Clase 2 - @September 1, 2022

- Pasar excel para que pongan sus cuentas de Github: Link
- Creamos grupo de teams Link
- Poner fecha para el día de chequeo de avance del obligatorio
- Comentar lo de .NET 3.1

- Entrar más en detalle en los conceptos vistos en primer clase
- Modelos
- Ejemplo integrador
- Práctico

Repaso clase pasada

En la primer clase:

- Presentamos los temas que vamos a dar en el curso
- Dimos un panorama general de .NET Core y la comparación con .NET Framework
  - **Multiplataforma**
  - **Coherente entre arquitecturas**
  - **Herramientas de línea de comandos**
  - **Implementación flexible**
  - **Compatible**
  - **Código Abierto**
- Comenzamos con el proyecto de WebApi
  - API la cual se encargara de almacenar pedidos de un restaurante de tacos
  - Vimos Algunos comandos básicos de dotnet
    - *dotnet new sln -n Tacos*
    - *dotnet new webapi -au none -n Tacos.WebApi*
    - *dotnet sln add Tacos.WebApi*
    - dotnet run -p Tacos.WebApi
  - Creamos el **OrderController**
  - Algunos verbos HTTP, como se especificaban en el controlador y diferentes combinaciones para lograr diferentes cosas en el endpoint
    - **[HttpGet("{id}")] → *api/orders/5***
    - **[HttpGet("vegan")] → *api/orders/vegan***
  - Attributes: estos especifican la ubicación en donde se encuentra el valor del parámetro en la request
    - **[FromBody] → ejemplo: voy a crear una orden y necesito especificar su información**

        - **[FromHeader] → ejemplo: content-type**
        - **[FromQuery] → ejemplo: especificar un filtro day api/orders?day='15-08-2022'**
        - **[FromRoute] → ejemplo: api/orders/2**
- Vimos un ejemplo de DataAccess
    - Creamos solución de consola
    - Creamos solución de tipo classlib
    - Instalamos algunos paquetes
        - dotnet add package Microsoft.EntityFrameworkCore → Para relaciones
        - Microsoft.EntityFrameworkCore.SqlServer → Para interacción con motor de base de datos
        - dotnet add package Microsoft.EntityFrameworkCore.Design → Para migrations
        - Y realizamos algunas migraciones

Manejo de errores

El manejo de errores es un aspecto critico de una buena API. Es muy importante poder explicarle al usuario de la API porque una request fallo, y brindarle toda la información necesaria para que pueda solucionarlo.

Usar HTTP Codes

Es importante usar los HTTP codes en las situaciones adecuadas para seguir un standard. Existen sobre 70 codigos, aunque solo un subconjunto es utilizado comunmente.

Cuántos usar?

Si se analiza los posibles flujos que pueden haber cuando se ejecuta un endpoint, hay 3 principales:

- Todo anduvo correctamente - Éxito
- El usuario hizo algo mal - Error del cliente
- La API hizo algo mal - Error de la API

Cada uno de estos se puede representar con los 3 siguientes codigos:

- **200** - OK
- **400** - Bad Request
- **500** - Server Error

A partir de esto, se pueden agregar los que se consideren necesarios. **201 - Created** es un código muy utilizado cuando se crea un elemento de un recurso. **401 - Unauthorized** también es muy utilizado, cuando el usuario no tiene permisos para realizar esa operación.

Retornar mensajes lo mas expresivos posibles

Mientras más expresivo y más información se le brinde al usuario, mas fácil sera de usar la API.

Siempre sera peor tener:

```
{"code" : 401, "message": "Authentication Required"}
```

que:

```
{
    "developerMessage" : "Verbose, plain language description of the problem for the app developer with hints about how to fix it.",
    "userMessage": "Pass this message on to the app user if needed.",
    "errorCode" : 12345,
    "moreInfo": "http://dev.teachdogrest.com/errors/12345"
}
```

En el segundo ejemplo, se brinda información descriptiva, se sabe donde ir a buscar mas información sobre el error, y se brinda un mensaje que se le puede mostrar a un usuario.

Modelos o DTOs

Estuvimos viendo ejemplos básicos de parámetros que se reciben en el body y seguramente ahora se están preguntando como es que se agarran valores más complejos de la request con los tipos en los parámetros. La respuesta a su pregunta es gracias al mecanismo de **Model Binding** que nos provee **ASP.NET Core.**

Model Binding es la automatización del proceso de convertir el string que escriben los clientes que son las requests a los tipos de **.NET** correspondientes.

Lo que hace es:

- Saca la información de varios lugares, del body, header, route, query strings.
- Esta información se la da a los controladores en forma de parámetros en métodos o properties publicas.

Que tanto exponer?

Veamos una buena practica de las APIs y relacionado también a la seguridad.
Imaginense un sistema que maneja login y creación de cuentas, al crear una cuenta recibe un **UserCreationModel** que tiene usuario y contraseña. Probablemente en la base de datos se guarde el nombre de usuario, contraseña de alguna forma encriptada entre otros datos, ahora tengamos un endpoint que retorna un **UserModel** con esta info el cual no debería tener la contraseña más allá que este encriptada por lo cual es útil manejar distintos modelos según lo que queramos exponer y esconder.

Las APIs en producción por lo general limitan la data que reciben y devuelven utilizando un modelo que tiene un sub-set de la entidad. Nos vamos a referir a esta técnica como **Data Transfer Object (DTO)**, **input model**, o **view model**.

Estos modelos son usados para:

- Prevenir mostrar de mas
- Esconder properties a clientes que no se suponen que vean
- Omitir algunas properties para reducir tamaño
- Minimizar las relaciones entre objetos.
- Minimizar el impacto de cambio a los clientes

Cuál es el objetivo de los modelos?

Como antes habíamos visto que la **api** era la implementación del **patron Fachada**, la realización de **modelos** o **dto** es la implementación del **patron Adapter.** De esta forma estamos **envolviendo** las entidades para que los diferentes clientes no se vean afectados si las entidades cambian su estado.

Ejemplo:

1. Creamos el OrderIntentModel y vamos a ver que lo definimos llega correctamente

```
using System.Collections.Generic;
using System.Linq;
using System;

namespace Tacos.WebApi.Controllers.In;

public class OrderIntentModel
{
    public List<string> Plates { get; set; }

    public List<string> Drinks { get; set; }

    public int Table { get; set; }

}
```

2. Creamos el OrderBasicInfoModel

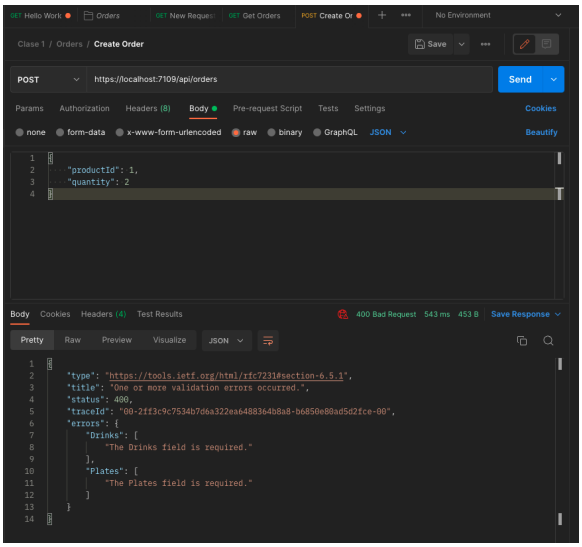
```
using System.Collections.Generic;
using System.Linq;
using System;
```

```
namespace Tacos.WebApi.Controllers.Out;

public class OrderBasicInfoModel
{

    public int Table { get; set; }

    public int Id { get; set; }

}
```

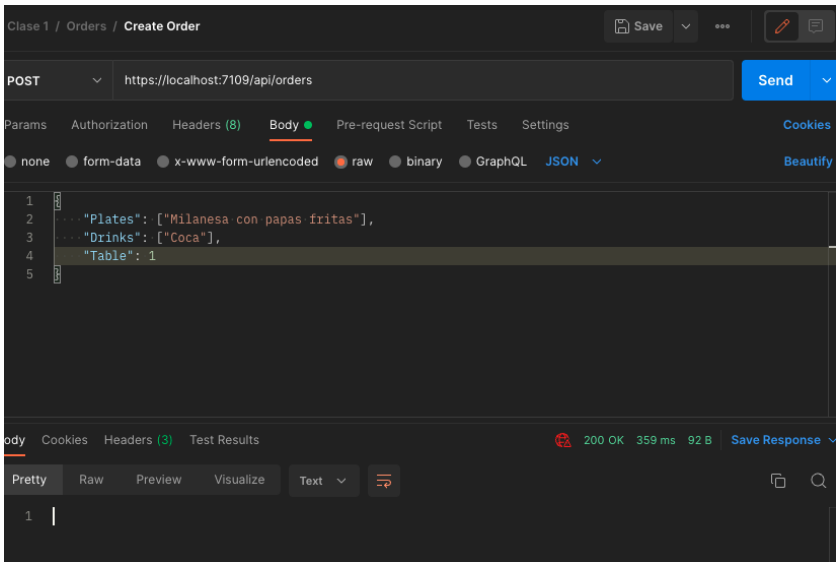
3. Utilizamos en la WebApi estos modelos

3. Ejecutamos la api y probamos enviando los campos incorrectos a api/orders

```
dotnet run -p Tacos.WebApi
```

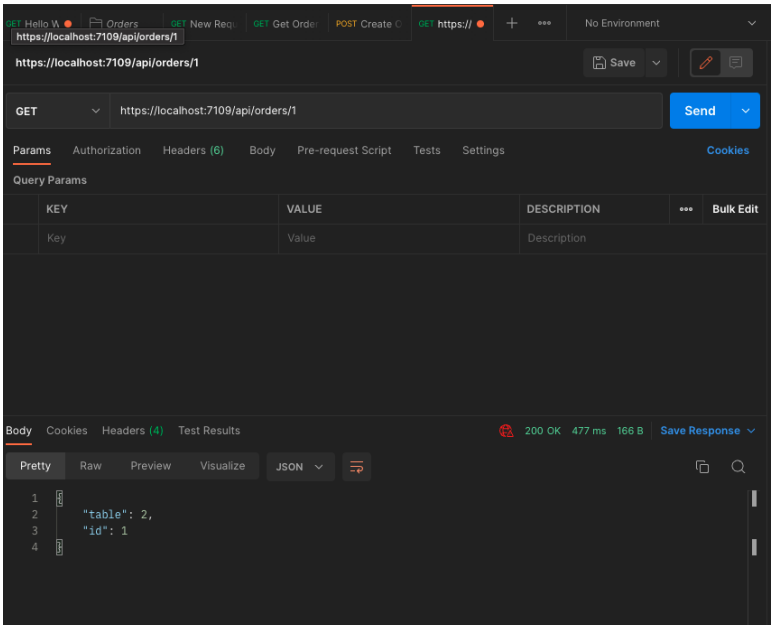


Si mandamos los datos que espera la api si funciona bien:



4. Ahora probamos retornando un OrderBasicInfoModel

```
// api/orders/2
[HttpGet("{id}")]
0 references
public IActionResult Get([FromRoute] int id){
    return Ok(new OrderBasicInfoModel()
        {
            Id = 1,
            Table = 2
        }
    );
}
```



Ejemplo Integrador

Previo: Tener levantado y corriendo SQL

1- Vamos a ver una solución completa con todas las capas para los que vamos a crear los proyectos:

- Domain → Class Library
- BusinessLogic → Class Library
- IBusinessLogic → Class Library (opcional)
- DataAccess → Class Library
- IDataAccess → Class Library (opcional)
- WebAPI → ASP.NET Core Web Application type web API

2- En el proyecto Domain comenzamos creando la clase Order que es el dominio que nuestro sistema va a manejar

```
public class Order
{
    public int Id { get; set; }
    public string Name { get; set; }
```

```
    public string Address { get; set;}
    public int PurchaseNumber { get; set;}
    public int Price { get; set;}
    public DateTime DeliveryDateTime { get; set; }
}
```

3- Creamos la clase contextDB dentro del paquete DataAccess

Para esto antes instalamos los paquetes:

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.SqlServer

```
public class ContextDb: DbContext
    {
        public DbSet<Order> Orders {get; set;}

        protected override void OnModelCreating (ModelBuilder modelBuilder)
        {

        }
        public ContextDb(DbContextOptions options) : base(options) {}

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
        }

    }
```

4- En el proyecto IDataAccess creamos la clase IOrderRepository que se encargara de explicitar los metodos que se van a usar para interactuar con el context que contiene los dbset de las representaciones de las tablas.

Nosotros ahora solo usaremos dos de estas funciones el add y get pero dejamos todas para que ya les quede de ejemplo la firma.

```
public interface IOrderRepository {

    void Add (Order entity);
    void Remove (Order entity);
    void Update(Order entity);
    IQueryable<Order> GetAll();
}
```

5- En DataAccess creamos la implementación de la interfaz anterior que es la que se encarga de interactuar con el contextDb (nuestra puerta a la bdd) para hacer las operaciones CRUD sobre este.

```
public class OrderRepository: IOrderRepository
{
    private ContextDb _contextDb;

    public OrderRepository(ContextDb contextDb)
    {
        _contextDb = contextDb;
    }

    public void Add(Order entity)
    {
        _contextDb.Orders.Add(entity);
        _contextDb.SaveChanges();
    }

    public void Remove(Order entity)
    {
```

```
        throw new NotImplementedException();
    }

    public void Update(Order entity)
    {
        throw new NotImplementedException();
    }

    public IQueryable<Order> GetAll()
    {
       return _contextDb.Orders;

    }
}
```

6- Creamos ahora en IBusinessLogic la initerface IOrderService

```
public interface IOrderService
{
    Order Get (int id);
    Order Create (Order order);
    IQueryable<Order> GetAll();
    void Delete(int id);
    void Update(int id, Order entity) ;
}
```

7- Creamos ahora en buisnessLogic la implementación de la clase anterior que sera OrderService

```
public class OrderService : IOrderService
{
    private IOrderRepository repository;
    public OrderService(IOrderRepository orderRepository)
    {
        repository = orderRepository;
    }

    public Order Get(int id)
    {
        throw new NotImplementedException();
    }

    public Order Create(Order order)
    {
         repository.Add(order);
         return order;
    }

    public IQueryable<Order> GetAll()
    {
        return repository.GetAll();

    }

    public void Delete(int id)
    {
        throw new NotImplementedException();
    }

    public void Update(int id, Order entity)
    {
        throw new NotImplementedException();
    }
}
```

8- Antes de comenzar a hacer el controller debemos ahcer algunas modificaciones en las clases Program, agregar la clase Startup y crear el connection string

```
"ConnectionStrings": {
    "DBClase2": "Server=.\\SQLSERVER;Server=localhost,1433;Database=TacosBDD;Trusted_Connection=true;MultipleActiveResultSets=True;User=sa;
  },
```

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<Startup>(); });
}
```

```
public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors(options =>
            {
                options.AddPolicy("AllowEverything",builder => builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod());
            });
            services.AddControllers();
            services.AddScoped<IOrderRepository, OrderRepository>();
            services.AddScoped<IOrderService, OrderService>();
            string directory = System.IO.Directory.GetCurrentDirectory();
            IConfigurationRoot configuration = new ConfigurationBuilder()
                .SetBasePath(directory)
                .AddJsonFile("appsettings.json")
                .Build();
            var connectionString = configuration.GetConnectionString("DBClase2");
            services.AddDbContext<ContextDb>(options =>
                options.UseSqlServer(connectionString));

        }


        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
        }
    }
```

9- Finalmente crearemos el controller

```
[ApiController]
[Route("api/Orders")]
public class OrderController: ControllerBase {

    private IOrderService _orderService;

    public OrderController(IOrderService orderService){
        _orderService = orderService;
    }

    [HttpGet]
    public IActionResult Get()
    {
        IQueryable<Order> orders = _orderService.GetAll();
        return Ok (orders);
    }


   [HttpPost]
   public IActionResult Add([FromBody] Order order)
   {
       Order orderCreated = _orderService.Create(order);
       return Created("", orderCreated);
   }


}
```

9- Instalamos el paquete EntityFrameworkCore.Design en el paquete DataAccess y en WebAPI para poder hacer las migrations, para ejecutar estos comandos nos paramos en DataAccess y referenciamos a webAPI que es donde iniciamos nuestra aplicacion y construimos el contextDB.

```
dotnet ef migrations add MyMigration --startup-project "../WebAPI"
dotnet ef database update --startup-project "../WebAPI"
```

10- Corremos la solucion y probamos los endpoints.

```
https://localhost:7142/api/Orders
```

11- Vamos a crear un modelo de order

```
public class OrderModel
{
    public string Name { get; set; }
    public string Address { get; set;}
    public string Plate { get; set; }
}
```

Ahora el metodo post queda

```
[HttpPost]
 public IActionResult Add([FromBody] OrderModel orderModel)
 {
     Order order = new Order()
     {
         DeliveryDateTime = DateTime.Now,
```

```
        Address = orderModel.Address,
        Name = orderModel.Name,
        Plate = orderModel.Plate,
        Price = 1232,
        PurchaseNumber = 123667
    };
    Order orderCreated = _orderService.Create(order);
    return Created("", orderCreated);
}
```