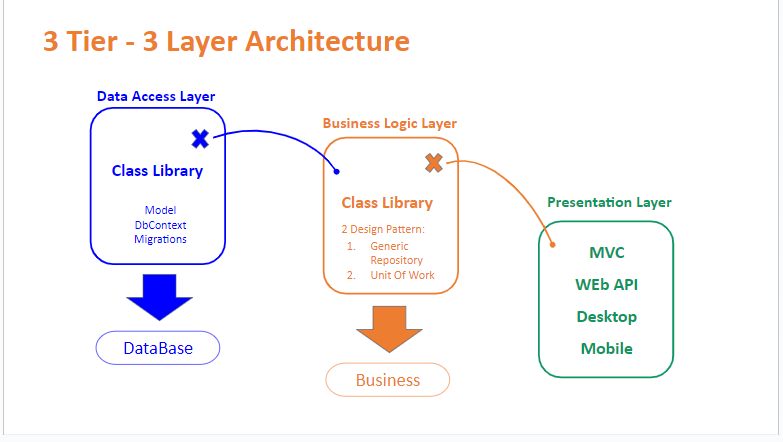
3 Tier – Architeture



Simply we can say it’s a way to organize our application into 3 Sections

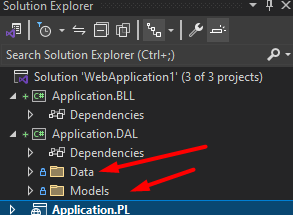
1 – Data Acess Layer -> This will hold all information about our models and database

2 – Bussiness Logic layer -> this will hold all logic behind our Models

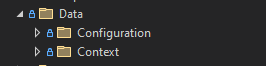
3 – Presination layer -> will hold all view to Our Client

**Data Acess Layer**

**This access layer will hold all Infromation About OUR DataBase configuration**



**Data Folder -> will Hold Context Folder and Configuration Folder**



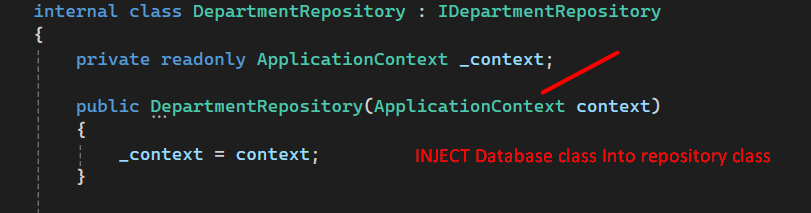
**In our Data access layer we create two Folders**

**Data -> will hold all Data connetions info**

**Models -> will hold all Models Info**

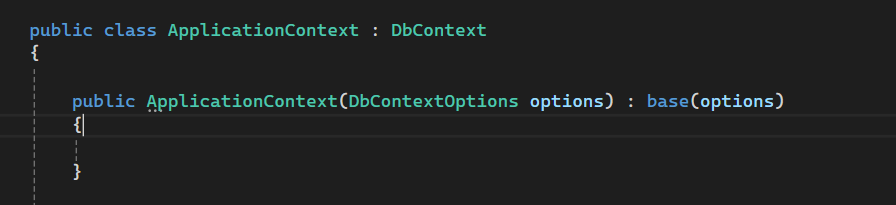
**DataConnection Configuration:**

**When we create A repository we told him to create object from ApplicationContext class to establish a database connection to make an operations on it**

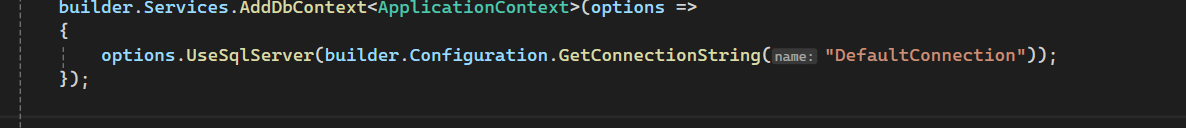


**When we run the application and request the Department Repository the CLR Will inject ApplicationContext then try to create object from it.**

**When clr go to ApplicationContext**

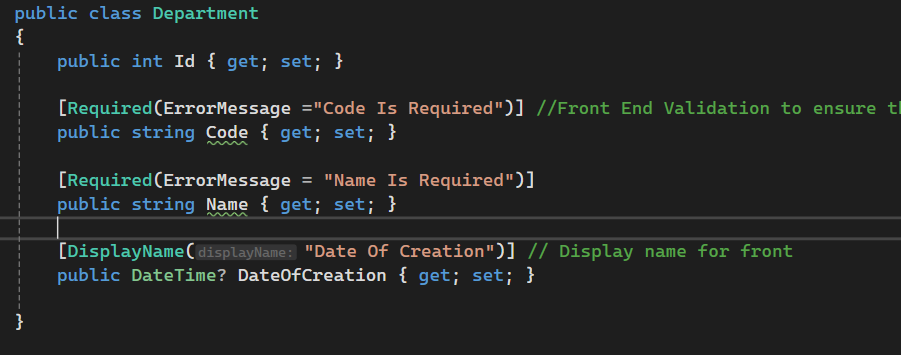


**He will find that we inject another DbcontextOption class to it so he need to create another object then pass it to the parent.**



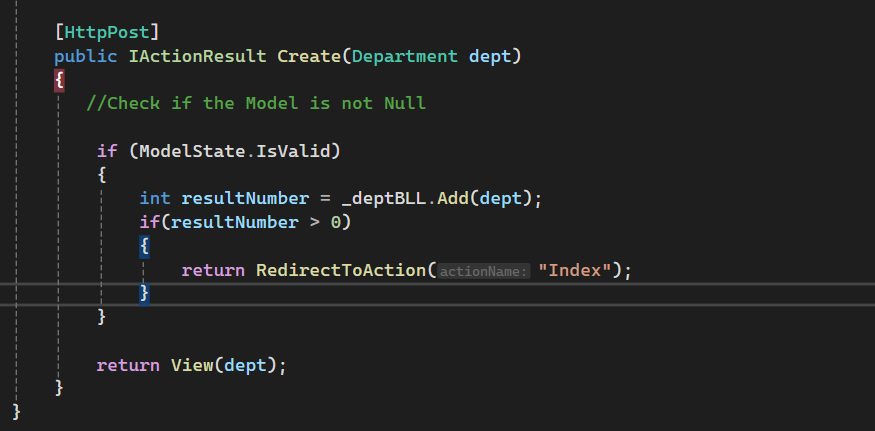
**In this clase we told the compiler that clr will handle creating object OF dBCONTEXTOPTION and use the connection string in the appsetting.json**

**Validate Inputs And Insert Data into Database:**



**[Required] attributes make this field is required into database and front end**

**And Error message this message will show if we put null into this field**



**[HttpPost] -> meaning this will be a Post request into database**

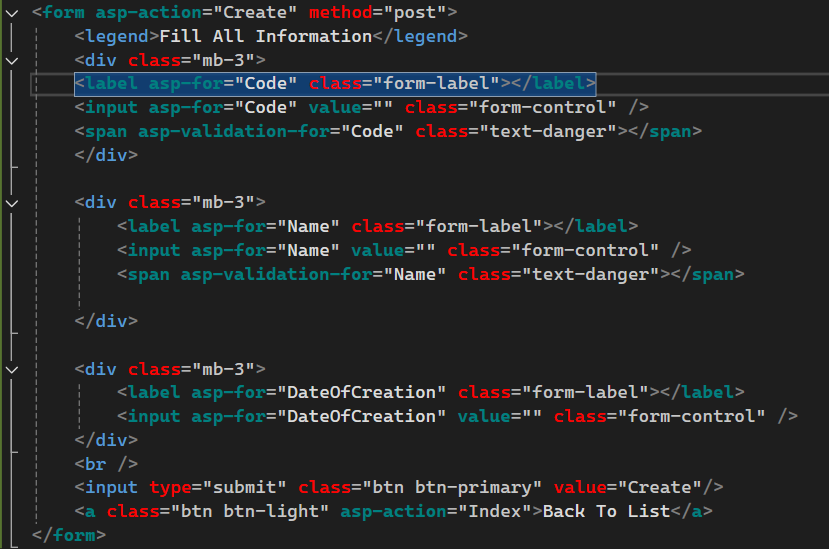
**Modelstate.IsValid -> this check if the model is empty or not and validate the Model in our database**

**Or our Model in DAL**

**If true then check if the number returned from database is larger than 0 that meaning it’s inserted sucssesfuly**

**So redirect the user to the index**

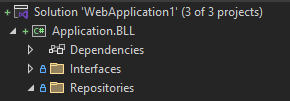
**Else make the request to the same Create page and show error message to him**



**Business Logic Layer:**

**This layer act as a kitchen it will cook all the meals.**

**In our case This layer act as a waiter take the Request from the controller in PL layer then go to talk to the Data acess layer then back with the information**



**Interface -> will hold the Defenetion or the Bluebrint of the Method in the repository**

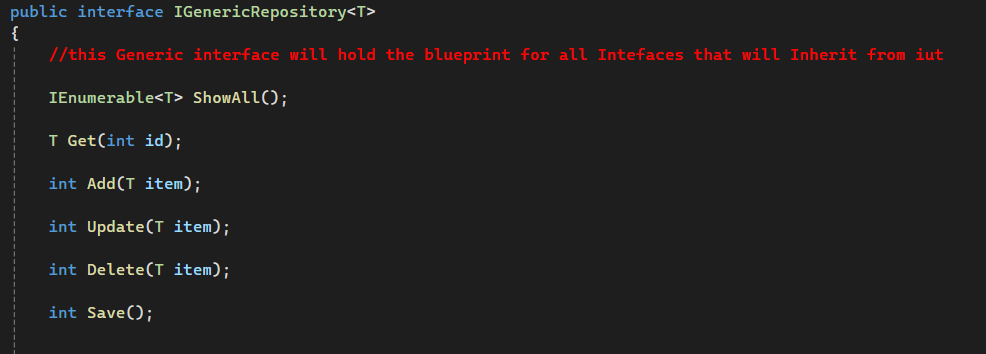
**Repository -> each model will has a repository (Waiter)**

**Asp.Net core MVC Session 04 Notes :**

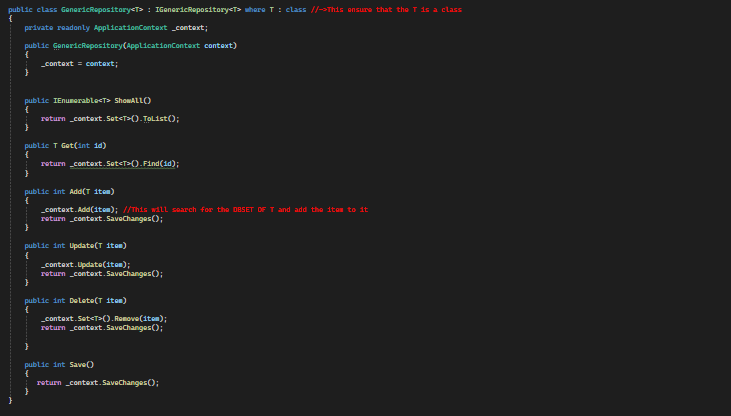
**Generic Repository**

**The Generic Repository is a design pattern that abstracts the application’s data layer, making it easier to manage data access logic across different data sources.**

* **Why Use a Generic Repository?** 
  + **Reduces Code Duplication: Common CRUD operations (Create, Read, Update, Delete) are written once in the generic repository and can be reused across all entities.**
  + **Centralizes Data Access Logic: All data access logic is managed in a centralized location, making the codebase easier to manage and understand.**
  + **Promotes DRY Principle: "Don't Repeat Yourself" by reducing redundancy in data access code.**
  + **Testability: Allows for easier unit testing by using dependency injection and mocking.**



**Implementing It**



* **A Partial View is essentially a view that renders a portion of HTML and can be reused across multiple views or layouts.**
* **They allow you to break down large views into smaller, reusable components that can be included in different views or layouts.**
* **Partial Views are similar to regular views, but they do not run on their own, they must be embedded within other views.**
* **Partial views are perfect for rendering small, reusable parts of the UI like navigation bars, forms, or lists that are used across different parts of an application.**

**MVC Session 05**

**viewData :**

* + **ViewData is a dictionary object of key-value pairs, That allows us to pass data from a controller action method to a view.**
  + **The ViewData is a property in the Controller class, and its Datatype is ViewDataDictionary.**
  + **The ViewDataDictionary class implements the IDictionary interface. So, ViewData in** [**ASP.NET**](http://ASP.NET) **Core MVC is a dictionary object.**
  + **ViewData stores data in the form of Key-Value Pairs, where each key must be a string, and the value that we pass to the dictionary will be stored as an object type ⇒ ViewData["KeyName"] = "Some Data";**
  + **Data stored in ViewData is available only for the current request. Once the view is rendered, ViewData is cleared.**

**ViewBag**

* **ViewBag is a dynamic property that provides a more convenient way to pass data from a controller to a view without the need to use a dictionary syntax.**
* **ViewBag is useful for passing simple data (e.g., strings, dates) that do not require a strongly typed view model.**
* **ViewBag is a dynamic object, which means you do not need to cast data when retrieving it. It uses C#'s dynamic keyword.**
* **We can store any value in ViewBag, but the type will be decided at run time rather than compile time.**
* **The data stored in ViewBag is available only during the current request lifecycle. Once the view is rendered and sent to the client, ViewBag data is no longer accessible.**

**TempData**

* **TempData is a dictionary object that is used to pass data from one action to another action within the same request or even across requests.**
* **It uses session storage to persist data across requests, ensuring data is available after a redirect.**
* **TempData is cleared after it is read, making it suitable for temporary data that doesn’t need to persist beyond a single request cycle.**
* **TempData is a property in the Controller class, and its Datatype is ITempDataDictionary.**

View Model (Mapping):

* **What is a View Model?**
  + A **ViewModel is a** class that contains properties representing the data that will displayed in the view.
  + It is specifically designed to serve the needs of the user interface and does not necessarily map directly to the domain model or database entities.
  + The primary purpose of a View Model is to encapsulate all the data that the view needs to render, making it easier to pass this data from the controller to the view.
  + It acts as an intermediary between the controller and the view, encapsulating the data displayed or edited on the view.

Why Use a View Model?

* + **Separation of Concerns:** ViewModels keep the logic and data that a view needs separate from the domain models, making it easier to manage and modify.
  + **Customization:** A ViewModel can combine data from multiple models, transform or format data, and only pass what the view requires.
  + **Simplification:** Using ViewModels reduces complexity, ensuring that the view only gets the data it needs and nothing more, enhancing security and performance.

**Step 1: Define the View Model:**

**Create a ViewModel Class in a ViewModels folder that includes properties relevant to the view’s needs.**

**ViewModel Class:**

**This class will hold the property that will be displayed in the view**

**What is AutoMapper?**

**AutoMapper is a simple library that helps us to transform one object type into another. It is a convention-based object-to-object mapper that requires very little configuration.**

**Unit of Work Pattern:**

* **The Unit of Work (UOW) is a design pattern in software development that helps manage changes to multiple entities and coordinates these changes so that they can be committed to the database as a single transaction.**
* **The Unit of Work Pattern groups one or more operations (usually database CRUD operations) into a single transaction and executes them by applying the principle of doing everything or nothing.**
* **The Unit of Work pattern is commonly used alongside the Repository pattern to decouple business logic from data access logic.**
* **It helps maintain a clean separation of concerns and ensures that all database operations are committed or rolled back in a transactional manner.**

**Sync vs Async :**

**Synchronous Programming**

**In synchronous programming, tasks are executed one after the other, sequentially. When a request is made to a synchronous API, the server processes the request and waits for it to be completed before moving on to the next task. This means that if one operation takes a long time to finish, it can block the execution of subsequent operations, potentially leading to slower response times for clients.**

**Asynchronous Programming**

**On the other hand, asynchronous programming allows tasks to be executed concurrently. When a request is made to an asynchronous API, the server can initiate tasks and continue processing other requests without waiting for the previous tasks to be completed. This can lead to better scalability and responsiveness, especially in scenarios where certain operations, such as I/O operations, may take some time.**

**url:**

[**https://www.c-sharpcorner.com/article/synchronous-vs-asynchronous-programming-in-asp-net-core-web-api/**](https://www.c-sharpcorner.com/article/synchronous-vs-asynchronous-programming-in-asp-net-core-web-api/)

**ASP.NET Core Identity:**

**https://dotnettutorials.net/lesson/asp-net-core-identity-tables/**