Clase 1 DA2 Tecnología @August 18, 2022

Profes:

Joselen Cecilia - joselencr@gmail.com

Marcos Fiorito - marcofiorito1@gmail.com

Introducción del curso

https://docs.google.com/presentation/d/1IeZZBRyDarKzpyRPCRfftqbM3nDU_u0PAqx4RzVDHaQ/edit#slide=id.g117df78bb49_(

Durante el transcurso del curso se construirá una aplicación completa vista desde la perspectiva de sus componentes, es decir, se va desarrollar un **back-end** con funcionalidad y base de datos, y también, se va generar una **SPA** (Single Page Application) para que el usuario pueda utilizar las funcionalidades provistas por el servidor.

Vamos a dividir esta construcción en dos partes:

- **La primera** es la construcción de la API (Application Programming Interface) REST la cual se creará utilizando .Net Core y WebApi. Cabe hacer énfasis que en esta parte no vamos a tener una GUI de usuario, sino que haremos el servicio y lo probaremos utilizando una aplicación para ello llamada Postman.
- **La segunda** es la construcción de una SPA (aplicación web) con la que el usuario podrá utilizar el sistema. La misma se realizará en Angular.

¿Qué tecnologías y herramientas vamos a usar para esta construcción?

- .Net CORE 6(SDK contiene al runtime)
- Visual Studio Code / Rider / Visual Studio
- Extensiones para un mejor uso del IDE Visual Studio Code
  - **C#**
  - **.NET Core Tools** https://marketplace.visualstudio.com/items?itemName=formulahendry.dotnet
  - **Auto-Using for C#** https://marketplace.visualstudio.com/items?itemName=Fudge.auto-using
  - C# Extensions https://marketplace.visualstudio.com/items?itemName=jchannon.csharpextensions
- SQL Server Express
- Angular y Node js
- Postman
- Es imprescindible tener GIT instalado en el equipo ya que es el repositorio que vamos a utilizar.
  - Se puede usar por línea de comandos o con una aplicación con GUI (recomendación **GitKraken**)

.NET Core

Cómo pueden recordar, en materias previas se utilizo **.NET Framework**.

Este permitía crear aplicaciones para Windows, Web, entre otros. Sin embargo, solo se podía correr en Windows, y al ser *closed-source*, era necesario el pago de licencias, depender de Microsoft para muchas cosas. También contaba con varios otros problemas que lo hacen una herramienta que a veces puede resultar poco atractiva o no la mejor en muchas situaciones.

Microsoft decidió cambiar el rumbo y "rehacer" .NET de una manera más ***developer-friendly***, adaptándose a las nuevas tecnologías y ambientes. Para esto, creo **.NET Core**. Cabe aclarar que no es que .NET framework dejo de existir, si no que existen en paralelo.

.NET Core es una plataforma de desarrollo de uso general de código abierto de cuyo mantenimiento se encargan Microsoft y la comunidad .NET en GitHub. Es multiplataforma, admite Windows, macOS y Linux y puede usarse para compilar aplicaciones de dispositivo, nube e IoT

Características de .NET Core en comparación con .NET Framework

- **Multiplataforma:** se ejecuta en los sistemas operativos Windows, macOS y Linux.
- **Coherente entre arquitecturas:** el código se ejecuta con el mismo comportamiento en varias arquitecturas, como x64, x86 y ARM.
- **Herramientas de línea de comandos:** incluye herramientas de línea de comandos sencillas que se pueden usar para el desarrollo local y en escenarios de integración continua. Esto es una ventaja **critica** en comparación con `.NET framework`.
- **Implementación flexible:** se puede incluir en la aplicación o se puede instalar de forma paralela a nivel de usuario o de equipo. Se puede usar con contenedores de Docker facilmente.
- **Compatible:** .NET Core es compatible con .NET Framework, Xamarin y Mono, mediante `.NET Standard`, el cual es un conjunto de APIs estables en común entre todos. Esto favorece enormemente él rehuso de código.
- **Código abierto:** la plataforma .NET Core es de código abierto, con licencias de MIT y Apache 2. .NET Core es un proyecto de .NET Foundation.
- **Compatible con Microsoft, entre otros:** .NET Core incluye compatibilidad con Microsoft y con muchísimas otras plataformas.

Para comenzar la clase vamos a:

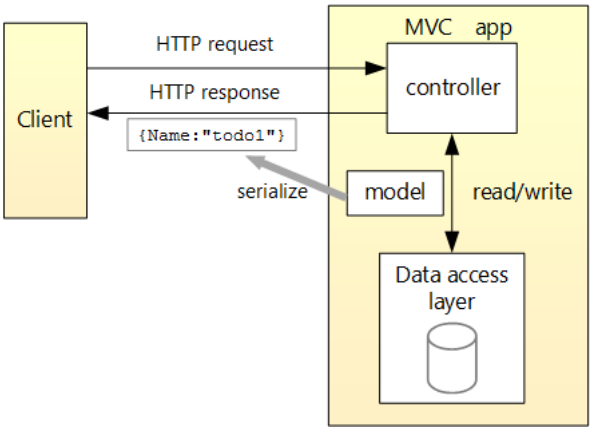
- Crear una proyecto WebApi
- Crear un controller con métodos CRUD
- Configurar endpoints y valores de retorno

Para realizar todo lo mencionado anteriormente, vamos a crear una API tacos la cual se encargara de almacenar pedidos de un restaurante de tacos. Para ello vamos a diseñar primeramente los siguientes endpoints:

Endpoints

API	Description	Body	Response Body
GET /orders	Retorna todas las ordenes existentes	-	Lista con todas las ordenes
GET /orders/{id}	Retorna una orden por el id	-	La orden con el id {id}
GET /orders?{day}	Retorna todas las ordenes de un día dado	-	Lista con todas las ordenes
POST /orders	Crea una orden	Orden	Orden con id
PUT /orders/{id}	Actualiza una orden existente	Orden	-
DELETE /orders/{id}	Elimina una orden	-	-

A continuación se muestra un diagrama que muestra la arquitectura de la app:



Pasos para crear el proyecto que vimos de ejemplo en clase

Para crear el proyecto se puede hacer tanto con VS code o VS directamente, en nuestro caso lo vamos a hacer con VS code para mostrarles los distintos comandos.

Para probar la **api** en **Visual Studio Code** se tiene que correr el comando ***dotnet run*** eso abrirá un puerto por defecto al cual le deberíamos de mandar las request y para **Visual Studio** basta con apretar **F5** o el **botón de play.**

1. Creamos la solución

```
dotnet new sln -n Tacos
```

2. Creamos un proyecto de tipo WebAPI

```
dotnet new webapi -au none -n Tacos.WebApi
```

3. Agregamos el proyecto a la solución

```
dotnet sln add Tacos.WebApi
```

Veremos que crea el siguiente controlador por defecto el cual ya podemos correr.

```
using Microsoft.AspNetCore.Mvc;

namespace Tacos.WebApi.Controllers;

[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };

    private readonly ILogger<WeatherForecastController> _logger;

    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
```

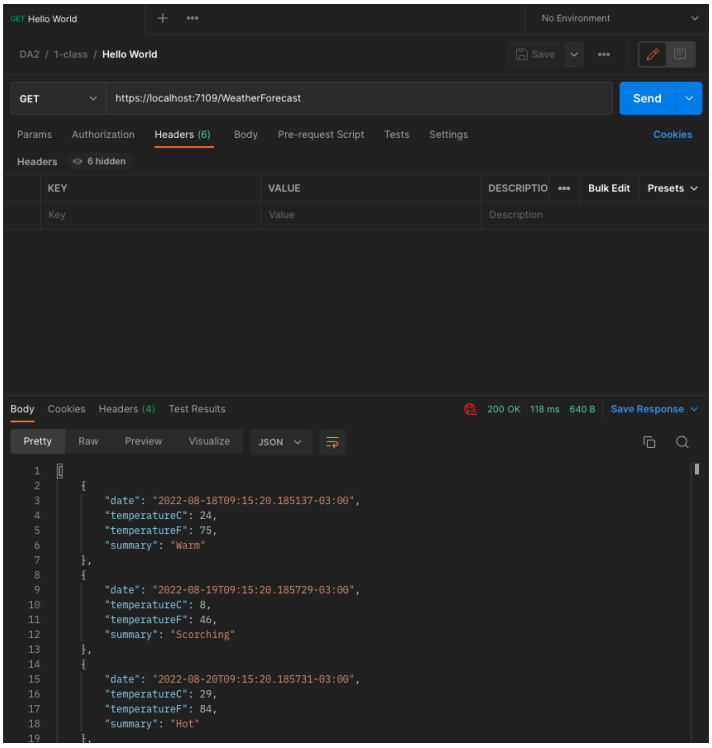
```
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

4. Corremos el proyecto

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

5. Vemos en postman el endpoint creado por defecto

```
https://localhost:7109/WeatherForecast
```



Vamos a recorrer la solución y a analizar ciertos puntos de la misma

Program.cs

Es el punto de entrada de nuestra aplicación donde configuramos el Web host. Esta clase configura aspectos de la infraestructura de la aplicación tales como Web host, Logging, Contenedor de Inyecciones de Dependencias, Integración con IIS, etc. La aplicación es creada, configurada y construida usando el método `CreateBuilder` en la clase Program.

Esta clase es similar a la clase Program de una aplicación de consola, eso es porque todas las aplicaciones .NET Core son básicamente aplicaciones de consola.

El principal objetivo de esta clase es configurar la infraestructura de la aplicación.

Al inicio, esta clase crea un Web Host, una vez creado, se le configura un montón de elementos como los que ya se mencionaron.

Web Host

El web host es el responsable de inicializar la aplicación y correrla. Los web hosts crean un servidor, el cual escucha request HTTP en un determinado puerto libre de la maquina. Es el encargado de configurar la request pipeline. También es el encargado de setear el contenedor de servicios donde nosotros podemos agregar nuestros propios servicios a ser usados, eso también incluye controlar el ciclo de vida de los servicios según el tipo de ciclo de vida seteado por nosotros.

En resumen, un Web Host, deja lista nuestra aplicación para recibir request, pero, este Web Host tiene que ser creado y configurado en la clase Program.

El WebServer usado por default en una api .NET Core es Kestrel. Kestrel es un HTTP Server open-source y cross-platform. Fue desarrollada para hostear aplicaciones .NET Core en cualquier plataforma.

Creamos un controlador

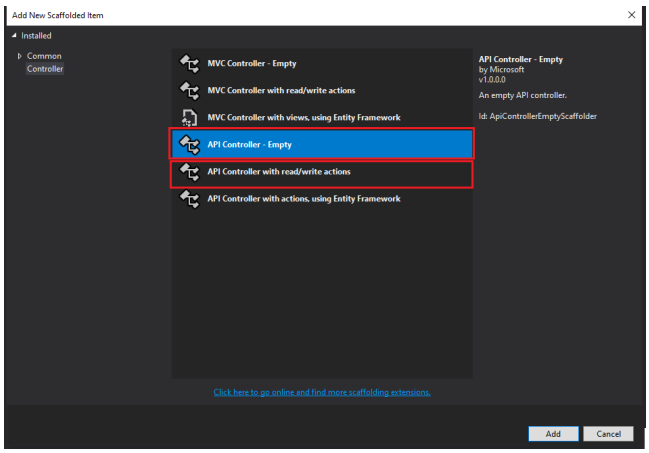
Un controlador es una clase que hereda de **ControllerBase**. Una **WebApi** contiene uno o varios controladores con sus respectivos endpoints.

ControllerBase provee muchas properties y métodos que son útiles para manejar requests HTTP. Por ejemplo **CreatedAction** retorna el código **201**, **BadRequest** retorna **400**, **NotFound** retorna **404**, entre otros.

Para mas información sobre los diferentes métodos y properties vean [ControllerBase Class]

Entonces, para crear nuestro primer controlador hacemos click derecho en la carpeta **controllers** que podemos encontrar en el proyecto **WebApi** y hacemos lo siguiente:

- Para **Visual Studio Code:**
  - Seleccionamos la opción **New File**
  - Escribimos el nombre **OrderController.cs.** La extension es necesaria.
- Para **Visual Studio:**
  - Seleccionamos la opción **Add → Controller** y nos mostrara la siguiente ventana



- Podemos seleccionar la opción **API Controller - Empty** la cual nos va a crear una clase controller vacía o la opción **API Controller with read/write actions** la cual nos va a crear una clase controller con las operaciones **CRUD.**
- Una vez seleccionada la opción se le pone el nombre **OrderController.cs,** acá no es necesario especificar la extension.

El controlador nos quedaría de la siguiente manera

```
using Microsoft.AspNetCore.Mvc;

namespace Tacos.WebApi.Controllers;

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

}
```

Attributes

Los attributes pueden ser usados para configurar el comportamiento de los controllers de la web API.

[ApiController] attribute

Este attribute se especifica en la clase controlador para permitir los siguientes comportamientos:

- Obligatorio el uso del attribute [Route]
- Respuestas HTTP 400 automáticas
- El uso de binding de parámetros
- Detalle de problemas para códigos de error

Attributes para endpoints

Los siguientes attributes serán usados para especificar que verbos HTTP nuestro controlador soporta:

Verbos HTTP

Aa Verbo	Attribute	Property
GET	[HttpGet] o [HttpGet(string template)]	Identifica una acción que soporta el verbo GET de HTTP
POST	[HttpPost] o [HttpPost(string template)]	Identifica una acción que soporta el verbo POST de HTTP
PUT	[HttpPut] o [HttpPut(string template)]	Identifica una acción que soporta el verbo PUT de HTTP
DELETE	[HttpDelete] o [HttpDelete(string template)]	Identifica una acción que soporta el verbo DELETE de HTTP

Como podemos observar en la tabla de arriba cada verbo tiene asignado dos attributes, uno que no recibe nada y otro que recibe por parámetro un string.

Ese string que le pasamos servirá para pasar valores por la uri (**[HttpGet("{id}")]** → ***api/orders/5***) o para crear un nivel mas sobre la ruta (**[HttpGet("vegan")]** → ***api/orders/premiere***), claramente hay una diferencia entre estos dos parámetros.

Tambien podemos encontrar algunas properties como por ejemplo **Name, Order, Template** y **HttpMethods;** estas properties son accedidas por cualquier verbo HTTP porque todos heredan de **HTTPMethodAttribute.**

La property **Name** la podemos usar para hacer re-direccionamiento de un método a otro a través del nombre, se usaría de la siguiente manera: ***[HttpGet("template",Name="Nombre")]*** el parámetro ***template*** puede ir como no.

Attributes para los parámetros

También usamos attributes en los parámetros, estos especifican la ubicación en donde se encuentra el valor del parámetro en la request. Los que vamos a usar son los siguientes:

- **[FromBody]** → Viene un body en la request
- **[FromHeader]** → Vienen headers en la request
- **[FromQuery]** → Vienen parámetros en la request
- **[FromRoute]** → Viene en la misma request la información

Creamos algunos endpoints de ejemplo en OrderController usando los atributos mencionados anteriormente

De cada uno solo se mostrara el verbo HTTP, la firma del método y se retornara 200 Ok:

1. Traer todas las películas que se encuentren en el sistema → GET *api/orders*

```
[HttpGet]
public IActionResult Get(){
  return Ok();
}
```

2. Traer todas las ordenes del día 15-08-2022 → GET *api/orders?day='15-08-2022'*

```
// api/orders?day='15-08-2022'
[HttpGet]
public IActionResult GetBy([FromQuery] string day){
  return 0k();
}
```

3. Traer ordenes con identificador 2 → GET api/orders/2

```
//1 api/orders/2
[HttpGet("{id}")]
public IActionResult Get([FromRoute] int id){
  return Ok()
}
```

4. Crear una orden → POST api/orders

```
//api/orders
[HttpPost]
public IActionResult Post([FromBody] MovieModel movie){
  return 0k();
}
```

5. Actualizar la orden con id 2 → PUT api/orders/2

```
//api/orders/2
[HttpPut (" {id}")]
public IActionResult Put([FromRoute] int id,[FromBody] MovieModel movie) {
  return Ok();
}
```

6. Eliminar la película con identificador 2 → DELETE api/orders/2

```
//api/orders/2
[HttpDelete("{id}")]
public IActionResult Delete(FromRoute] int id){
  return Ok();
}
```

7. Eliminar todas las ordenes del sistema → DELETE api/orders

```
//api/orders
[HttpDelete]
public IActionResult Delete() {
  return Ok();
}
```

Ya vista la capa de la Web API veremos un ejemplo de la capa DataAccess que es similar a lo que creábamos en DA1, pero en este caso como halamos antes usaremos **EntityFrameworkCore**

Pasos para crear el proyecto

1. Creamos la solución

```
dotnet new sln -n Clase1
```

2. Creamos un proyecto de consola

```
dotnet new console -n Main
```

3. Agregamos el proyecto a la solucion

```
dotnet sln add Main
```

```
using System;

namespace Main
{
   class Program
   {
        static void Main(string[] args)
        {
            Console.WriteLine("Ejemplo clase 1");
        }
   }
}
```

4. Corremos el proyecto

```
dotnet run --proyect Main
```

5. Creamos un proyecto de library

```
dotnet new classlib -n Library
```

6. Agregamos el proyecto a la solucion

```
dotnet sln add Library
```

```
// Creo la clase Person
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string LastName {get; set; }
  public List<PersonDocument> Persons documents { get; set;}
}
```

8. Agregamos referencia de Main a Library

```
dotnet add Main reference Library
```

Esto me va a permitir usar la clase persona en la clase program de Main

9. Instalo entity framework en el paquete Library

```
// Me muevo a library
cd Library

//Agrego el paquete
dotnet add package Microsoft.EntityFrameworkCore
```

EntityFrameworkCore vs EntityFramework

Junto con .NET Core se han desarrollado del mismo modo algunas tecnologías relacionadas, entre ellas Entity Framework Core, una version ligera, extensible y multi-plataforma de Entity Framework

Diferencia en las relaciones n a n

EntityFrameworkCore mejora en ciertas cosas como las relaciones many to many → te fuerza a tener una tabla intermedia.

Relacion n a n

EntityFrameworkCore mejora en ciertas cosas como las relaciones many to many → te fuerza a tener una tabla intermedia.

Ejemplo relacion n a n → Documento tiene personas y cada persona tiene Documentos

```
// Creo la Clase Document
public class Document {
  public int Id { get; set; }
  public string Name { get; set; }
  public string Author { get; set; }
  public List<PersonDocument> Persons documents { get; set;}
}
```

```
// Creo la clase intermedia PersonDocument
public class PersonDocument{
  public int Personld { get; set; }
  public Person Person { get; set; }
  public int DocumentId { get; set; }
  public Document Document { get; set;}
}
```

10. Creamos en el proyecto Library la clase **contextDb**, esta clase es una implementacion del unit of work y los dbset de los repositorios

La clase **contextDb** es la puerta de acceso a la base de datos, aca tengo definidas en forma de DbSet cada una de las tablas de las entidades que voy a persistir

```
public class ContextDb: DbContext
{
  public DbSet<PersonDocument> PersonsDocuments {get; set;}
  public DbSet<Person> Persons {get; set;}
  public DbSet<Document> Documents {get; set;}
}
```

11. Le digo cual es la key de **PersonDocument**

```
protected override void OnModelCreating (ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<PersonDocument>()
        .HasKey(pp => new { pp.Personld, pp.DocumentId });
      modelBuilder.Entity<Person>(person =>
      {
        person.HasKey(x => x.Id);
        person.HasMany(x => x.Personsdocuments)
          .WithOne(personDocument => personDocument.Person)
          .HasForeignKey(personDocument => personDocument.Personld);
      });

      modelBuilder.Entity<Document>(document =>
      {
        document.HasKey(x => x.Id);
        document.HasMany(x => x.Personsdocuments)
          .WithOne(personDocument => personDocument.Document)
          .HasForeignKey(personDocument => personDocument.DocumentId);
      });
 }
```

12. Definimos en el contextDb también que motor de base de datos vamos a usar

En este caso vamos a usar SQL Server para lo que vamos a instalar EntityFrameworkCore.SqlServer en el paquete Library

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
        optionsBuilder.UseSqlServer(
            "Server=.\\SQLSERVER;Server=localhost,1433;Database=DB;Trusted_Connection=false;MultipleActiveResultSets=True;User=sa;Pass
}
```

13. Instalamos el paquete EntityFrameworkCore.Design en el paquete Library para poder hacer las migrations

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

14. Usamos el comando ef para generar la base de datos y crear migraciones desde el proyecto Library

```
dotnet ef migrations add MyMigration --startup-project "../Main"
```

```
dotnet ef database update --startup-project "../Main"
```

Links de interes para profundizar en los temas dados en la clase:

Introducción a ASP .NET Core

https://www.tektutorialshub.com/asp-net-core/asp-net-core-introduction/

Armado de ambiente

https://www.tektutorialshub.com/asp-net-core/asp-net-core-installing-setting-up-development-environment/

DotNet CLI

https://www.tektutorialshub.com/asp-net-core/asp-net-core-dotnet-cli-command-line-tool/

Construcción de proyecto base

https://www.tektutorialshub.com/asp-net-core/asp-net-core-getting-started/

Estructura del proyecto

https://www.tektutorialshub.com/asp-net-core/asp-net-core-solution-structure/

Create web APIs with ASP.NET Core

https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.1#apicontroller-attribute

Model Binding in ASP.NET Core

https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1

Tutorial: Create a web API with ASP.NET Core

https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio-code