C#

▼ Notas

▼ Solicitud HTTP

La solicitud se hace hacia el WebService y este consultara a la base de datos

Verbos

[HttpGet] → Obtener datos

[HttpPost] → Crear dato

[HttpPut] → Actualizar dato

[HttpPatch] → Actualizar un solo atributo de un dato

[HttpDelete] → Borrar dato

▼ Atributos HTTP

Los atributos HTTP se utilizan para definir que action method se va a invocar a partir de una request, estos son las *rutas* y los *verbos* asociados a esos metodos

Estos serian todos los verbos de CRUD, el [Route], y algunos no vistos como [HttpHead],[HttpOptions] y [AcceptVerbs(”GET”, “POST”]

Estos van en Models

▼ Entities

Las Entities representan la entidad de un dato y su estructura, pero no contienen ningun tipo de logica. su mayor utilidad es para llevar (mapear) una clase a la base de datos

▼ Repository Pattern

Los IRepositorios son interfaces que definen sin implementacion los metodos para acceder y/o modificar los datos de un modelo, como ICollection<”Modelo”> Get”Modelo”(); para recibir toda la lista de ese modelo

O “Modelo” Get”Modelo”(int id); para recibir uno en especifico por ID

Estos metodos se llaman operaciones CRUD, Crear, Leer, Actualizar, Eliminar

Luego debe crearse una clase que implemente esos metodos, ésta en Repository y que se llame igual pero sin la I

▼ Contexto

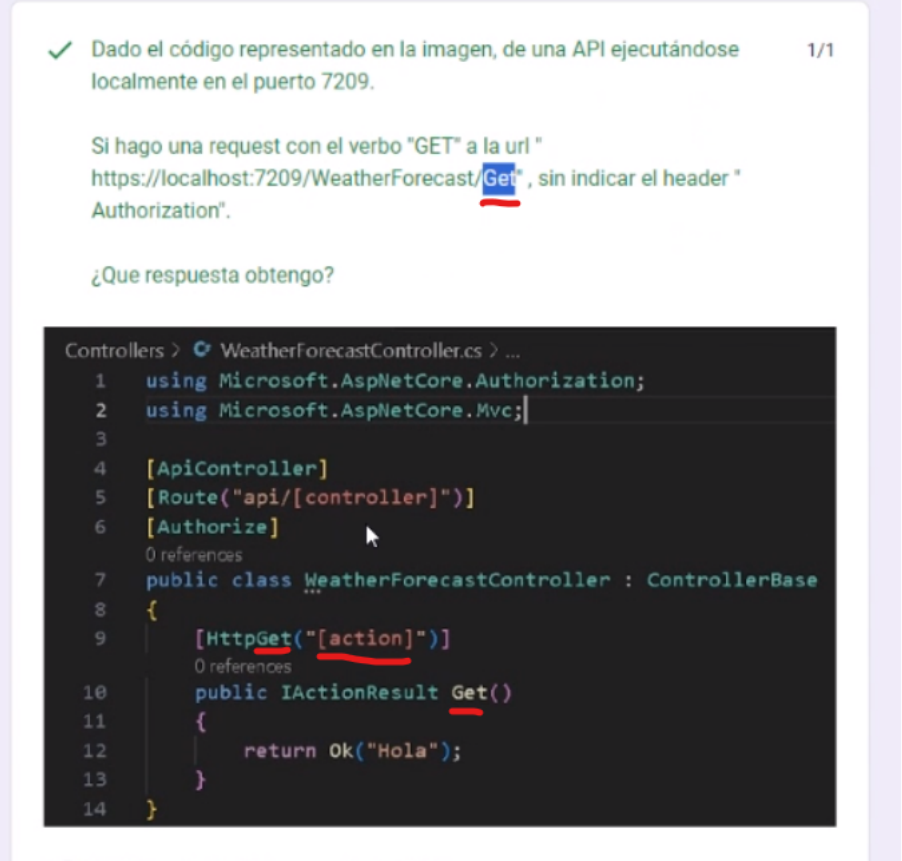
El Context es una clase que actua de intermediario entre la aplicacion y la base de datos (incluyendo su contenido), para hacer las migraciones y realizar operaciones CRUD, ahi se definen como atributos cada modelo del proyecto

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<Applicati

    }

    //Modelos aqui
    public DbSet<Modelo> Modelo { get; set; }
}
```

▼ Data Transfer Model

Un DTO es un objeto que solo contiene datos, sin logica, y que sirven mayoritariamente para ocultar atributos que un servicio no deberia ver

Los DTO tambien sirven para definir que datos queremos retornar cuando se hace una request a la API.



si pide id y doy uno no valido y no hay try catch es internal error

controller ruteo authorizacion y llamar metodo adecuado

implementar siempre patron dto? no, si no hay datos para ocultar no hace falta

clean architecture dependen de abstracciones

Los datos sensibles no pueden llegar por FromRoute porque son sensibles

El rol al asignarlo deberia tener validacion de ser admin

Chequear tipos de dato al revisar codigo

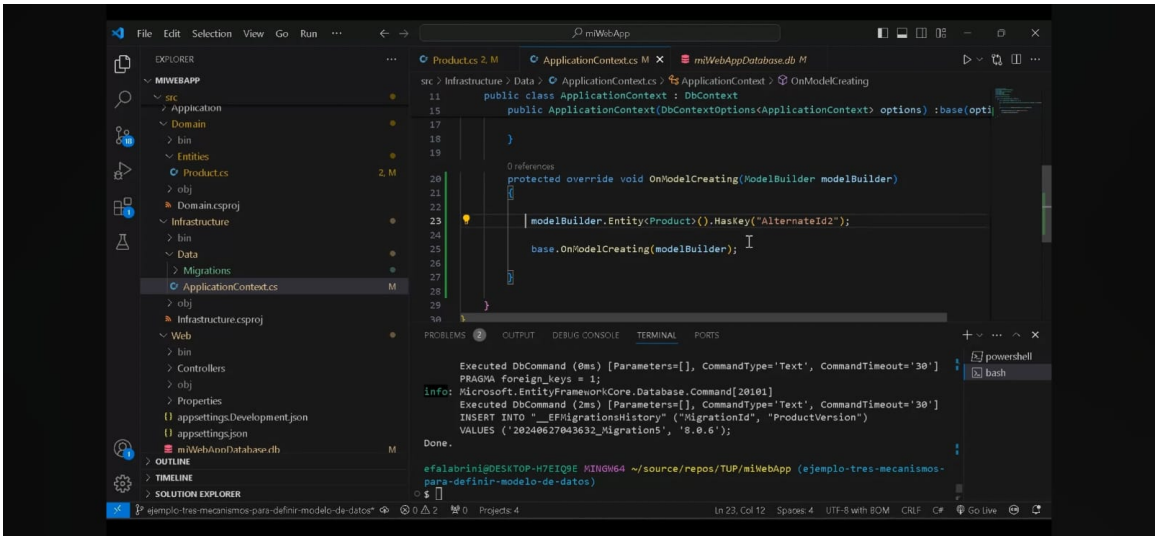
-H '' (Header)

autorize bearer token valido

Indique cuales de las siguientes opciones son correctas respecto al patrón de diseño Data Transfer Objects.

Las clases para transferencia de datos sirven para tomar datos que vienen informados en la request que se hacen a la API

Los clases para transferencia de datos sirven para definir que datos queremos retornar cuando se hace una request a la API.



[Key] y [Required] Son ejemplos de data notations

Id es el key por defecto
[Key] es para especificar una key
Y la imagen es para especificar una key a traves del modelbuilder

JWT - header-payload-signature

lo del medio se llama payload, y es donde están las claims, que son los pares clave-valor con datos del usuario

▼ Follow up

▼ Start

- Borrar WeatherReport.cs y WeatherReport controller
- Crear carpetas en el proyecto: Data,Models/Dtos,"Proyecto"Mapper,Repository/IRepository(interfaces)
- Crear Archivos en el proyecto: Models/Modelo.cs , Data/ApplicationDbContext.cs

▼ Models

Un modelo es una tabla dentro de la base de datos, osea crea una nueva columna y cada dato sera una nueva linea

▼ [Key]

Dentro de un model, convencion al asignar una propiedad id a un nuevo modelo, indica que la propiedad de abajo es el identificador de la linea en la base de datos.

▼ [Required]

Dentro de un model, Propiedad obligatoria

▼ Extensiones

Estas deben instalarse en cada projecto, Tools → NuGet Package Manager → Manage NuGet Packages for Solution

- EntityFrameworkCore.SqlServer
- EntityFrameworkCore.Tools

▼ ApplicationDbContext

Clase necesaria para migraciones y demas operaciones

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<Applicati

    }
```

```
    //Modelos aqui
    public DbSet<Modelo> Modelo { get; set; }
}
```

▼ Conexion a Base de datos SQL

```
en appsettings.json

"ConnectionStrings": {

  "SqlConnection": "NombreServidorSql;Database=NombreDB;Tr

}
```

```
en Program.cs

builder.Services.AddDbContext<ApplicationDbContext>(opcior
{
opciones.UseSqlServer(builder.Configuration.GetConnectionS
});
```

Tools → NuGet Package Manager → Package Manager Console

add-migration CreacionTabla”Modelo”

update-database //push a la base de datos

▼ Interfaz de Repository Pattern

Repository/IRepository/I”Modelo”Repository.cs

En esta *Interfaz* se crean los metodos para acceder y/o modificar los datos de un modelo, como ICollection<”Modelo”> Get”Modelo”(); para recibir toda la lista de esa clase

O “Modelo” Get”Modelo”(int id); para recibir uno en especifico por ID

Estos metodos se llaman operaciones CRUD, Crear, Leer, Actualizar, Eliminar

El IRepository es como una clase abstracta, los metodos se implementan en el Repository normal

▼ Implementando la Interfaz

Luego se crea una clase que implemente esos metodos CRUD, por ejemplo, Repository/”Modelo”Repository.cs

```
public class "Modelo"Repository : I"Modelo"Repository
{
    private readonly ApplicationDbContext _db; // Se i

    // Se usa los atajos del visual para implementar t

}
```

▼ Ejemplo Peliculas

1. Models/Category.cs
2. /*SQL appsettings.json
3. Extensiones Entity SQL y Entity TOOLS
4. Data/ApplicationDbContext.cs
5. Program.cs
6. add-migration CreacionTablaModelo
7. update-database