Build .NET 8 APIs in Clean Architecture, user Identity system and Azure deployment using CI/CD + practical exercises

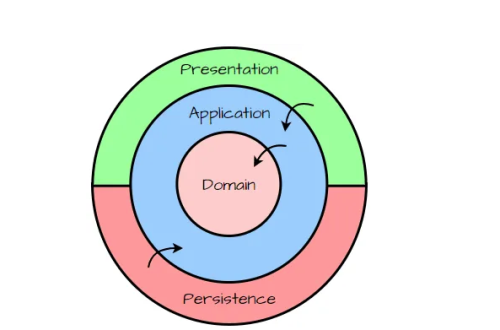
# Learning Objective

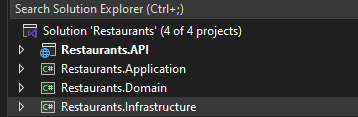
* REST Web API
* Solution Architecture
* Infrastructure Setup
* API Controller
* CQRS + MediatR
* WEB API Utilities
* Sub-entities
* Authentication
* Authorization
* Pagination
* Automated tests
* Azure Cloud
* CI/CD pipelines

# CLI Commands

* dotnet new webapi –n <project-name> --no-openapi –controllers “Create a webapi project with controller”
* dotnet new sln “Create sln with folder name”
* dotnet sln add “<project path>” like “./Restaurants.API” “Adding the project on sln”

# Solution Architecture

**Benefits**

* Testability
* UI independence
* Free to choose/change the database
* Flexibility
* Transparency

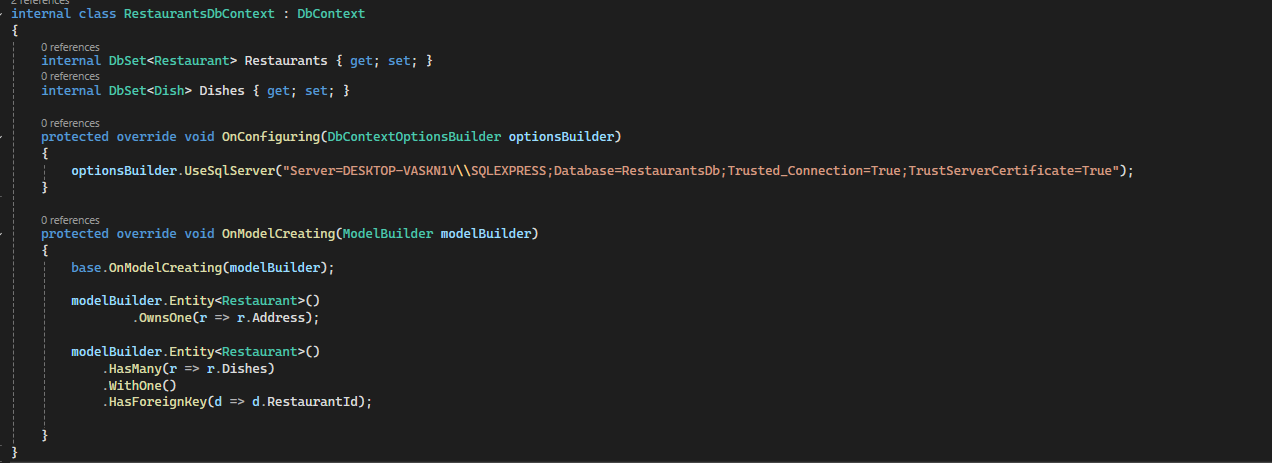
# Setting up the Infrastructure

* Installing the entity framework packages

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Tools

* Creating the dbContext class



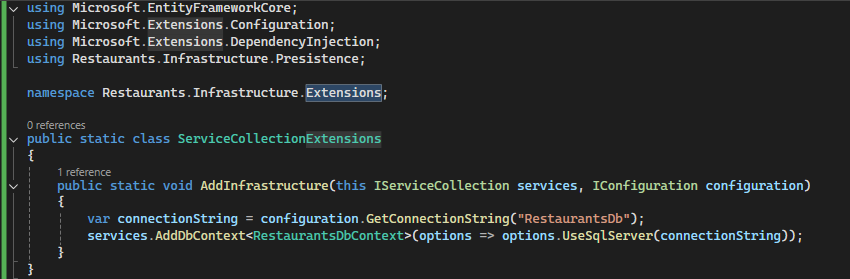
* Add the migration and update the database via package manager console.

## Flexible Database Connection

* Update the DbContext File to removing the **OnConfiguring** method and use the constructor



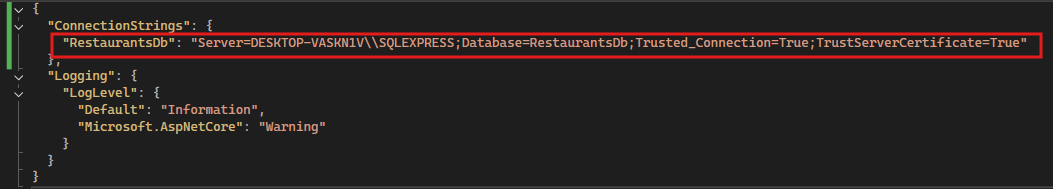
* Create an Extensions methods



* Inject the method into main project **program.cs** file

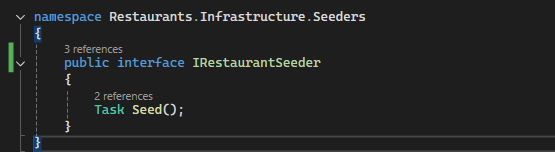


* Move the connection string to main project **appsettings.Development.json**

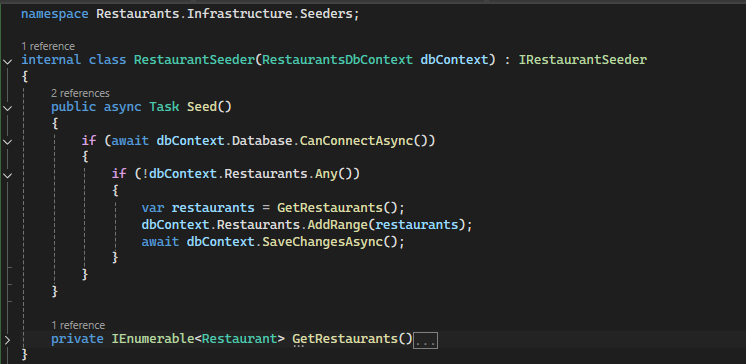


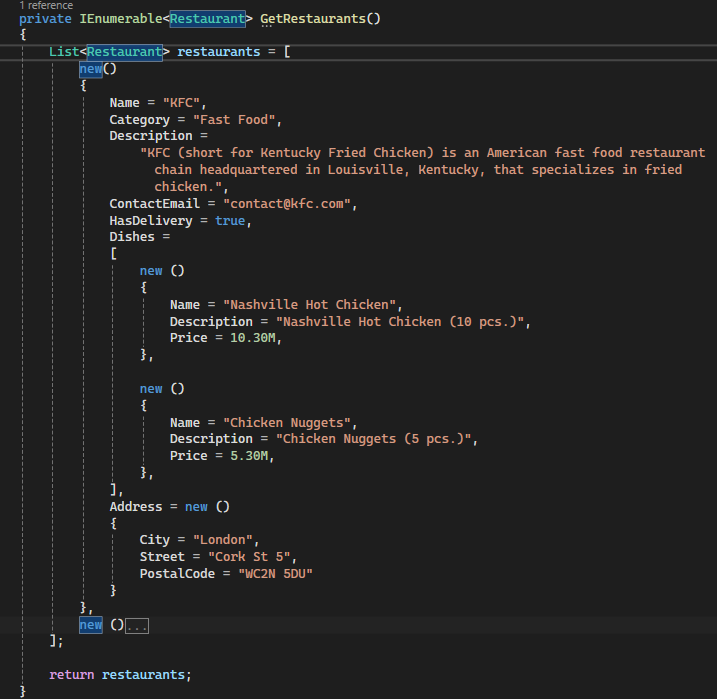
## Seeding Data

* Create an Interface on the Infrastructure project

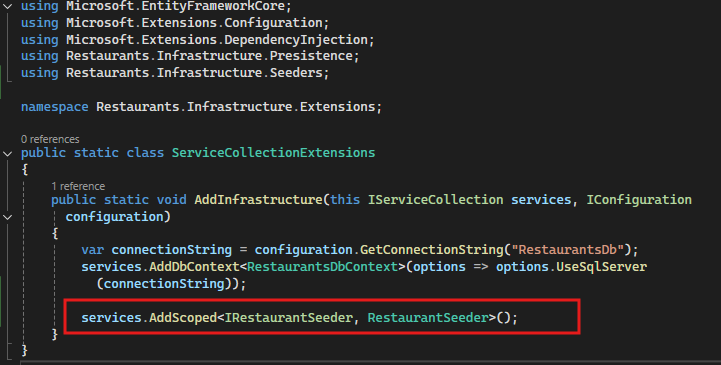


* Create a class to implement the seeder interface.





* Update the extension method for dependency injection

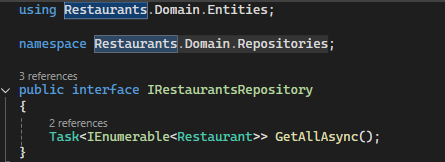


* Update the program.cs file

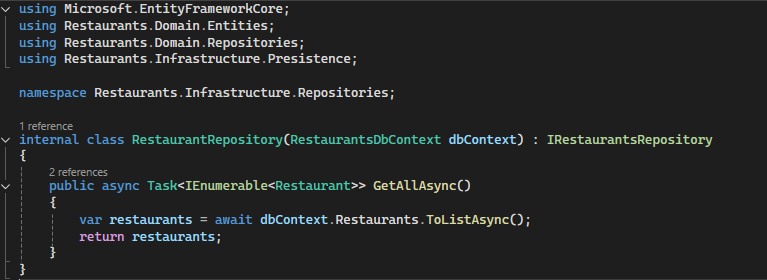


# API Controller

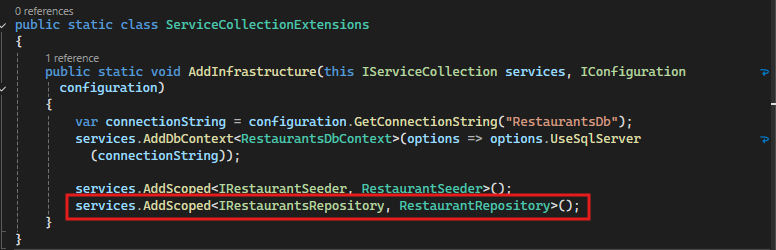
* Create repository interface on the Restaurants.Domain project inside the Restaurants.Domain.Repositories folder



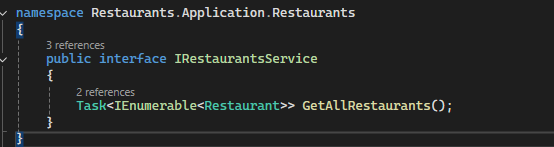
* Create a Repository class on the Restaurants.Infrastructure project inside the Restaurants.Infrastructure.Repositories folder, due the dbContext file in the project



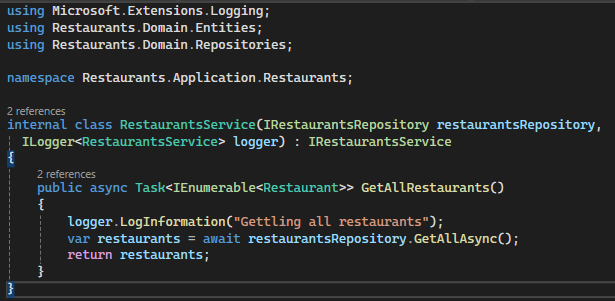
* Update the ServiceCollectionExtensions class for adding the dependency between the newly created repository interface and class



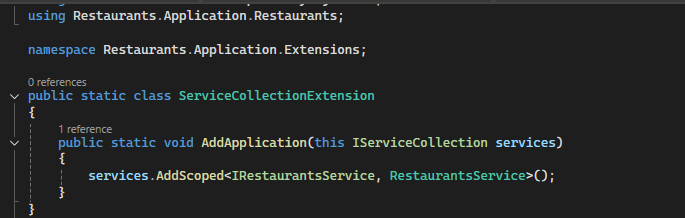
* Under the Restaurants.Application project inside the Restaurants.Application.Restaurants folder create the interface for service layer



* Under the same folder implement the interface and retrieve the repository method



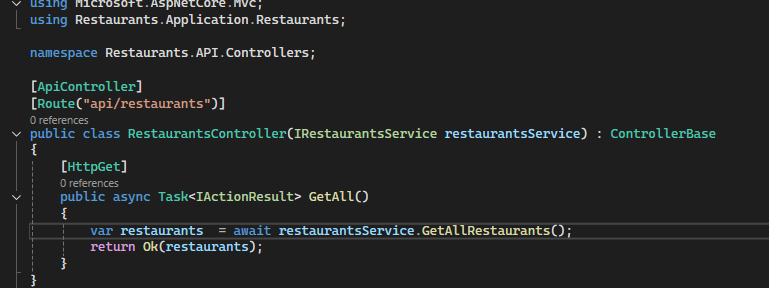
* Create an Extensions method under the Restaurants.Application project inside the Restaurants.Application. Extensions folder



* Inject to the program.cs file



* Create a controller with GET method

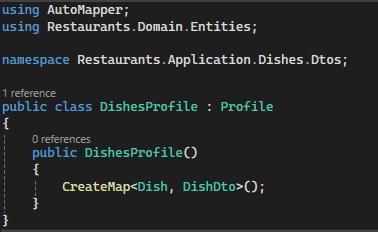


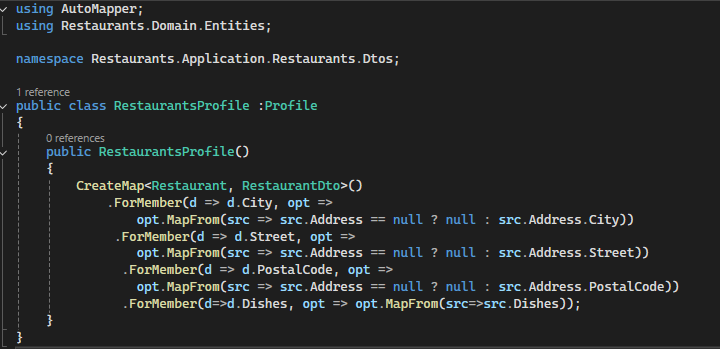
# Automapper

* Package needed

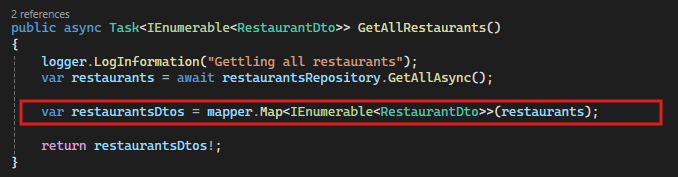
AutoMapper.Extensions.Microsoft.DependencyInjection

* Create a profile class under the Restaurants.Application project





* Update the service layer

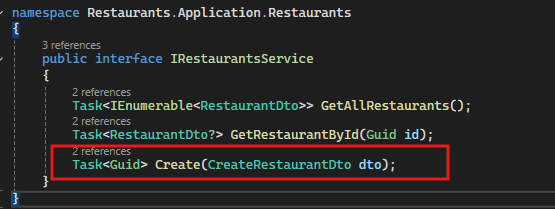


* Update the ServiceCollectionExtension

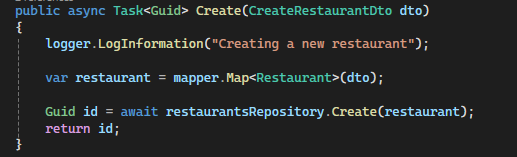


# Creating Resources

* Update the interface on the Restaurants.Application project



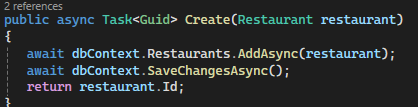
* Implement the update interface method



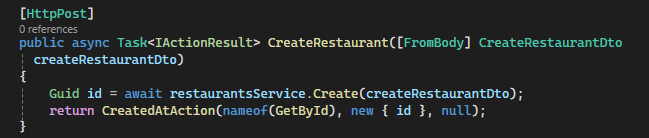
* Update the interface on the repository interface inside the Restaurants.Domain project



* Implement the update interface method

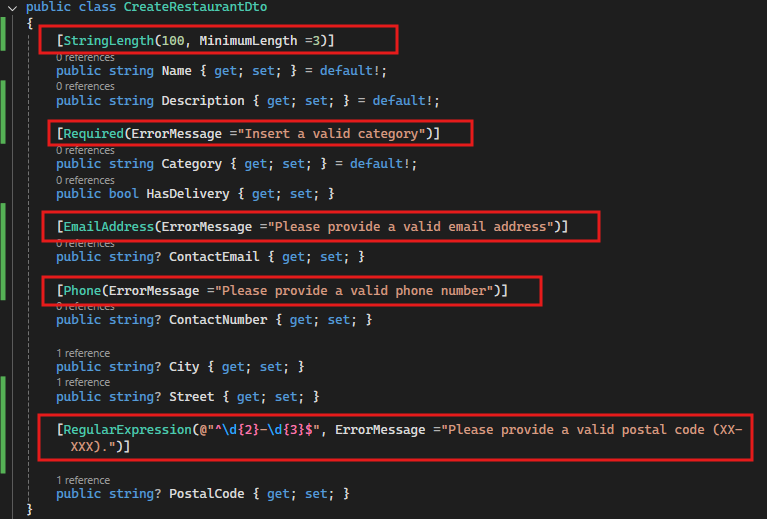


* Crate a new method on the controller for post



# Model Validation

* Update and add DataAnnotations the Dto which is passing through POST body

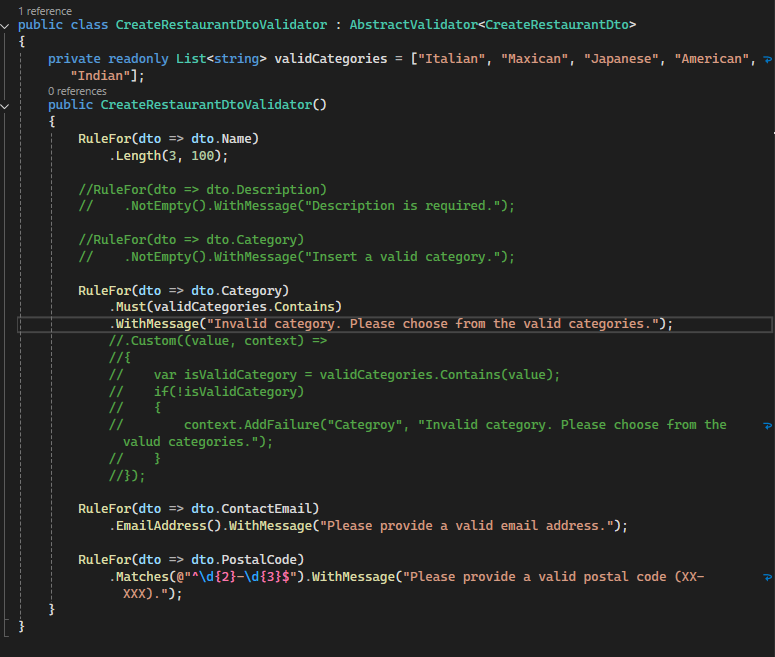


# FluentValidation

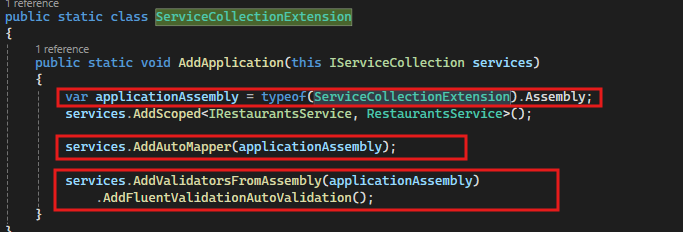
* Install the NuGet Package on the Restaurants.Application project

FluentValidation.AspNetCore

* Create a validator Class



* Update the ServiceCollectionExtension Class



# CQRS + MediatR

## CQRS Introduction

* **Command/Query Responsibility Segregation** – a design pattern that segregates responsibilities depending on whether a given operation writes or retrieves data.

|  |  |
| --- | --- |
| **Command** | **Query** |
| Responsible only for writing/saving the data | Responsible only for retrieving the data |
| Validates the model | Checks the request context |
| Processes application logic | It is idempotent |

**Mediator**

**Mediator**

GetAll Restaurants Handler

Create Restaurant Handler

GetAll Restaurants Query

Create Restaurant Command

* Install the package on the **Restaurants.Application** project
* MediatR

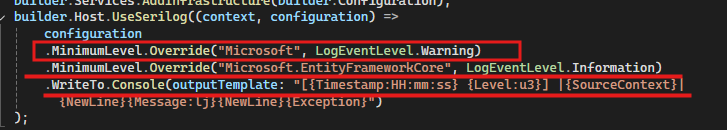
# Web API utilites

## Adding Loggers (Console)

* Adding Serilog on the project

Serilog.AspNetCore

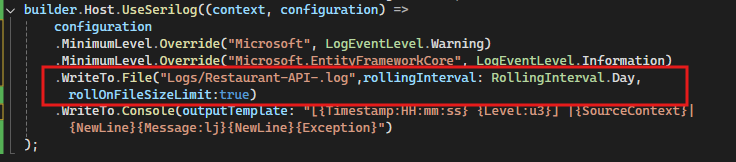
* Update the program.cs file

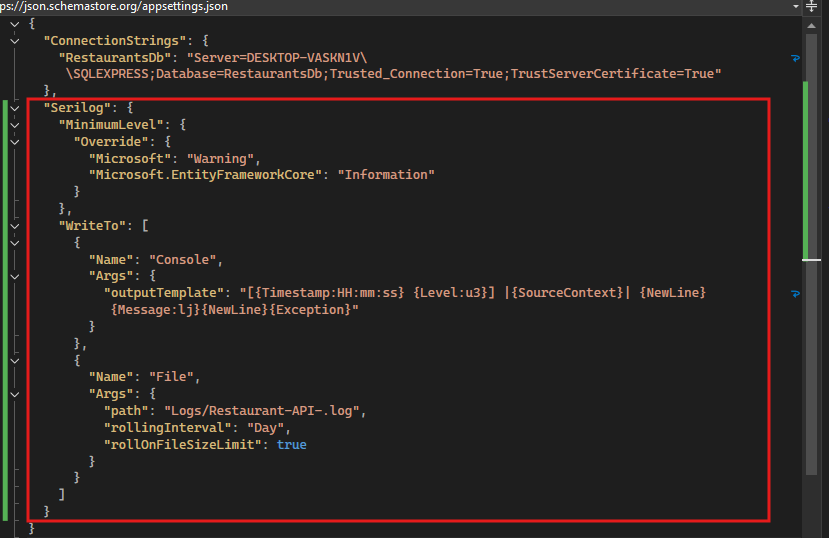


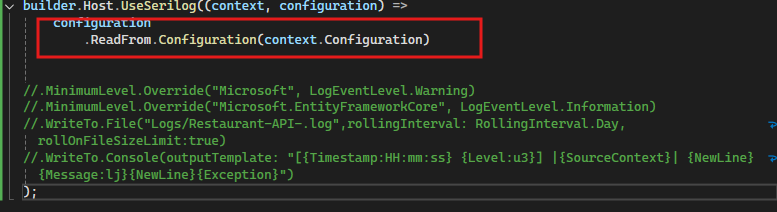


## Adding Loggers (Into File)

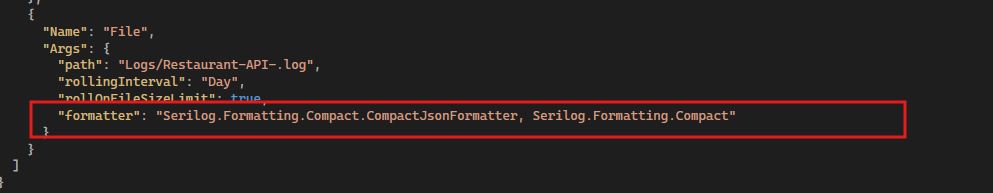
* To log the logs into the file first update the set of line on the program.cs file where provide the file name and some of the args



* To getting all the Serilog settings from the appsettings.development.json file we update the json file like:
* 
* For reading the configuration from the json we update the program.cs now



* For formatting the logs



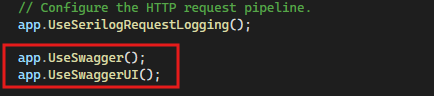
## Swagger docs

* Install the package

Swashbuckle.AspNetCore

* Update the program.cs file add service and middleware





## Exception Handling Middleware

* Create an Exception Handling Middleware



* Created newly created Middleware Dependency on program.cs file

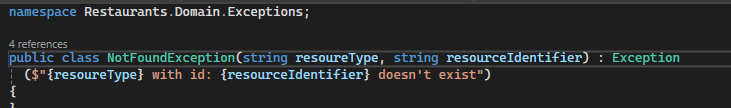


* Register the Middleware on the program.cs file

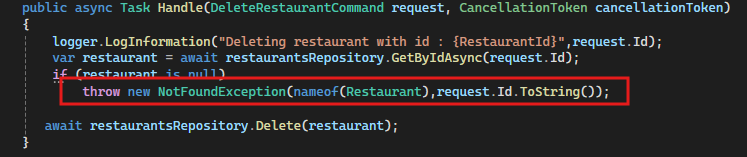


## Custom Exception Handling

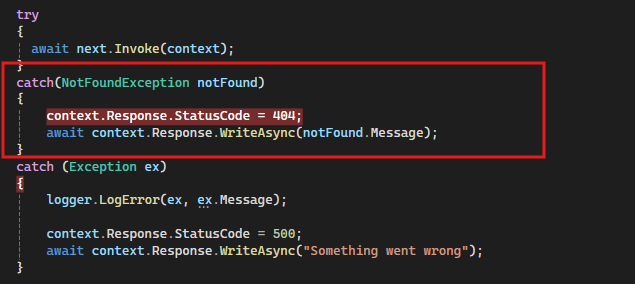
* Create a separate exception file inherit the ***Exception*** class



* Throw the exception each time if code not found the id



* Update the Middleware ErrorHandlingMiddle, so each time if exception throws then it’s execute the custom exception.



# Subentities

## Subentities in REST

* Like if we need to create such type of apis like:

|  |  |
| --- | --- |
| /api/restaurants/{restaurantId}/dishes | GET |
| /api/restaurants/{restaurantId}/dishes | POST |
| /api/restaurants/{restaurantId}/dishes/{id} | GET |
| /api/restaurants/{restaurantId}/dishes/{id} | PATCH |
| /api/restaurants/{restaurantId}/dishes/{id} | DELETE |

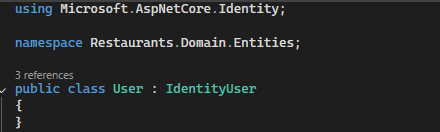
# Authentication

## ASP.NET Identity

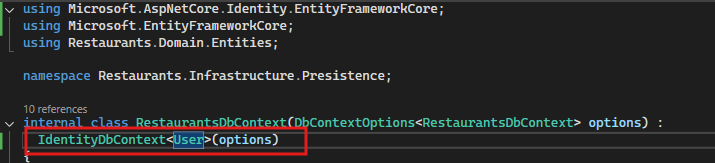
* Install the package

Microsoft.AspNetCore.Identity.EntityFrameworkCore

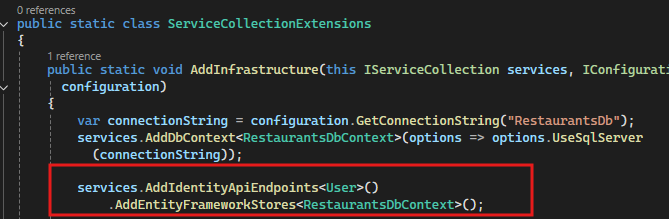
* Now create the class inside the *Restaurants.Domain.Entities* name of *User* and inherit the class ***IdentityUser***



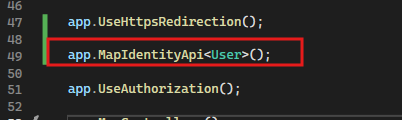
* Update the DbContext file also replace IdentityDbContext class to DbContext class like:



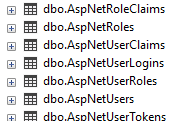
* Update the **ServiceCollectionExtensions** class



* Update the **Program.cs** file



* Then initiate the database migration via cli commands, after update the database new table created on the database like:



* POST the http request to register for register the user

POST {{Restaurants.API\_HostAddress}}/register

Content-Type: application/json

{

"email":"testuser@test.com",

"password":"Password1!"

}

* POST the login to get the Bearer token

Post {{Restaurants.API\_HostAddress}}/login

Content-Type: application/json

{

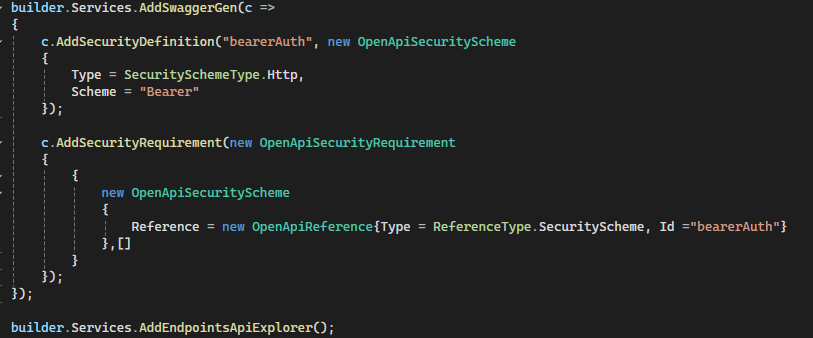
"email":"testuser@test.com",

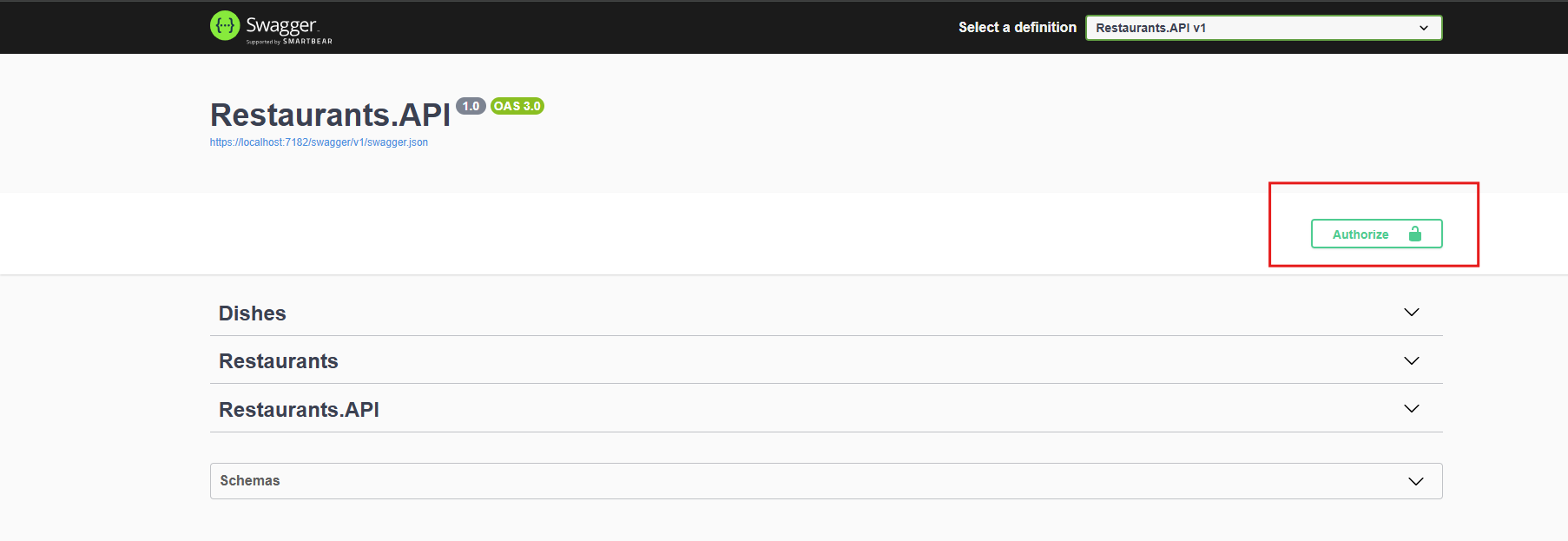
"password":"Password1!"

}

## Swagger support of identity

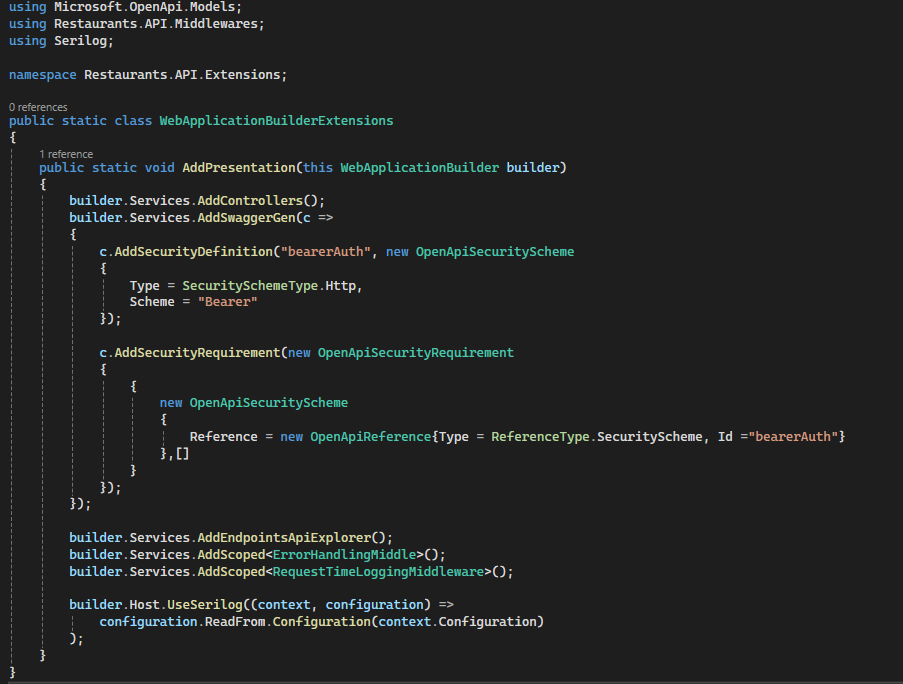
* Update the program.cs file



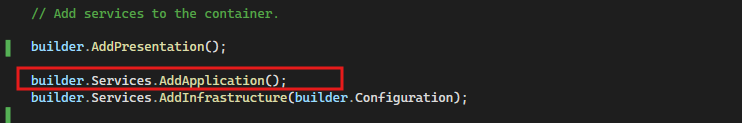


## Extracting presentation layer services

* Create a new extenstions method under the Restaurants.API project name of the folder like: Extensions
* Update the class to move the services from program to newly created static class

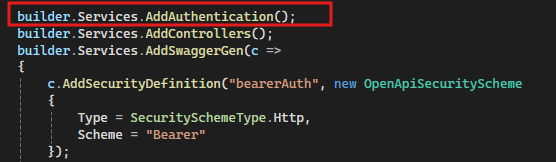


* Update the program.cs file

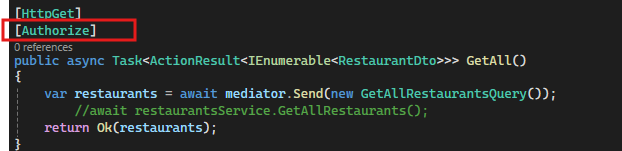


## Authentication setup

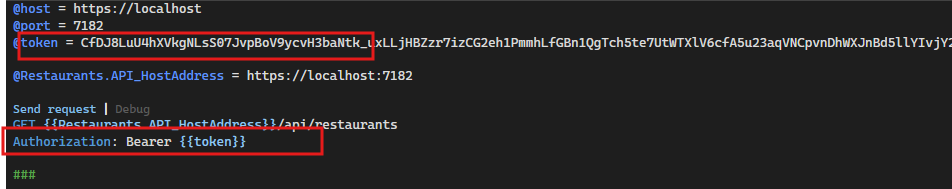
* Update the AddPresentation extension method



* Update the controller to adding the new attribute above the request



* Send the request using the authorization token



# Authorization

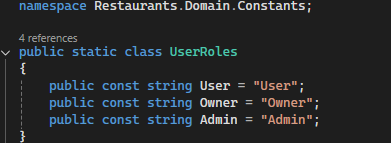
In a .NET Core API, "authorization" refers to the process of determining whether a verified user (authenticated) has the necessary permissions to access a specific resource or perform an action on it, essentially deciding what a user is allowed to do within the API based on their identity and assigned roles or claims; it's a security mechanism that controls access levels beyond just confirming a user's identity

**Access control**

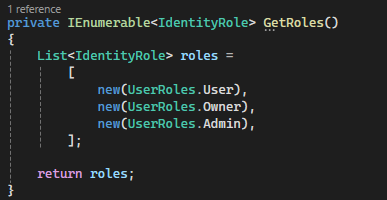
* Role Based
* Attribute Based
* Resource based

## Role Based Authorization

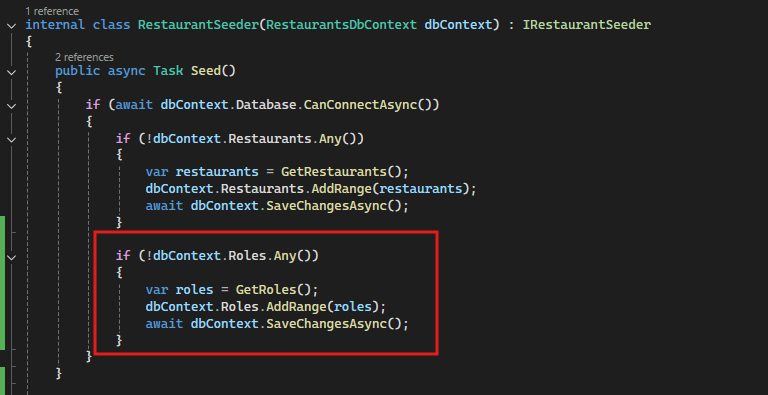
* Create a static UserRoles class under the project **Restaurants.Domain**



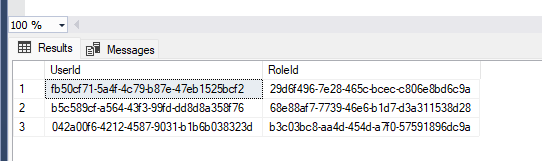
* Update the RestaurantSeeder class under the project **Restaurants.Infrastructure** adding new method which return new **IdentityRoles**



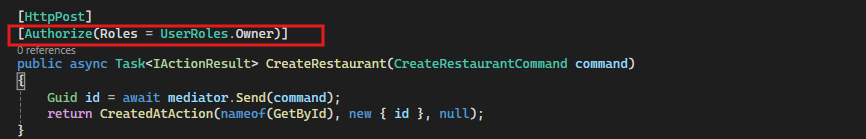
* Update the Seed method so before the project run seeding complete if roles not exist.



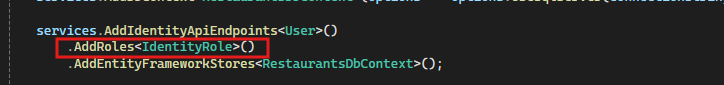
* Update the **AspNetUserRoles** table to assign the user to respective roles



* Update the controller

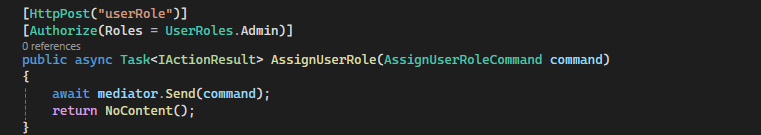


* Update the Extension method under the **Restaurants.Infrastructure** to adding the roles

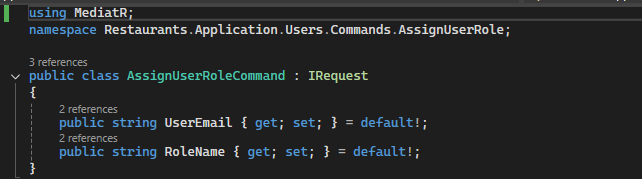


## Assigning user roles

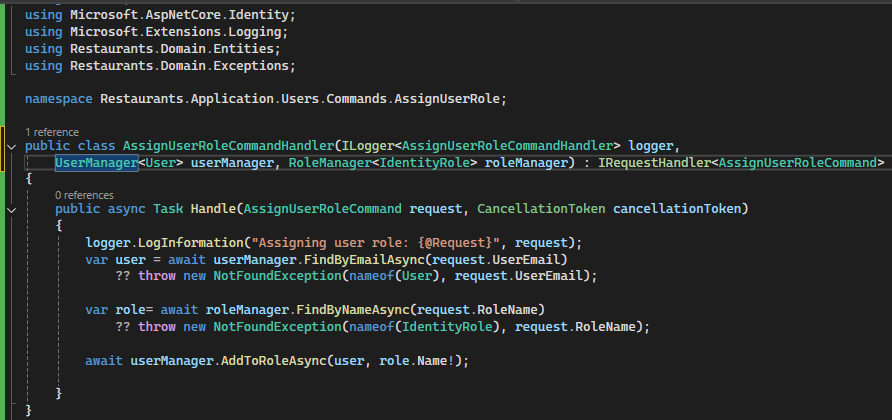
* Updating the **IdentityController**



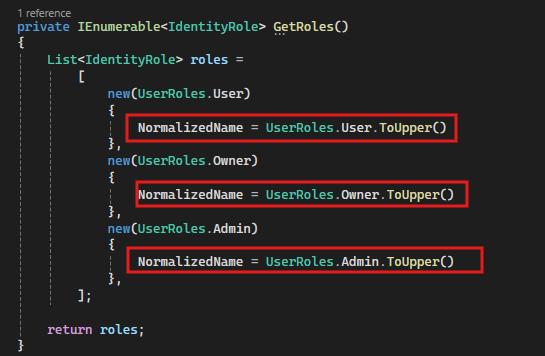
* Create a **Command** under the **Restaurants.Application** project under folder **Restaurants.Application.Users.Commands.AssignUserRole**



* Create a **Handler** under the folder **Restaurants.Application.Users.Commands.AssignUserRole**

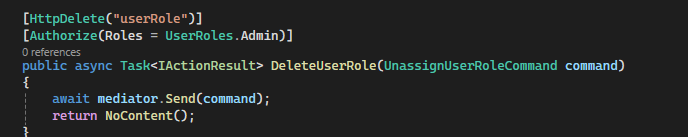


* If getting error while assigning the role to the user check the database table **AspNetRoles** is normalized of not or update the **RestaurantSeeder** class.

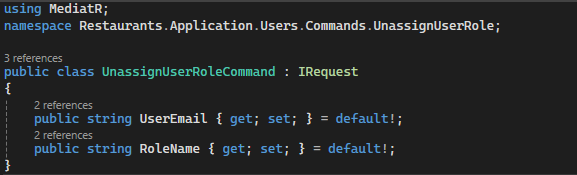


## Delete User Role

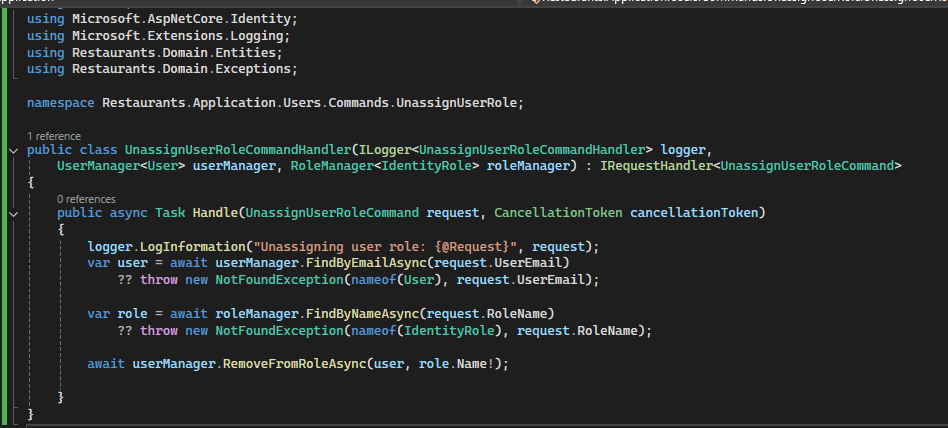
* Update the IdentityController



* Create the command

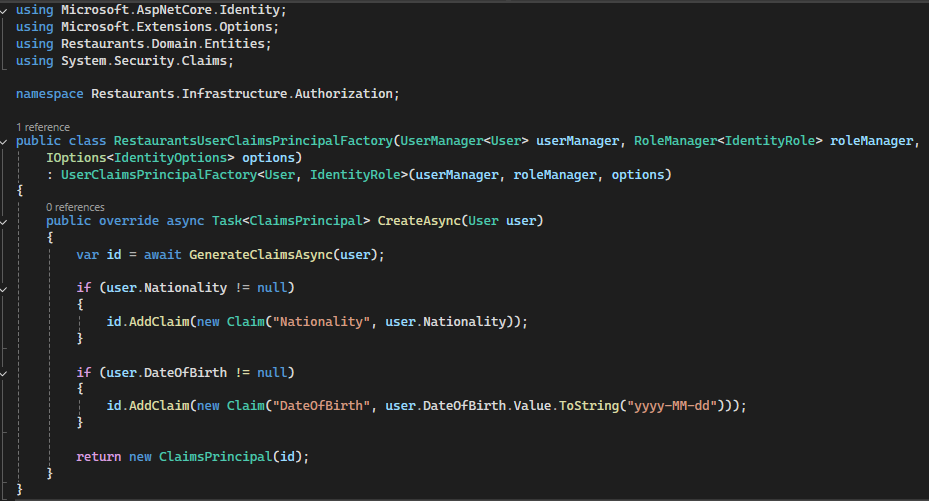


* Create the handler

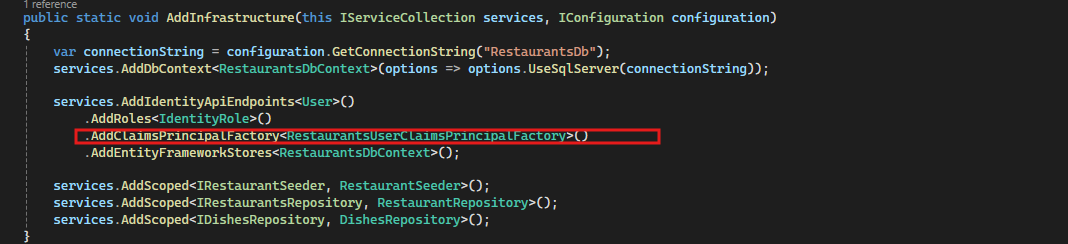


## Custom User claims

* Create the RestaurantsUserClaimsPrincipalFactory under the project Restaurants.Infrastructure under folder Restaurants.Infrastructure.Authorization

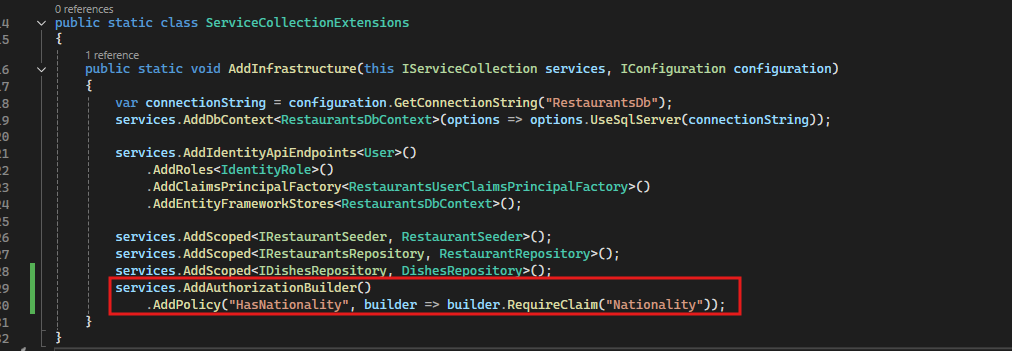


* Update the ServiceCollectionExtensions class

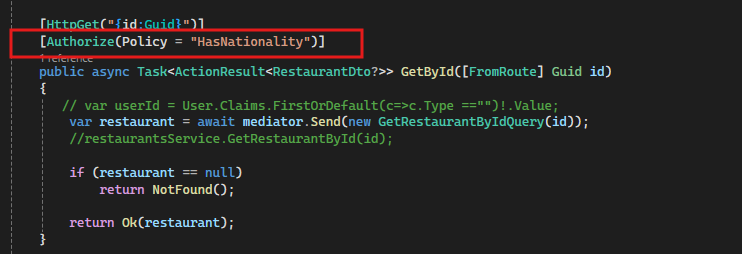


## Claim Based access control

* Update the ServiceCollectionExtensions class under the Restaurants.Infrastructure.Extensions project



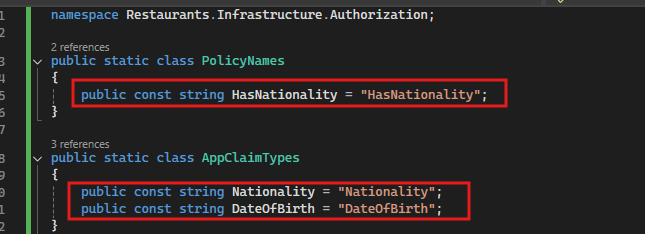
* Then update the controller for claim based authorization



* If you to add parameters first is **claimtype** and second is **parms**



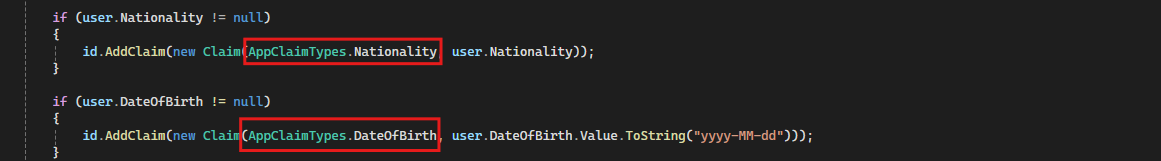
* Create a common Constant class to access the values



* Replace the hardcoded value with these variables



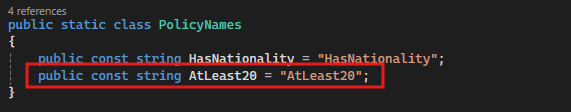




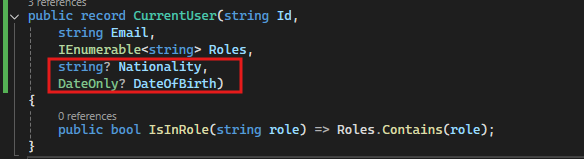
## Custom Authorization requirements

Here we adding new functionality for adding custom authorization to check user age is more then or equal to 20 or not for this process we follow these steps:

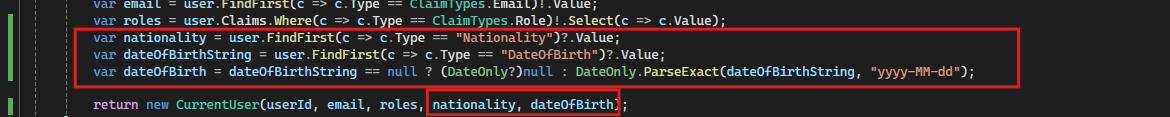
* Update the **Constants.cs** class under the Restaurants.Infrastructure.Authorization project



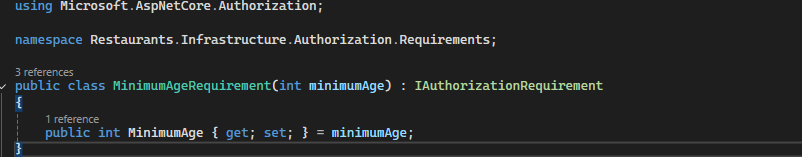
* Update the **CurrentUser** record under the Restaurants.Application.Users project



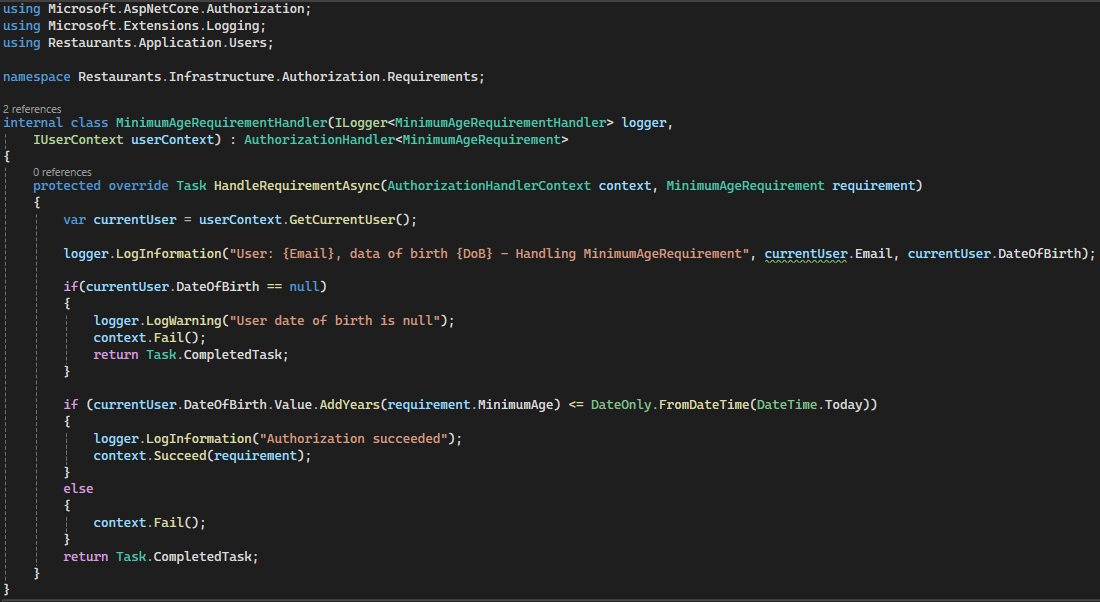
* Update the **UserContext.cs** file under the project Restaurants.Application.Users for adding new details on the current context user



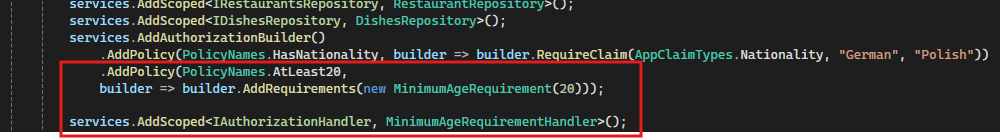
* Create a new class **MinimumAgeRequirement** which inherit the interface IAuthorizationRequirement under the Restaurants.Infrastructure.Authorization.Requirements project



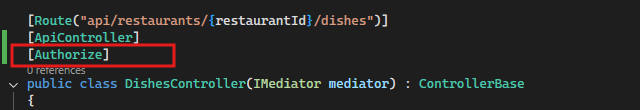
* Create a handler MinimumAgeRequirementHandler under the Restaurants.Infrastructure project to check that minimumAge requirement

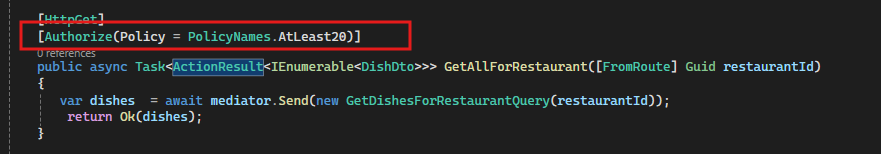


* Update the extensions class ServiceCollectionExtensions under the Restaurants.Infrastructure.Extensions project to implement into middleware



* After that update the DishesController controller

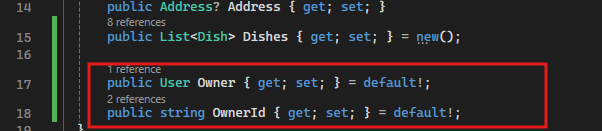




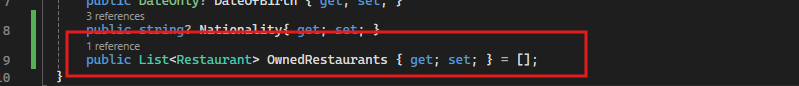
## User Resources

In this part we learn how to update the existing table with foreign key and see how one to many relationship works:

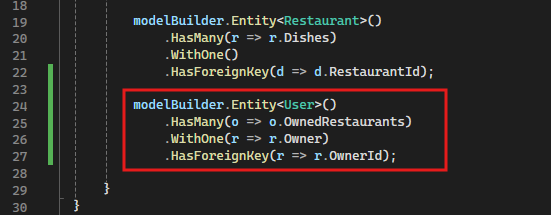
* Update the entity class Restaurant under the project Restaurants.Domain



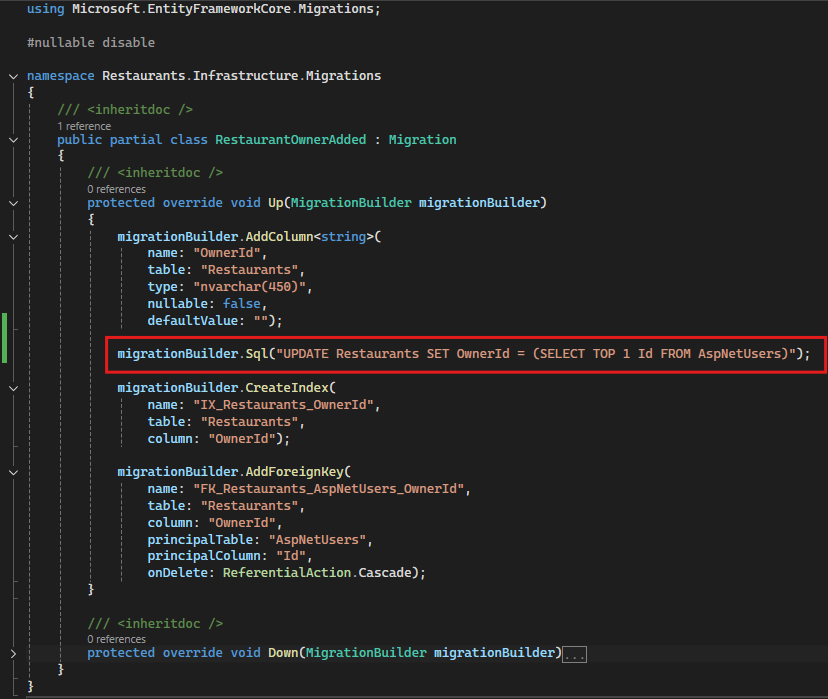
* Update the User entity class User under the project Restaurants.Domain.Entities



* Update the DbContext class RestaurantsDbContext under the project Restaurants.Infrastructure.Presistence



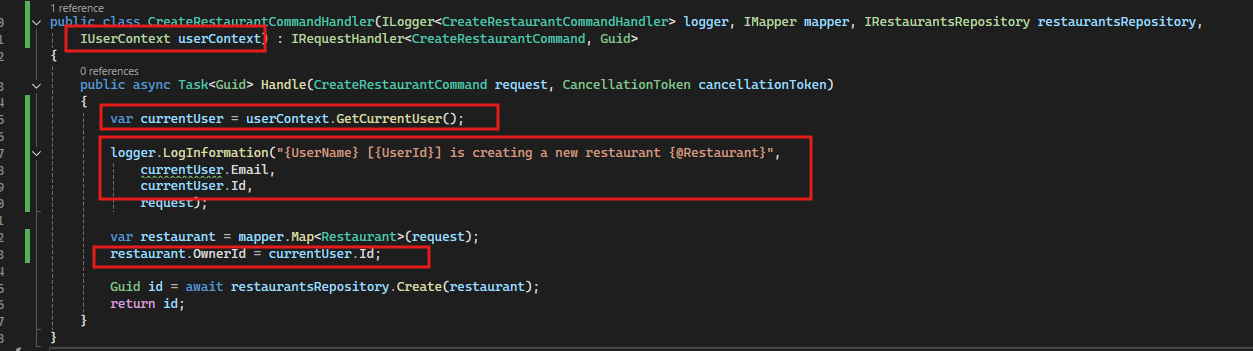
* After the add-migration on the Restaurants.Infrastructure project, once migration run successfully then update the migration class



* On the above image the highlighted section added that because the container foreign key concept if we not adding this so we getting this error



* Update update-database command on CLI database update now update the CreateRestaurantCommandHandler class under the Restaurants.Application.Restaurants.Commands.CreateRestaurant project for creating a new restaurant request

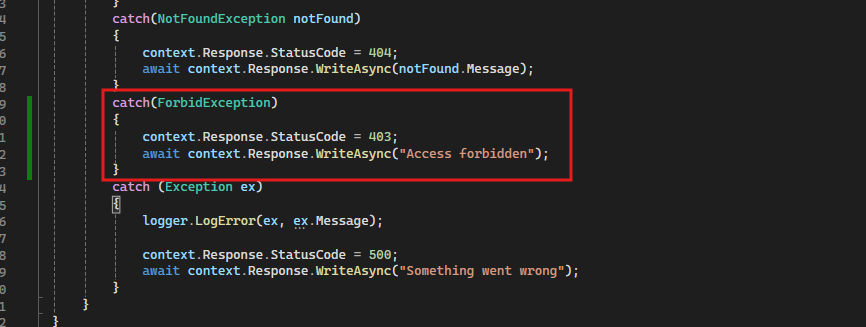


## Resource based authorization

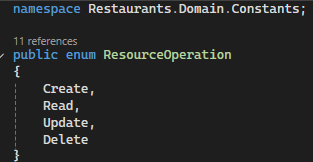
When authorizing requests in ASP.NET Core Web API projects, it’s not enough to know the endpoint and the user; we also need to be aware of the specific resource the user is trying to access. In this article, we will learn how to tackle these situations using resource-based authorization.

Steps to follow

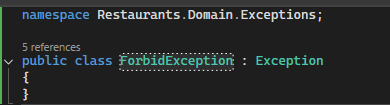
* Update the ErrorHandlingMiddle class under the Restaurants.API.Middlewares project for implementing forbidden exception related to access.



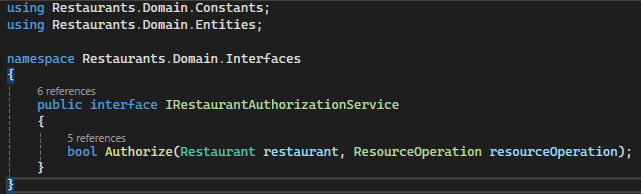
* Create a enum for operation related under the project Restaurants.Domain.Constants



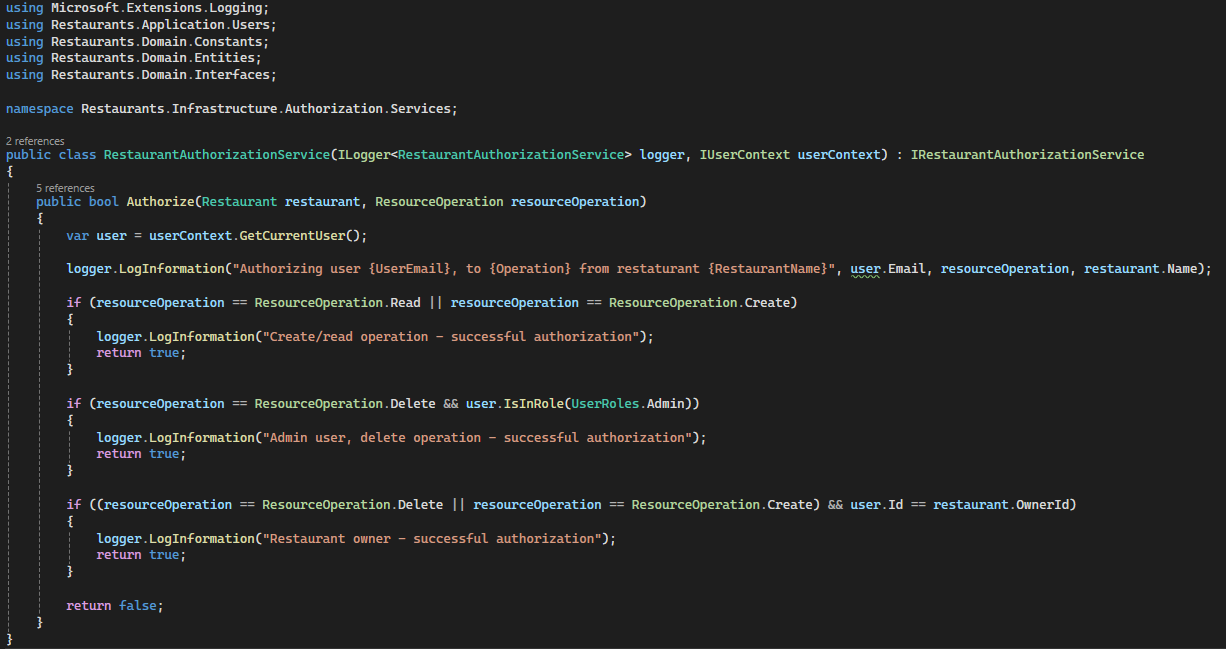
* Create a ForbidException class to implement the Exception class under the project Restaurants.Domain.Exceptions



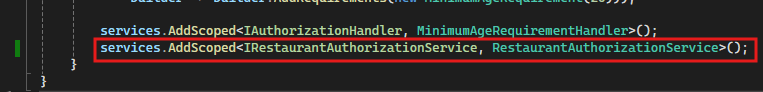
* Create an interface IRestaurantAuthorizationService under the Restaurants.Domain.Interfaces project



* Create a service RestaurantAuthorizationService class under the project Restaurants.Infrastructure.Authorization.Services



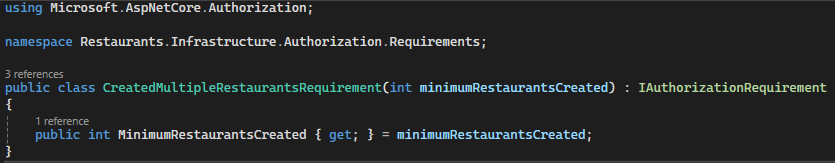
* Update the ServiceCollectionExtensions class under the project Restaurants.Infrastructure.Extensions



## Exercise – authorization policy

Create custom policy, that will authorize users who have are the owners of a least 2 restaurants.

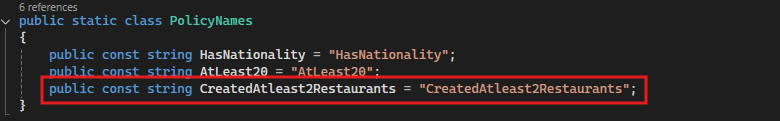
* Create the CreatedMultipleRestaurantsRequirement class under the project Restaurants.Infrastructure.Authorization.Requirements



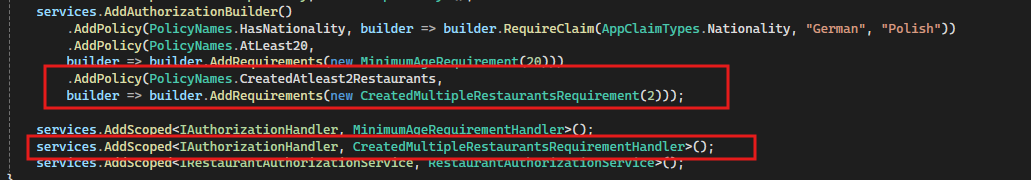
* Create the CreatedMultipleRestaurantsRequirementHandler class under the project Restaurants.Infrastructure.Authorization.Requirements



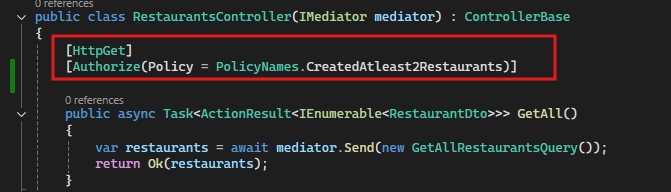
* Update the Constants.cs file



* Update the ServiceCollectionExtensions class under the project Restaurants.Infrastructure.Extensions



* Update the RestaurantsController controller, update the attribute



# Results Pagination

**Pagination Server Perspective benefits:**

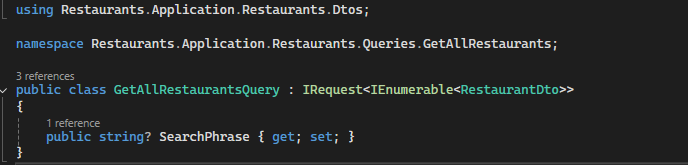
* **Resource Efficiency –** by paginating results, servers can limit the amount of data they need to process and transmit in a single request. This helps in optimizing server resources and ensures that can server can handle requests from multiple clients efficiently.
* **Scalability –** Paginating results allows the server to scale more effectively. Handling large volumes of data in a single request can strain server resources and affect the overall performance. Pagination enables the server to manage requests more effectively, especially in scenarios where the dataset is large.
* **Load Balancing –** Paginating results helps in load balancing across servers. When multiple servers are handling request, distributing the load evenly becomes essential for optimal performance. Pagination allows for a more balanced distribution of data retrieval tasks across servers.
* **Reduced Latency –** By fetching and transmitting smaller chunks of data at a time, pagination can help reduce latency for clients. Clients receive data more quickly because they don’t have to wait for the entire dataset to be processed and transmitted in a single request.

**Pagination Client Perspective benefits:**

* **Improved performance –** Paginating results allows clients to fetch data incrementally, which can improve the responsiveness of the application. Clients can display partial results to users while fetching additional data in the background, providing a smoother user experience.
* **Bandwidth Optimization –** Paginating results reduces the amount of data transferred over the network in each request. This is particularly important for clients with limited bandwidth or when accessing the API over mobile networks. By fetching smaller chunks of data, pagination helps optimize bandwidth usage.
* **Reduced Processing –** Fetching smaller batches of data reduces the amount of processing required on the client side. Clients can focus on rendering and displaying the retrieved data without having to handle large datasets all at once.
* **Scalability –** Pagination allows clients to navigate through large datasets efficiently. Users can browse through pages of result without experiencing performance issues of overwhelming the client application with a massive amount of data at once.

## Filtering results

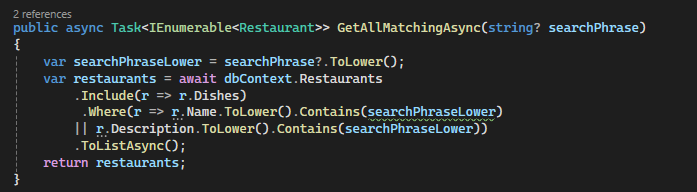
* Update the GetAllRestaurantsQuery under the Restaurants.Application.Restaurants.Queries.GetAllRestaurants project



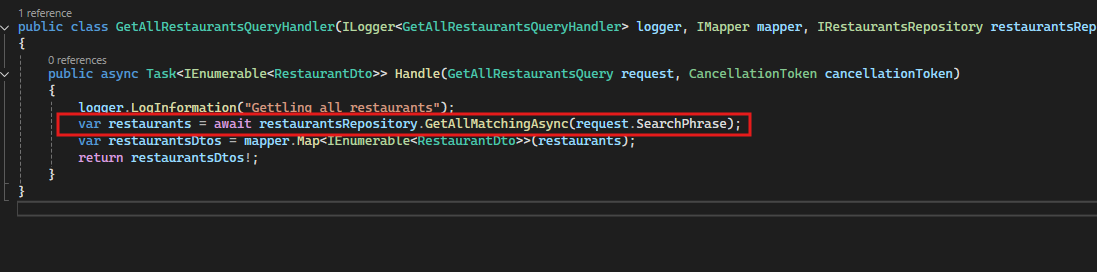
* Update the interface IRestaurantsRepository under the project Restaurants.Domain.Repositories



* Implement the update method of the interface



* Update the QueryHandler GetAllRestaurantsQueryHandler under the project Restaurants.Application.Restaurants.Queries.GetAllRestaurants



* Update the controller
* 