[Build ASP.NET Core Web API - Scratch To Finish (.NET 7 API)](https://cotiviti.udemy.com/course/build-rest-apis-with-aspnet-core-web-api-entity-framework/)

Middleware:

Middleware is a software component or a piece of code that sits between different parts of an application’s requests/response processing pipeline. Allows us to handle various tasks such as authentication, routing, logging ,caching and more.

REST: (Representational State Transfer):

Style of architecture for building web services

Set of principles.

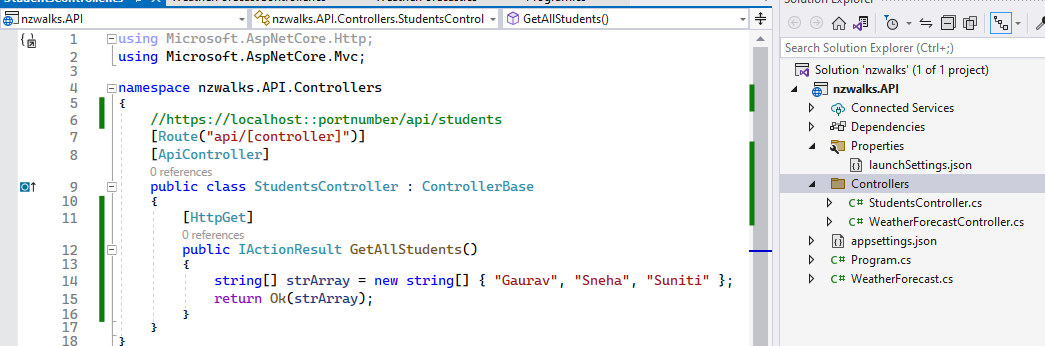
Everything acts as a resource URL and the server doesn’t store any state of the application in the server.

HTTP Verbs:

GET ,POST ,PUT ,DELETE

Whenever creating a new controller follow the format

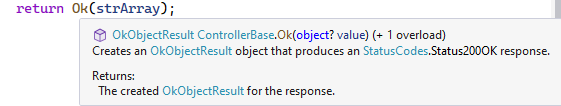
Name+Controller  
eg. StudentsController.cs



For example while creating a new controller

The [Route("api/[controller]")]

Takes the api name of Students



Navigation Property

Navigation properties in ASP.NET models are used to define relationships between entities in an application. They allow for easy navigation and traversal between related entities. Navigation properties represent associations between entities and can be used to access related data.

In ASP.NET, navigation properties are typically defined using object-oriented principles. For example, if you have two entities, such as Author and Book, and there is a one-to-many relationship between them (one author can have multiple books), you can define a navigation property in the Author entity to represent the collection of books associated with that author.

Here's an example of how navigation properties can be defined in ASP.NET models:

public class Author

{

public int Id { get; set; }

public string Name { get; set; }

public ICollection<Book> Books { get; set; }

}

public class Book

{

public int Id { get; set; }

public string Title { get; set; }

public Author Author { get; set; }

public int AuthorId { get; set; }

}

In this example **1**, the Author entity has a navigation property called Books, which represents the collection of books associated with that author. Similarly, the Book entity has a navigation property called Author, which represents the author associated with that book.

Navigation properties allow you to easily navigate between related entities. For example, you can access the books of an author using the Books property of an Author object, or you can access the author of a book using the Author property of a Book object.

By defining navigation properties in ASP.NET models, you can establish relationships between entities and easily access related data, making it convenient to work with complex data structures and perform operations involving related entities.

In the provided code, the Book object knows to link the AuthorId property to the Id field of the Author model because of the naming convention used.

By default, Entity Framework (EF) follows a convention-based approach for establishing relationships between entities. In this case, EF recognizes the AuthorId property in the Book model as the foreign key that references the Id field in the Author model. This convention is based on the naming pattern of <NavigationProperty><PrimaryKeyProperty>.

If you change the property name from AuthorId to AuthorUser\_Id, EF will not recognize it as the foreign key for the Author model's Id field. To establish the relationship correctly, you can use data annotations or Fluent API to explicitly define the foreign key property.

Here's an example of how you can explicitly define the foreign key property using data annotations:

public class Book

{

public int Id { get; set; }

public string Title { get; set; }

[ForeignKey("Author")]

public int AuthorUser\_Id { get; set; }

public Author Author { get; set; }

}

In this example **12**, the [ForeignKey("Author")] attribute is used to specify that the AuthorUser\_Id property is the foreign key for the Author navigation property.

Alternatively, you can use Fluent API in the DbContext class to define the relationship:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Book>()

.HasOne(b => b.Author)

.WithMany(a => a.Books)

.HasForeignKey(b => b.AuthorUser\_Id);

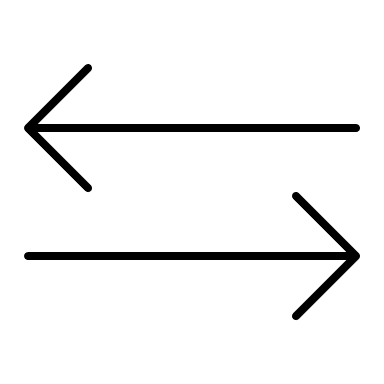
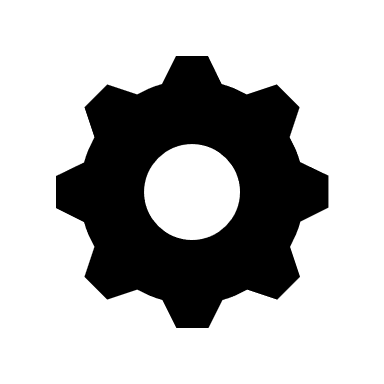
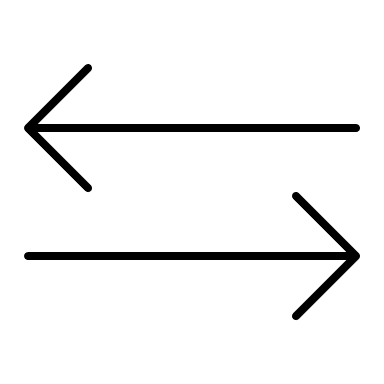
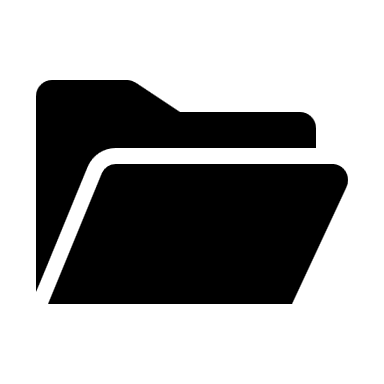
}

In this case, the HasOne, WithMany, and HasForeignKey methods are used to define the relationship between the Book and Author entities, specifying the AuthorUser\_Id property as the foreign key.

By explicitly defining the foreign key property, you can ensure that the relationship between the Book and Author models is correctly established, regardless of the property name.

DBContext Class

1. Maintain Connection to Database
2. Track Changes
3. Perform CRUD operations.
4. Bridge between domain models and the database



Controller DbContext Database

**Dependency Injection**

1. Design pattern to increase maintainability, testability
2. DI is built into ASP.NET Core
3. DI container is responsible for creating and managing instances

At its core, DI works on this fundamental that instead of instantiating objects within a class, those objects are passed in as parameters to the class, like passing it to the constructor or the method instead.

This allows for greater flexibility in the way objects are created and managed, as well as easier testing of individual components.

Good explanations here :  
[Build ASP.NET Core Web API - Scratch To Finish (.NET 7 API) | Udemy Business](https://cotiviti.udemy.com/course/build-rest-apis-with-aspnet-core-web-api-entity-framework/learn/lecture/36980064#reviews)

Next to insert the migration   
use commands

Add-Migration “Name Of Migration”

Update-Database

DTOs (DATA TRANSFER OBJECTS)

Client -> DTO -> API -> Domain Model -> Database

[HttpGet]

      public IActionResult Getall()

      {

          //Get Data from the database-Domain Model

          var result=\_dbcontext.Regions.ToList();

          //Map Domain Model to DTO's

          var regionsDTO = new List<regiondto>();

          foreach (var r in result)

          {

              regionsDTO.Add(new regiondto()

              {

                  id = r.id,

                  Code = r.Code,

                  Name = r.Name,

                  RegionImageUrl = r.RegionImageUrl

              });

          }

          //Return DTO's

          return Ok(regionsDTO);

      }

Httppost method along with DTO

[HttpPost]

       public IActionResult Create([FromBody] addregiondto AR)

       {

           //MAP or create DTO to DOMAIN MODEL

           var regionDomainModel = new Region

           {

               Code = AR.Code,

               Name = AR.Name,

               RegionImageUrl = AR.RegionImageUrl

           };

           \_dbcontext.Regions.Add(regionDomainModel);

           \_dbcontext.SaveChanges();

           //MAP domainModel back to dto

           var regionsDTO = new regiondto

           {

               id = regionDomainModel.id,

               Code = regionDomainModel.Code,

               Name = regionDomainModel.Name,

               RegionImageUrl = regionDomainModel.RegionImageUrl

           };

           return CreatedAtAction(nameof(GetById),new {id=regionsDTO.id},regionsDTO);

       }



Async Programming

The return type of every async method is a task

So we enclose it inside a Task<>

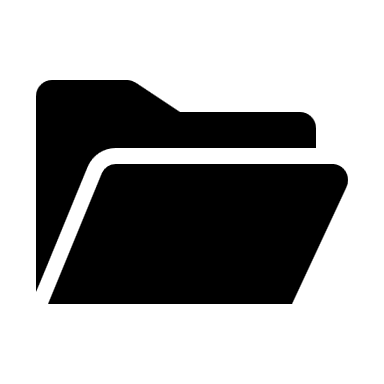
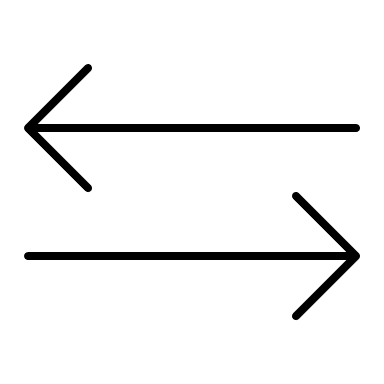
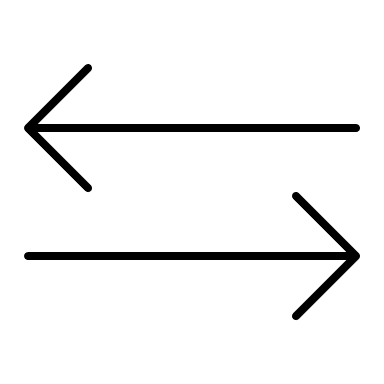
Async (Asynchronous with **await**):

* When you use asynchronous programming with the **await** keyword, the application doesn't block while waiting for an asynchronous operation. Instead, it can continue executing other tasks, such as handling user input, processing requests, or performing other work.
* The application can initiate an asynchronous operation, like a database query, and then immediately move on to other tasks without waiting for the database operation to finish.
* When the awaited operation (e.g., the database query) completes, the code after the **await** keyword is picked up and executed. In the meantime, other tasks could have been processed.

Remove Method doesn’t have an async method.

**Repository Pattern**

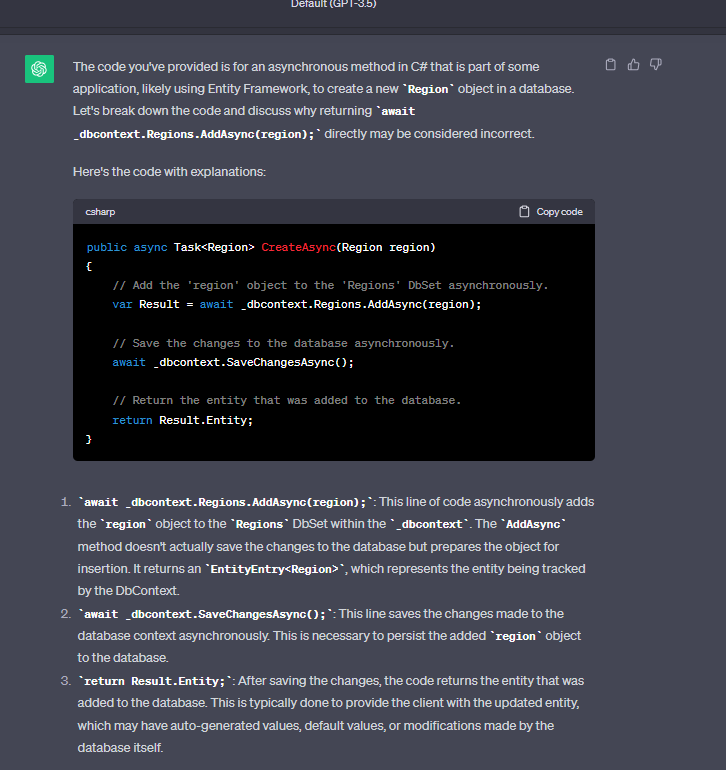
1. Design pattern to separate the data access layer from the application.
2. Provides interface without exposing implementation.
3. Helps create abstraction.



Controller -> Repository -> Database

DB Context is injected inside the Repository instead of the Controller

1. **Purpose**:
   * **Data Access Abstraction**: The primary purpose of a repository is to abstract and encapsulate the data access logic for your application. It provides a clean and consistent way to interact with the underlying data store, whether it's a database, a web service, or any other data source.
2. **Components**:
   * **Repository Interface**: Typically, a repository is defined by an interface that specifies a set of methods for performing common data operations, such as Create, Read, Update, and Delete (CRUD). These methods are often based on the data model of your application.
   * **Concrete Repository**: Implementations of the repository interface that interact with a specific data source (e.g., a database) are referred to as concrete repositories. These classes contain the actual data access code and are responsible for executing the database queries or web service requests.

While creating a POST Method api in repository  


Automapper

1. Object to object mapping
2. Simplification
3. Map between DTOs and Domain Models and Vice-versa
4. Quite powerful apart from just simple object to object mapping