do not know anything about outer layers. Applying the Dependency Inversion Principle, we ensure `UserSaveHandler` do not depend on `UserRepository`, but both depend on the `IUserRepository` interface. This inversion of dependencies except from achieving loose coupling, flexibility, testability guarantees that clean architecture core principles like Dependency Rule can be applied.

Abstraction Placement:

When we invert dependency from a high-level policy that ‘uses’ a low-level detail high-level policy to an idea where high level policy ‘uses’ an abstraction and the low-level policy ‘implements’ the abstraction. Since we want higher level policy not to depend on the low level, the abstraction belongs with the high-level policy either on application layer if it’s related to the application or external services (`IJwtBearerService`, `IEncryptionSimpleService`, `IOtpService`) either to the domain layer if it’s related to business logic (`IUserRepository`, `IEmployeeRepository`).

The concrete implementation of an interface should not be placed in domain layer but in an outer layer like Infrastructure Layer (`EmailService`) when communicate with external service providers or in Application Layer (`JwtBearerService`, `EncryptionSimpleService`) when is application specific. By adding abstractions to Domain or Application layer and implementations to Application or Infrastructure layer we ensure that Dependency Rule is applied. [19]

Chapter 3

Frameworks and Libraries

3.1 C#	31
3.2 Angular	31
3.3 SQL Server	32
3.4 ASP.NET Core Framework	32
3.4.1 Minimal APIs	34
3.4.2 Middlewares Filters	35
3.4.3 Filters	39
3.4.4 Authentication & Authorization	39
3.4.3 .Net Libraries	43
3.4.3.1 MediatR	43
3.4.3.2 Fluent Validation	46
3.4.3.3 Dapper	52

3.1 C#

Our application will be developed in ASP.NET Core 7.0 Framework and the language that is most popular for .NET platform applications is C# language. C# is a cross-platform language used by millions of developers with rich support. The most important is that it contains a lot of features that support object-oriented principles. Developers can write code fast with high performance. Some of the features that we are going to use is record, generics, Task based asynchronous programming with async, await keywords, interfaces, classes, inheritance, etc. [26]

3.2 Angular

Angular is an application-design framework and development platform built on Typescript for creating efficient and sophisticated single-page apps. Typescript is the programming language of Angular that is based on JavaScript. Angular is a component-based framework for building

scalable web applications that contains a collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more. Along with angular we will use HTML, CSS, Bootstrap and PrimeNG library that provide a rich source native Angular UI Components. Enterprise-level applications can be built using Angular which also has a large community with over 1.7 million developers. Angular provides a rich variety of features such as routing, forms, components, dynamic properties, property binding from typescript file to html template, event handling, dependency injection and a lot of others. [27]

3.3 SQL Server

SQL Server, developed by Microsoft, is a robust and highly scalable relational database management system (RDBMS) designed for enterprise-level applications. It offers a comprehensive suite of tools and features for data storage, management, and analysis, making it a preferred choice for businesses of all sizes. Its built-in security features ensure data protection and compliance with industry standards, while its recovery solutions provide reliability. With its powerful performance tuning and optimization tools, SQL Server enables efficient handling of large volumes of data, making it an essential component for our applications. [29]

3.4 ASP.NET Core Framework

ASP.NET Core is a cross-platform, high-performance framework for building modern, cloud-based, and internet-connected applications. ASP.NET Core supports a variety of application models, including web applications, microservices, and serverless functions. ASP.NET Core allows developers to create web APIs using two primary approaches: controller-based APIs and minimal APIs. [29] Our Web APIs was developed using a simplified way from ASP.NET Core with Minimal APIs focusing as the name suggest on minimalism. They are designed for rapid development scenarios and offers exposing and handling endpoints in a single line via lambdas expressions. Traditional Controllers Web APIs come with a considerable amount of boilerplate code while having a large overhead to expose endpoints. Let's compare the steps for each approach. [43]

Controller-Based Approach – Figure 3.4.1:

- HelloController class inherits from ControllerBase which is a base class in ASP.NET Core for API controllers.
- To add default API behaviors, it uses [ApiController] attribute.
- The [Route] attribute defines the base route path for the controller.
- Get method is a GET HTTP Request endpoint that responds with a message.

Minimal-API Approach – Figure 3.4.2:

- CreateBuilder method of WebApplication class creates a minimal API Web Application.
- MapGet method specifies an HTTP Get endpoint for the specified route “api/hello”.
- The lambda expression inside MapGet method contains the endpoint Handler

```
[ApiController] → Add default API Behaviours
[Route("api/[controller]")] → Defines base route path
public class HelloController : ControllerBase
{
    [HttpGet] → HTTP Request method
    public IActionResult Get() → Inherits from API Controller's base class
    {
        return Ok("Hello, API!");
    }
}
```

Figure 3.4.1: Controller-Based Approach [43]

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/api/hello", () => "Hello, API!");
app.Run();
```

HTTP Request method →

Endpoint Handler with lambda expressions →

Figure 3.4.2: Minimal API Approach [43]

Obviously, the above figures proves that controller-based API requires more code with class declarations, attributes and method signatures while Minimal APIs eliminate boilerplate code and abstractions, are simpler and quicker to create and handle an endpoint . Minimal APIs use

a functional approach with direct route to handling mapping, but Controllers follow use methods in classes for this purpose. [43] Latest Benchmarks proves that Minimal APIs perform with reduced resource consumption producing slightly better than traditional APIs with about 40% faster response and less memory.

ASP.NET Core supports dependency injection, Middlewares, Minimal APIs, Filters, Authentication, Authorization. Let's analyze some of those features now and how they can be used in an ASP.NET Core 7.0 Web Minimal API app. [29] Before that Figure 3.4.3 represent a high-level overview of our Web API based on the design pattern we introduce in the previous section (mediator and CQRS) and on technologies we will analyze in the following sections like Dapper ORM, Minimal APIs, Fluent Validation, MediatR. **In the upcoming chapter research will focus on building the base software and how we can integrate each technology, library and use features provided from frameworks into our base software project without breaking our generic architecture.**

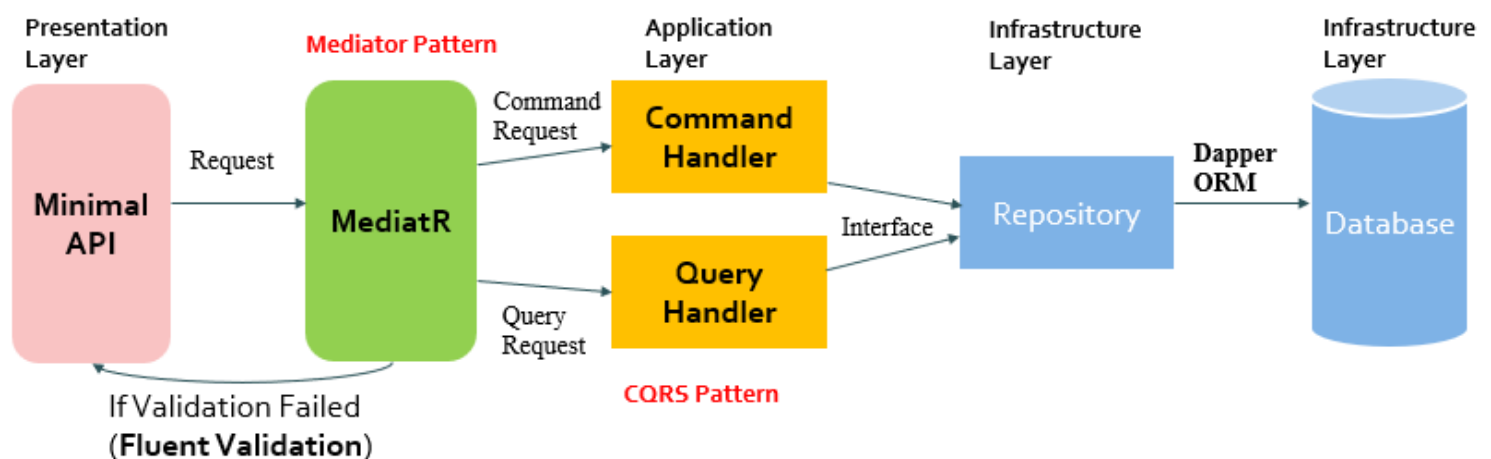


Figure 3.4.3: Web API Overview

3.4.1 Minimal APIs

Minimal APIs leverage the `WebApplicationBuilder` and `WebApplication` classes to define routes and handle HTTP Requests. Developers can handle those endpoints with a functional approach using lambda expressions. With lambda expressions you can create an anonymous function that will handle the HTTP Request for the specific endpoint. Minimal APIs supports features such as Route Handlers, Endpoint Filters, Parameter Mapping, interfaces to create responses, etc. Next, we are going to define the core concepts for using Minimal APIs. [30]

RouteGroupBuilder & RouteHandlerBuilder classes

The RouteGroupBuilder class in ASP.NET Core is a tool for organizing and managing routes in a web application. It is a builder for defining a group of endpoints that share a common prefix. This can help in structuring your API routes clean and logical. When you use the MapGroup() from RouteGroupBuilder class you can define a route pattern or prefix that will be applied to all endpoints within the group. MapPost() method adds a RouteEndpoint class. This RouteEndpoint is associated to a specific URL path with a function that should run when an HTTP POST requests for the specified route is made. A configured WebApplication also supports HTTP methods like MapGet, MapPatch, MapPut and MapDelete. Another valuable method supported is AddEndpointFilter() method that allows to add filters to routes within a specific group. RouteGroupBuilder also provides WithTags() and WithGroupName() extension methods that helps to categorize endpoints by tags into related groups in the SwaggerUI. [31] RouteHandlerBuilder class provides WithMetadata() method that can add the provided metadata items to the EndpointBuilder metadata. The RouteHandlerBuilder class also contains AllowAnonymous() method that allow anonymous access to the endpoint meaning that no authorization is required. [32] More Details will be explained in the next section with practical implementation examples.

3.4.2 Middlewares

Middleware is software that refers to components in the ASP.NET Core framework pipeline that handle requests and responses. Each middleware can choose if the request can be passed to the next middleware in the pipeline. Middlewares could do work before and after the next component in the pipeline. Request delegates handle HTTP requests, and they are used to build the request pipeline. It's a powerful tool for implementing cross-cutting concerns such as logging, authentication, authorization, error handling, and more. [33]

How do they work?

Those middlewares components is what we call ASP.NET Core request pipeline. In Figure 3.4.2.1, the black arrows represents the flow through the request pipeline. An important note is the reverse order for the response.

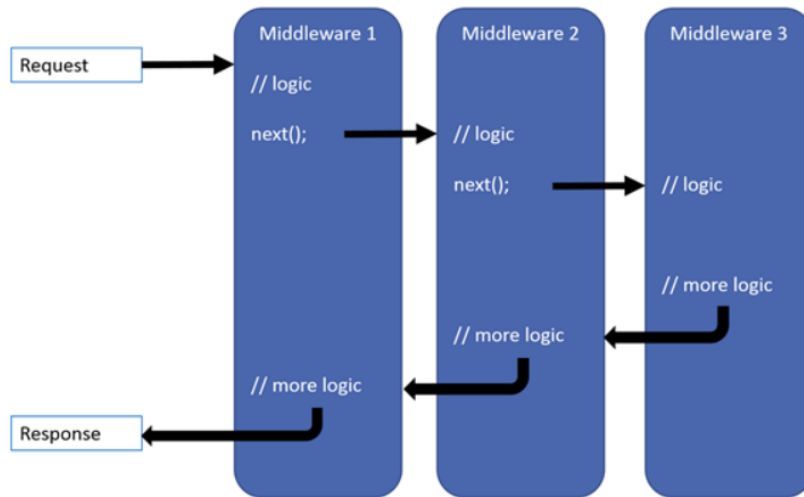


Figure 3.4.2.1: Request Pipeline With Middlewares [34]

A middleware invoking the next delegate can pass the request to the next middleware component in the pipeline allowing it to process the request further or stops the request processing at the current middleware (short-circuiting the pipeline). This idea allows us to do some processing before passing the request to the next middleware before next() method. The code after next() method is the processing after the next middleware has completed. [33]

Middlewares Ordering

When we create a new ASP.NET Core app, many of the middleware components are already registered in the order of the following Figure 3.4.2.2. Middleware order is crucial as it defines the sequence in which middleware components are executed. You can specify the order of middleware by the order in which you add them in your Program.cs file. You have full control over how to reorder existing middlewares or inject new custom middlewares as necessary for your scenarios. Ordering is critical for security and functionality.

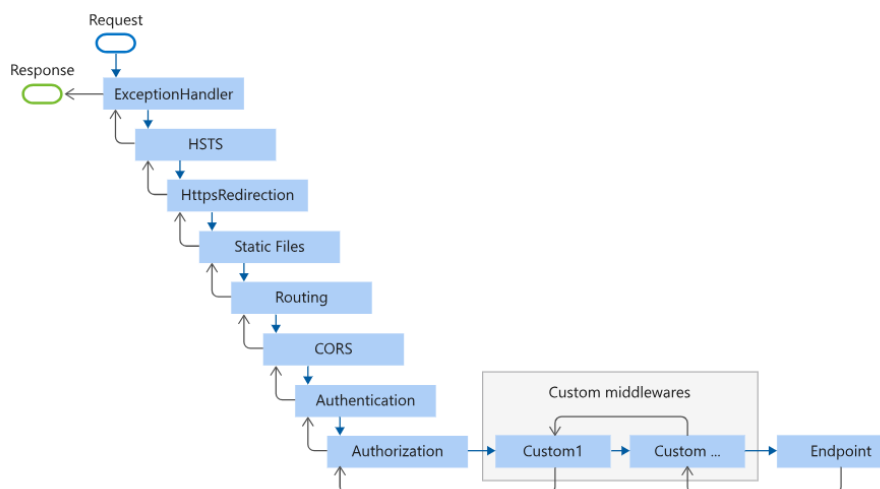



Figure 3.4.2.2: Default Middlewares Ordering [34]

Custom Middlewares with Minimal APIs in our System

In this section, we are going to create custom middlewares using request pipeline. Even though try-catch blocks are the most common approach wherever an error can occur ASP.NET Core middleware mechanism offer an innovative strategy for error-handling in a global way. Utilizing middleware concept we can remove all the exception-handling logic from the classes to a single centralized file called `ExceptionHandlerMiddleware` (see Figure 3.4.2.3). This custom middleware can be created by implementing a function called `Invoke()` passing the `HttpContext` as an argument. Inside this class, when no error occurs try block will execute the `await _next(context)` where `_next` is the `RequestDelegate` parameter responsible to pass the execution to the next middleware component. Notice that `_next(context)` has no code before or after indicating that middleware doesn't want to perform any logic when no error occurred. If an exception was thrown our middleware will trigger the catch block and call the `HandleExceptionAsync()` method. In that method, `GetHttpStatusCode()` (see Figure 3.4.2.4) method is called and based on the exception it set up the appropriate HTTP response status code, content type returning a structured error response. In the case where the exception is unknown HTTP status code is set to 500 indicating internal server error. This middleware handles exceptions that occurs during request processing maybe from other middlewares, handler, repository, database, dapper, etc. If an exception is thrown this middleware intercepts it doing some logic. This strategy makes exception-handling more readable and maintainable avoiding repeating the same code. Middleware can be registered to request pipeline using the `UseMiddleware()` method provided by the `IApplicationBuilder` interface (see Figure 3.4.2.5) [18].



```
2 references
public class ExceptionHandlingMiddleware
{
    private readonly RequestDelegate _next;

    0 references
    public ExceptionHandlingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    0 references
    public async Task Invoke(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            await HandleExceptionAsync(context, ex);
        }
    }
}
```

Figure 3.4.2.3: `ExceptionHandlerMiddleware` class



```
private HttpStatusCode GetHttpStatusCode(Type exceptionType)
{
    // custom exceptions:
    if (exceptionType == typeof(ApiAuthenticationException))
        return HttpStatusCode.BadRequest;

    else if (exceptionType == typeof(ApiBadRequestException))
        return HttpStatusCode.BadRequest;

    else if (exceptionType == typeof(ApiFluentValidationException))
        return HttpStatusCode.BadRequest;

    else if (exceptionType == typeof(ApiNotFoundException))
        return HttpStatusCode.NotFound;

    // system's exceptions:
    else if (exceptionType == typeof(Exception))
        return HttpStatusCode.InternalServerError;

    else if (exceptionType == typeof(ImplementedException))
        return HttpStatusCode.NotFound;

    else if (exceptionType == typeof(UnauthorizedAccessException))
        return HttpStatusCode.Unauthorized;

    else if (exceptionType == typeof(KeyNotFoundException))
        return HttpStatusCode.Unauthorized;

    return HttpStatusCode.InternalServerError;
}
```

Figure 3.4.2.4:
`GetHttpStatusCode()` method of
`ExceptionHandlerMiddleware` class

```

public static WebApplication AddCoreApplication(this WebApplication app, IConfiguration configuration)
{
    app.UseMiddleware(typeof(LogMiddleware));
    app.UseMiddleware(typeof(ExceptionHandlingMiddleware));

    #region ENDPOINTS
    var apiMapGroup = app.MapGroup("api")
        .AddEndpointFilter<DataIntegrityValidatorFilter>()
        .AddEndpointFilter<PermissionsFilter>()
        .WithOpenApi();

    //auth-management
    app.AddAuthEndpoints(apiMapGroup);
    //user-management
    app.AddUsersEndpoints(apiMapGroup);
    app.AddPermissionsEndpoints(apiMapGroup);
    app.AddRolesEndpoints(apiMapGroup);
    app.AddGroupsEndpoints(apiMapGroup);
    app.AddPortalsEndpoints(apiMapGroup);
    app.AddRelationsEndpoints(apiMapGroup);
    //employee-management
    app.AddEmployeesEndpoints(apiMapGroup);
    app.AddDepartmentsEndpoints(apiMapGroup);
    app.AddEmployeeRolesEndpoints(apiMapGroup);
    app.AddProjectsEndpoints(apiMapGroup);
    app.AddEmployeeProjectsEndpoints(apiMapGroup);
    #endregion ENDPOINTS

    return app;
}

```

Figure 3.4.2.5: AddCoreApplication() method

LogMiddleware is another custom middleware that logs various details about the request and response, such as request method, path, headers, and response status (see Figure 3.4.2.6). It also handles exceptions thrown during request processing and logs them. It's important to mark that exceptions are handled in different middleware so catch block is empty but try block will implement the logging of the request and the response. LogMiddleware should be placed first in the order of the custom middlewares (see Figure 3.4.2.5:) because we need to log requests before processed by other middleware components. Another reason is that if an exception happens at the response, then log needs to be executed after ExceptionHandlingMiddleware so error can be logged.

```

public class LogMiddleware
{
    private readonly RequestDelegate _next;

    public LogMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        try
        {
            await MiddlewareInfo(context, context.Request);
        }
        catch (Exception)
        {
            /* any app/system's exceptions are handled on different middleware */
        }
    }
}

```

Figure 3.4.2.6: LogMiddleware

In summary, middleware components in ASP.NET Core provide a way to encapsulate cross-cutting concerns and intercept requests and responses as they flow through your application. Custom middleware components, such as logging, and exception handling can be easily

integrated into the middleware pipeline and help maintain separation of concerns in your application architecture.

3.4.3 Filters

Minimal API filters give the change to the developer to implement logic that will run a code before and after the endpoint handler. You can inspect and modify parameters before reaching the endpoint handler or modify the response after execute endpoint handler. In Minimal APIs, filters can be implemented using `IEndpointFilter` interface and providing `InvokeAsync` method() that takes an `EndpointFilterInvocationContext` and an `EndpointFilterDelegate` as arguments. The `EndpointFilterInvocationContext` provides access to the `HttpContext` of the request and an `Arguments` list indicating the arguments passed to the endpoint handler. Endpoint Filters can be registered as shown in Figure 3.4.2.5 method using `AddEndpointFilter()` function. [35]

3.4.4 Authentication and Authorization

ASP.NET Core with Minimal APIs support provide functionality for authentication and authorization. Authentication is the process of determining a user's identity while authorization refers to checking if the user is permitted to access a resource.

JSON Web Tokens

To achieve authentication, we utilize JWTs. JWTs are compact and self-contained tokens that securely transmit information between parties as a JSON object. They are digitally signed, using HMAC algorithm and public/private key pair using RSA ensuring their authenticity and integrity. JSON Web Tokens are useful for authorization since once the user is logged in, he/she can access all endpoints of API that are permitted with that token. [36]

Implementing Authentication System

When a user tries to login in, his credentials are validated from our authentication system (Figure 3.4.4.1 line 37-43). If authentication is successful, a JSON Web Token is generated with an expiration time and returned to client as a response of a successful authentication request (see Figure 3.4.4.2). Therefore, clients in subsequent requests should send the JWT in the authorization header of the request as proof of authentication. The question is how to perform this process of validating JWT. [36]

```

20 internal class LoginHandler : IRequestHandler<LoginCommand, LoginResponse>
21 {
22     private readonly IUserRepository _userRepo;
23     private readonly IJwtBearerService _jwtBearerService;
24     private readonly IPasswordHasher<UserEntity> _passwordHasher;
25     private readonly IEncryptionSimpleService encryptionSimpleService;
26
27     public LoginHandler(IUserRepository userRepo, IJwtBearerService jwtBearerService, IPasswordHasher<UserEntity> passwordHasher, IEncryptionSimpleService encryptionSimpleService)
28     {
29         _userRepo = userRepo ?? throw new Exception("ArgumentNullException");
30         _jwtBearerService = jwtBearerService ?? throw new Exception("ArgumentNullException");
31         _passwordHasher = passwordHasher;
32         this.encryptionSimpleService = encryptionSimpleService;
33     }
34
35     public async Task<LoginResponse> Handle(LoginCommand request, CancellationToken cancellationToken)
36     {
37         var portalId = request.IsAdmin ? PortalsSmartEnum.AdminPortal.Value : PortalsSmartEnum.CustomerPortal.Value;
38         var user = await _userRepo.GetUserByPortalAndEmail(portalId, request.Email) ?? throw new ApiAuthenticationException("UserNotFound");
39         if (user.UserStatusId == (long)UserStatusEnum.Inactive || user.UserStatusId == (long)UserStatusEnum.AccountDeleteRequest)
40             throw new ApiAuthenticationException("InactiveAccount");
41         var passwordVerificationResult = _passwordHasher.VerifyHashedPassword(null, user.Password, encryptionSimpleService.Decrypt(request.Password));
42         var userHasValidPassword = passwordVerificationResult != PasswordVerificationResult.Failed;
43         if (!userHasValidPassword)
44             throw new ApiAuthenticationException("InvalidCredentials");
45     }

```

Figure 3.4.4.1: LoginHandler

```

// auth model
AuthResult authResult = await _jwtBearerService.GenerateTokens(user.UserId, user.Email);

return new LoginResponse
{
    AccessToken = authResult.AccessToken,
    AccessTokenExpiryUTCdatetime = authResult.AccessTokenExpiryUTCdatetime,
    RefreshTokenGuid = authResult.RefreshTokenGuid,
    RefreshTokenIdExpiryUTCdatetime = authResult.RefreshTokenIdExpiryUTCdatetime,
    User = authResult.User,
};

```

Figure 3.4.4.2: LoginHandler

The responsibility to perform the authentication process is assigned to JwtBearer. JwtBearer is a NuGet package that comes in with an ASP.NET Core middleware that enables an application to receive an OpenID Connect bearer token. The middleware that comes with JwtBearer package performs authentication by extracting JWT token from the authorization header and validating it. Therefore, each incoming request to our application is intercepted by this middleware checking whether the token is valid and not expired to allow the request to proceed. For long-lived sessions, clients can refresh their access tokens using a refresh token. When a token refresh request is received, the existing JWT is exchanged for a new one with an extended expiration time. This process helps maintain session security while minimizing the need for users to reauthenticate frequently. The following figure 3.4.4.3 is responsible to configure a JWT bearer-based authentication using AddAuthentication() and AddJwtBearer() extension methods of ASP.NET Core with JwtBearer NuGet package . First, we need to enable authentication by calling AddAuthentication() that registers the required authentication services on the app's service provider. It also configures the authentication options and sets the default authentication schema. Then, a specific authentication strategy is required which in our case is JWT bearer-based authentication. Using AddJwtBearer(), we can register JWT Bearer Authentication middleware so it's not necessary to invoke Authentication middleware of ASP.NET Core. [37] However, there are some case such as when a user tries to log in or need to refresh a token as shown in Figure 3.4.4.4 that we need to disable JWT Bearer Authentication

middleware using `AllowAnonymous()` method because user doesn't have any JSON Web Token and he/she must get one. To sum up, our Authentication system consists of validating user credentials when tries to login and validating JWT using JWT Bearer-based Authentication middleware.

```
services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(x =>
```

Figure 3.4.4.3: JWT Bearer-based Authentication

```
11 internal static class AuthEndpoints
12 {
13     1 reference
14     public static void AddAuthEndpoints(this WebApplication app, RouteGroupBuilder apiMapGroup)
15     {
16         RouteGroupBuilder auth = apiMapGroup.MapGroup(TagNames.AUTH)
17             .WithTags(TagNames.AUTH)
18             .WithGroupName(SwaggerGroupNames.AUTH)
19             .WithOpenApi();
20
21         auth.MapPost<LoginCommand, LoginResponse>("Login").AllowAnonymous();
22         auth.MapPost<RefreshTokenCommand, RefreshTokenResponse>("RefreshToken").AllowAnonymous();
23     }
24 }
25
```

Figure 3.4.4.4: AuthEndpoints class

Authorization System

Except from user authentication for each request, we also need to ensure that the user has the appropriate permission to access a specific endpoint. Our Authorization is permission-based where each user has a specific role that contains a set of permissions indicating the access-level of each user in the application. Our application implements authorization using custom endpoints filters. We can use `AddEndpointFilter()` method to add a filter called `PermissionFilter` under all routes of the application (see Figure 3.4.2.5). `InvokeAsync()` method of `PermissionFilter` (see Figure 3.4.4.5) class is executed for each request after finishing the request pipeline and before allowing access to specific endpoint handler. In Figure 3.4.4.5 shows that if the user doesn't have all the required permissions access is denied and `UnauthorizedAccessException` is thrown. Since endpoint filter raise an exception the control flow of the app is passed to the middleware pipeline in reverse order. Therefore, in Figure 3.4.2.3 catch block will execute modifying the response that flows back through the middleware pipeline adding the appropriate HTTP Status Code and a structured error message. If the User has all the permissions, then return `await next(context)` stops the execution of filter pipeline and pass the execution to endpoint handler.

```

public class PermissionsFilter : IEndpointFilter
{
    private readonly IRequestContext<UserContextModel> _reqCtx;
    0 references
    public PermissionsFilter(IRequestContext<UserContextModel> reqCtx)
    {
        _reqCtx = reqCtx;
    }
    0 references
    public async ValueTask<object?> InvokeAsync(EndpointFilterInvocationContext context, EndpointFilterDelegate next)
    {
        //List of permissions in metadata
        if (context.HttpContext.GetEndpoint()?.Metadata.GetMetadata<List<PermissionsSmartEnum>>() is { } meta)
        {
            //ENDPOINT'S PERMISSIONS
            var endpointPermissions = meta;
            if (endpointPermissions == null || endpointPermissions.Count() == 0)
            {
                return await next(context);
            }
            // USER'S PERMISSIONS
            var userPermissions = _reqCtx?.User?.PermissionIds;

            // Unauthorized for user with no permissions
            if (userPermissions == null || userPermissions.Count() == 0)
            {
                throw new UnauthorizedAccessException("PermissionError");
            }
            // Unauthorized for user that does not have ANY OF the corresponding endpoint permissions
            foreach (var endpointPermission in endpointPermissions!)
            {
                if (!userPermissions.Contains(endpointPermission.Value))
                {
                    throw new UnauthorizedAccessException("PermissionError");
                }
            }
        }
        return await next(context);
    }
}

```

Figure 3.4.4.5: PermissionFilter class

In that part, it's obvious that we implement cross-cutting concerns like authentication, authorization, exception-handling and logging using middlewares and filters features from ASP.NET Core Minimal APIs. Figure 3.4.4.6 shows the middlewares sequence of our application while Figure 3.4.4.7 shows the flow of a request in our application combining Middlewares, Filters and the high-level overview of our Web API explained. In the following section, we are going to analyze the missing parts of Figure 3.4.4.7 like MediatR, Dapper ORM and Fluent Validation.

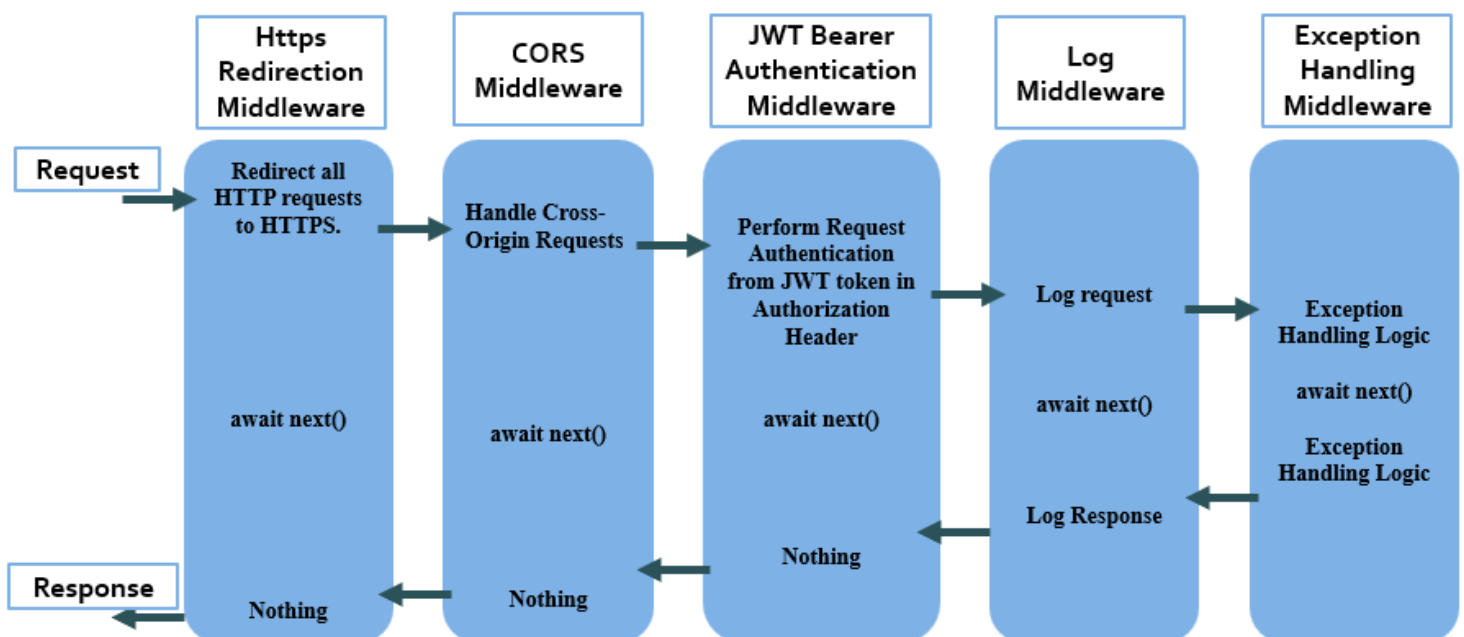


Figure 3.4.4.6: Middlewares Sequence

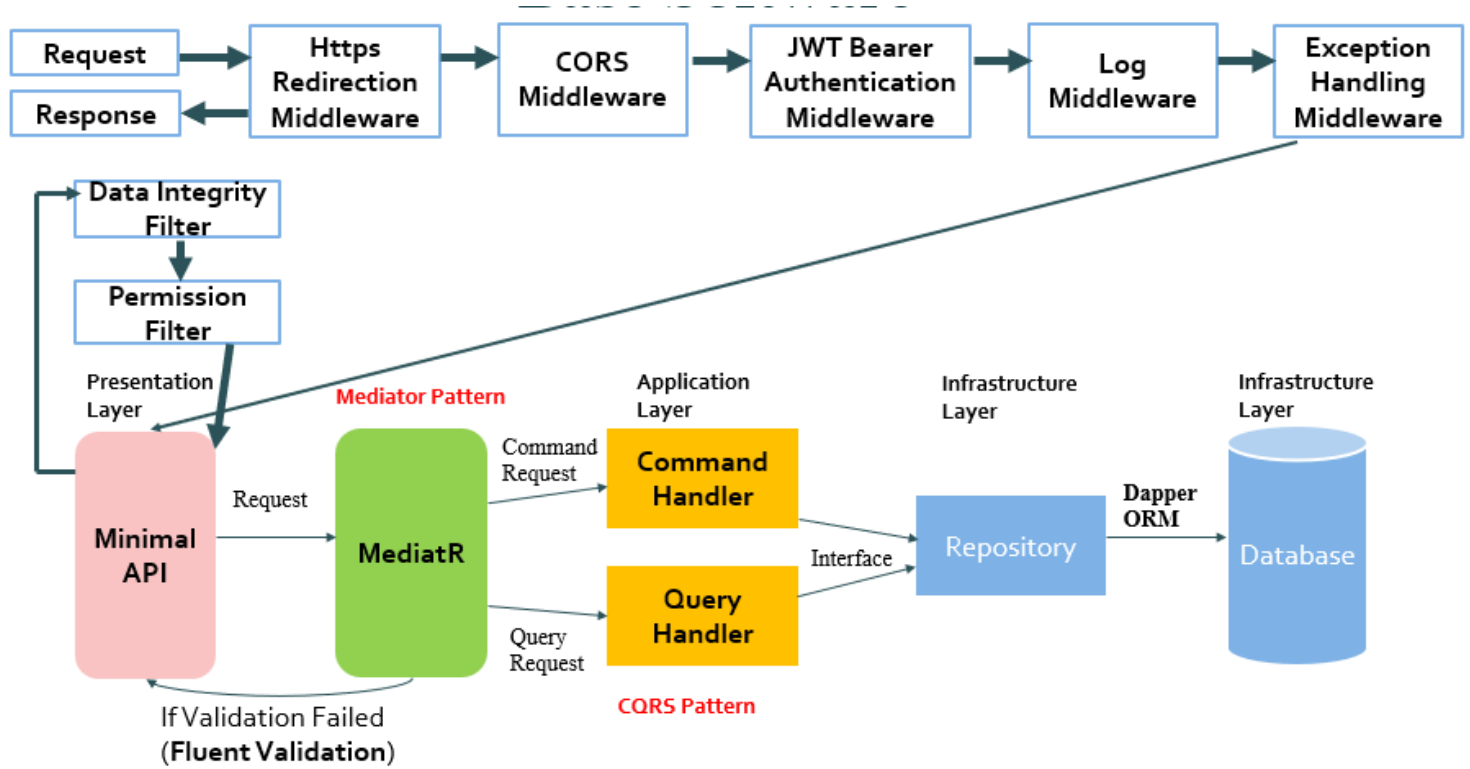


Figure 3.4.4.7: Request Flow in our Web API

3.4.5 Net Libraries

3.4.5.1 MediatR

MediatR is a library that contains mediator pattern implementation in .Net. This library is used for processing messages without dependencies in software applications. [38] MediatR library provides a clear separation between layers that wants to communicate through classes making them independent of each other. [10]. In the view of this statement, using MediatR library we can practically apply a core principle of clean architecture principle that refers to loose coupling between layers as explained in Mediator Pattern section. Another key point is that MediatR is the way to apply CQRS pattern in our System and will be analyzed later. [10]

MediatR Interfaces

In our project, there are 3 interfaces that we will use from MediatR library. ISender, IRequest and IRequestHandler. IRequest (see Figure 3.4.5.1: line 10) is an interface that will be implemented by the classes that will represent command or query **requests**. In other words, it requires to **declare the response object returned for this request** [10]. IRequestHandler aims to **define a handler for request** (see Figure 3.4.5.2 line 11). Therefore, IRequestHandler<request, response> is the interface implemented from the Handler classes to

handle command or query requests. In General, it requests the request and response classes to be declared and force Handler to implement Handle method. [10] IRequestHandler enforces request to implement IRequest<TResponse> interface and that TResponse of IRequest is the same class with response class in IRequestHandler (see Figure 3.4.5.2 line 21).

```

6 namespace MediatR;
7
8 //
9 // Summary:
10 //   Marker interface to represent a request with a response
11 //
12 // Type parameters:
13 //   TResponse:
14 //     Response type
15 public interface IRequest<out TResponse> : IBaseRequest
16 {
17 }
18 #if false // Decompile log

```

Figure 3.4.5.1: IRequest interface

```

9 namespace MediatR;
10
11 //
12 // Summary:
13 //   Defines a handler for a request
14 //
15 // Type parameters:
16 //   TRequest:
17 //     The type of request being handled
18 //
19 //   TResponse:
20 //     The type of response from the handler
21 public interface IRequestHandler<in TRequest, TResponse> where TRequest : IRequest<TResponse>
22 {
23     //
24     // Summary:
25     //   Handles a request
26     //
27     // Parameters:
28     //   request:
29     //     The request
30     //
31     //   cancellationToken:
32     //     Cancellation token
33     //
34     // Returns:
35     //   Response from the request
36     Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken);
37 }
38 #if false // Decompile log

```

Figure 3.4.5.2: IRequestHandler interface

Send() method from ISender interface is responsible to achieve the communication between the classes. Internally, this method finds the IRequestHandler of a query or a command request passed as an argument and executes the method handle inside the class implementing that interface.

```

6 namespace MediatR;
7
8 //
9 // Summary:
10 //   Defines a mediator to encapsulate request/response and publishing interaction
11 //   patterns
12 public interface IMediator : ISender, IPublisher
13 {
14 }
15 #if false // Decompile log

```

Figure 3.4.5.3: IMediator interface

```

10 namespace MediatR;
11
12 //
13 // Summary:
14 //   Send a request through the mediator pipeline to be handled by a single handler.
15 public interface ISender
16 {
17     //
18     // Summary:
19     //   Asynchronously send a request to a single handler
20     //
21     // Parameters:
22     //   request:
23     //     Request object
24     //
25     //   cancellationToken:
26     //     Optional cancellation token
27     //
28     // Type parameters:
29     //   TResponse:
30     //     Response type
31     //
32     // Returns:
33     //   A task that represents the send operation. The task result contains the handler
34     //   response
35     Task<TResponse> Send<TResponse>(IRequest<TResponse> request, CancellationToken cancellationToken = default(Cancellation));

```

Figure 3.4.5.4: ISender interface

How to achieve loose coupling using MediatR in our System?

In Figure 3.4.5.5 an anonymous route handler function for endpoint “baseUrl/GetEmployeesReportExcell” located in the Presentation layer wants to communicate with a handler located in Application layer. Therefore, using Send() method from the ISender interface through IMediator interface, we achieve communication between two different layers of clean architecture through mediator pipeline. In that phase, it’s important to clarify that MediatRGet() extension method coming from an internal company library is the corresponding method to MapGet method provided from Minimal APIs. Those extension methods basically contains Send() method and returning the response to help developer minimizing the code into just a single line.

```
//test your report from swagger
employeeProjects.MapPost("download", async (IMediator mediator, ViewsGetEmployeesReportExcelQuery request) =>
{
    var response = await mediator.Send(request);
    return Results.File(
        fileContents: response.Content,
        fileDownloadName: response.FileName,
        contentType: response.ContentType
    );
});
employeeProjects.MediatRGet<ViewsGetEmployeesReportExcelQuery, ReportDto>("GetEmployeesReportExcell");
```

Figure 3.4.5.5: EmployeeProjectsEndpoints

How to apply CQRS pattern using MediatR in our System?

Our record UsersSaveCommand (see Figure 3.4.5.7) inherits from UsersSaveRequestDto and IRequest interface. By implementing the IRequest interface it forces our request to return an object UsersSaveResponse that contains a Boolean value whether the insert or update was successful. Then, UserSaveHandler class inherits from IRequestHandler<UsersSaveCommand, UsersSaveResponse > which means that UserSaveHandler class will handle UsersSaveCommand (request) which return a UsersSaveResponse (response). IRequestHandler checks that UsersSaveCommand has implemented IRequest<TResponse> interface and that TResponse object is also UsersSaveResponse object otherwise it will throw an error. As a result, whenever Send() method is called the request parameter will be sent through mediator pipeline to be handled by the query or the command that implements the IRequestHandler interface and the request parameter matches the request of the interface. Then, the method handle inside the Handler will be executed.

```
public record UsersSaveResponse : BooleanDto;
public record UsersSaveCommand : UsersSaveRequestDto, IRequest<UsersSaveResponse>;
public class UsersSaveHandler : IRequestHandler<UsersSaveCommand, UsersSaveResponse>
```

Figure 3.4.5.7: UsersSaveHandler figure

3.4.5.2 Fluent Validation

Validation is a cross-cutting concern in a RESTful API where we need to ensure that request is valid before processing it. In Minimal APIs there is not any built-in validation mechanism to extend. The most used approach is to check request right before processing command in the same class with the handler (see Figure 3.4.5.2.1). This means that validation is tightly coupled to the command handler making the maintenance harder as the complexity of the validation increases. Each change in the validation of the request will affect the handler since its code can grow a lot. [14]

Filling the class with if statement and exception throwing is a beginner practice and not a professional one. A modern way to validate a request is to use the data annotation attribute directly to the request class (see Figure 3.4.5.2.2). Using this approach, POCO class/record will be polluted by the data annotation attribute making the entity class tightly coupled to validation rather than act as a DTO. [15]

```
internal sealed class ShipOrderCommandHandler
    : IRequestHandler<ShipOrderCommand>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IShippingService _shippingService;
    private readonly ShipmentSettings _shipmentSettings;

    public async Task Handle(
        ShipOrderCommand command,
        CancellationToken cancellationToken)
    {
        if (!_shipmentSettings
            .SupportedCountries
            .Contains(command.ShippingAddress.Country))
        {
            throw new ArgumentException(nameof(ShipOrderCommand.Address));
        }

        var order = _orderRepository.Get(command.OrderId);

        _shippingService.ShipTo(
            command.ShippingAddress,
            command.ShippingMethod);
    }
}
```

Figure 3.4.5.2.1: Common Validation[14]

```
public record UserResource(
    [Required]
    [MaxLength(20, ErrorMessage = "Username maximum length is 20 character")]
    string Username,

    [Required]
    [MaxLength(20, ErrorMessage = "Password maximum length is 20 character")]
    string Password,

    [Required]
    [MaxLength(50, ErrorMessage = "FirstName maximum length is 50 character")]
    string FirstName,

    [Required]
    [MaxLength(50, ErrorMessage = "LastName maximum length is 50 character")]
    string LastName,

    [Required]
    [EmailAddress]
    [MaxLength(50)]
    string Email,
```

Figure 3.4.5.2.2: Modern Validation [15]

Fluent Validation

Fluent Validation is the library that will replace data annotation validation attributes by separating validation rules from entity and DTO classes. This way we can also make dynamic and conditional validations. [16] Fluent Validation is a popular validation open-source library for .NET, which uses a fluent interface and lambda expressions for building strongly typed validation rules. [16]

Why Fluent Validation?

A good rule to follow for each developer in Restful APIs is that an invalid command or query request should never reach the handler. That's why we use Fluent Validation to separate request validation from request handling. Fluent Validation can be used to validate user input checking for null values, empty strings, required fields, string length (input validation) or it can be used to validate business rules such as checking for required preconditions (business validation) before processing a request [14]. It allows developers in a very easy way to define built-in or custom validation rules either synchronous or asynchronous for a request using a simple, readable, and maintainable syntax accelerating development cycle. Integration with various frameworks and libraries such as ASP.NET Core, Minimal API and Web API is easy since it is a .Net library. Fluent Validation has also a small initial learning curve. [17]

Business & Input Validation Examples Through Project

In this section we are going to focus on business validation examples but through this you can see that input validation is also done since before ensuring that business rules are satisfied you need to make sure that input is valid. A good example is when we want to ensure that VatNo of a company is unique then this refers to a rule related about business validation (see Figure 3.4.5.2.3). First, we use `NotNull`, `NotEmpty`, `MaxLength` to validate input and then we can use the asynchronous Method `MustAsync` to make sure that VatNo in the Company Entity that we are trying either to insert or update does not already exist in database. This can be achieved by getting an `ICompanyRepository` instance where inside `BeUniqueVatNo` we will use this instance to call a method to get all Vat Numbers in database. After, we can simply return true or false whether the VatNo passed as an argument is contained in existing Vat Numbers. This way `MustAsync` will throw an error in case `BeUniqueVatNo` returns false. Leveraging Fluent Validation, we ensure both input data and business logic are satisfied before handling the request achieving greater performance since our request will not pass from all those layers until database will throw unique constraint error. As a result, we can focus on the application-logic by avoiding polluting handlers, entities, database, and user interface with validation checks.


```

public class CompanySaveValidator : AbstractValidator<CompanySaveCommand>
{
    private readonly ICompanyRepository _companyRepository;

    0 references
    public CompanySaveValidator(ICompanyRepository companyRepository)
    {
        _companyRepository = companyRepository;

        RuleFor(x => x.Company.VatNo).NotNull().NotEmpty().MaxLength(50)
        // Check That Unique Constraint on VatNo is satisfied Either on Insert or Update
        .MustAsync(async (company, vatNo, cancellationTokens) =>
        {
            return await BeUniqueVatNo(vatNo, company.CompanyId, cancellationTokens);
        })
        .WithMessage(Resource.CompanyVatNoExist);
    }
}

```

Figure 3.4.5.2.3: CompanySaveValidator

Based on this, the next example refers to a situation where fluent validation is not used, and we pollute our typescript file in angular project with validation. As you see in Figure 3.4.5.2.4, we just check whether the two passwords match. However, a better practice would be to add this validation to ChangePasswordValidator in the API project. In Figure 3.4.5.2.5, we need to decrypt the password (line 9) since passwords are encrypted when Angular and API share those sensitive data and then we need to check whether the new password matches the regular expression (line 11) that enforce user to add a strong password. Then, (line 14) we check whether the new password matches the verification password. That's why fluent validation is so powerful because it helps us to remove this logic from typescript file in those cases and move it to the corresponding separate validator class.

```

if (this.form.value.password !== this.form.value.verifyPassword) {
    this.form.controls.verifyPassword.setErrors({ 'NotMatched': true });
    return;
}

```

Figure 3.4.5.2.4: Validation in Angular

```

5 public class ChangePasswordValidator : AbstractValidator<ChangePasswordCommand>
6 {
7     0 references
8     public ChangePasswordValidator(IEncryptionSimpleService encryptionSimpleService)
9     {
10         RuleFor(r => encryptionSimpleService.Decrypt(r.NewPassword))
11         .NotEmpty()
12         .Matches("^(?=.*{8,})(?=.*[\\d])(?=.*[a-z])(?=.*[A-Z])(?=.*[!@#$%^&*+=]).*$")
13         .WithMessage("PasswordValidation");
14
15         RuleFor(r => r).Must(request =>
16         {
17             if (encryptionSimpleService.Decrypt(request.NewPassword) != encryptionSimpleService.Decrypt(request.VerifyPassword))
18                 return false;
19             return true; //rule passed
20         }).WithMessage("NotMatchedPasswords");
21     }
22 }

```

Figure 3.4.5.2.5: ChangePasswordValidator

The last business-related example is associated with tokens because when a user wants to set a new password this is done via a token URL sent in email for safety reasons. In SetNewPasswordValidator we need to check that some constraints such as tokens are not used or expired after communicating with database through repository interface to get that info.


```

7  public class SetNewPasswordValidator : AbstractValidator<SetNewPasswordCommand>
8  {
9
10     0 references
11     public SetNewPasswordValidator(IUserRepository userRepo )
12     {
13         ClassLevelCascadeMode = CascadeMode.Stop;
14
15         RuleFor(r => r.UserToken).MustAsync(async (userToken, cancellation) =>
16         {
17             if (string.IsNullOrEmpty(userToken))
18                 return false;
19
20             var validatorInfo = await userRepo.GetValidatorInfoForResettingPassword(userToken!);
21
22             if (validatorInfo is null)
23                 return false;
24
25             if (validatorInfo.IsUserTokenUsed)
26                 return false;
27
28             if (DateTimeOffset.UtcNow > validatorInfo.UserTokenExpireAt)
29                 return false;
30
31             return true; //rule passed
32         }).WithMessage("InvalidUser");
33     }

```

Figure 3.4.5.2.6: SetNewPasswordValidator

Exploring Fluent Validation Validators & Features

Fluent Validation give us the ability to check whether a property is not null, empty string, whitespace, empty collection and not the default value of the type of the property (not zero if is integer for example) giving us the change to use it with different types of properties (see Figures 3.4.5.2.7, 3.4.5.2.8 and 3.4.5.2.9). All this can be achieved using the built-in validator `NotEmpty()` method of Fluent Validation Library (`NotEmpty()`). `NotEmpty()` can replace `NotNull().GreaterThan(0)` or `NotNull().GreaterThanOrEqualTo(1)` (see Figure 3.4.5.2.7) since it contains those checks inside the function. We can also provide our custom message of error along `WithMessage()` method and check whether a value is contained in a list using `In()` method (see Figure 3.4.5.2.9). It gives us the chance to use functions like `GreaterThan(0)`, `MaxLength()`, `EmailAddress()`, `Length()` (see Figure 3.4.5.2.10 and Figure 3.4.5.2.11). Another important function that library provides is `When()` since you can use it if you want to apply more than one rule based on a condition (see Figure 3.4.5.2.12). Moreover, developer could check if all the elements in the list inside `RuleForEach` satisfy the rules given under `ChildRules()` (see Figure 3.4.5.2.13). A very powerful feature that this library offers is the custom rules as shown in Figure 3.4.5.2.14. The use case is that we have a request we have a list with permissions and role so we can update the database with the given permissions for that role. In the example, we just replace the commented and most obvious validation rules for checking if an integer is greater than zero or if a list is empty just to show how we can define custom rule. Obviously, library has a lot of other methods and features that can be useful for specific use cases of the application (for more info <https://docs.fluentvalidation.net/en/latest/index.html>).

```

0 references
public AuthoritiesDeleteValidator()
{
    RuleFor(r => r.Value).NotNull().GreaterThanOrEqualTo(1);
}

```

Figure 3.4.5.2.7: Fluent Validation Features 1

```

RuleFor(r => r.ConsumerCountryId).NotEmpty();

```

Figure 3.4.5.2.8: Fluent Validation Features 2

```

RuleFor(r => r.Document.FileExtension)
    .NotEmpty()
    .In(AllowedFileTypes).WithMessage(string.Format(Resource.Validator_In, "{PropertyName}", string.Join(", ", AllowedFileTypes)))
;

```

Figure 3.4.5.2.9: Fluent Validation Features 3

```

RuleFor(r => r.Pagination.PageNumber).GreaterThan(0);
RuleFor(r => r.Pagination.PageSize).GreaterThan(0);

RuleFor(r => r.DocumentFilters.Name).MaxLength(150).When(x => !string.IsNullOrEmpty(x.DocumentFilters.Name));
RuleFor(r => r.DocumentFilters.CategoryId).GreaterThan(0).When(x => x.DocumentFilters.CategoryId is not null);

```

Figure 3.4.5.2.10: Fluent Validation Features 4

```

RuleFor(r => r.CompanyEmail).EmailAddress().Length(3, 255);

```

Figure 3.4.5.2.11: Fluent Validation Features 5

```

//general complaint
When(r => r.ComplaintCategoryId == (int)ComplaintCategoriesEnum.General, () =>
{
    RuleFor(r => r.GeneralComplaintDescription).NotEmpty().MaxLength(400);
    RuleFor(r => r.GeneralComplaintActionsDescription).MaxLength(4000).When(
        RuleFor(r => r.GeneralComplaintIsGDPRAccepted).NotNull());
});

//other complaint

```

Figure 3.4.5.2.12: Fluent Validation Features 6

```

RuleForEach(r => r.LawTranslations).Cascade(CascadeMode.Stop).ChildRules(translation =>
{
    translation.RuleFor(r => r.Code).NotEmpty().MaxLength(100).When(x => !string.IsNullOrEmpty(x.Code)); ;
    translation.RuleFor(r => r.Value).NotEmpty().MaxLength(4000).When(x => !string.IsNullOrEmpty(x.Value)); ;
    translation.RuleFor(r => r.LanguageId).NotNull().GreaterThanOrEqualTo(1);
});

```

Figure 3.4.5.2.13: Fluent Validation Features 7

```

public class RelationsUpdatePermissionsForRoleCommandValidator
    : AbstractValidator<RelationsUpdatePermissionsForRoleCommand>
{
    0 references
    public RelationsUpdatePermissionsForRoleCommandValidator()
    {
        //RuleFor(r => r.RoleId).GreaterThan(0);
        RuleFor(r => r.RoleId).Custom((id, context) =>
        {
            if (id is not > 0)
                context.AddFailure("Role Id must greater than 0!");
        });

        //RuleFor(r => r.PermissionsIds)
        // .Must(list => list.Any())
        // .WithMessage("Permissions must NOT be null");
        RuleFor(r => r.PermissionsIds).Custom((list, context) =>
        {
            if (list is null)
                context.AddFailure("Permissions must NOT be null");
        });
    }
}

```

Figure 3.4.5.2.14: Fluent Validation Features 8

Fluent Validation Integration With MediatR and CQRS & Design Decisions

Based on all the previous example it should be clear to understand that to define a set of validation rules for a particular request, you will need to create a class that inherits from `AbstractValidator<T>`, where T is the request object that is going to be validated (see Figure 3.4.5.2.14). To specify a validation rule for a particular property you can use any of the methods explained in Fluent Validation features, passing a lambda expression that indicates the property that you wish to validate. That's all for now. Obviously, someone can wonder now how we can change the flow of the project so when we use `IMediator Send()` method instead of going to the appropriate handler to go first in the validator and if request validation is satisfied then to go to the handler. Before explaining how mediator pipeline can solve this problem let's discuss some logical approaches to this issue. A custom approach is before `Send()` method to pass the control to the Validator. However, this violates clean architecture core principle about separation of concerns since validating is not related to the presentation layer but in application layer where use cases logic is placed. Therefore, a better approach is to pass the control to the validator from the use case (handler) through `ValidateAndThrow()` method provided from library (see Figure 3.4.5.2.15). This requires injecting `IValidator<T>` service where T stands for the query or command handler to be able to run validation through this instance. But this forces you to define an explicit dependency in every handler to `IValidator` interface.[14] To overcome the disadvantages of the previous approaches we validate a request through MediatR pipeline. `AddValidatorsFromAssemblies()` method (see Figure 3.4.5.2.16) tells `FluentValidation` to look for any validator in the assembly (API project) and automatically register them all to the application [14]. However, registering MediatR first through `AddMediatR()` (see Figure 3.4.5.2.16) and then Validators we basically add `FluentValidation Middleware` into MediatR pipeline so it can pass the control to the corresponding Validator of the command or query request passed in `Send()` method and then go to handler of that request.

```
internal sealed class ShipOrderCommandHandler
    : IRequestHandler<ShipOrderCommand>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IShippingService _shippingService;
    private readonly IValidator<ShipOrderCommand> _validator;

    public async Task Handle(
        ShipOrderCommand command,
        CancellationToken cancellationToken)
    {
        _validator.ValidateAndThrow(command);

        var order = _orderRepository.Get(command.OrderId);

        _shippingService.ShipTo(
            command.ShippingAddress,
            command.ShippingMethod);
    }
}
```

Figure 3.4.5.2.15: Fluent Validation inside Handler

```

2. References
void RegisterMediatorFluentValidationAutoMapper(IServiceCollection services, IEnumerable<Type> types)
{
    //register mediator, mapper and fluent validation for all the above domains
    services
        .AddMediatR(types.Select(t => t.Assembly).ToArray())
        .AddValidatorsFromAssemblies(types.Select(t => t.Assembly))
        .AddAutoMapping(types.Select(t => t.Assembly))
    ;
}

```

Figure 3.4.5.2.16: Fluent Validation Through MediatR pipeline

3.4.5.3 Dapper

Dapper is an open-source object-relational mapping (ORM) library for .NET applications, designed to provide developers with a quick and easy way to access data from databases. Available as a NuGet package, it is lightweight and fast, making it an ideal choice for applications that require low latency and high performance. Dapper specializes in reading data from a database and mapping it to objects, it excels in scenarios where there's a lot of reading happening, but not much writing. Therefore, the main reason we use Dapper instead of entity framework is speed because application most requests are to get data. [39]

Dapper boasts a range of features that make it an attractive option for data access in .NET applications. Dapper allows you to execute raw SQL queries and stored procedures, giving you full control over your database interactions. Dapper can map query results directly to .NET objects. Dapper supports both asynchronous and synchronous database queries and batching multiple queries together into a single call, which can significantly reduce the number of round trips to the database and improve performance. Dapper supports parameterized queries, helping to protect against SQL injection attacks by ensuring that user input has properly escaped. [39]

Dapper in our System

In our project, IDataAccessLayerContext interface encapsulates dapper integration with SQL Server. IDataAccessLayerContext is the instance where every repository class needs to connect to get access to the database and use the dapper functionalities. This interface from an internal company package injects an ADO.NET IDbConnection object because dapper works with any database system where there is an ADO.NET provider. Therefore, this IDbConnection represents an open connection to a data source which in our case is SQL Server. The only thing that a developer needs to do is to add the connection string for the database in settings.

Why Parameterized Queries are important?

Parameterized queries and stored procedure improving query execution plan caching. Using Dapper functions with parameters allows the database engine to parse and compile the query or stored procedure once and reuse the execution plan for subsequent executions with different parameter values. This reduces the overhead of parsing and compilation, leading to faster write operation execution.

Using parameterized queries with Dapper it ensures that data is inserted/updated/deleted safely without exposing vulnerabilities to SQL injection attacks. Parameterized queries are SQL statements where parameters values are used to represent variable values within the query string. Instead of directly embedding values into the query string, parameters are placeholders that are substituted with actual values at execution time. When you execute a parameterized query, you provide the parameter values separately from the query string and these parameters values are then securely passed to the database engine to replace the placeholders and execute the query. This allows the database engine to treat the SQL code and data separately. Since user input is not directly interpreted as part of the query Dapper will automatically sanitize the user input and if SQL injection attack is detected dapper escapes before executing the query and prevent SQL injection attacks. To sum up parameter values are treated as data rather than being a part of SQL executable code ensuring that structure of the query cannot be altered. [39]

How Dapper Provides Better Performance

Considering dapper documentation, we realize that dapper does not manage the caching of execution plans. Instead, it delegates this responsibility to the underlying database provider (e.g., SQL Server). Overall, Dapper with parameterized queries leverages benefits from the optimizations performed by SQL Server, to reuse cached execution plans efficiently.

Dapper contributes to speed improvements with efficient result mapping and optimized data access. Dapper excels at mapping query results to .NET objects with minimal overhead. Its lightweight and high-performance object mapping capabilities allow for fast and efficient processing of query results, even when dealing with large datasets. Dapper provides faster data access compared to heavier ORMs since it utilizes ADO.NET under the hood, leveraging its capabilities for connection management and data retrieval. [39]

To conclude, this chapter analyze integration of libraries and frameworks into the base software explaining why to choose each one. Moreover, it uses the features provided from those technologies to implement common features and add them to the base software. The process of creating this base software followed the principles of Generic architecture as proposed in chapter.

Chapter 4

Application Design

4.1 Database Design	54
4.1.1 Schemas	54
4.1.2 Tables	55
4.1.3 Database Diagrams	59
4.2 Clean Architecture in .Net Projects Structure	61
4.2.1 Infrastructure	61
4.2.2 Domain	62
4.2.3 Application	64
4.2.4 Presentation	66
4.3 Code Decisions	68
4.2.1 Domain	68
4.2.2 Infrastructure	69
4.2.3 Presentation	71

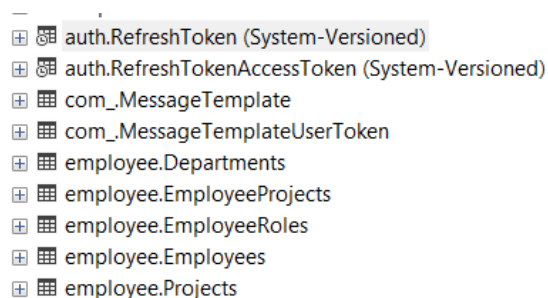
4.1 Database Design

This chapter focus on the implementation of the Employee Management System as a proof of concept of the generic architecture and the base software. In the following section, we deal with the database design of our application, creation of the .Net Projects based on our Generic Architecture. This chapter proves that having a set of technical details and common features from the base software we can use them to implement application-specific features without affected from technical details allowing us to focus on business logic. The code implementation of Employee Management Features must also align with the generic Architecture as proposed in Chapter 2.

4.1.1 Schemas

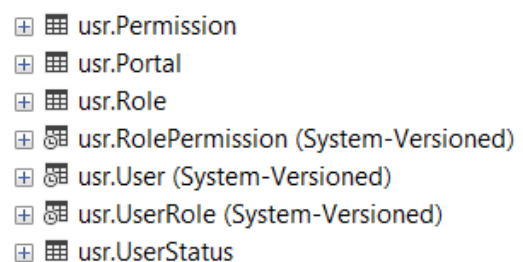
In our database design, we have organized the database into multiple schemas, with each schema containing a set of related tables. Chosen schemas are com that refers to common

management, usr to user management, employee to employee management and auth to authentication management. In figures 4.1.1.1 and 4.1.1.2, you can see the schema that each table belongs. This approach offers several significant benefits such as logical separation of database tables, stored procedures, user-defined objects and all database related stuff. This helps in organizing tables based on their functionality making the database more manageable and easier to understand. By using schemas, we can assign different permissions to different schemas to restrict the accessing or modifying specific parts of the database. Schemas help avoid name collisions by allowing the same table name to exist in different schemas. This is particularly useful in large applications where different modules might have tables with similar names. By structuring our database with schemas and tables, we achieve a well-organized, secure, and efficient database design that supports scalability, maintainability, and collaborative development. [40]



- + [grid icon] auth.RefreshToken (System-Versioned)
- + [grid icon] auth.RefreshTokenAccessToken (System-Versioned)
- + [grid icon] com_.MessageTemplate
- + [grid icon] com_.MessageTemplateUserToken
- + [grid icon] employee.Departments
- + [grid icon] employee.EmployeeProjects
- + [grid icon] employee.EmployeeRoles
- + [grid icon] employee.Employees
- + [grid icon] employee.Projects

Figure 4.1.1.1 Auth, Com, Employee Schemas



- + [grid icon] usr.Permission
- + [grid icon] usr.Portal
- + [grid icon] usr.Role
- + [grid icon] usr.RolePermission (System-Versioned)
- + [grid icon] usr.User (System-Versioned)
- + [grid icon] usr.UserRole (System-Versioned)
- + [grid icon] usr.UserStatus

Figure 4.1.1.2 Usr schema

4.1.2 Tables

In this section, we are going to explain the design of the database tables (see figures 4.1.2.1 – 4.1.2.5) to manage employees, departments, roles, and projects. Database design needs to aligns with normalization principles and I was based on this achieving to reach the third normal form (3NF). The design follows the principles of database normalization to ensure data integrity, reduce redundancy, and improve the efficiency of data operations. Below is the analysis of the design choices made for each table and why i created those tables based on normalization.

Project table stores information about various within the organization. Departments keeps track of different departments in the organization. The EmployeeRoles table contains the roles that an employee can have in a project. Employee table contains information about employees while having a reference on Department table since an employee belongs to one department and a department can have many employees. This means that is 1-N relationship and no further table

like EmployeeDepartment is needed. Obviously, it's critical to understand why we created that EmployeeProject Table. Based on the requirements an employee can work in many projects and a project can have many employees working on it. An employee can work in a project having different roles and a role can be chosen by many employees. So, we have Employee-Roles N-N relationship, Employee-Project N-N. However, an employee that works in a project needs to select roles and an employee that has roles those roles must also be specified in which project it has each role. That lead us to create a single table storing EmployeeId, EmployeeRoleId and ProjectId as composite key so we can force to add a record that contains the info of who is the employee in what project works and which role he/she has in the specific project this employee. To sum up, composite primary key ensures that each combination of EmployeeId, ProjectId, and EmployeeRoleId is unique. This accurately models the many-to-many relationship between employees, projects, and roles.

Let's examine now in what Normal Form our tables are. All tables have atomic columns without multivalues, or composite columns and they also have a unique identifier which is the primary key. This means that 1NF is satisfied. In 2NF, all non-key attributes must depend on the primary key. Since each table has a single-column primary key except for EmployeeProjects, all non-key columns depend on the primary key. For example, in the Employees table all columns depend on EmployeeId. Imagine department names stored in Employees table instead of Departments table then this would violate 2NF because department name would be dependent on a non-key-attribute like departmentId instead of the primary key like EmployeeId. 3NF states that no transitive dependencies must exist which means that non-key attributes are not dependent on other non-key attributes. In the Employees table, the non-key attributes (FirstName, LastName, SignOffStatus, DepartmentId) depend only on the primary key (EmployeeId), and not on each other's relationships. A possible violation of 3NF in Employees Table could be if SignOffStatus for some departments is the same. Let's say DepartmentA has SignOffStatus=0, DepartmentB has SignOffStatus=1 while other departments have mixed SignOffStatus values. This dependency means that knowing the DepartmentId can help us determine SignOffStatus for some departments introducing a transitive dependency (EmployeeId -> DepartmentId -> SignOffStatus). However, our data doesn't have this relation between those 2 columns so splitting DepartmentId and SignOffStatus to a different table is not necessary. EmployeeProjects satisfies both 2NF and 3NF because we don't have non-key attributes at all in the table. [41]

The reason why we follow normalization is because by splitting data into related tables, you minimize redundancy. For example, department names are stored once in the Departments

table rather than being repeated for each employee. Normalization helps maintain data integrity by ensuring that data dependencies make sense. For example, the foreign key constraints prevent orphaned records and maintain consistency across tables. Changes to data like updating a department name need to be made in only one place, reducing the risk of anomalies. [41]

In summary, database design effectively normalizes the data and maintains referential integrity through appropriate foreign key constraints and primary keys. This design minimizes data redundancy and ensures data integrity, aligning with the principles of database normalization.

```
CREATE TABLE [employee].[Departments] (  
    [DepartmentId] INT IDENTITY (1, 1) NOT NULL,  
    [Name] VARCHAR (50) NOT NULL,  
    CONSTRAINT [PK_Departments] PRIMARY KEY CLUSTERED ([DepartmentId] ASC)  
);
```

Figure 4.1.2.1: Departments Table

```
CREATE TABLE [employee].[Projects] (  
    [ProjectId] INT IDENTITY (1, 1) NOT NULL,  
    [Name] VARCHAR (50) NOT NULL,  
    CONSTRAINT [PK_Projects] PRIMARY KEY CLUSTERED ([ProjectId] ASC)  
);
```

Figure 4.1.2.2: Projects Table

```
CREATE TABLE [employee].[EmployeeRoles] (  
    [EmployeeRoleId] INT IDENTITY (1, 1) NOT NULL,  
    [Type] VARCHAR (50) NOT NULL,  
    CONSTRAINT [PK_EmployeeRoles] PRIMARY KEY CLUSTERED ([EmployeeRoleId] ASC)  
);
```

Figure 4.1.2.3: EmployeeRoles Table

```
CREATE TABLE [employee].[Employees] (  
    [EmployeeId] INT IDENTITY (1, 1) NOT NULL,  
    [FirstName] VARCHAR (50) NOT NULL,  
    [LastName] VARCHAR (50) NOT NULL,  
    [SignOffStatus] BIT NOT NULL,  
    [DepartmentId] INT NULL,  
    CONSTRAINT [PK_Employees] PRIMARY KEY CLUSTERED ([EmployeeId] ASC),  
    CONSTRAINT [FK_Departments_DepartmentId] FOREIGN KEY ([DepartmentId])  
    REFERENCES [employee].[Departments] ([DepartmentId]) ON DELETE SET NULL ON UPDATE CASCADE  
);
```

Figure 4.1.2.4: Employees Table

```

CREATE TABLE [employee].[EmployeeProjects] (
    [EmployeeId] INT NOT NULL,
    [ProjectId] INT NOT NULL,
    [EmployeeRoleId] INT NOT NULL,
    CONSTRAINT [PK_EmployeeProjects] PRIMARY KEY CLUSTERED ([EmployeeId] ASC, [ProjectId] ASC, [EmployeeRoleId] ASC),
    CONSTRAINT [FK_EmployeeProjects_EmployeeId] FOREIGN KEY ([EmployeeId])
    REFERENCES [employee].[Employees] ([EmployeeId]) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT [FK_EmployeeProjects_EmployeeRoleId] FOREIGN KEY ([EmployeeRoleId])
    REFERENCES [employee].[EmployeeRoles] ([EmployeeRoleId]) ON DELETE CASCADE ON UPDATE CASCADE,
    CONSTRAINT [FK_EmployeeProjects_ProjectId] FOREIGN KEY ([ProjectId])
    REFERENCES [employee].[Projects] ([ProjectId]) ON DELETE CASCADE ON UPDATE CASCADE
);

```

Figure 4.1.2.5: EmployeeProjects Table

Analyzing the tables related to user management is crucial to understand authorization in our application. I want to mention that normalization principles used in the previous section are also followed to the creation of the tables below. In User table we have all the users, Roles table has all Role of the system and Permission tables stored all the permission for the application. A user can have multiple roles and a role can be given to many Users. A role can have many permissions and each permission can be given to many roles. Those many to many relationships forced us to create UserRole (see Figure 4.1.2.6) and RolePermission tables (see Figure 4.1.2.7) to resolve those relationships adhering with normalization principles explained in previous section.

```

CREATE TABLE [usr].[UserRole] (
    [UserId] BIGINT NOT NULL,
    [RoleId] BIGINT NOT NULL,
    [ValidFrom] DATETIME2 (7) GENERATED ALWAYS AS ROW START HIDDEN DEFAULT (sysutcdatetime()) NOT NULL,
    [ValidTo] DATETIME2 (7) GENERATED ALWAYS AS ROW END HIDDEN DEFAULT (CONVERT([datetime2], '9999-12-31 23:59:59.9999999')) NOT NULL,
    CONSTRAINT [PK_UserRole] PRIMARY KEY CLUSTERED ([UserId] ASC, [RoleId] ASC),
    CONSTRAINT [FK_UserRole_RoleId] FOREIGN KEY ([RoleId]) REFERENCES [usr].[Role] ([RoleId]),
    CONSTRAINT [FK_UserRole_UserId] FOREIGN KEY ([UserId]) REFERENCES [usr].[User] ([UserId]),
    PERIOD FOR SYSTEM_TIME ([ValidFrom], [ValidTo])
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE=[usr].[UserRoleHistory], DATA_CONSISTENCY_CHECK=ON));

```

Figure 4.1.2.6: UserRole Table

```

CREATE TABLE [usr].[RolePermission] (
    [RoleId] BIGINT NOT NULL,
    [PermissionId] BIGINT NOT NULL,
    [ValidFrom] DATETIME2 (7) GENERATED ALWAYS AS ROW START HIDDEN DEFAULT (sysutcdatetime()) NOT NULL,
    [ValidTo] DATETIME2 (7) GENERATED ALWAYS AS ROW END HIDDEN DEFAULT (CONVERT([datetime2], '9999-12-31 23:59:59.9999999')) NOT NULL,
    CONSTRAINT [PK_RolePermission] PRIMARY KEY CLUSTERED ([RoleId] ASC, [PermissionId] ASC),
    CONSTRAINT [FK_RolePermission_PermissionId] FOREIGN KEY ([PermissionId])
    REFERENCES [usr].[Permission] ([PermissionId]),
    CONSTRAINT [FK_RolePermission_RoleId] FOREIGN KEY ([RoleId]) REFERENCES [usr].[Role] ([RoleId]),
    PERIOD FOR SYSTEM_TIME ([ValidFrom], [ValidTo])
)

```

Figure 4.1.2.7: RolePermission Table

4.1.3 Database Diagrams

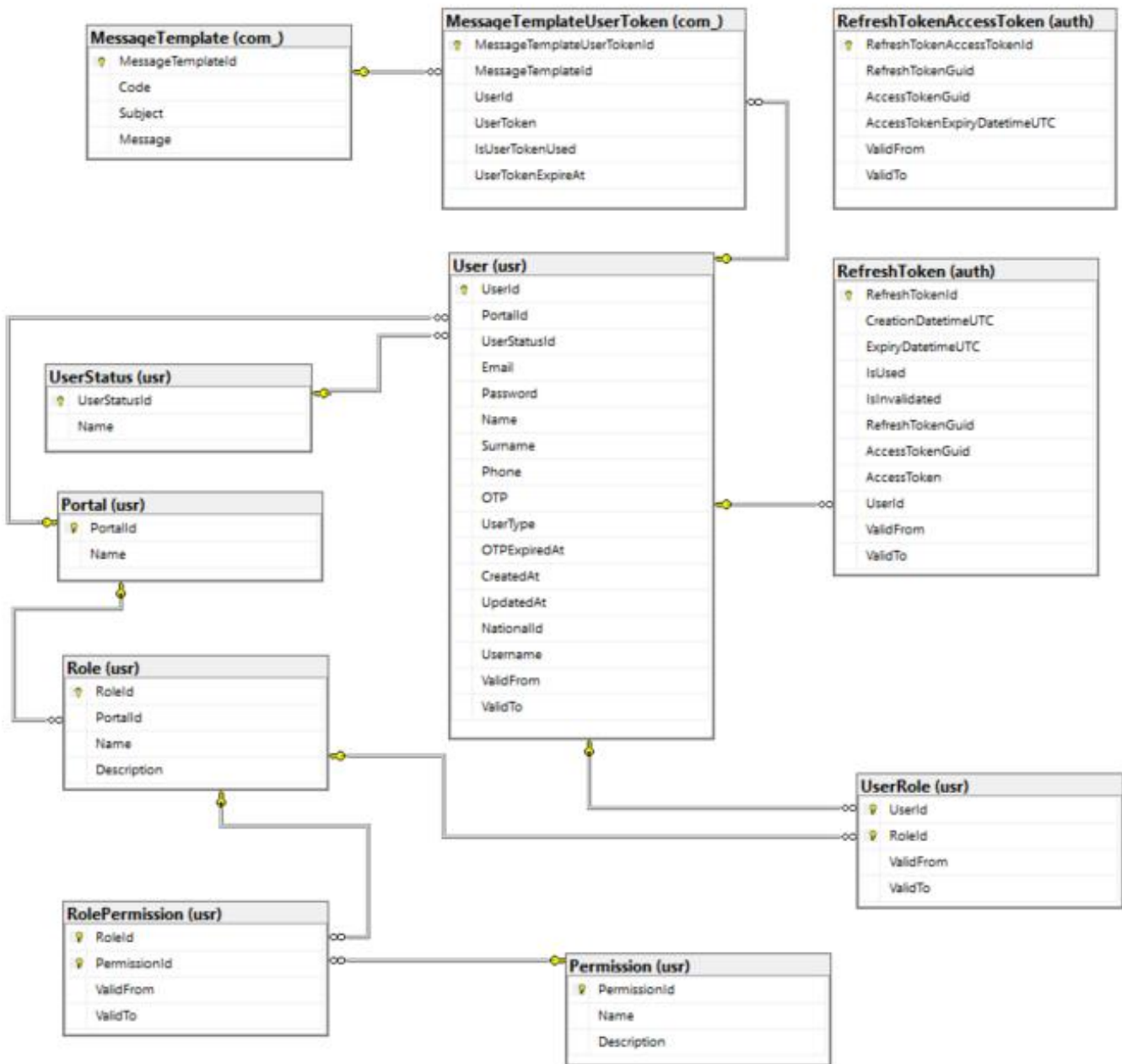


Figure 4.1.3.1: Usr, Com, Auth Schema Diagrams

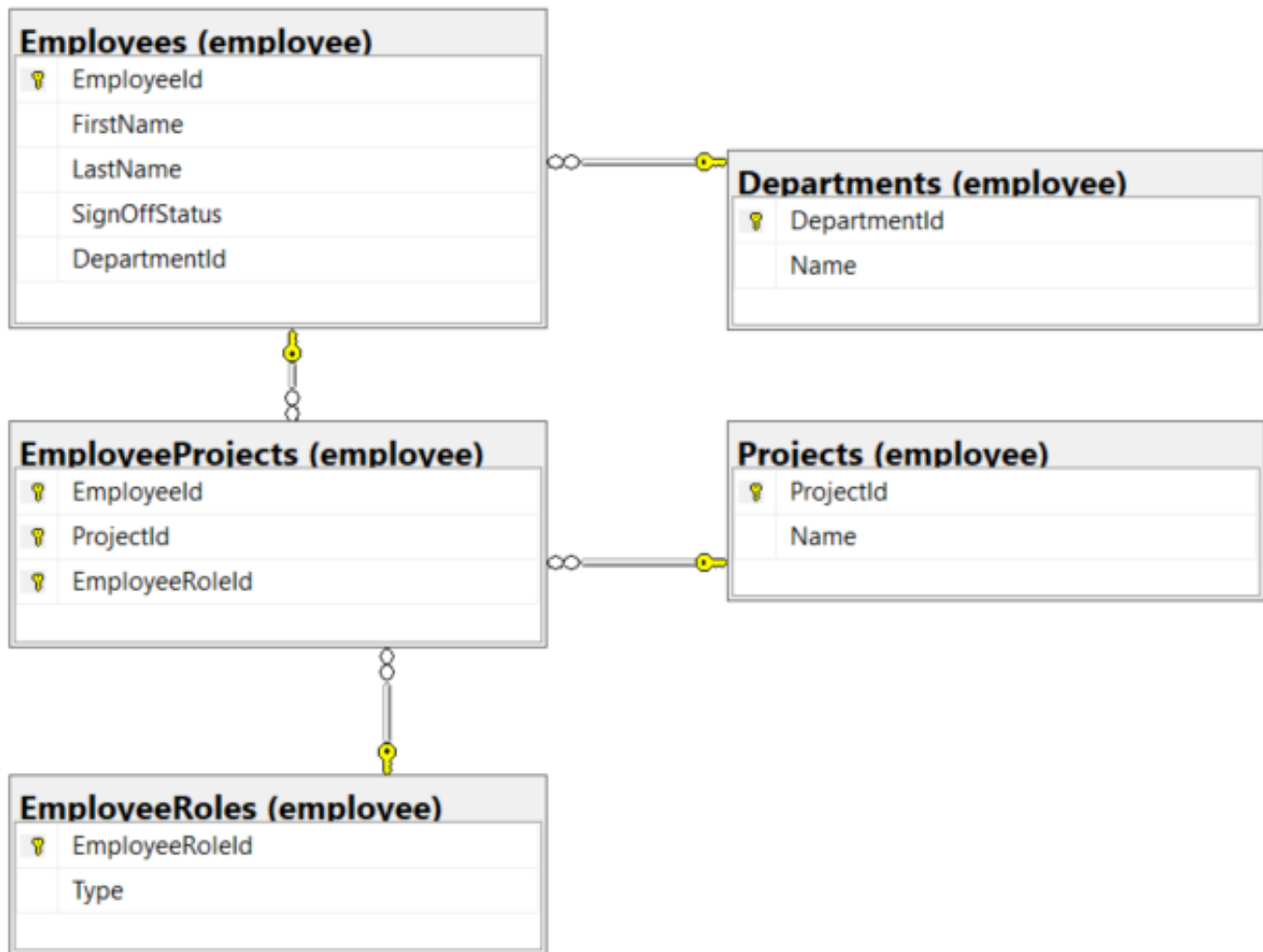


Figure 4.1.3.2: Employee Schema Diagram

4.2 Clean Architecture in .Net Projects Structure

Clean Architecture is a layered architecture that splits the project into four layers. Each of the layers is one or more .Net project. Our Foldering both in APIs and Angular project follows the logical splitting of schemas in database. In the following paragraphs, we are exploring the different layers of Our Clean Architecture and how each one of those can be implemented as a .Net project. Even though our solution contains multiple .Net Projects for each layer our Web API will be compiled to a single Assembly.

4.2.1 Infrastructure Layer

The Infrastructure layer is directly above the Application Layer and is responsible for communicating with database and external services (see figure 2.1.2.1). This layer contains repository implementation, external-services implementation, table entities, Database project (tables, stored procedures, etc). In figure 4.2.1.2, all .Net projects of Infrastructure Layer are shown. The Core.Infra.Database .NET project is our Database project in C#. This means that a developer can manage table writing C# in .Net project or using Microsoft SQL Server Management Studio writing SQL since those 2 are the exact same project. Table entities can be found under Infrastructure/DataAccessProject/Entities/{SchemaName}_Management and follow {TableName} naming convention. Repositories are positioned in Infrastructure/DataAccessProject/Repositories/{SchemaName}_Management and named like {DatabaseTableName}_Repository. Table entities and Repositories belongs to DataAccess project because they are essential to communicate with SQL Server using Dapper. Last, external services implementations like Emailing are placed in Common project under Infrastructure folder.

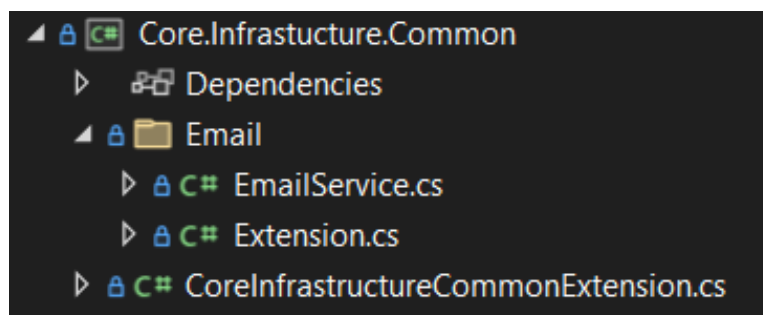


Figure 4.2.1.1: ExternalServices

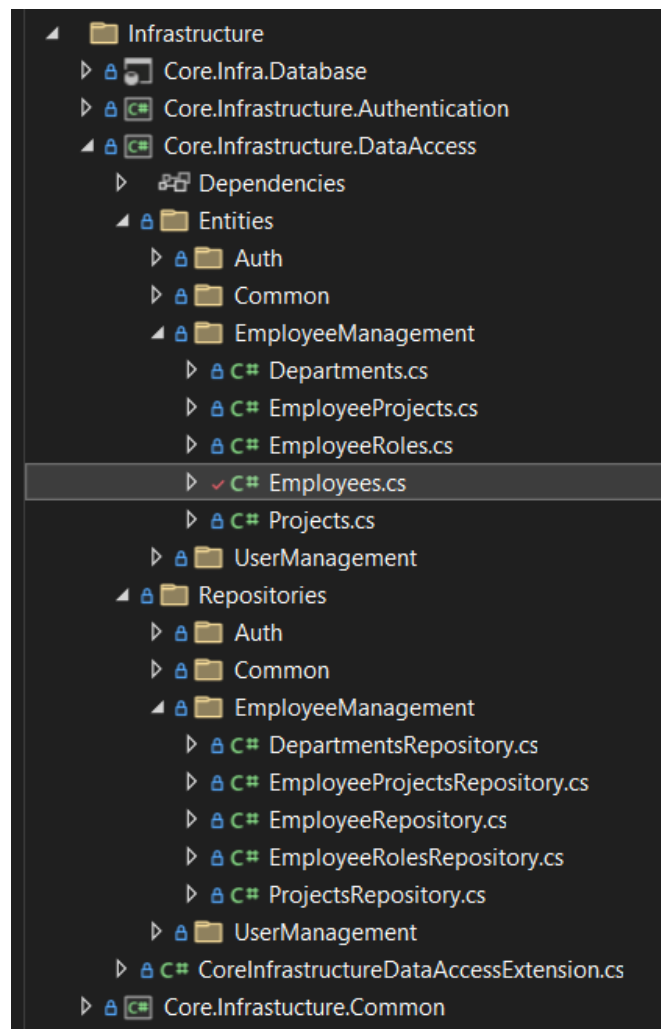


Figure 4.2.1.2: Infrastructure Layer

4.2.2 Domain Layer

In Domain Layer, which is the core layer of our project, we defined entities, abstractions, models and enums. This layer doesn't have any dependencies on any project since it's the innermost layer of the application. Figure 4.2.2.4 indicates that Domain Project doesn't reference any other project which confirms that inner layers should not depend on outer layer. Enforcing dependency Rule through managing references indicates how we apply Clean Architecture in our Project.

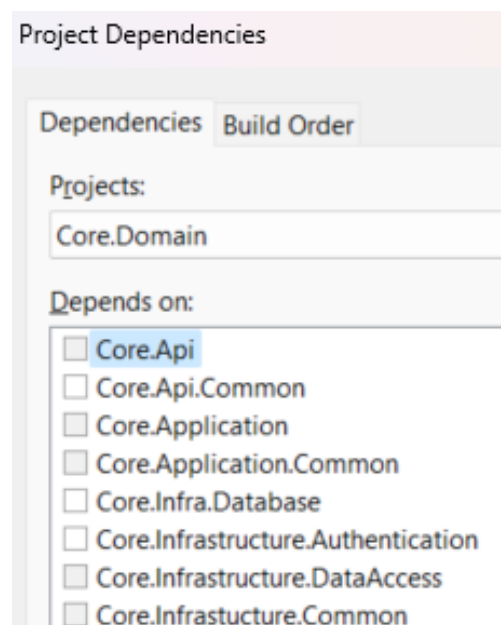


Figure 4.2.2.4: Domain Layer References

Domain Entities should be named from {DatabaseTableName}_Entity and foldering should be Domain/Entities/{SchemaName}_Management (see figure 4.2.2.1). Models name comes from {UseCaseName}_Model and they must be placed under Domain/Models/{SchemaName}_Management/{TableName} (see figure 4.2.2.2).

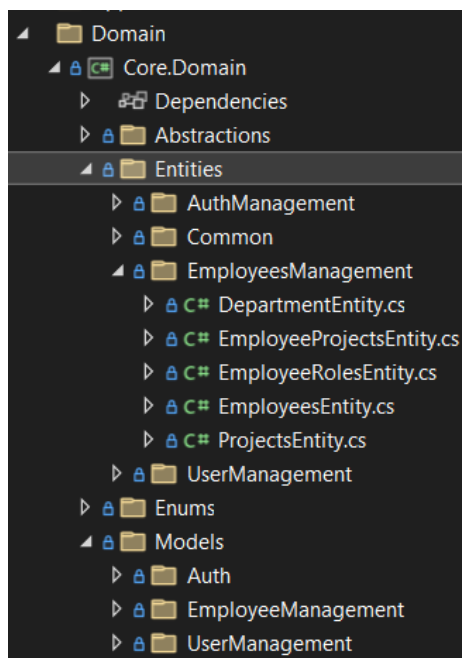


Figure 4.2.2.1 : Domain Entities

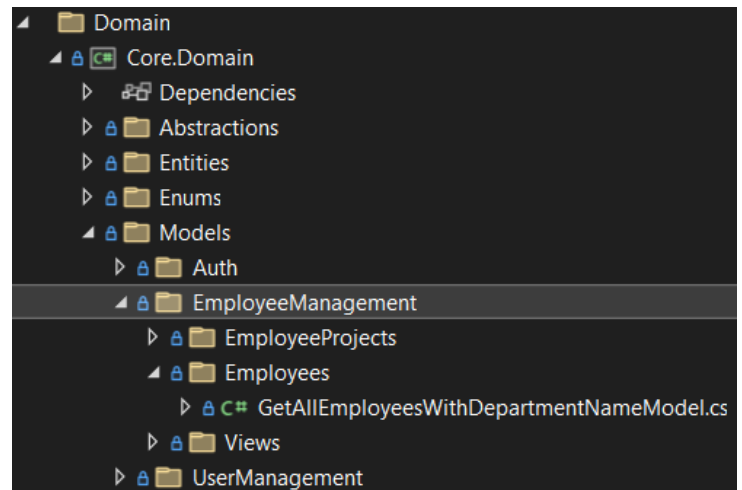


Figure 4.2.2.2: Models

This layer must also contain the repository interfaces under Domain/Abstractions/Repositories/{SchemaName}_Management and names comes from I_{DatabaseTableName}_Repository. The repositories folder contains business-related abstractions while Services and Email has all the application-related abstractions (see figure 4.2.2.3). All interfaces are placed under abstraction folder. An improvement should be to place

each application and infrastructure interfaces in application layer under an abstractions folder but since Dependency Rule is not violated, adding them in domain layer is still correct.

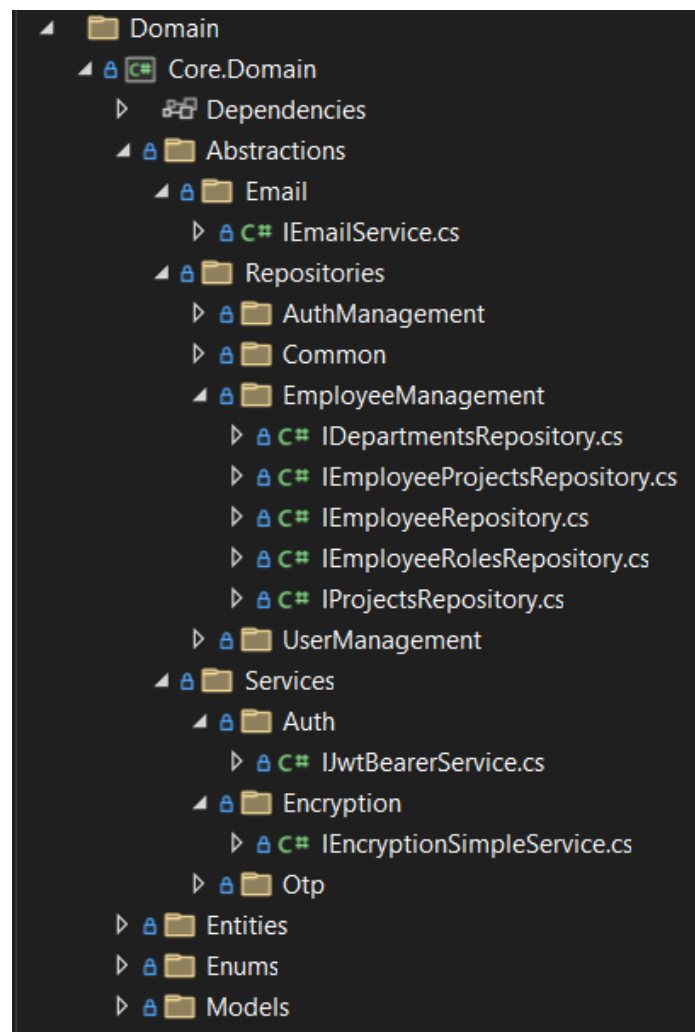


Figure 4.2.2.3 : Repository Interfaces

4.2.3 Application Layer

The Application layer is directly above Domain Layer and contains the use cases. Handlers are the classes responsible for implementing those use cases communicating with database. This layer references Domain Layer project to communicate through the abstraction with the database and doesn't have any other dependencies to outer layer projects. Figure 4.2.3.5 indicates that Application Project flow the dependencies inward by referencing Domain project. Moreover, it does not on any outer layer satisfying this way the Dependency Rule. Here, you can find the implementation of application-specific services, Handlers, Endpoints, Filters and Middlewares.

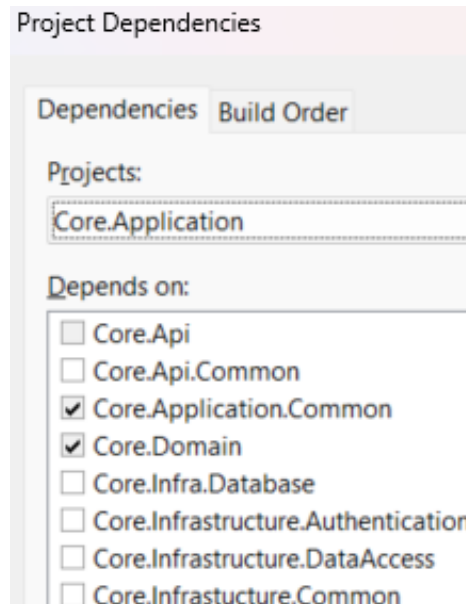


Figure 4.2.3.5 : Application Layer Dependencies

Application-specific Service implementations such as JWT Bearer, Excel Reports and Encryption should be under `Application/Services/{ServiceName}` and `{ServiceName}_Service` is the name of each service (see figure 4.2.3.1). Handlers must be under `Application/Handlers/{SchemaName}_Management/{TableName}/{UseCaseName}` and named like `{TableName}_{UseCaseName}_Handler` (see figure 4.2.3.2). The same naming convention and foldering need to be followed for Requests and Responses and Validators.

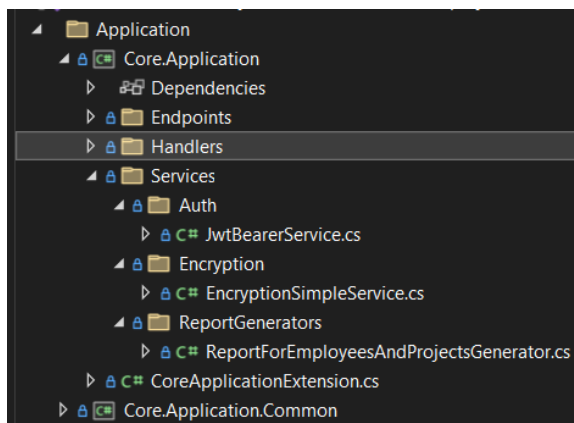


Figure 4.2.3.1 : Application Services

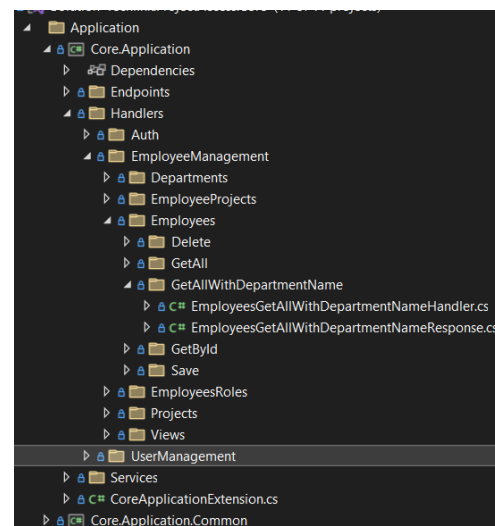


Figure 4.2.3.2 : Handlers

Cross-cutting concerns like exception-handling, logging, authorization, and data integrity are mostly related with application-specific logic and that's why Middlewares and Filters are placed in Application Layer (see figure 4.2.3.4).

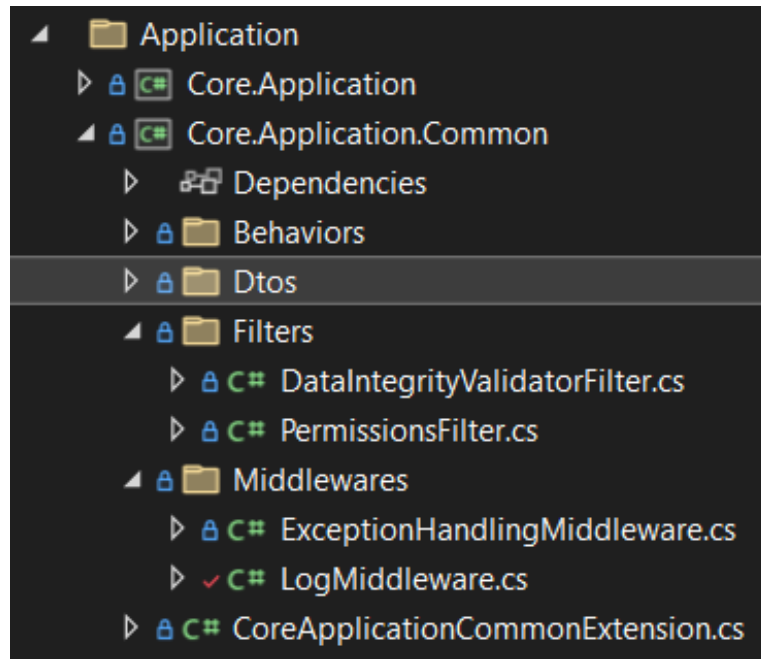


Figure 4.2.3.4 : Filters & Middlewares

4.2.4 Presentation Layer

This layer is responsible for building a solution based on the framework used. It is configured to run first. This happens because it contains a Program.cs file that serves as the startup file in a .Net Web API project. This file connects all the other layers to this layer so they can run as a single Web API project. Wiring can be achieved by using all those classes under Definitions folder (see figure 4.2.4.1) that scan each .Net Project in the solution and register services, interfaces, repositories, configure the pipeline and everything that is related to .Net to build the solution. Moreover, it contains docker, settings and scanner files.

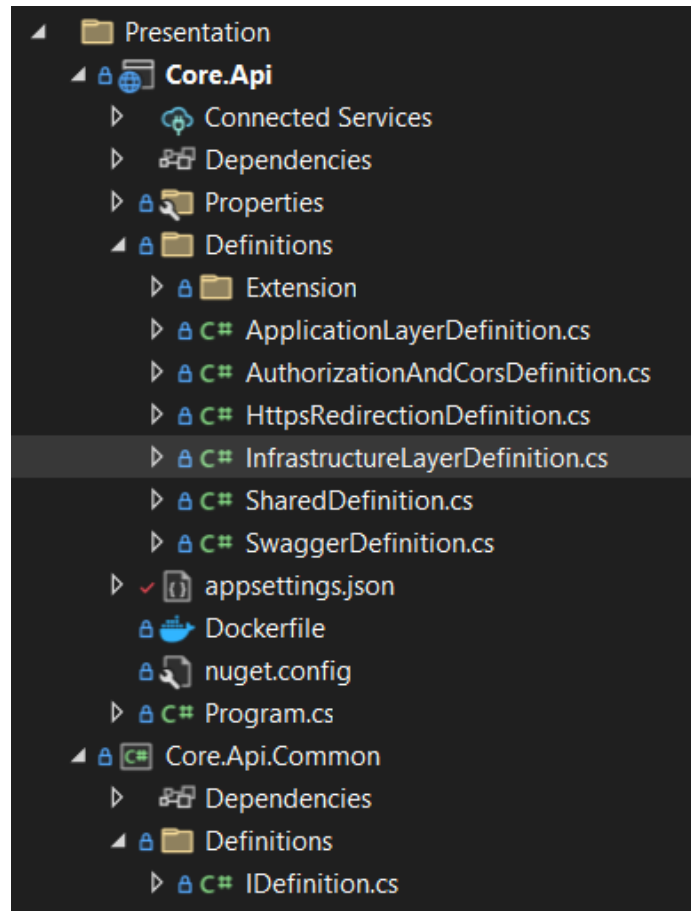


Figure 4.2.4.1 : Presentation Layer

This layer also needs **to ensure that request is validated before passed to the handler** as user can send invalid data. Requests need to be stopped in presentation layer, so error does not pass to the next layer slowing the performance of the application. As explained in How to achieve loose coupling using MediatR in our System paragraph this can be achieved by **utilizing mediator pipeline. When send() method is called of IMediator Interface it pass the control to the corresponding Validator of the command or query request and then go to handler of that request.** This means that except achieving loose coupling we can also achieve early exit of an invalid request before reaching application layer and the request handler.

In Figure 4.2.3.3, we see that endpoints are placed under Application/Endpoints/{SchemaName}_Management and named like {TableName}_Endpoints.

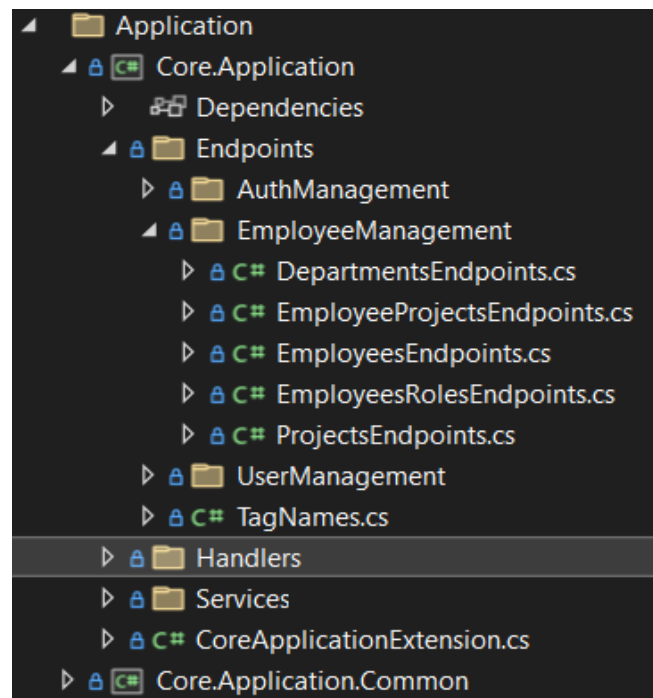


Figure 4.2.3.3 : Endpoints

4.3 Code Decisions

4.3.1 Domain Layer

As “Uncle Bob” explained, entities are objects that encapsulate the most high-level and general rules of the application which is business logic. Figure 4.3.1.1. states exactly this idea that entities are business rules independent from the application’s use cases. Domain Entities should strictly be the same as database tables (Table Entities) because those are the business rules of our application . Models represents the state of the system at a particular time in the application. You can create a model when you want to map data to a specific format to communicate between the layers of the application or with external systems. For example, **GetAllEmployeesWithDepartmentNameModel** class is a model that will communicate with the UI to return the employees along with the DepartmentName. Here is the difference between Models and Entities since models are used for the application’s use cases while entities should never be affected. [42] A small detail that needs to be highlighted is that you can communicate with UI both using Models and Domain Entities in our Clean Architecture implementation since they both act like DTOs however you cannot change a Domain or Table Entity.

```

21 references
public class EmployeesEntity
{
    11 references
    public int EmployeeId { get; set; }
    6 references
    public string FirstName { get; set; } = null!;
    6 references
    public string LastName { get; set; } = null!;
    4 references
    public bool SignOffStatus { get; set; }
    4 references
    public int DepartmentId { get; set; }
}

```

Figure 4.3.1.1 : EmployeesEntity class

```

public class GetAllEmployeesWithDepartmentNameModel : EmployeesEntity
{
    4 references
    public string DepartmentName { get; set; } = null!;
}

```

Figure 4.3.1.2: GetAllEmployeesWithDepartmentNameModel class

4.3.2 Infrastructure layer

Repositories are responsible for communicating with our database using dapper ORM. As explained in Dapper section IDataAccessLayerContext instance contains an IDbConnection which provides us all the available dapper functionalities. In figure 4.3.2.1, _db instance provides QueryAsync dapper method that execute our Stored Procedure. Then, by providing the .Net Object GetAllEmployeesWithDepartmentNameModel inside <>, we benefit from the efficient mapping of the query result to the object that Dapper provides with minimal overhead.

```

public class EmployeeRepository : IEmployeeRepository, ITransientRegistration
{
    private readonly IDataAccessLayerContext _db;
    private readonly IMapper _mapper;

    0 references
    public EmployeeRepository(IDataAccessLayerContext db, IMapper mapper)
    {
        _db = db ?? throw new Exception("ArgumentNullException");
        _mapper = mapper ?? throw new Exception("ArgumentNullException");
    }

    #region CRUD
    2 references
    public async Task<IEnumerable<EmployeesEntity>> GetAllEmployees() =>
        _mapper.Map<IEnumerable<EmployeesEntity>>(await _db.GetAllAsync<Employees>());

    2 references
    public async Task<EmployeesEntity> GetById(int id) =>
        _mapper.Map<EmployeesEntity>(await _db.GetAsync<Employees>(id));

    2 references
    public async Task<int> Insert(EmployeesEntity employee) =>
        await _db.InsertAsync(_mapper.Map<Employees>(employee));

    2 references
    public async Task<bool> Update(EmployeesEntity employee) =>
        await _db.UpdateAsync(_mapper.Map<Employees>(employee));

    2 references
    public async Task<bool> Delete(EmployeesEntity employee) =>
        await _db.DeleteAsync(_mapper.Map<Employees>(employee));
    #endregion CRUD

    2 references
    public async Task<IEnumerable<GetAllEmployeesWithDepartmentNameModel>> GetAllWithDepartmentName() {
        return await _db.QueryAsync<GetAllEmployeesWithDepartmentNameModel>("[employee].[Employees_GetAllWithDepartmentName]");
    }
}

```

Figure 4.3.2.1: EmployeeRepository class

In that part, we must explain why we need to have both Table Entities and Domain Entities. Obviously, somebody can think why not mapping the result directly to Employee Entity instead of Employee for CRUD operations in EmployeeRepository class (see Figure 4.3.2.1). However, introducing Dapper Contrib .Net library will clarify the reason for this choice. Dapper Contrib is a 3rd party library that extends the Dapper functionality by providing methods for providing CRUD operations and mapping database results to strongly typed objects. This means that you don't have to write any SQL code to implement CRUD, but only provide an object that represents the table of the database. As shown in Figure 4.3.2.2, simply adding the Table and Key Attribute and library can automatically provide you with these methods. All you need is whenever you are using dapper CRUD methods to Map the result or to pass that Table entity.

```

[Table("[employee].[Employees]")]
5 references
public class Employees : ITableEntity, IMap<EmployeesEntity>
{
    [Key]
    0 references
    public int EmployeeId { get; set; }
    0 references
    public required string FirstName { get; set; }
    0 references
    public required string LastName { get; set; }
    0 references
    public required bool SignOffStatus { get; set; }
    0 references
    public required int DepartmentId { get; set; }
}

```

Figure 4.3.2.2: Employee class

4.3.3 Presentation Layer

As explained in Minimal APIs section, using `RouteGroupBuilder` class can help to structure our APIs routes. In figure 3.4.2.5, calling `MapGroup()` on `WebApplication` instance the common prefix “api” for all routes of the application was added. Then, we pass the current `RouteGroupBuilder` instance called `apiMapGroup` to each `method()` to add endpoints. Examining `AddGroupsEndpoints()` method as shown in figure 4.3.3.1, we added another prefix for the current subset of endpoints using `MapGroup()`. If you compare figures 4.3.3.2 and figures 4.3.3.4 you will see that the new prefix is common only for routes contained in the `AddGroupsEndpoints()` method. . Then each endpoint adds something indicating what it does and a handler for the route using `MediatRGet()` or `MediaRPost()` method. We will explain later what those 2 methods do.

```

internal static class GroupsEndpoints
{
    1 reference
    public static void AddGroupsEndpoints(this WebApplication app, RouteGroupBuilder apiMapGroup)
    {
        RouteGroupBuilder groups = apiMapGroup.MapGroup(TagNames.GROUPS)
            .WithTags(TagNames.GROUPS)
            .WithGroupName(SwaggerGroupNames.USER_MANAGEMENT);

        groups.MediatRGet<GroupsGetAllQuery, GroupsGetAllResponse>("GetAll")
            .WithMetadata(PermissionsSmartEnum.GroupsGetAllEndpointPermission);

        groups.MediatRPost<GroupsGetByIdQuery, GroupsGetByIdResponse>("GetById")
            .WithMetadata(PermissionsSmartEnum.GroupsGetByIdEndpointPermission);

        groups.MediatRPost<GroupsDeleteCommand, GroupsDeleteResponse>("Delete")
            .WithMetadata(PermissionsSmartEnum.GroupsDeleteEndpointPermission);

        groups.MediatRPost<GroupsSaveCommand, GroupsSaveResponse>("Save")
            .WithMetadata(PermissionsSmartEnum.GroupsSaveEndpointPermission);
    }
}

```

Figure 4.3.3.1 : Groups Endpoints

Common Prefix for all Endpoints

Authorize 

Groups		^
GET	/api/Groups/GetAll	⌵ 🔒
POST	/api/Groups/GetById	⌵ 🔒
POST	/api/Groups/Delete	⌵ 🔒
POST	/api/Groups/Save	⌵ 🔒
Permissions	Common Prefix for Groups Endpoints	⌵
Portals		⌵
Relations		⌵

Figure 4.3.3.2: Groups Endpoints in Swagger


```

0 references
internal static class EmployeesEndpoints
{
    1 reference
    public static void AddEmployeesEndpoints(this WebApplication app, RouteGroupBuilder apiMapGroup)
    {
        var employees = apiMapGroup.MapGroup(TagNames.EMPLOYEES)
            .WithMetadata(PermissionsSmartEnum.ViewEmployeesMenuPermission)
            .WithTags(TagNames.EMPLOYEES)
            .WithGroupName(SwaggerGroupNames.EMPLOYEE_MANAGMENT)
            .WithOpenApi();

        employees.MediatRGet<EmployeesGetAllQuery, EmployeesGetAllResponseDto>("GetAll");
        employees.MediatRGet<EmployeesGetAllWithDepartmentNameQuery, EmployeesGetAllWithDepartmentNameResponse>
            ("GetAllWithDepartmentName");
        employees.MediatRPost<EmployeesGetByIdQuery, EmployeesGetByIdResponseDto>("GetById");
        employees.MediatRPut<EmployeesSaveCommand, BooleanDto>("Save");
        employees.MediatRPost<EmployeesDeleteCommand, BooleanDto>("Delete");
    }
}

```

Figure 4.3.3.3: Employees Endpoints

Departments	▼
EmployeeProjects	▼
EmployeeRoles	▼
Employees	Common Prefix for Employees Endpoints
GET /api/Employees/GetAll	▼ 🔒
GET /api/Employees/GetAllWithDepartmentName	▼ 🔒
POST /api/Employees/GetById	▼ 🔒
PUT /api/Employees/Save	▼ 🔒
POST /api/Employees/Delete	▼ 🔒
Projects	▼

Figure 4.3.3.4: Employees Endpoints in Swagger

In figures 4.3.3.1 and 4.3.3.3, WithGroupName() and WithTags() method are used to separate each set of endpoints in Swagger based on schema and table name they belong (see figures 4.3.3.5 and 4.3.3.6). Swagger is a tool from Swashbuckle .Net library that was used in the project to visualize our Web API and test endpoints. WithGroupName() is used to specify in which management-schema name (see figure 4.3.3.7) this set of endpoints while WithTags() is used to specify table name.

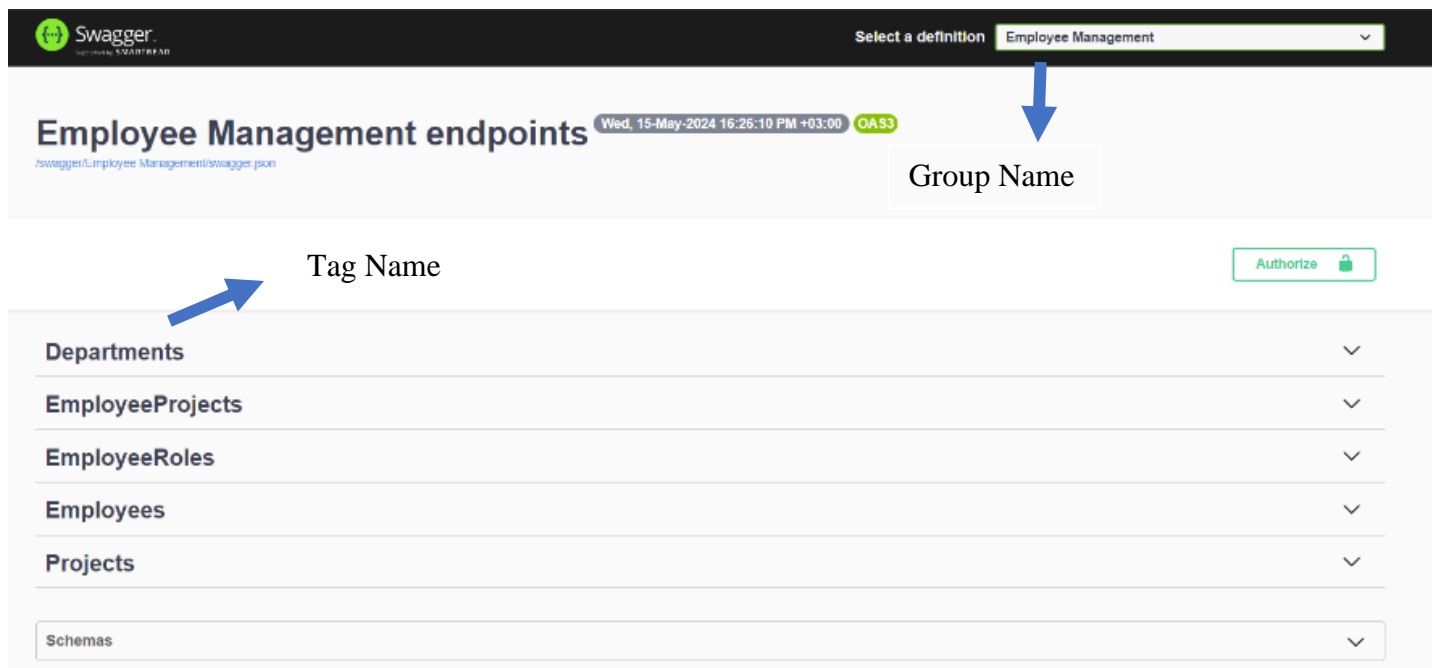


Figure 4.3.3.5: Employees Management Endpoints in Swagger

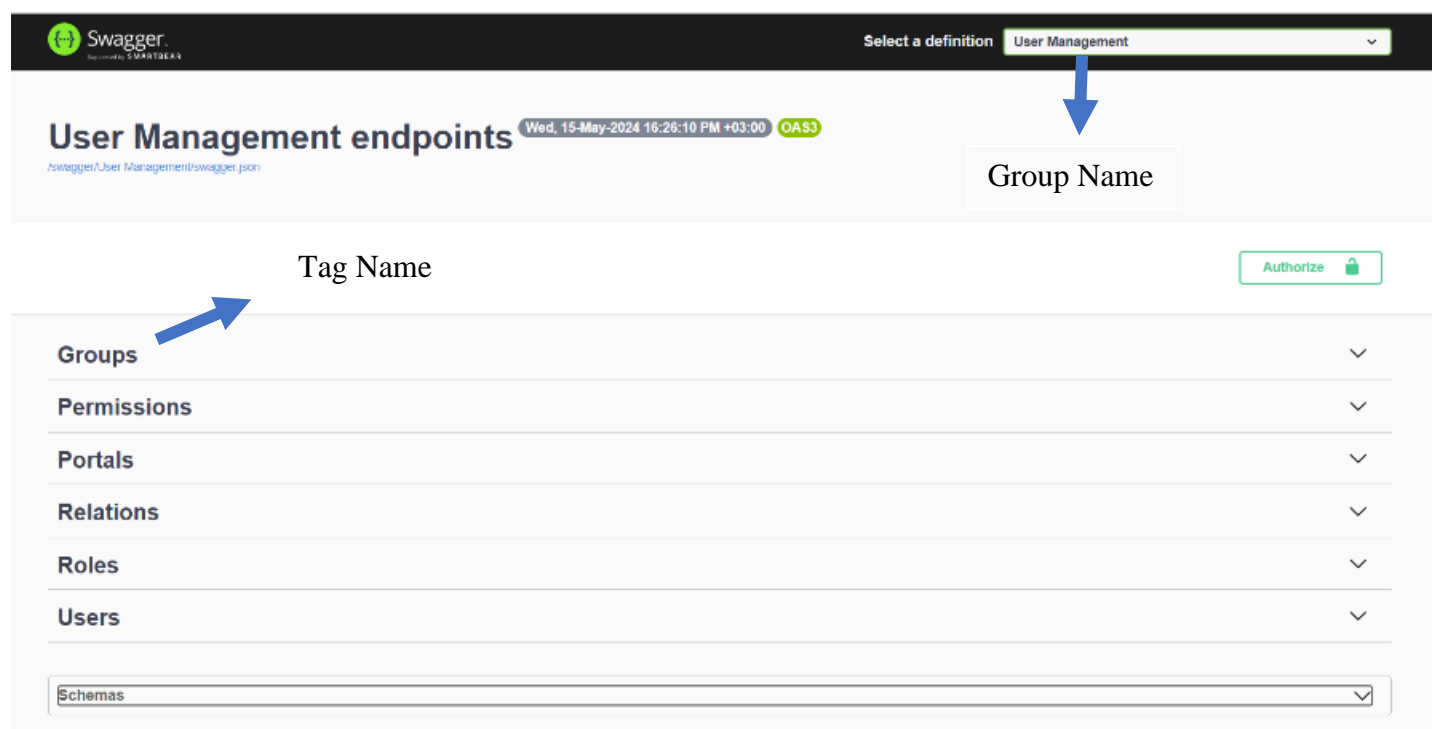


Figure 4.3.3.6: User Management Endpoints in Swagger

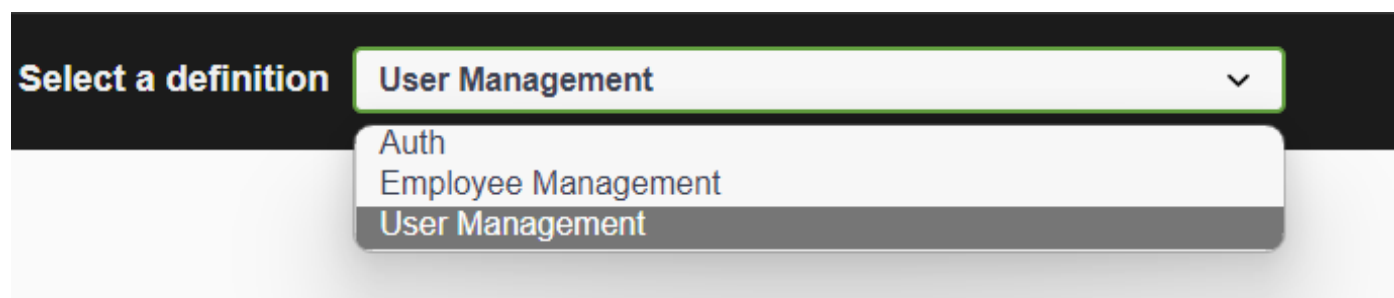


Figure 4.3.3.7: Group Name Dropdown

Let's give an example of what happened when an endpoint is called. Assume User **click on GET Endpoint with /api/Employees/GetAll route in SwaggerUI** (see Figure 4.3.3.4). This request flows through the request pipeline and when finish **Minimal API mechanism will handle the request**. It will **find the Route Endpoint associated with that route** and pass the control to **the route handler function which is MediatRGet** with generic arguments **EmployeesGetAllQuery** and **EmployeesGetAllResponseDto** (see Figure 4.3.3.3). **This function will utilize mediator pipeline to validate the request and pass the control to the handler associated with that request**. This handler is **EmployeesGetAllHandler** located in **Application Layer**. Then, the **handler will communicate with Infrastructure Layer through Repository Interface** (see Figure 4.3.3.8) calling **GetAllEmployees()** method (see Figure 4.3.2.1 for implementation). Then, it **returns with EmployeesGetAllResponseDto class the data** towards the mediator pipeline and from there to the **Minimal API mechanism to return this response** for user request (see Figure 4.3.3.9).

```

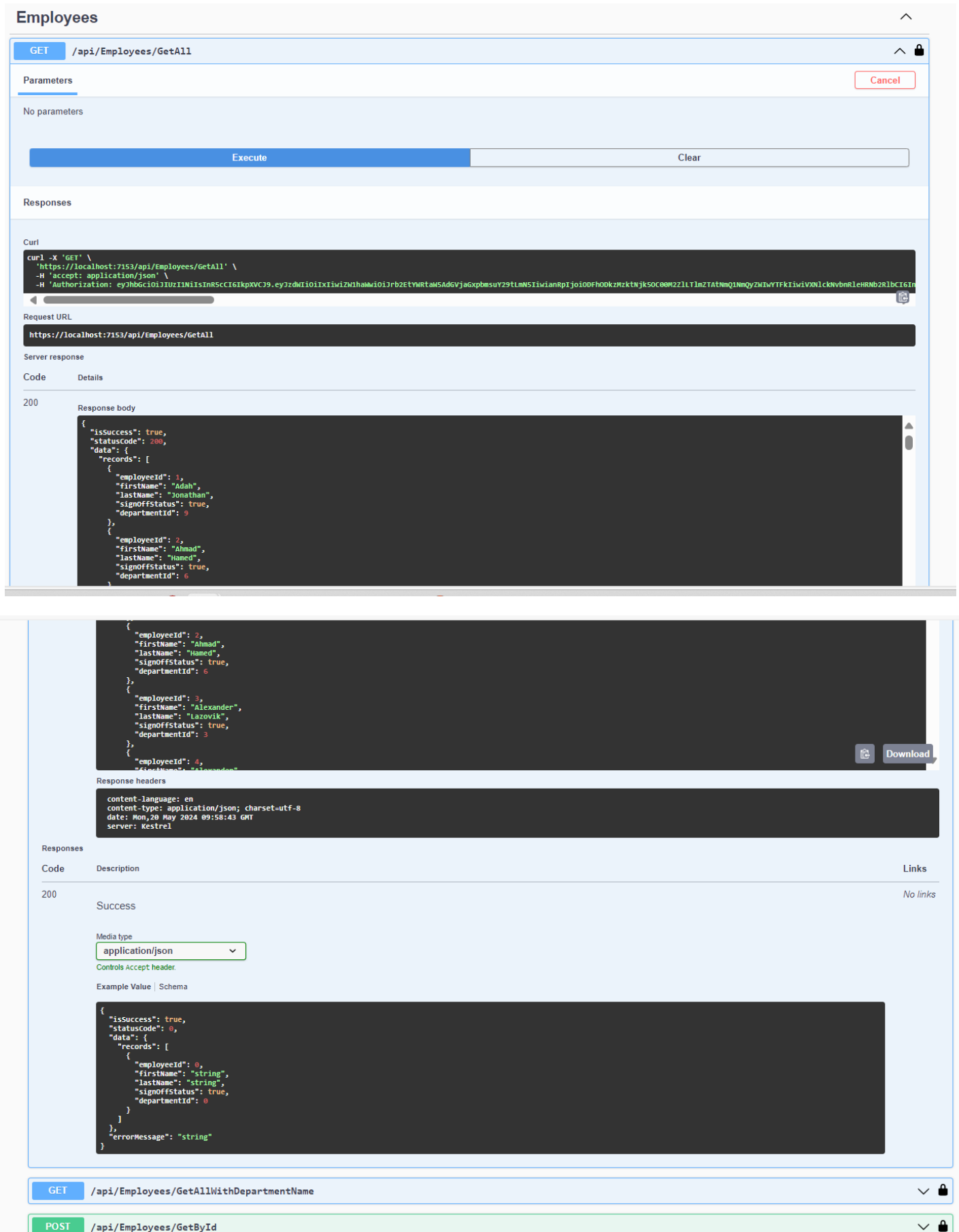
public record EmployeesGetAllQuery : IRequest<EmployeesGetAllResponseDto>;
1 reference
public class EmployeesGetAllHandler : IRequestHandler<EmployeesGetAllQuery, EmployeesGetAllResponseDto>
{
    private readonly IEmployeeRepository _employeeRepository;

    0 references
    public EmployeesGetAllHandler(IEmployeeRepository employeeRepository)
    {
        _employeeRepository = employeeRepository;
    }

    0 references
    public async Task<EmployeesGetAllResponseDto> Handle(EmployeesGetAllQuery request, CancellationToken cancellationToken)
    {
        return new EmployeesGetAllResponseDto { Records = await _employeeRepository.GetAllEmployees()
        };
    }
}

```

Figure 4.3.3.8: EmployeesGetAllQuery



Chapter 5

Employee Management System User Interface

5.1 User Interface

77

5.1 User Interface

In this chapter, we will demonstrate the User Interface of Employee Management System Application. To start with, in our application is an internal system so we don't support API users but only Internal Users. This means that only Super Admin Persons can create accounts for the Team Leaders or HR departments. Super Admin Account will be created from database and use those credentials to be authenticated.

Authentication

Route: <http://localhost:4200/auth>

Super Admin log in with the credentials of a user created in the database. After, he/she will navigate to welcome page.



Figure 5.1: Authentication Page

Welcome Page

Route: <http://localhost:4200/employee-management/employees-view-home>

The default Welcome Page is the Report of the Employees and Roles per Project. In the sidebar, the system appears all the functionalities Super Admin can do. At this stage, we need to mention that Super Admin has all the Permissions of the system, and he/she can interact with all the features that the application supports.

Name	Department	Roles	Full Name
Ablebook			
	Software Developer	DEV	Andreas Constantinou
	Software Developer Team Leader	TLEAD	George Mitsis
	Junior Software Developer	DEV	Haris Kountouris
APIHall			
Carats			
	Software Developer	DEV	Adah Jonathan
	Software Developer	DEV	Andreas Constantinou
	Software Developer Team Leader	PROD-TLEAD	George Mitsis

Figure 5.2: Report Page

User Management

Route: No navigation

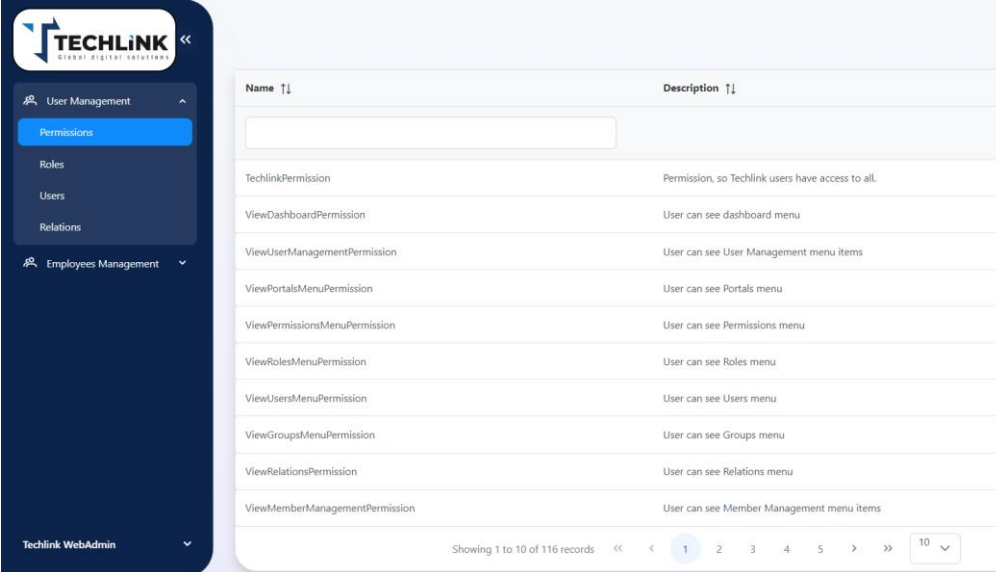
Assuming the user clicked on User Management in the menu. No navigation will happen, just a dropdown with user management options will appear as shown in Figure 5.3. Then the user clicks on Permissions.

Name	Department	Roles	Full Name
Ablebook			
	Software Developer	DEV	Andreas Constantinou
	Software Developer Team Leader	TLEAD	George Mitsis
	Junior Software Developer	DEV	Haris Kountouris
APIHall			
Carats			
	Software Developer	DEV	Adah Jonathan
	Software Developer	DEV	Andreas Constantinou
	Software Developer Team Leader	PROD-TLEAD	George Mitsis

Figure 5.3: User Management Options Page

Route: <http://localhost:4200/user-management/permissions-home>

Angular application will load Permissions listing page (see Figure 5.4). Then the user clicks on Roles.

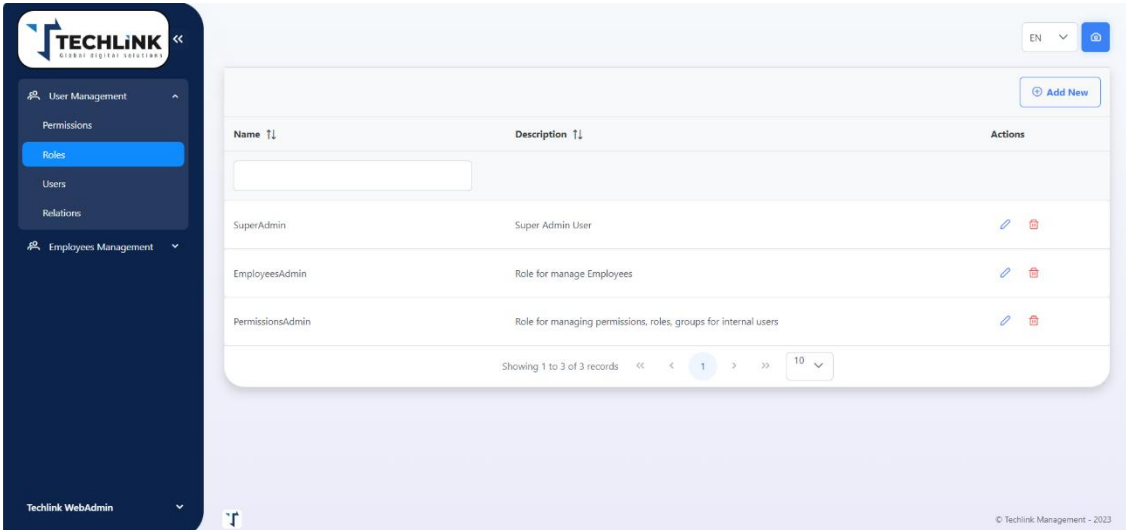


Name	Description
TechlinkPermission	Permission, so Techlink users have access to all.
ViewDashboardPermission	User can see dashboard menu
ViewUserManagementPermission	User can see User Management menu items
ViewPortalsMenuPermission	User can see Portals menu
ViewPermissionsMenuPermission	User can see Permissions menu
ViewRolesMenuPermission	User can see Roles menu
ViewUsersMenuPermission	User can see Users menu
ViewGroupsMenuPermission	User can see Groups menu
ViewRelationsPermission	User can see Relations menu
ViewMemberManagementPermission	User can see Member Management menu items

Figure 5.4: List Permissions Page

Route: <http://localhost:4200/user-management/roles-home>

Angular application will load Roles listing page (see Figure 5.5). Then the user clicks on Edit on Employees Admin.



Name	Description	Actions
SuperAdmin	Super Admin User	Edit Delete
EmployeesAdmin	Role for manage Employees	Edit Delete
PermissionsAdmin	Role for managing permissions, roles, groups for internal users	Edit Delete

Figure 5.5: List Roles Page

Route: Same

Angular application will appear a modal on the same page (see Figure 5.6). Note that modal has prepopulated the fields with the previous details. Then the user clicks Save after finishing edit.

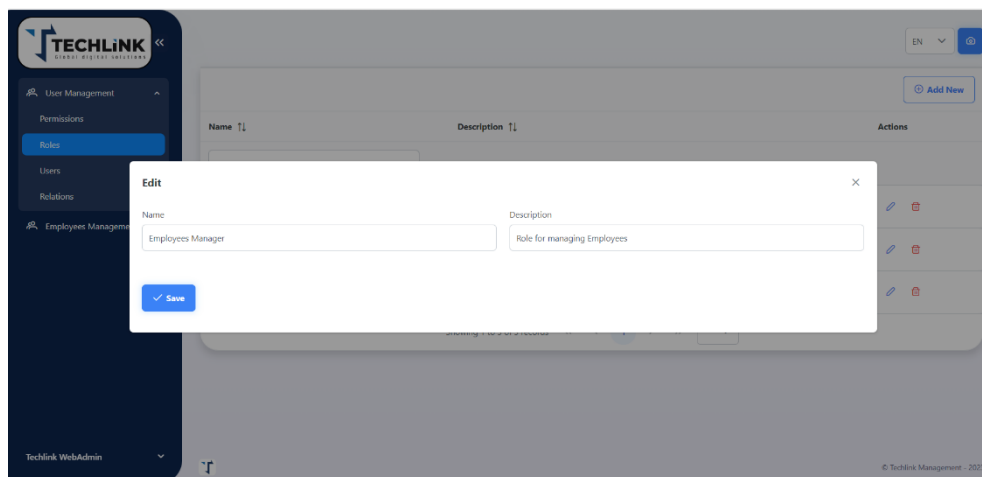


Figure 5.6: Edit Roles Modal

Route: Same

A success Message will appear to the screen and modal close. Changes was correctly applied to the Role (see Figure 5.7). Add or Delete Role options are also available. Then the user clicks on the Users option.

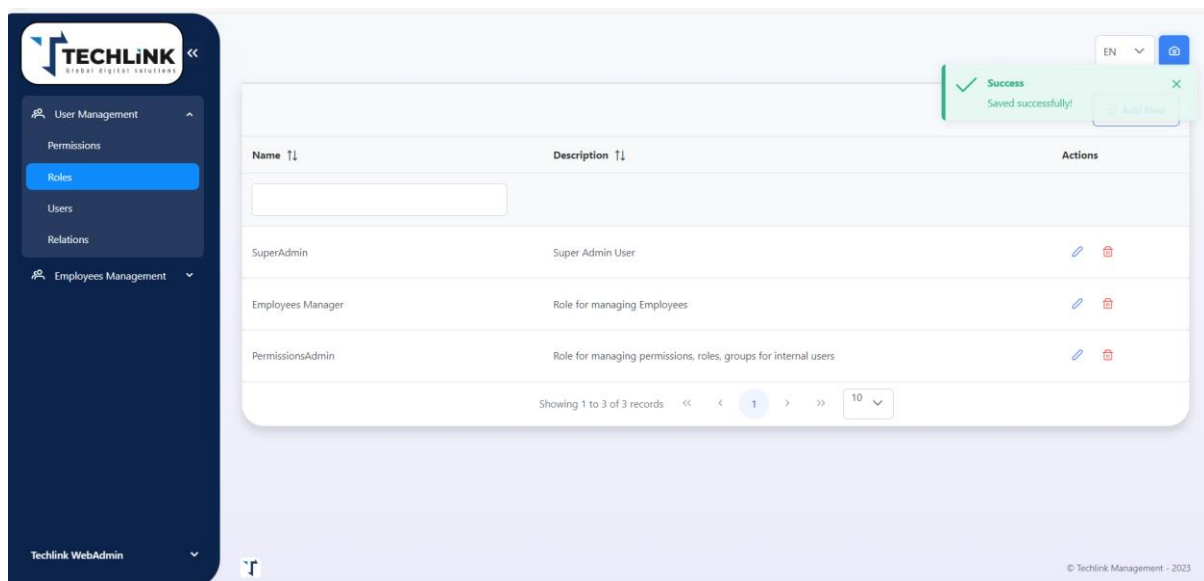


Figure 5.7: List Roles After Edit Page

Route: <http://localhost:4200/user-management/users-home>

Angular application will load Users listing page (see Figure 5.8). In the following steps, we will show the process to create an Account for Team leaders with the appropriate permissions. Then the user clicks on Users where all Internal Users are listed and click Add new to create a new Internal User.

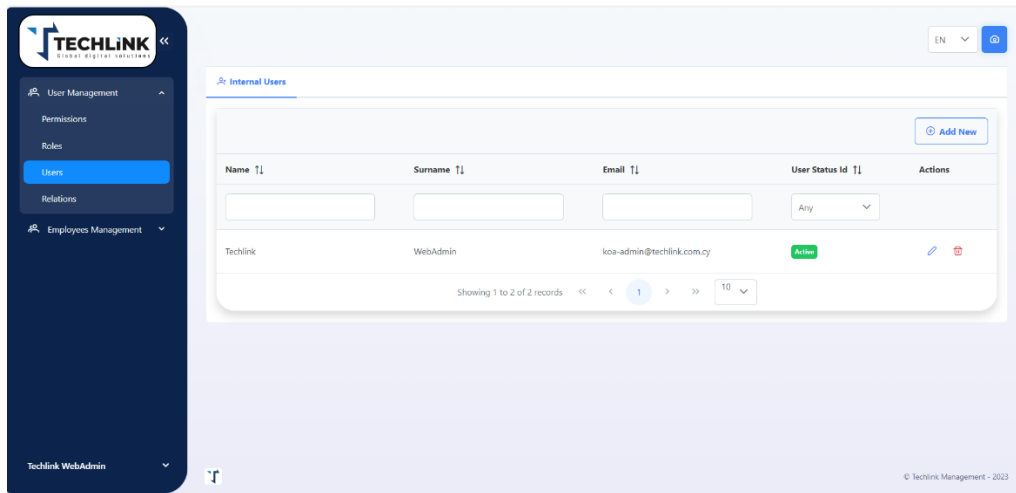


Figure 5.8: List Users

Route: same

Super admin user needs to fill the modal appear with the details of the Account that are given to him by the Team Leader that wants to create the account (see Figure 5.9). It is important for the Super Admin to select Active Dropdown option for User Status and select only the Employee Manager Role in the Roles Dropdown options (see Figure 5.10). When user clicks Save, a success Message will appear to the screen, modal closes and Figure 5.11 appears in the screen. Now, Super Admin needs to assign the permissions related to Employee Manager Role. The user clicks on Relations, then on Role-Permissions selecting Employee Manager Role .

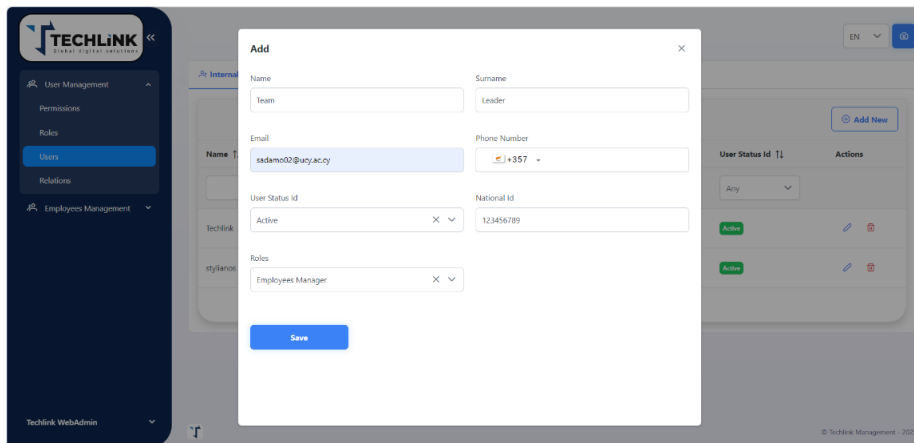


Figure 5.9: Add User (Team Leader)

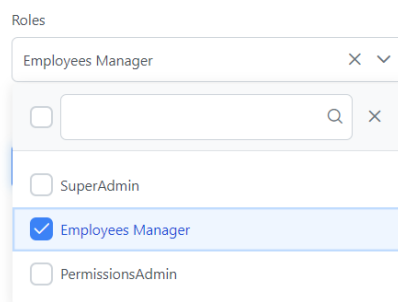


Figure 5.10: Roles Dropdown Options

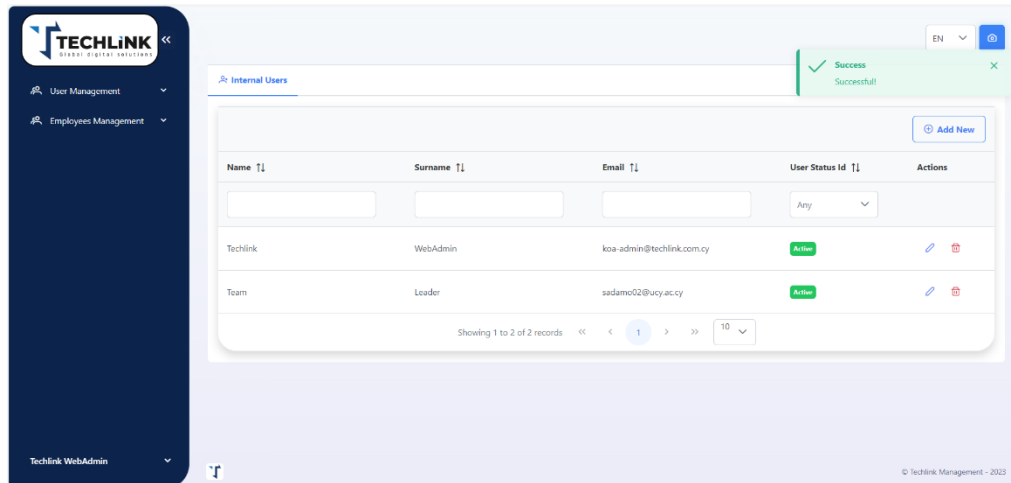


Figure 5.11: List User After Add Team Leader

Route: <http://localhost:4200/user-management/relations-home>

Figure 5.12 will be loaded where super admin will choose the permissions that Employee Manager Role should have. Permissions are separated by Schema Name to be easier for the Super Admin to assign them to a Role. Note that this Role is reusable, and it will be given to each Team Leader that wants to create an account. Then the user navigate back to Users List and click Edit on Team Leader User.

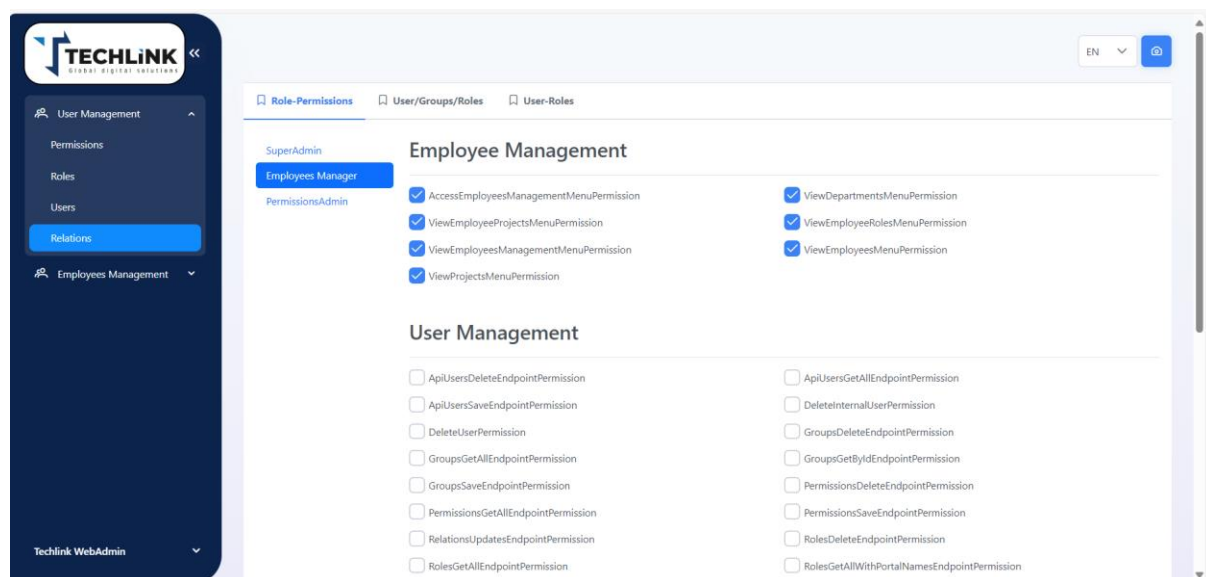


Figure 5.12: Role Permissions Management User

Route: <http://localhost:4200/user-management/users-home>

Super Admin needs to click Reset Password button and choose yes (see Figure 5.13) to send an email to the Team Leader where he/she can get a token URL to navigate and set his password. The reason why the button didn't appear before when we were trying to create the user was because the button appeared only when editing an active user and not when creating

one. This was an introduction on how to interact with User Management features that is only allowed through Super Admin Account. Now, we will proceed with Team Leader Account to explore Employee Management features. Then, the Team Leader needs to click on the token URL in his email.

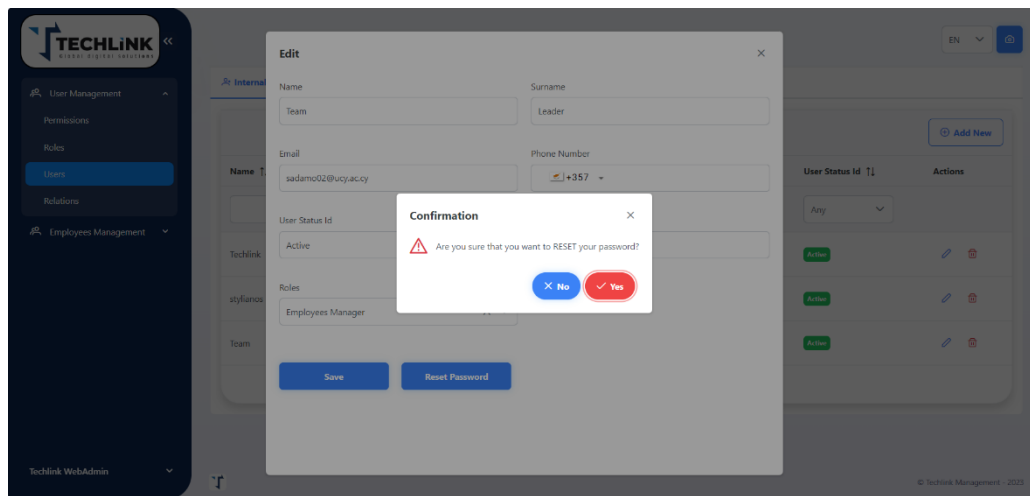


Figure 5.13: Send Reset Password Email to Team Leader

Route: <http://localhost:4200/reset-password/614aee2f-ed3f-4aca-963c-24c7a1b4eb25>

Team Leader must fill in the form with his password (see Figure 5.14). If nothing goes wrong, he/she will automatically get authenticated successfully and redirected to the Welcome Page (see Figure 5.2). In the following steps, we will analyze Employee Management Features through Team Leader Account.

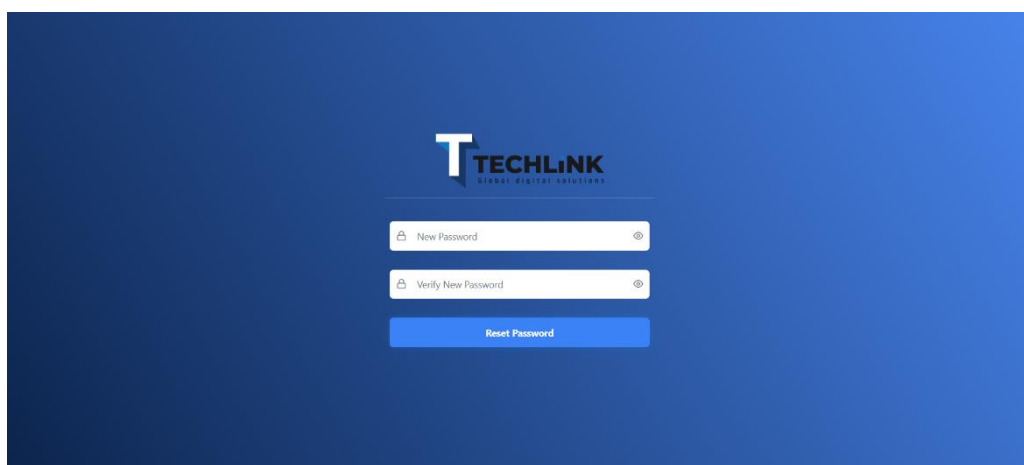


Figure 5.14: Reset Password Form

Employee Management

Route: <http://localhost:4200/employee-management/employees-view-home>

As you see in Figure 5.16 permissions are correctly assigned to Team Leader since menu doesn't contain the User Management. The following examples demonstrate some filters that we can apply to the Employees Report. We can find all employees working for a specific Project (see Figure 5.15). A project that matches the name without any employee working on it will also be returned. Searching by employee full name brings all Projects where this employee is currently working (see Figure 5.16). Last, we can also filter by role which means that returns all projects where an employee is working with that Role (see Figure 5.17). Before proceeding with assigning employees to a project, this page also provides the functionality to download this report in excel file with the button at the bottom left of the page. Figure 5.18 shows the employee report in an excel file. Then the user clicks on the Employees option.

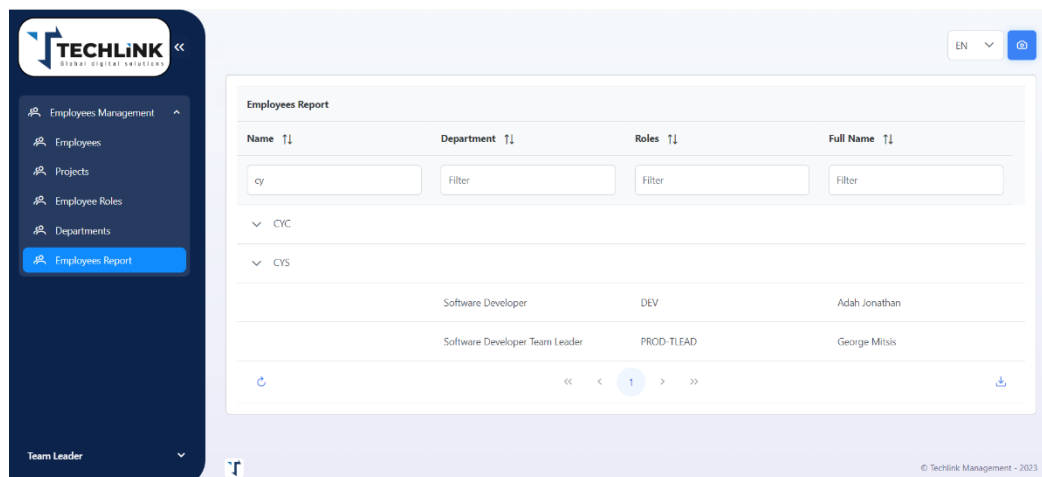


Figure 5.15: Filter Employees Report by Project

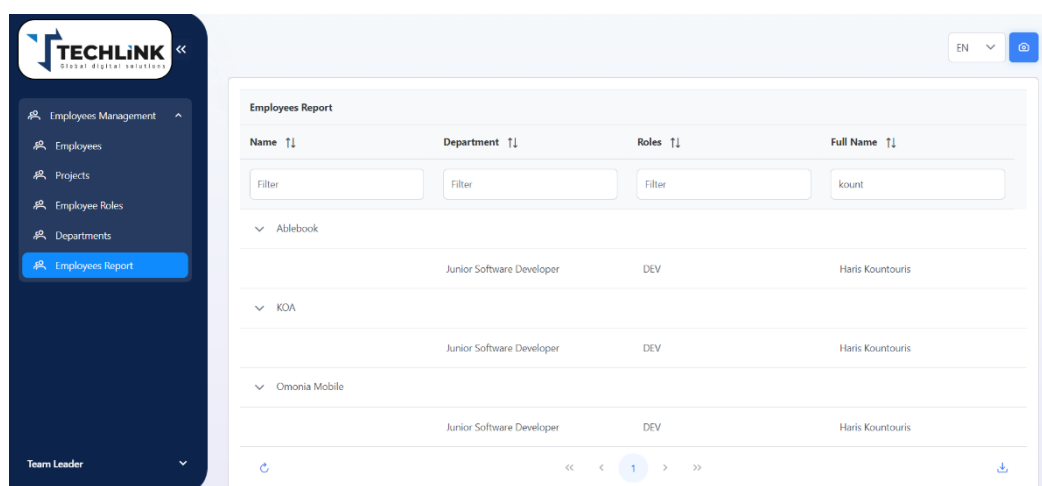


Figure 5.16: Filter Employees Report by Employee Full Name

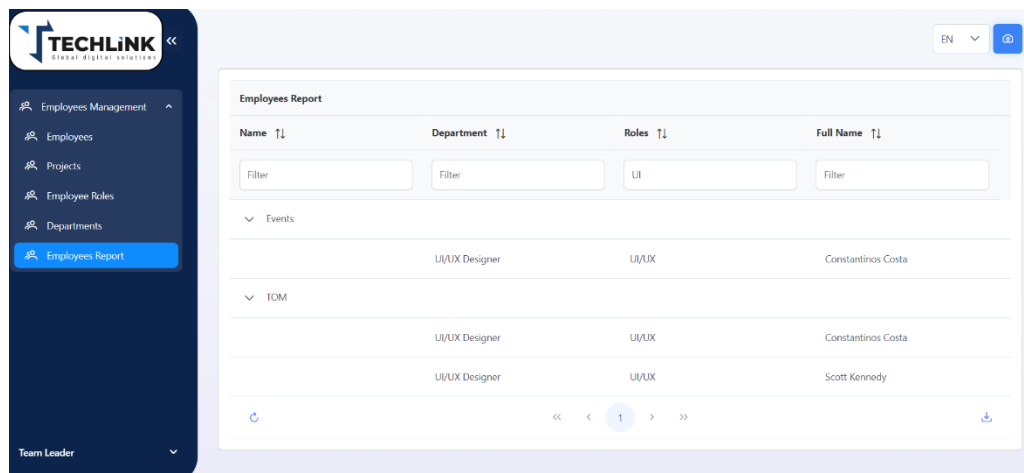


Figure 5.17: Filter Employees Report by Role

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Employee Name	Ablebook	API4all	Carats	CYC	CYS	Events	GEHRET	GEHRET APP	GPTW	Hellenic	Infocredit ISG	Infocredit Debt Collection	Infocredit ECheck	JCC	KO
2	Andreas Constantinou	DEV		DEV								DEV		DEV		DE
3	George Mitsis	TLEAD		PROD-TLEAD		PROD-TLEAD	PROD-TLEAD					PROD-TLEAD	PROD-TLEAD	PROD-TLEAD		PROD-T
4	Haris Kountouris	DEV														DE
5	Adah Jonathan			DEV		DEV	DEV					DEV	DEV	DEV		
6	Ahmad Hamed						QA									
7	Alexander Tyutyunik						DEV									
8	Augoustinos Papastavrou						DEV									DE
9	Christos Mappouras						DEV									DE
10	Constantinos Costa						UI/UX									
11	Hasan Jahouse						DEV									
12	Kyriakos Georgiades						DEV					DEV				
13	Muayyad Diab						DEV									
14	Nicos Anastasiou						QA									
15	Anastasis Adamou											DEV	DEV	DEV		DE
16	Nicholas Christodoulou											DEV				QA
17	Anton Iarovoi											DEV				QA
18	Alexander Raznikov															DE
19	Stylanos Adamou															
20	Khalil Berakdar															
21	Alexis Kourmatze															
22	Alona Csonka															
23	Andreas Vasileiou															
24	Andreas Herodotou															
25	Andriy Kravchenko															
26	Arya Thakur															
27	Christos Aglitsiotis															

Figure 5.18: Employees Report Excel

Route: <http://localhost:4200/employee-management/employees-home>

In figure 5.19, All Employees of the Company appear on the Screen with. Team Leader authorized to make all the CRUD operations in each Employee. However, focus on Figure 5.20 icon since when click it you can manage where the selected employee can work and with what role. User clicks on the Employees option.

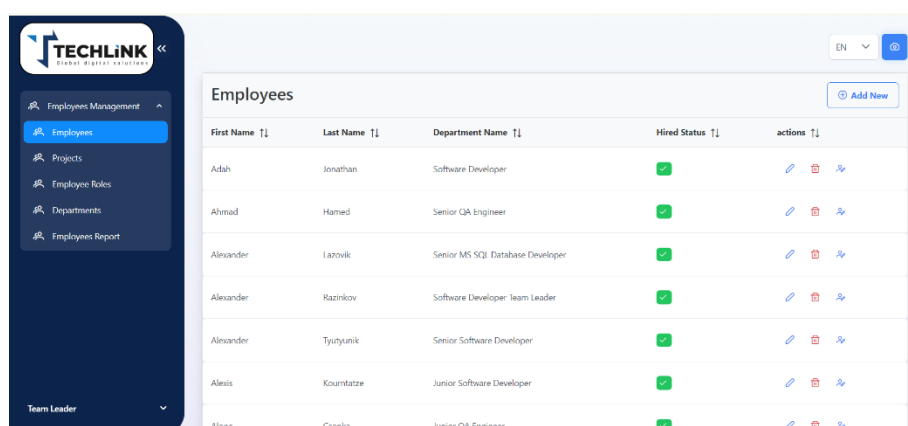


Figure 5.19: List Employees

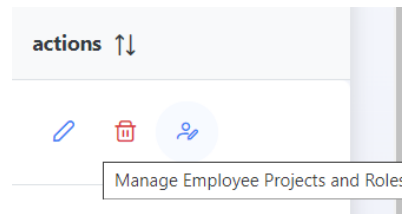


Figure 5.20: Icon to Manage Employee Projects and Roles

Route: same

Angular Application will appear a modal in the screen of Team leader (see Figure 5.21). Modal shows at left all the project that the selected employee is working. The blue highlight on projects indicates the selected project. Within this selected project at the left side of the modal we can see all the roles that the company has. Roles that are checked indicate that Selected Employee work on Carats Project with Role DEV. Therefore, Team Leader for example can add new or delete existing roles in Carats Project for the selected Employee (see Figure 5.22). When an employee stopped working on a project, we can simply press the red button and remove the employee from that project. In figure 5.23, we try to remove the selected employee from Carats Project, and we click yes to proceed. As you can see the project no longer appears on the right side of the modal (see Figure 5.24). Team leader could also assign the selected employee to a new project. By pressing the Add Project button a modal appears where the Team Leader must select the project to add the employee and set of the roles that he/she will have on that project (see Figure 5.25 and 5.26). When user clicks on save button, a success Message will appear to the screen and modal will close. Back in the previous modal we can notice that a new project was added along with the set of roles (see Figure 5.27).

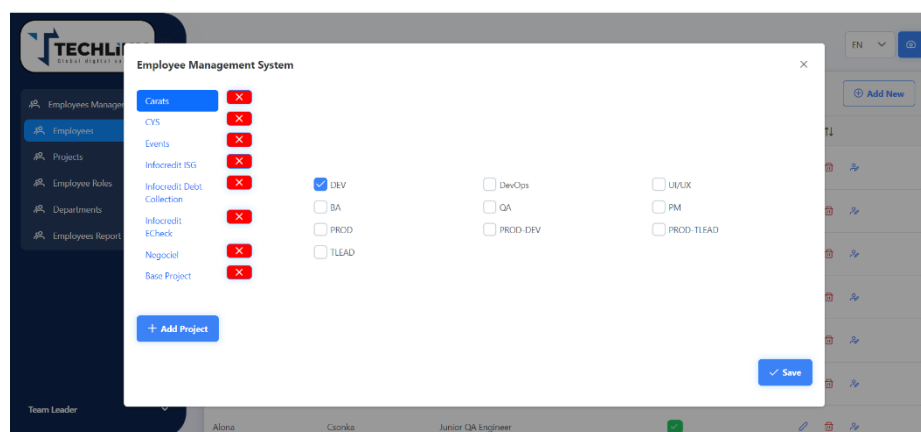


Figure 5.21: Assign Project with Roles To Selected Employee



Figure 5.22: Assign Roles To Selected Employee in Carats Project

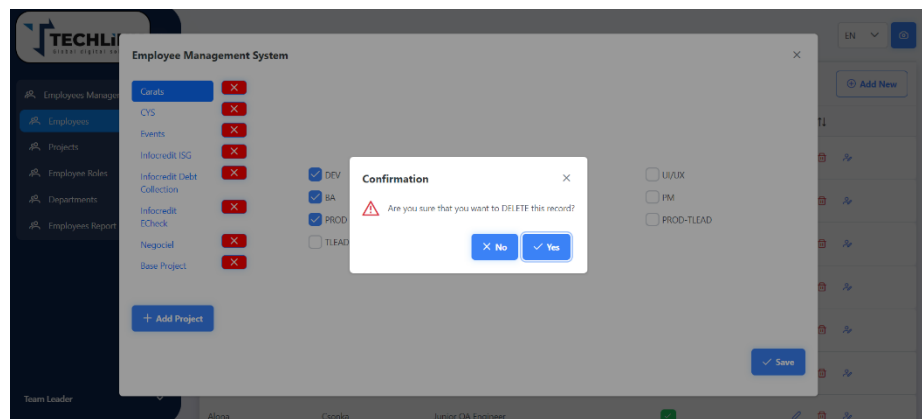


Figure 5.23: Delete Selected Employee From Carats Project

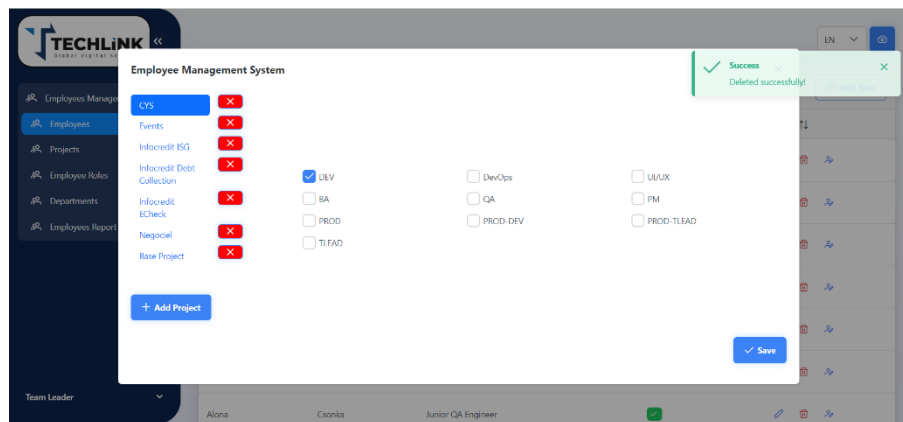


Figure 5.24: Selected Employee After Deleting Carats Project

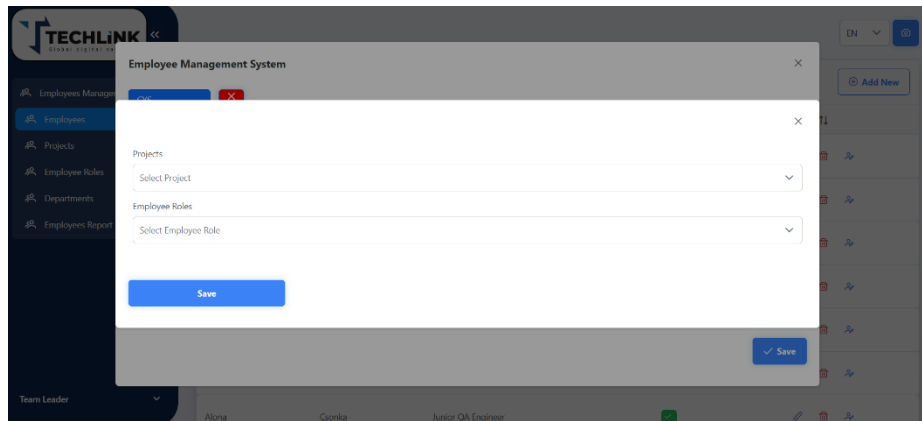


Figure 5.25: Add Project to Selected Employee Modal

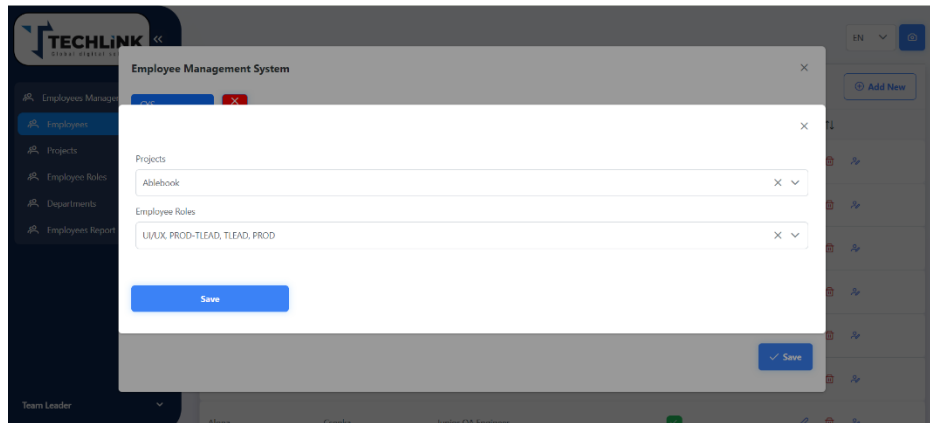


Figure 5.26: Fill Data to Add Project to Selected Employee Modal

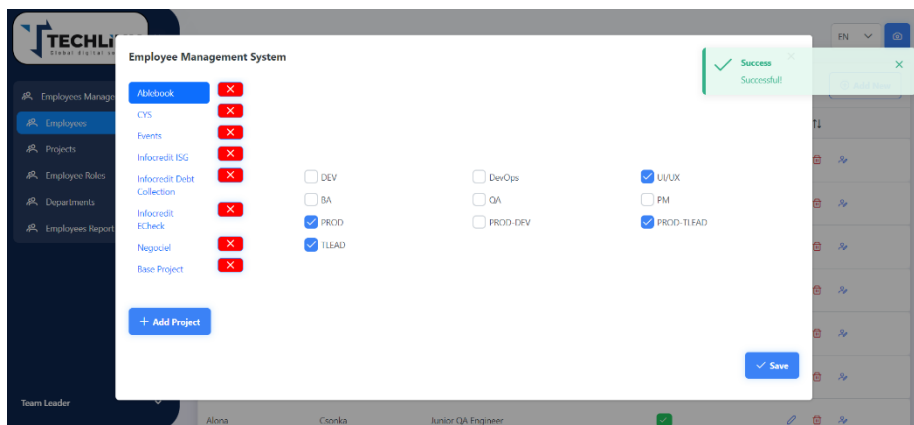


Figure 5.27: Selected Employee After Adding Project

Chapter 6

Conclusion

6.1 Evaluation	89
6.2 Future Work	90
6.3 Feedback	90
6.4 Conclusion	91

6.1 Evaluation

The Evaluation aspect of the Research will be done under the aspect of research questions, performance, and security.

- **Question 1:** In Chapter 3, we analyze the technologies, libraries and common features among different applications that integrated into our base software. Our application was written using C# and SQL and with SQL Management Studio and ASP.NET Core frameworks. Libraries used are MediatR, Fluent Validation, Dapper, Swashbuckle while code was developed using CQRS and mediator patterns. It also contains external services communication like database and emailing, a feature for User Management, while used features like Minimal APIs, Middlewares and Filters to implement Authentication. Authorization, Global Error-Handling and Logging. Chapter 4 screenshots related to the .Net projects proves that everything related to base software was placed under the correct folder and follow what our generic architecture proposes. Using the reference idea between .Net Projects and Dependency inversion we ensure that we align with our architecture.
- **Question 2:** This question is satisfied since we used an Employee Management System as proof of concept that we use that base software without affecting the business logic of the specific application. Therefore, we prove that any different business logic can be simply added to our base software by following our generic architecture rules.
- **Performance:** Applying 3NF to database tables ensures data integrity, reduce data redundancy, and improves the efficiency of data operations. The choice of Dapper ORM to communicate with the database is because of the performance. Dapper is specialized

in reading data from a database and efficiently mapping query results to .Net objects with minimal overhead. We also reduce multiple round trips to the database by bringing multiple results in a single store procedure. Additionally, we use indexes in columns that was used in Where clause in some store procedures. Moreover, Minimal APIs also provides faster response for API calls.

- **Security:** Utilizing Https through middleware pipeline and using DataIntegrityValidatorFilter to endpoints ensures data protection and that they are not changed when API communicate with the UI. Implementing mechanisms like Authentication and Authorization using JWT provides security. Another measure is the custom permission-based system by adding permission to each endpoint that ensures user is authorized to access a resource outer circle. Moreover, when a user tries to login or to reset his password is encrypted when sending the request to the API.

6.2 Future Work

Future work to this project could be to implement CQRS Pattern along with 2 different database schemas. By doing this approach our indexes will be more meaningful in the application and we can get the actual benefit of CQRS Pattern. It's important to mention that having separate schemas for read and write operations allows you to scale them independently with appropriate resources. Another interesting Future Work is to try to take this solution and try to break it to multiple projects to implement Microservices without breaking Clean Architecture principles. Last, an important feature requested for future development in the Employee Management System was to keep track of who worked in a project and the projects that an employee work even though we delete an employee from a project when a project finishes.

6.3 Feedback

The feedback from the company was very positive and rewarding for me. Many Team Leaders use the Base Software as a starting point in their new projects to save time. About the architecture, most of the developer's state that this is a very powerful architecture that can make development easier simplifying the tasks by splitting them into steps as proposed by the architecture. Code is cleaner, extensible and maintainable and can help to minimize bugs. However, it is noticed that in times of high-pressure to deliver tasks developers violates some of the theoretical principles that are not adopted for «faster development». In my opinion and

as my experience shows whenever this happens instead of saving time we spend more time in the future when we need to maintain a project or to add new features. It's very important for Team Leaders to require from developers to follow this architecture to gain the maximum of its benefits. About the employee management system Team Leaders use it on daily basis to manage the employees and the projects automatically without using excel anymore. The generic Architecture, Base Software and Employee Management System was the reason for my promotion in the company indicating how valuable was the research and the application.

6.4 Conclusion

To conclude, this diploma thesis has achieved to build a modern software application that solves the problem of a reusable project so companies can have a starting point for incoming projects. Importantly that was achieved through solving another problem of managing employees by creating an internal management system. This Web Application has achieved to integrate Microsoft recommended libraries such as FluentValidation, Dapper, MediatR, Swashbuckle, JwtBearer, following Object-Oriented Principles, CQRS Pattern and all this under Clean Architecture and ASP.NET Core Minimal API.

References

- [1] Singh A., “A Deep Dive into Clean Architecture and Solid Principles”, Building Software for the Future, 2023. https://medium.com/@unaware_harry/a-deep-dive-into-clean-architecture-and-solid-principles-dcdcec5db48a
- [2] Šahbaz E., “Comprehensive Guide to SOLID Principles in C#, 2023. [Illustration] <https://medium.com/@edin.sahbaz/comprehensive-guide-to-solid-principles-in-c-54d79e19b7d7>
- [3] Šahbaz E., “Comprehensive Guide to SOLID Principles in C#, 2023. <https://medium.com/@edin.sahbaz/comprehensive-guide-to-solid-principles-in-c-54d79e19b7d7>
- [4] Ghosh T., “Liskov Substitution Principle (LSP). 2023. <https://tusharghosh09006.medium.com/liskov-substitution-principle-lsp-744eceb29e8>
- [5] Singh A., “A Deep Dive into Clean Architecture and Solid Principles”, Building Software for the Future, 2023. [Illustration] https://medium.com/@unaware_harry/a-deep-dive-into-clean-architecture-and-solid-principles-dcdcec5db48a
- [6] Ghosh T., “Liskov Substitution Principle (LSP). 2023. [Illustration] <https://tusharghosh09006.medium.com/liskov-substitution-principle-lsp-744eceb29e8>
- [7] Colton, “Differences Of Abstract And Virtual Methods”, 2021. <https://medium.com/the-crazy-coder/differences-of-abstract-and-virtual-methods-4ff833f2cd49>
- [8] Shekhawat S. S., “Virtual Method in C#”, 2024. [Illustration] <https://www.c-sharpcorner.com/UploadFile/3d39b4/virtual-method-in-C-Sharp/>
- [9] Hasan R., “SOLID principles explanation using Clean Architecture and CQRS”, 2023. <https://medium.com/@rabbyofc/solid-principles-explanation-using-clean-architecture-and-cqrs-caf209f6bd68>

- [10] Abay Ö., “MediatR”, 2023. <https://omerabay1.medium.com/mediatr-795155841b27>
- [11] Shukla A., “Implementing Mediator Patter using the MediatR Library”, 2023. <https://www.codeproject.com/Articles/5368707/Implementing-Mediator-Pattern-using-the-MediatR-Li>
- [12] Jovanovic M., “CQRS PATTERN WITH MEDIATR”, 2023. <https://www.milanjovanovic.tech/blog/cqrs-pattern-with-mediatr>
- [13] Kurbegovic E., “CQRS Softwarer Architecture Pattern: The Good, the Bad, and the Ugly, 2023. <https://medium.com/@emer.kurbegovic/cqrs-software-architecture-pattern-the-good-the-bad-and-the-ugly-efe48e8dcd14>
- [14] Jovanovic, M. “CQRS validation with MediatR Pipeline and FluentValidation”, 2023. <https://www.milanjovanovic.tech/blog/cqrs-validation-with-mediatr-pipeline-and-fluentvalidation>
- [15] Hengkyawan, J. “Model Validation using Fluent Validation”, 2022. <https://juldhais.net/model-validation-using-fluent-validation-eaae2f4952b8>
- [16] Pandey, U. “ Fluent Validation ASP.NET Core Web API 6.0”, 2023. <https://www.c-sharpcorner.com/article/fluent-validation-asp-net-core-web-api-6-0/>
- [17] Thevathas, D. A., “FluentValidation Introduction and Setup In. NET”, 2023. <https://atdilakshan.medium.com/fluentvalidation-introduction-and-setup-in-net-1d1024812786>
- [18] Spasojevic M., “Global Error Handling in ASP.NET Core Web API”. 2024. <https://code-maze.com/global-error-handling-aspnetcore/>
- [19] Tilleuil D., & Dechamps G., “The importance of the dependency inversion principle”, n.d. <https://www.tripled.io/07/05/2019/dependency-inversion-principle/>

- [20] Nanavaty, R., “Clean Architecture”, 2023.
<https://medium.com/@rudrakshnanavaty/clean-architecture-7c1b3b4cb181>
- [21] Rizal, D. H. F., “101 Clean Code Architecture”, 2023. <https://medium.com/bento-tech-innovation/101-clean-code-architecture-651e2650bbe8>
- [22] Jovanovic, M., “Why it’s great for complex projects”, 2023.
<https://www.milanjovanovic.tech/blog/why-clean-architecture-is-great-for-complex-projects>
- [23] Shirsath R., “Summarized Clean Code Architecture”, 2020.
<https://reemishirsath.medium.com/summarized-clean-code-architecture-concept-3b947ad44ef1>
- [24] Abstract Class vs Interface in C#: Analyzing the Pros and Cons.
<https://dev.to/bytehide/abstract-class-vs-interface-in-c-analyzing-the-pros-and-cons-32mj>
- [25] C# Multiple inheritance using interfaces, 2023: <https://www.geeksforgeeks.org/c-sharp-multiple-inheritance-using-interfaces/>
- [26] C# documentation: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>
- [27] Introduction to the Angular docs: <https://angular.io/docs>
- [28] What is SQL Server?: <https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16>
- [29] Overview of ASP.NET Core:
<https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16>
- [30] Minimal APIs overview: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/overview?view=aspnetcore-8.0>

- [31] RouteGroupBuilder Class: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.routing.routebuilder?view=aspnetcore-8.0>
- [32] RouteHandlerBuilder Class: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.routehandlerbuilder?view=aspnetcore-8.0>
- [33] ASP.NET Middleware: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0>
- [34] ASP.NET Middleware [Illustration]: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0>
- [35] Filters in Minimal API apps: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/min-api-filters?view=aspnetcore-8.0>
- [36] Introduction to JSON Web Tokens: <https://jwt.io/introduction>
- [37] Authentication and authorization in minimal APIs: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/security?view=aspnetcore-8.0>
- [38] MediatR: <https://github.com/jbogard/MediatR>
- [39] Learn Dapper: <https://www.learndapper.com/>
- [40] Create a database schema: <https://learn.microsoft.com/en-us/sql/relational-databases/security/authentication-access/create-a-database-schema?view=sql-server-ver16>
- [41] Description of the database normalization basics: <https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description>
- [42] Allies P., “Clean Architecture: Entities and Models”, 2023.
<https://nanosoft.co.za/blog/post/clean-architecture-entity-model>

- [43] Siva V., “Minimal APIs in ASP.NET Core: Compare With Controller”, 2023.
<https://www.c-sharpcorner.com/blogs/minimal-apis-in-asp-net-core-a-lean-approach-to-web-development>