

Fundamentos .Net Core

Ángel Carlos López Quevedo

CURSO PARA GTT



13.12.2021



Rediscover
the meaning of technology

Founded in 2006 by 4 Microsoft MVPs, Plain Concepts was created to help companies adopt new technologies aimed at improving their productivity and processes.

Awarded in 2016 as **Microsoft Partner of the Year**, we currently have over **350 employees**, reaching a milestone in the technology sector by having more than **10 professionals** recognized as **Microsoft MVP** and over a dozen certifications at business level.

Present in **Spain, USA, Australia, United Kingdom, Germany and the Netherlands**, we have developed over 2,500 projects for companies across all industrial vertical.

You can find us



SPAIN



USA



AUSTRALIA



UNITED
KINGDOM



GERMANY



NETHERLANDS

Our services





Ángel Carlos López

SOFTWARE DEVELOPER ENGINEER

Soy un apasionado de la programación. Me gusta darle vueltas a los problemas y decidir implementar la solución que más valor aporte al proyecto. Por lo que me gusta pensar!

En mi tiempo libre, soy ajedrecista, casi siempre podrás encontrarme moviendo piezas en algún tablero virtual.

@_aclopez

<https://github.com/acnagrellos>



Web & App



DevOps

- Nombre

- Puesto de trabajo

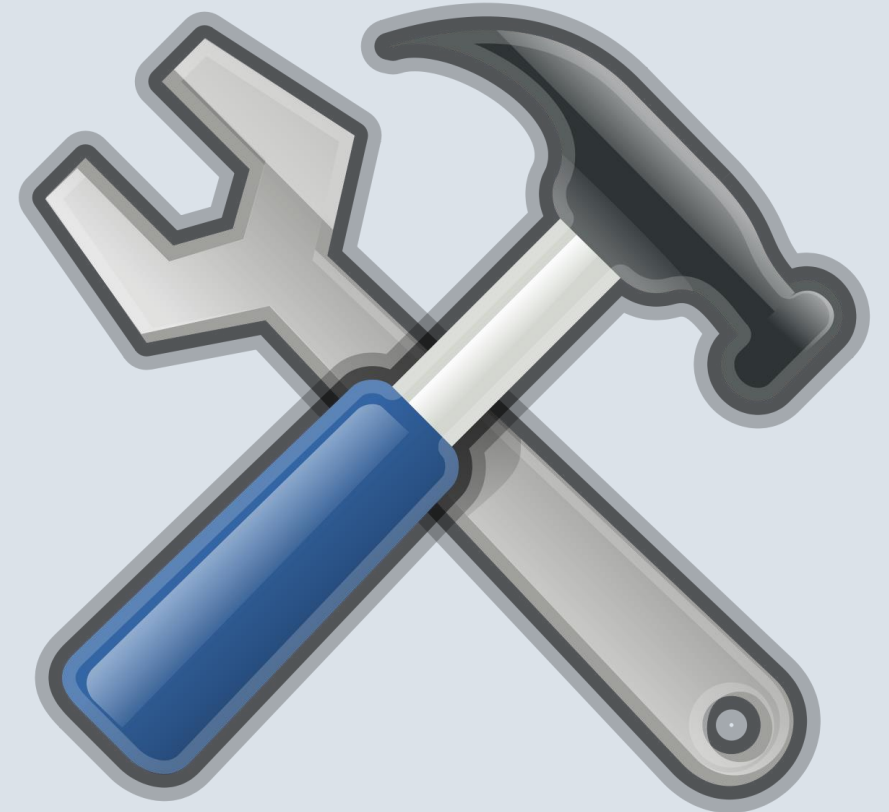
- Experiencia en programación

- ¿Que experiencia tienes con NetCore?



Herramientas

Herramientas que
vamos a usar
durante el curso

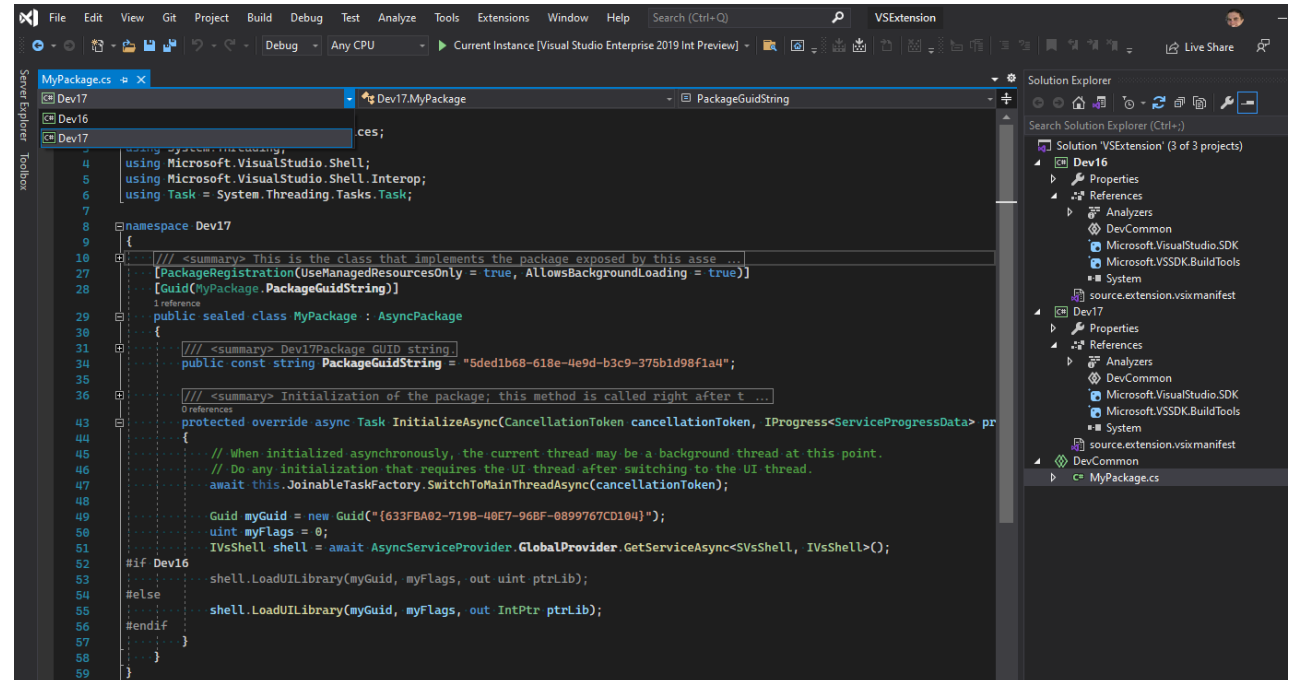


Visual Studio



Visual Studio 2022 es el IDE oficial de Microsoft

- Productivo
- Moderno
- Innovador



Microsoft Teams



Microsoft Teams, como Plataforma de Comunicación

- Reunión de las clases
- Reuniones partidas para resolver dudas de manera individual

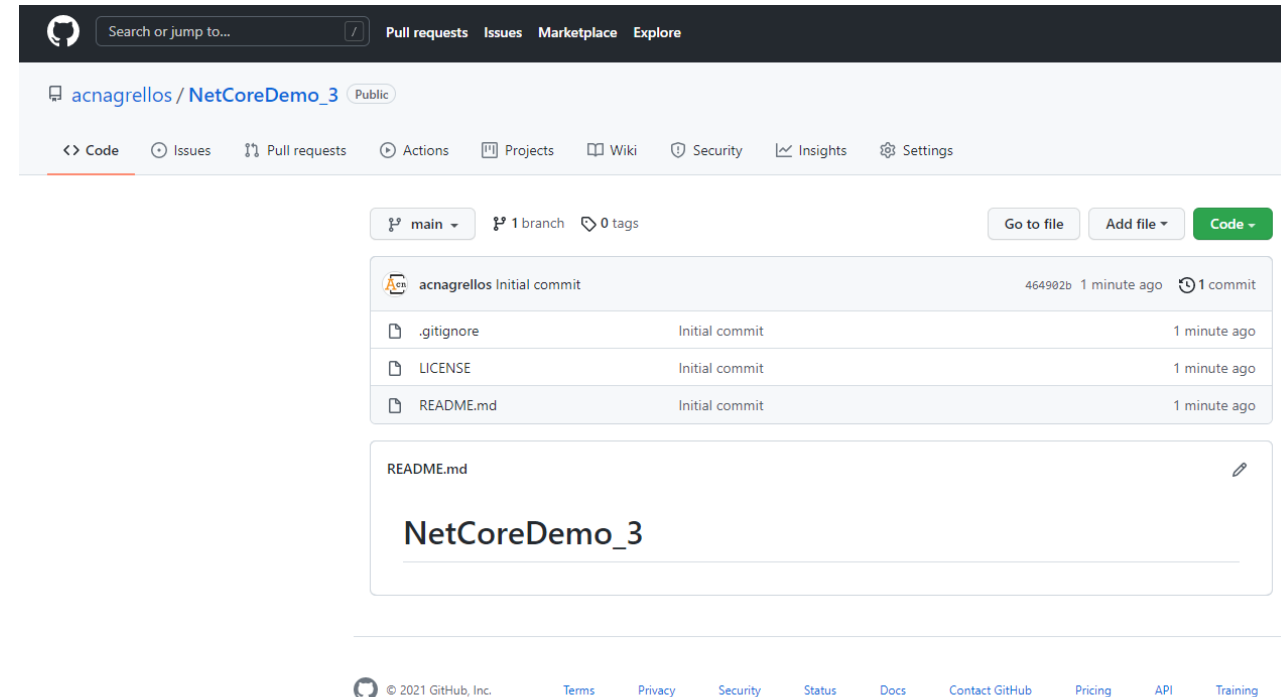


Github



Github, como repositorio del Código Fuente que desarrollemos

- Material subido por temas
- Resolución de todos los ejercicios
- Subidas por día de clase



Email

Resolución de dudas por email o Teams

- No os quedéis con dudas, ¡preguntad!
- aclopez@plainconcepts.com



Agenda



13 December

08:00 – 08:30

Presentación

08:30 – 09:00

¿Por qué Net Core?

09:00 – 10:00

DotNet Cli

10:00 – 10:20

Descanso

10:20 – 10:45

Ejercicios DotNet Cli

10:45 – 12:00

API Rest

12:00 – 12:30

Ejercicio Diseño de una API

12:30 – 12:45

Descanso

12:45 – 15:00

Fundamentos de Net Core

¿Por qué NetCore?

Que es NetCore y
cual es el motivo por
el que usarlo para
construir nuestras
aplicaciones



¿Qué es NetCore?

.NET Core es un framework de desarrollo de código libre y abierto para desarrollar aplicaciones multiplataforma dirigidas a Windows, Linux y macOS. De acuerdo con Open Source For U, esta tecnología es capaz de ejecutar aplicaciones en dispositivos, la nube y el IoT.

- Framework de Desarrollo
- Código libre y abierto
- Aplicaciones multiplataforma

¿Qué es NetCore?

¿Donde se pueden hacer aplicaciones NetCore?

- **Web:** Sirve para crear aplicaciones y servicios web para Windows, Linux, macOS y Docker.
- **Móvil:** Usa una sola base de código para crear aplicaciones nativas en iOS, Android y Windows.
- **Escritorio:** Cree aplicaciones de escritorio atractivas para Windows y macOS.
- **Microservicios:** Crea microservicios de forma independiente que se ejecuten en contenedores de Docker.
- **Desarrollo de juegos:** Desarrolla juegos 2D y 3D para los escritorios, teléfonos y consolas más populares.
- **Aprendizaje Automático:** Agregue algoritmos de visión, procesamiento de voz, modelos predictivos y muchos más a sus aplicaciones.
- **IoT:** Crea aplicaciones de IoT (Internet de las Cosas), con soporte nativo para Raspberry Pi, HummingBoard, BeagleBoard, Pine A64, y otros.

¿Por qué Net Core?

Multiplataforma: NET Core es soportado por los tres principales sistemas operativos: Linux, Windows and OS X.

- Las librerías .NET Core pueden ejecutarse sin modificaciones en los sistemas operativos compatibles..
- Las aplicaciones deben recompilarse por entorno
- Los usuarios pueden elegir el mejor entorno compatible con .NET Core para sus aplicaciones



¿Por qué Net Core?

Código Abierto: [NET Core](#) está disponible en Github

- Licencia con el MIT y Apache 2
- También hace uso de un conjunto significativo de dependencias de la industria de código abierto
- Ser OSS (Open Source Software) es fundamental para tener una comunidad próspera más "imprescindible" para muchas organizaciones donde OSS es parte de su estrategia de desarrollo



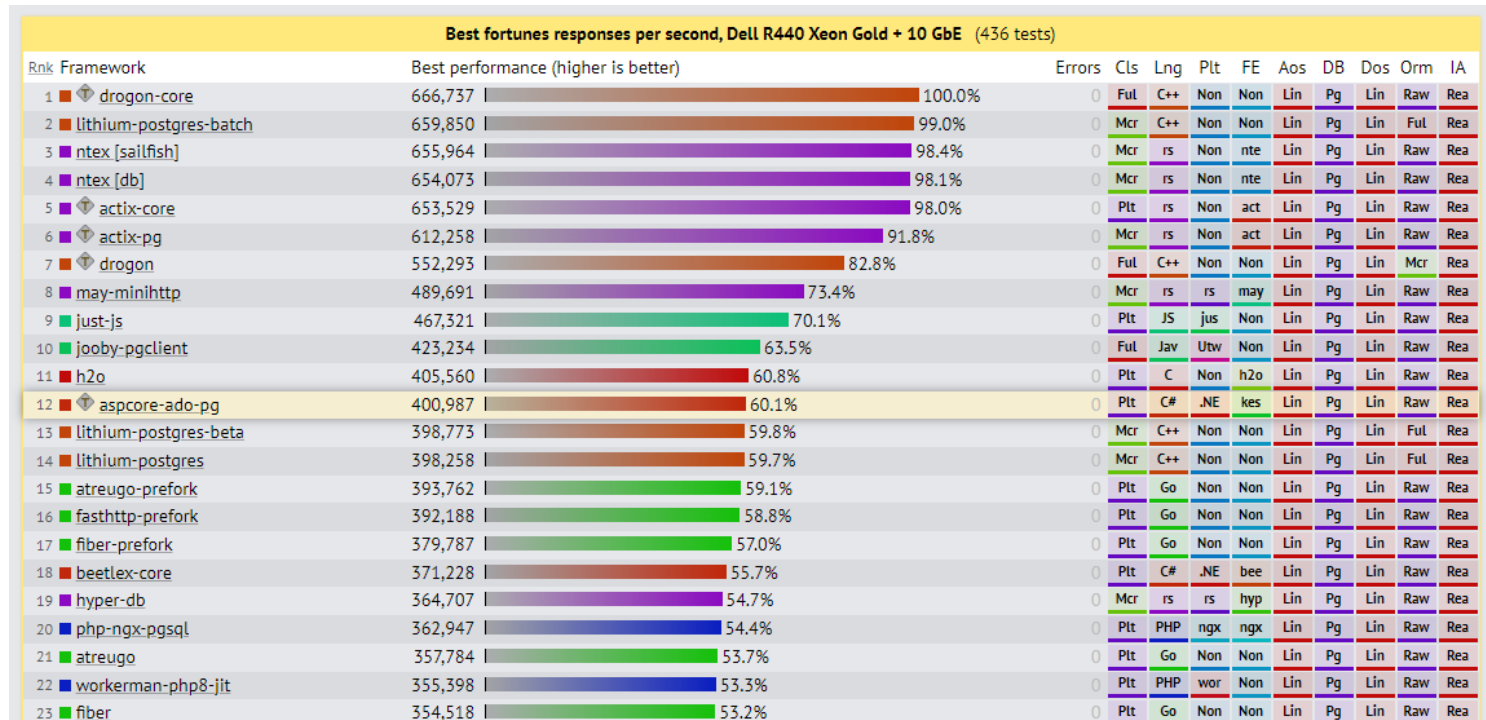
¿Por qué Net Core?

Natural Acquisition: .NET Core está distribuido en una serie de paquetes Nugets que el usuario puede escoger acorde a sus necesidades.

- Nuget como gestor de paquetes de .NET
- El runtime y la base del framework se pueden obtener en NuGet y otros OS-specific package managers
- Docker images están disponibles en docker hub.
- Las librerías del framework de alto nivel están disponibles en Nuget

¿Por qué Net Core?

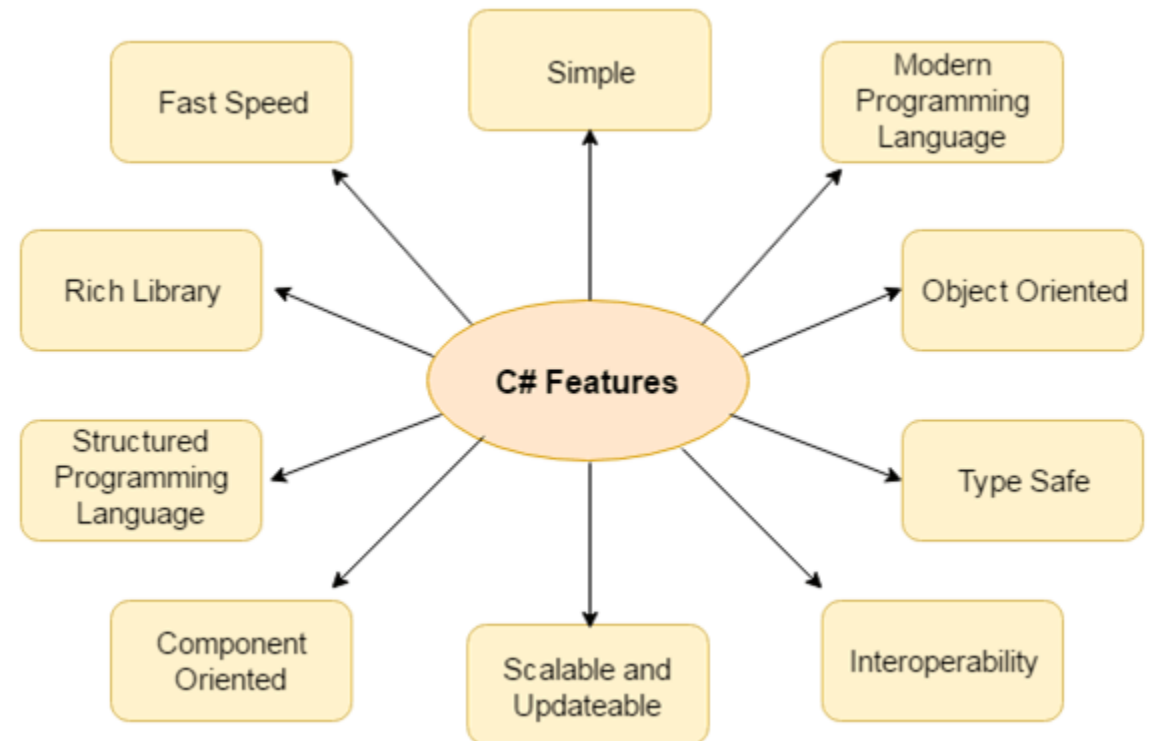
Rápido: El rendimiento de .NET Core en las versiones 5 y 6 ha mejorado muchísimo y es uno de los frameworks más rápidos: [Fuente](#)



¿Por qué Net Core?

Moderno: .Net Core tiene como lenguaje de programación C# uno de los lenguajes más modernos que existen.

- C# evoluciona constantemente, ahora ha salido la versión 10.



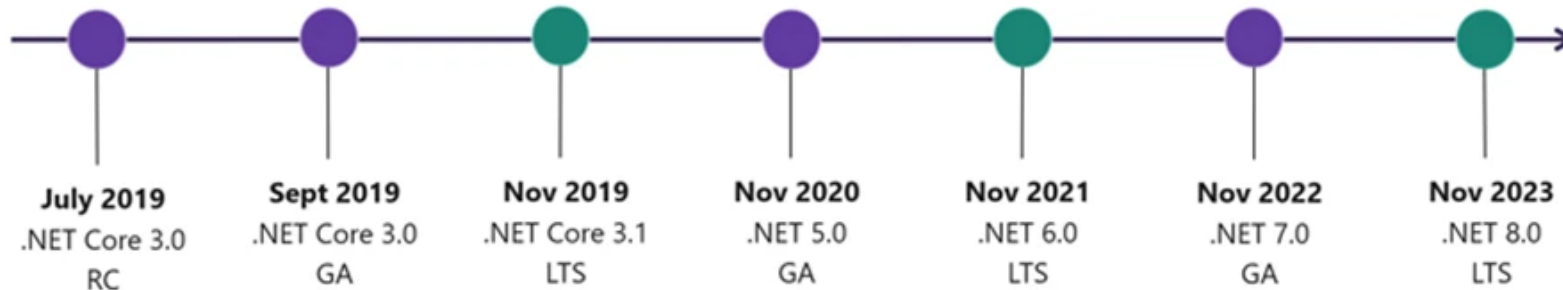
Estructura de NetCore

.NET – A unified platform



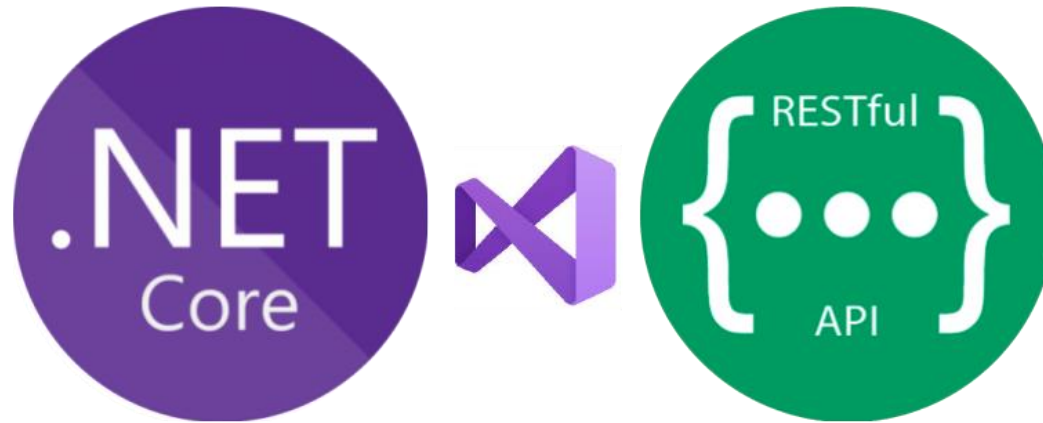
Evolución de NetCore

.NET Schedule



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

Net Core – Web APIs



Create



Read



Update



Delete

Instalación Net Core

Instalar Net Core es tan fácil como ir a la página de descarga del framework: [Download NetCore](#)

- SDK: Son las tools para desarrollar en .NET
- Runtime: Los binarios que necesitamos instalar en una máquina que queramos que corra una aplicación .NET



DotNetCli

La interfaz de la línea
de comandos (CLI)
de .NET

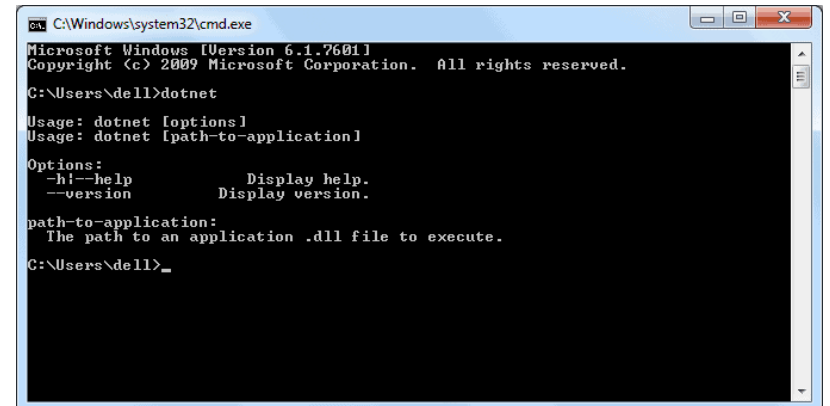


.NET CLI

DotNetCli

La interfaz de la línea de comandos (CLI) de .NET es una cadena de herramientas multiplataforma que sirve para desarrollar, compilar, ejecutar y publicar aplicaciones .NET.

- La documentación de Microsoft es:
 - <https://docs.microsoft.com/es-es/dotnet/core/tools/>
- Los comandos más importantes son
 - dotnet
 - dotnet new
 - dotnet run
 - dotnet build
 - dotnet publish



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>dotnet

Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help           Display help.
  --version           Display version.

path-to-application:
  The path to an application .dll file to execute.

C:\Users\dell>
```

dotnet new

Con dotnet new podemos crear crea un nuevo proyecto, archivo de configuración o solución según la plantilla especificada

<https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-new>

```
dotnet new <template> -n <name>
```

```
-- crea una nueva solución  
dotnet new sln -n prueba
```

```
-- crea un nuevo Proyecto de Consola  
dotnet new console -n MiConsola -o MiConsola
```

```
-- crea una nueva librería de clases  
dotnet new classlib
```

dotnet build

compila un proyecto y todas sus dependencias.

<https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-build>

```
dotnet build <project.csproj>
```

```
-- build MiConsola csproj  
dotnet build MiConsola/MiConsola.csproj
```

```
-- build de una solucion sln  
dotnet build prueba.sln
```

dotnet run

Ejecuta el código fuente sin comandos explícitos de compilación o inicio.

<https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-run>

```
dotnet <archive.dll> | dotnet run --project<project.csproj>
```

-- ejecutar el Proyecto que hay en la carpeta
dotnet run

-- ejecutar un Proyecto
dotnet run --project MiConsola/MiConsola.csproj

-- ejecutar un archivo dll
dotnet MiConsola/bin/Debug/net6.0/MiConsola.dll

-- no se puede ejecutar una Solucion
dotnet run prueba.sln ✗

dotnet publish

publica la aplicación y sus dependencias en una carpeta para la implementación en un sistema de hospedaje.

<https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-publish>

```
dotnet new <template> -n <name>
```

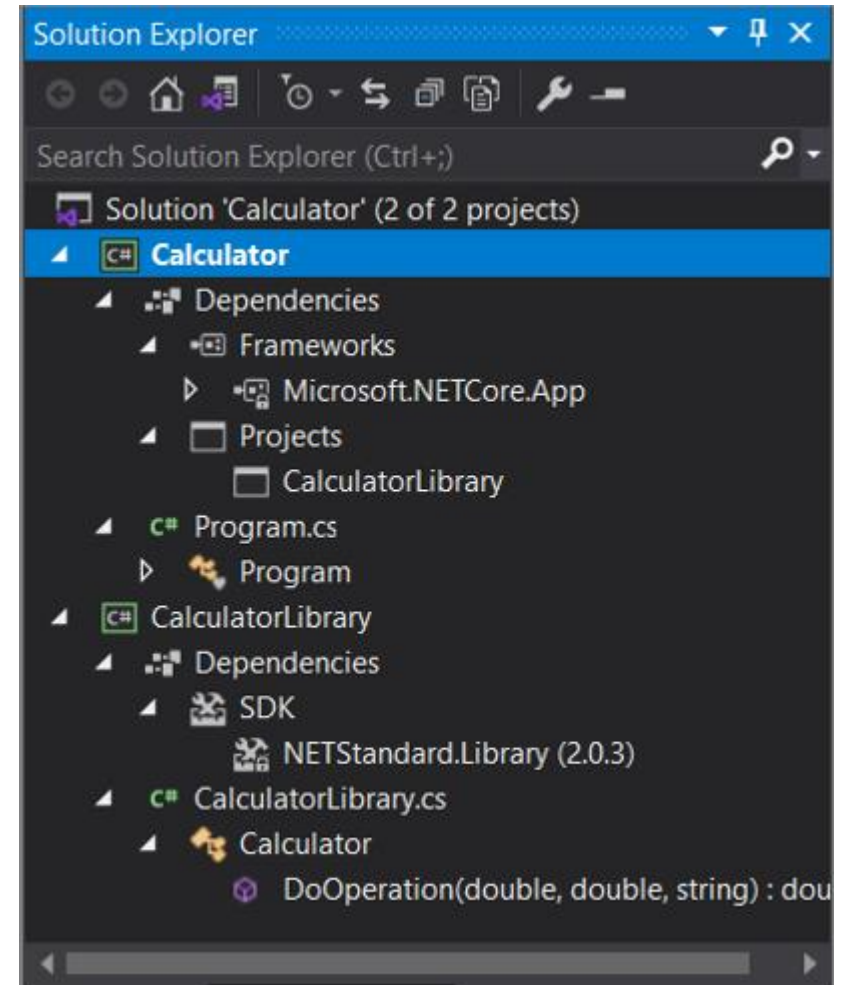
```
-- crea una nueva solución  
dotnet new sln -n prueba
```

```
-- crea un nuevo Proyecto de Consola  
dotnet new console -n MiConsola
```

```
-- crea una nueva librería de clases  
dotnet new classlib
```

SLN y CSPROJ

- La solución engloba todos los proyectos
- Los proyectos pueden tener referencias de otros proyectos
- No se pueden dar referencias circulares
- Los proyectos de tipo classlib no se pueden ejecutar, solo están para ser referenciados



dotnet sln

enumera o modifica los proyectos en un archivo de solución de .NET.

<https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-sln>

```
dotnet sln <solution.sln> <parameters>
```

```
-- Enumera los proyectos de una solución  
dotnet sln prueba.sln list
```

```
--Añade un Proyecto a la solución  
dotnet sln add MiConsola/MiConsola.csproj
```


dotnet <command> reference

Gestiona las referencias de un Proyecto.

<https://docs.microsoft.com/es-es/dotnet/core/tools/dotnet-add-reference>

```
dotnet list <project.csproj> references
```

```
dotnet add <project.csproj> reference <project.csproj>
```

```
dotnet remove <project.csproj> reference <project.csproj>
```

```
-- Lista las referencias de un Proyecto
```

```
dotnet list MiConsola/MiConsola.csproj reference
```

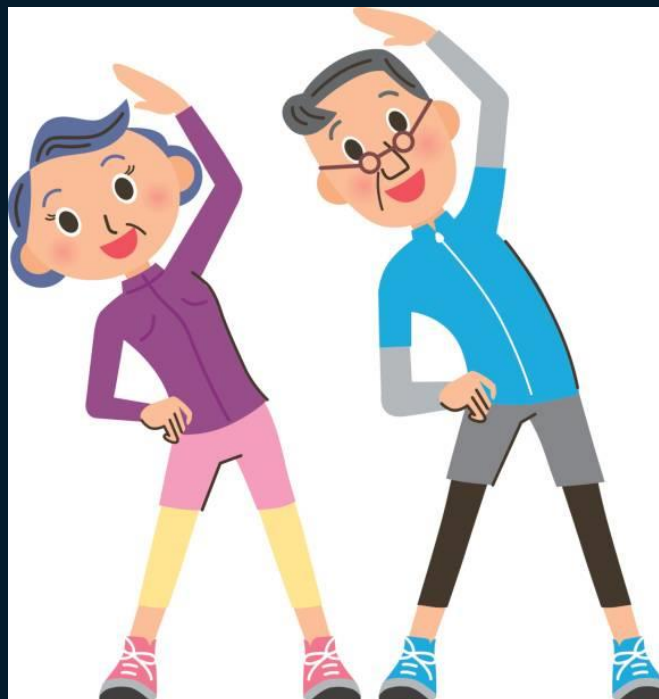
```
-- Añade una referencia a un Proyecto
```

```
dotnet add MiConsola/MiConsola.csproj reference  
ClassLib/ClassLib.csproj
```

```
-- Elimina una referencia a un Proyecto
```

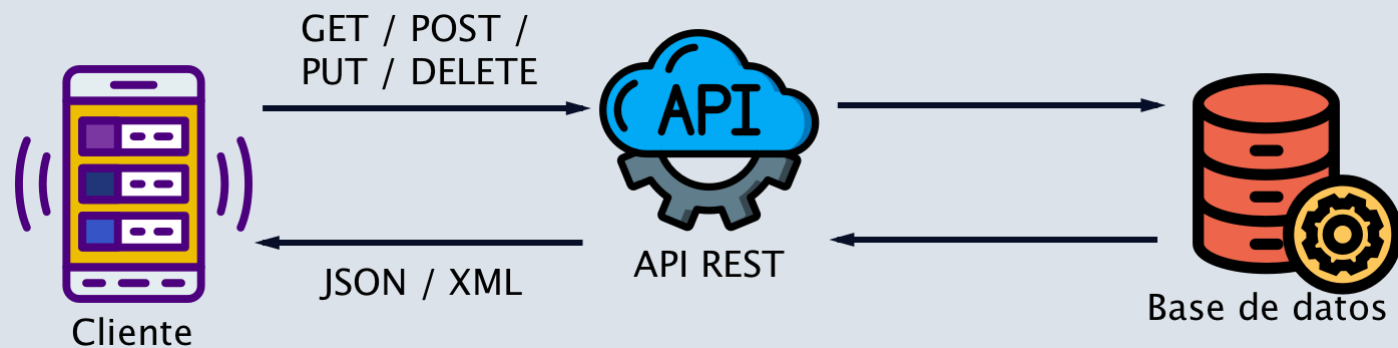
```
dotnet remove MiConsola/MiConsola.csproj reference  
ClassLib/ClassLib.csproj
```

Ejercicios time!



API REST

APIs HTTP Restful



¿Qué es un API REST?

Fundamentals of REST API

API REST: Definición

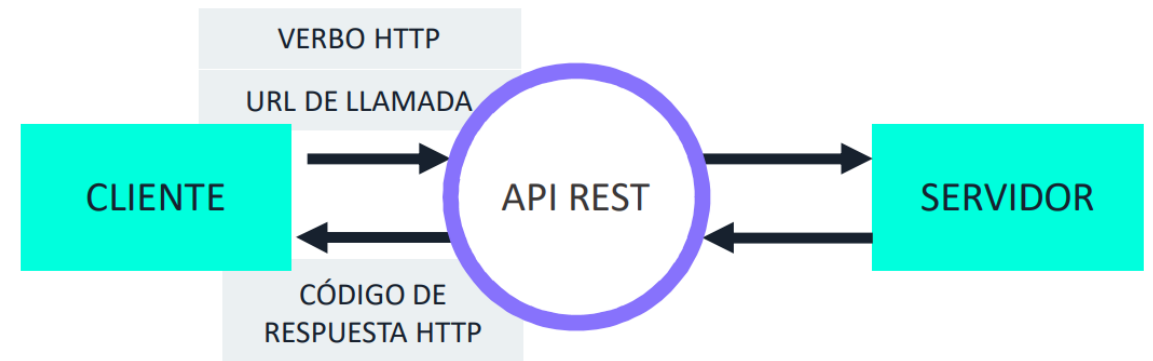


- **API** (Application Programming Interface) será el contrato que nuestro servidor le ofrezca a todos sus clientes.
- **REST** (Representational State Transfer) es un estilo de arquitectura para diseñar aplicaciones en la red.
 - Cada **petición HTTP contiene toda la información necesaria para ejecutarla**, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla.
 - Se compone de una lista de reglas que se deben cumplir en el diseño de la arquitectura de una API.
 - Hablaremos de **servicios web restful** si cumplen la arquitectura REST (restful = adjetivo; rest = nombre)

¿Qué es un API REST?

API REST: ¿Como funciona?

- Las llamadas al API se implementan como peticiones HTTP
- La URL representa el recurso al que queremos acceder
 - <https://reqres.in/api/users?page=2>
- El verbo HTTP representa la acción
 - GET, POST, PUT, DELETE
- El código de estado representa el resultado devuelto por el API
 - 200 (OK), 201 (CREATED), 404 (NOT FOUND), 500 (INTERNAL SERVER ERROR)
- Los datos que devuelve el API junto con el código de respuesta suele estar en formato JSON o XML



¿Qué es un API REST?

Reglas de la Arquitectura REST. Las reglas que debe cumplir una implementación de un API para ser restful son

- Interfaz uniforme
- Peticiones sin estado
- Cacheable
- Separación entre cliente y servidor



Reglas Arquitectura REST

Interfaz uniforme

- La interfaz de basa en recursos
 - El recurso Empleado (Id, Nombre, Apellido)
- El servidor manda los datos en el formato que le pidan (json, xml...)
 - Nunca se envían estructuras internas (BBDD por ejemplo)
- La información del recurso que le llega el cliente podrá ser utilizado en otras operaciones
 - Si se tienen permisos
- Procurar una API sencilla y jerárquica
- Nomenclatura de los recursos siempre en plural

RESOURCE

```
1 {  
2   "id": 1,  
3   "name": "Administration",  
4   "description": "Category for administrators",  
5   "familyId": "1"  
6 }
```

DDBB TABLE

Categories *	
Id	
Name	
Description	
CreatedOn	
UpdatedOn	
FamilyId	

Reglas Arquitectura REST

Peticiones sin estado. El protocolo HTTP es un protocolo sin estado

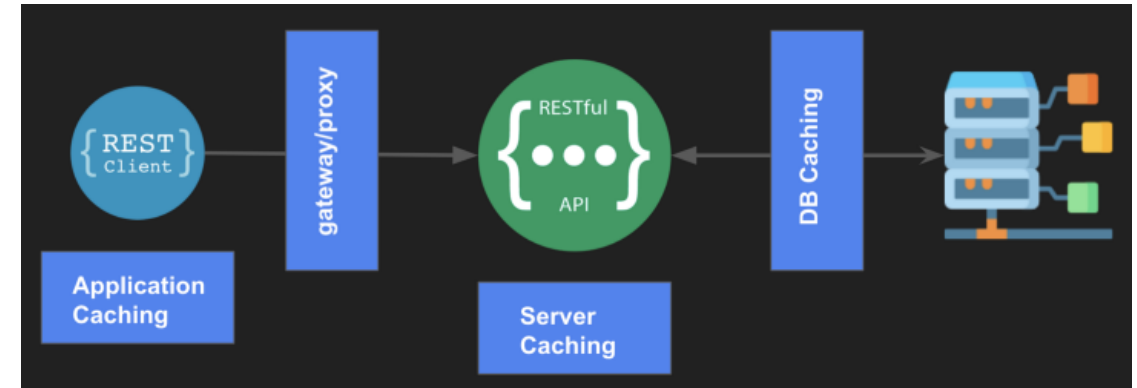
1. GET {baseurl}/customers/1234/invoices => 200, json: [2345, 2346, 2347]
2. DELETE {baseurl}/customers/1234/invoices/2345 => 200

- En la segunda petición hemos tenido que indicar el identificador del recurso que venía en la primera petición
- El servidor no guardaba los datos de la consulta previa que tenía el cliente en particular
- Después de borrar el recurso un GET a la invoice 2345 dará error, ya que ¡no existe el recurso!

Reglas REST

Cacheable

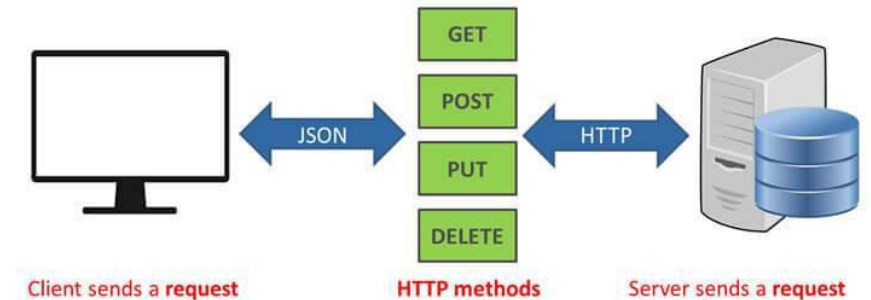
- En la web los clientes pueden cachear las respuestas del servidor
- Las respuestas se deben marcar de forma implícita o explícita como cacheables o no, apoyándose en las cabeceras
- En futuras peticiones, el cliente sabrá si puede reutilizar o no los datos que ha obtenido, y cuanto tiempo puede cachearlos
- Si ahorramos peticiones, mejoramos el rendimiento en cliente (evitamos latencia)



Reglas Arquitectura REST

Separación entre cliente y servidor

- El cliente y servidor están separados, su unión es mediante la API.
- Los desarrollos en frontend y backend se hacen por separado, cumpliendo el contrato que define la API
- Mientras la interfaz no cambie, podremos cambiar el cliente o el servidor sin problemas.

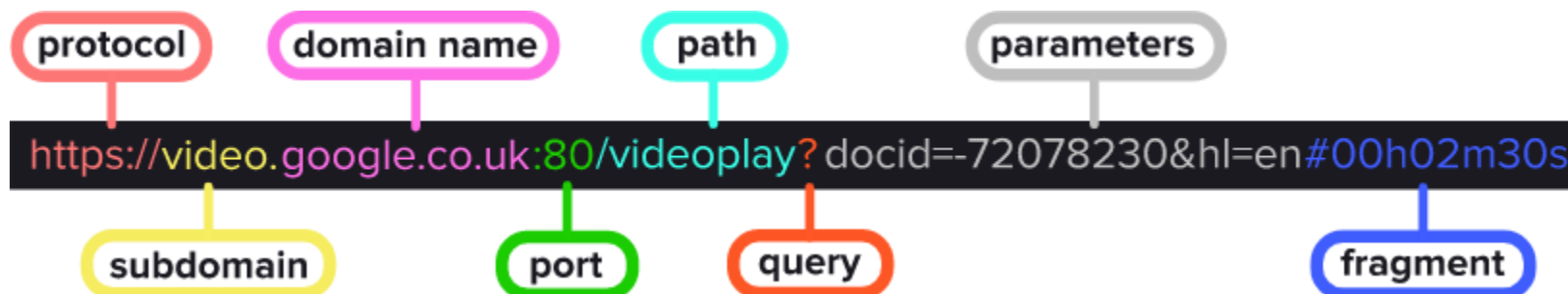


URIs, URLs

URI: LA URI es el recurso, por lo tanto, es la información a la que queremos acceder o que queremos modificar o borrar, independientemente de su formato.

URL: (Uniform Resource Locator) son un tipo de URI (Uniform Resource Identifier) que además de permitir identificar de forma única el recurso, nos permite localizarlo para poder acceder a él o compartir su ubicación.

- URL: {protocolo}://{dominio o hostname}[:puerto (opcional)]/{ruta del recurso}?{consulta de filtrado}



URIs, URLs y QueryString

Reglas de una URI

- Los nombres de URI no deben implicar una acción, por lo tanto, debe evitarse usar verbos en ellos.
- Deben ser únicas, no debemos tener más de una URI para identificar un mismo recurso.
- Deben ser independiente de formato
- Deben mantener una jerarquía lógica
- Los filtrados de información de un recurso no se hacen en la URI

URIs, URLs y QueryString

Las URIs no deben implicar acciones y deben ser únicas

- /invoices/234/edit ✗
 - Por ejemplo, la URI /invoice/234/edit sería incorrecta ya que tenemos el verbo editar en la misma
- /invoices/234 + HTTP VERB ✓
 - Para el recurso factura con el identificador 234, la URI anterior sería la correcta, independientemente de que vayamos a editarla, borrarla, consultarla o leer sólo uno de de sus conceptos. Esta URI luego irá acompañada de su verbo correspondiente, que indicará su acción.

URIs, URLs y QueryString

Las URIs deben ser independientes de formato

- /invoices/234.pdf ✗
 - No es una URI correcta ya que estamos indicando la extensión. Los formatos se marcan con las cabeceras HTTP.
- /invoices/234 + HTTP HEADER FORMATO PDF ✓
 - Para el recurso factura con el identificador 234 la URI anterior es correcta, ya que indicaríamos el formato de como queremos recibir el recurso en una cabecera, ya sea formato pdf, epub, txt, xml o json.
 - La cabecera para el formato de la petición suele ser: Content-Type

URIs, URLs y QueryString

Las URIs deben mantener una jerarquía lógica

- /invoices/234/customers/007 ✗
 - No es una URI correcta ya que no sigue una jerarquía lógica. Es raro que las facturas tengan clientes. Los clientes deberían tener facturas.
- /customers/007/invoices/234 ✓
 - Ahora la jerarquía si es correcta.

URIs, URLs y QueryString

Filtrados y otras operaciones.

- Para filtrar, ordenar, paginar o buscar información en un recurso, debemos hacer una consulta sobre la URI, utilizando parámetros HTTP en lugar de incluirlos en la misma.
- `/invoices/order/desc/date/2007/page/2` ✗
 - Es una URI incorrecta, ya que para filtrar recursos debemos utilizar el Query String.
- `/invoices?date=2007&order=DESC&page=2`
 - Esta sería la URI correcta usando el QueryString
 - Lo que hay a partir de la ? es lo que se denomina QueryString.

Verbos HTTP

Existen 4 Verbos HTTP que indican las operaciones que vamos a llevar a cabo.

- GET: Para consultar y leer recursos.
- POST: Para crear recursos
- PUT: Para editar recursos
- DELETE: Para eliminar recursos
- PATCH: Para editar partes concretas de un recurso (muy poco usado)

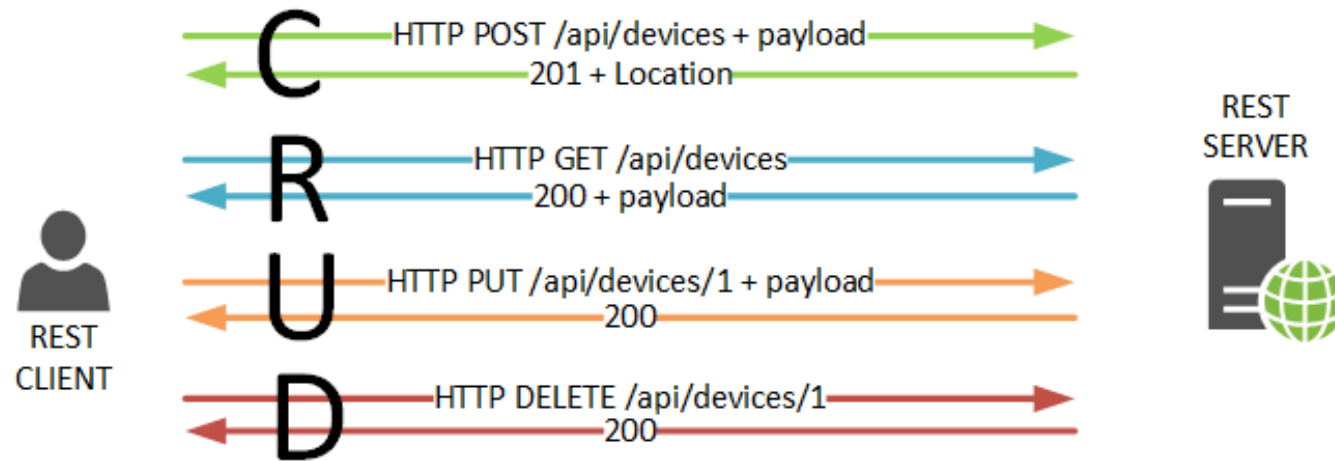
Verbos HTTP

Ejemplos de consultas con verbos HTTP

- GET /invoices Nos permite acceder al listado de facturas
 - GET /invoices/321 Nos permite acceder a la factura 321
 - POST /invoices Nos permite crear una factura nueva.
 - Su resultado normalmente es 201 (Created)
 - PUT /invoices/123 Nos permite editar la factura, sustituyendo la totalidad de la información anterior por la nueva.
 - DELETE /facturas/123 Nos permite eliminar la factura
-
- Es un error emplear solo métodos GET y POST para todas las operaciones de nuestra API

Verbos HTTP

Ejemplos de consultas con verbos HTTP



Ejercicios URI

Ejercicios de URI dentro del módulo de API-Teoría



Códigos de Estado

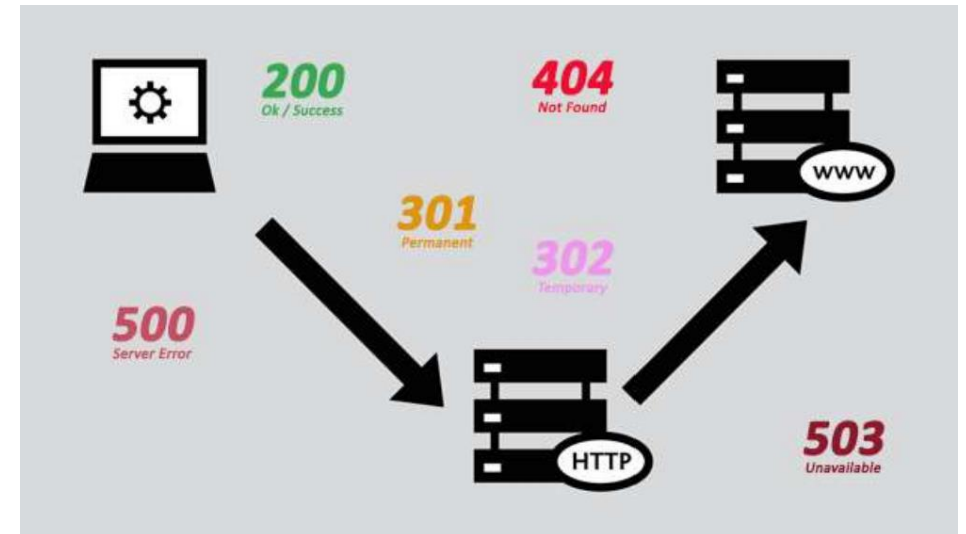
Códigos de Estado

- Si tenemos una petición PUT /facturas/123 y la respuesta que nos devuelven es:
 - Status Code 200
 - Content: { success: false, code: 754, errormessage: "datos insuficientes" }
- Esto está devolviendo un código 200 (Respuesta correcta), y en el cuerpo de la respuesta el error.
 - Esto no es una respuesta standard ni es Restfull
 - El cliente se tendría que adaptar a este tipo de respuestas.
 - Tenemos que mantener nuestros propios códigos en el servidor
- No hay que reinventar la rueda con las respuestas en un API que va por HTTP, hay que respetar el protocolo.
 - Para ello el protocolo HTTP dispone de unos códigos de respuesta standard

Códigos de Estado

Códigos de Estado. Http tiene un abanico muy amplio de códigos de estado, incluidos los códigos de error. Su codificación genérica es:

- 100-199: Respuestas informativas
 - 200-299: Peticiones correctas
 - 300-399: Redirecciones
 - 400-499: Errores del cliente
 - 500-599: Errores del servidor
- [Anexo:Códigos de estado HTTP - Wikipedia, la enciclopedia libre](#)



Códigos de Estado

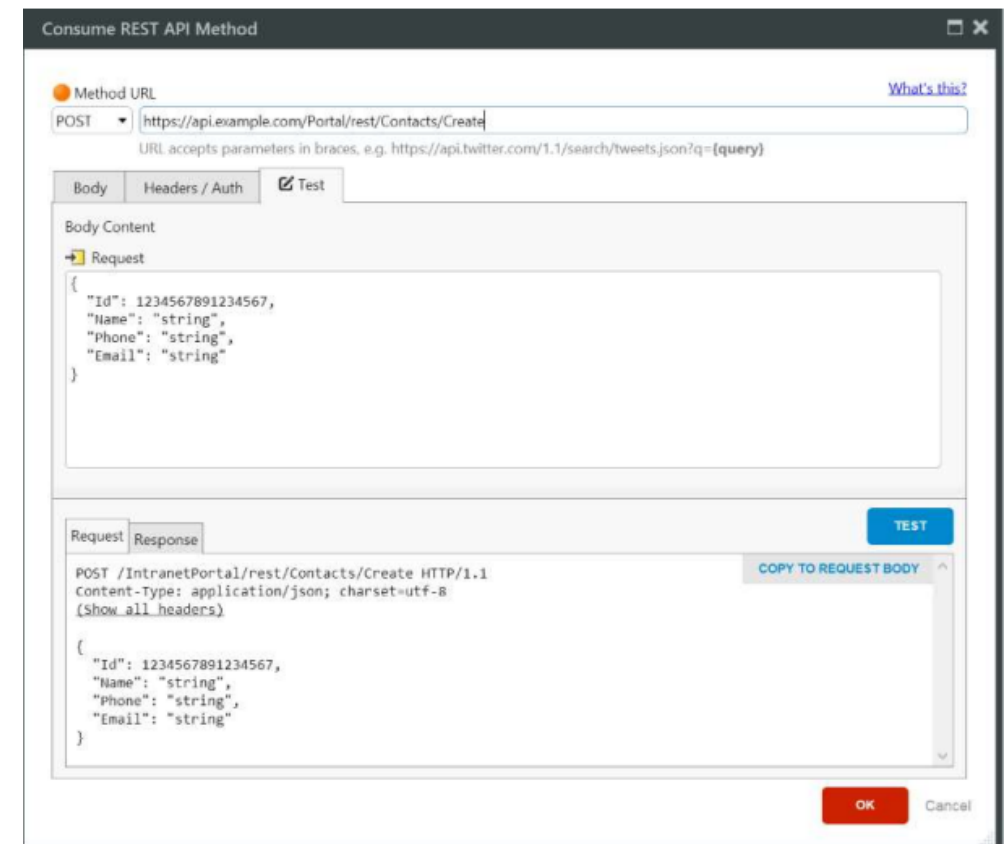
Códigos de Estado más importantes

Código	Significado	Descripción
200	OK	Todo ha ido correctamente.
201	CREATED	El nuevo recurso ha sido creado
204	NO CONTENT	Todo ha ido correctamente.
400	BAD REQUEST	La petición es inválida o no puede ser servida
401	UNAUTHORIZED	La petición requiere de la autenticación del usuario.
403	FORBIDDEN	El acceso al recurso no está permitido.
500	INTERNAL SERVER ERROR	Error en el servidor.

Body en Put y Post

Body. Las peticiones y respuestas HTTP tienen una estructura similar donde comparten un campo llamado Body que es opcional

- Es el campo donde se envía la información del recurso a modificar o añadir
- No puede usarse en peticiones GET.
- Solo se usa en PUT y POST
- Muy raramente se usa en algún DELETE



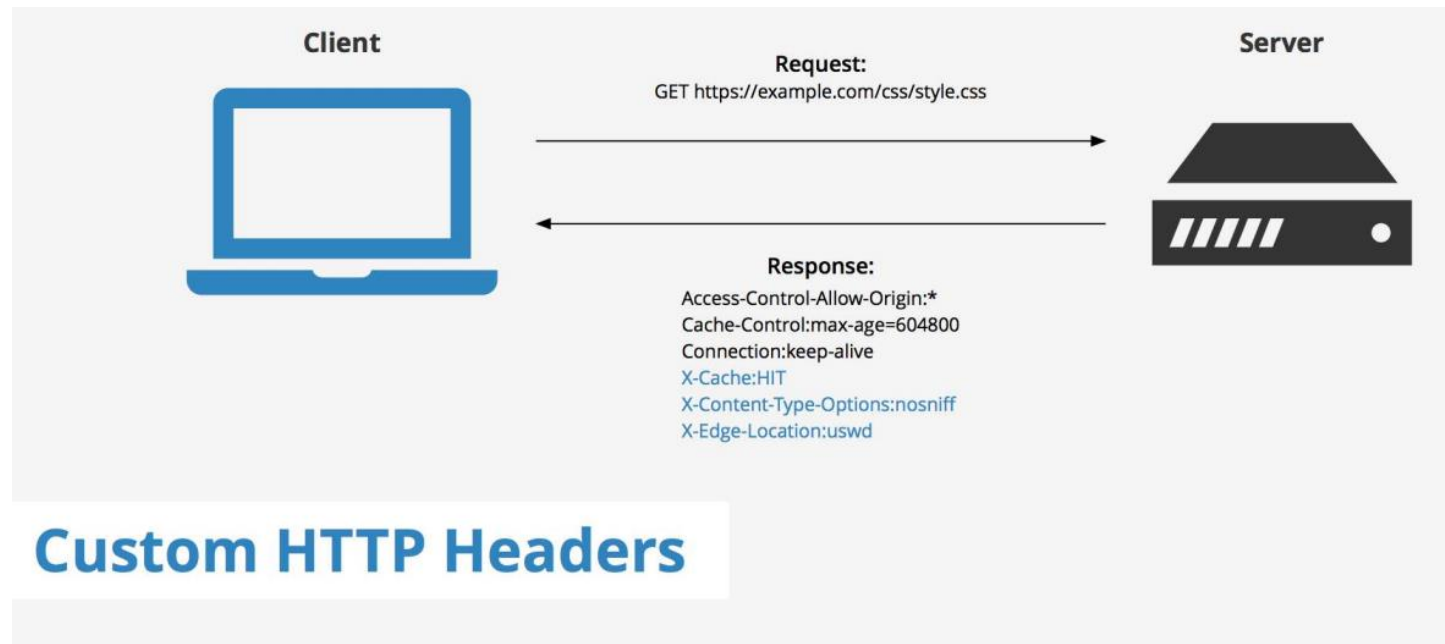
Cabeceras HTTP

Cabeceras HTTP. Las peticiones y respuestas HTTP tienen otra peculiaridad: las cabeceras

- Las cabeceras permiten al cliente y servidor enviar información adicional junto a una petición
- Una cabecera está compuesta por su nombre y su valor
 - (<header-name>: <value>)
- Existen muchas cabeceras standards
 - [HTTP headers - HTTP | MDN \(mozilla.org\)](#)
- Podemos añadir nuestras propias cabeceras usando el prefijo 'X-'
 - X-Custom-Header: value10

Cabeceras HTTP

Cabeceras HTTP. Las peticiones y respuestas HTTP tienen otra peculiaridad: las cabeceras



HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) Otra condición para cumplir el API REST es que deben disponer de enlaces a otros recursos referenciamos.

- Si por ejemplo en una petición de una facture, esa factura tiene un cliente, nuestro json debería contener algo parecido a:

```
{
  id: 344,
  amount: 525,
  orderId: 266,
  client: {
    id: 155,
    href: '/clients/155'
  }
}
```

Ejercicios Diseño API

Ejercicios de diseño de una API dentro del módulo de API-Teoría



Demo POSTMAN

POSTMAN tool para consumir una API

- Es una aplicación que nos permite realizar pruebas API
- También puede servir como repositorio para las llamadas de una API



POSTMAN

Ejercicios POSTMAN

Ejercicios de POSTMAN dentro del módulo de API-Teoría



Agenda



13 December

08:00 – 10:30

ASP.NET Core Fundamentals

10:30 – 10:50

Descanso

10:50 – 11:30

ASP.NET Core Fundamentals

11:30 – 13:00

ASP.Net Core Controladores

13:00 – 13:15

Descanso

13:15 – 15:00

ASP.Net Core Controladores

ASP.NET Core Fundamentals

ASP.NET Core –
Estructura de
proyectos.



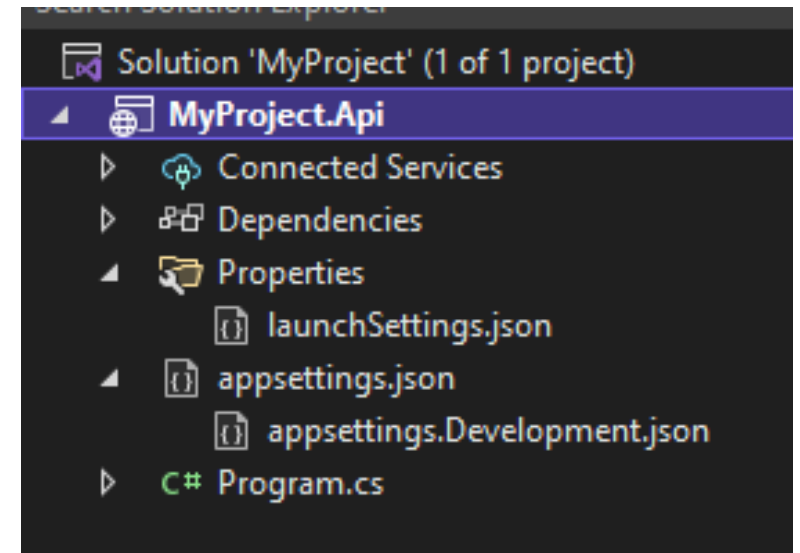
.NET 6

```
references  
static void Main(string[] args)  
    Product myProduct = new Product { Name = "My Product", Price = 10.99m };  
    var pos = myProduct.Position; // 1 reference  
    static double Postage(double price) => {  
        {  
            < 20 => 5.99,  
            20 and < 40 => 3.99,  
            40 and < 60 => 2.99,  
            60 and < 80 => 1.99,  
            80 and < 100 => 0.99,  
            > 100 => 0.49;  
        }  
    }  
}
```


Project NetCore Estructura

La Estructura de un Proyecto ASP.NET Core

- launchSettings.json
 - Fichero de Configuración
- Appsettings.json
 - Appsettings.Development.json
- Program.cs
- Nuget Packages



Project NetCore Estructura

launchSettings.json

- Fichero de configuración de la ejecución en Visual Studio
- Se pueden crear varios perfiles
- Se puede modificar el puerto de la app
- Se pueden setear variables de entorno
 - Por ej: "ASPNETCORE_ENVIRONMENT: Development"

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:23762",
      "sslPort": 44334
    }
  },
  "profiles": {
    "MyProject.Api": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7105;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Project NetCore Estructura

Program.cs

- Punto de entrada de .Net Core
- La función `WebApplication.CreateBuilder` crea la aplicación ASP.NET Core.
- Con `app.Run()` se ejecuta la aplicación

```
1  var builder = WebApplication.CreateBuilder(args);  
2  var app = builder.Build();  
3  
4  app.MapGet("/", () => "Hello World!");  
5  
6  app.Run();  
7
```

Project NetCore Estructura

appsettings.json

- Fichero de configuración de las variables de entorno de la app.
- Se pueden tener tantas configuraciones como entornos tengamos añadiendo:
 - appsettings.<environment>.json

```
var secret = builder.Configuration["Secret"];  
app.MapGet("/secret", () => secret);
```

```
builder.Configuration.AddJsonFile("appsettings.json", optional: false, reloadOnChange: false)  
                      .AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.json", optional: false, reloadOnChange: true);  
  
var secret = builder.Configuration["Secret"];  
  
app.MapGet("/secret", () => secret);
```

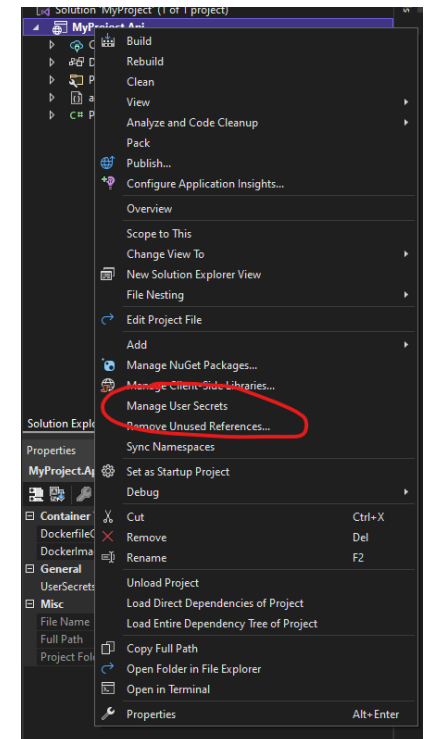
Project NetCore Estructura

UserSecrets

```
builder.Configuration.AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.json", optional: false, reloadOnChange: true)
builder.Configuration.AddJsonFile("appsettings.json", optional: false, reloadOnChange: false)
builder.Configuration.AddUserSecrets<Program>();
```

- Sobrecribir en appsettings solo en local
- Se habilitan con el botón derecho en el proyecto
 - Botón Derecho => Manage User Secrets
- Se crea en el fichero .csproj una variable de user secrets que apunta a un json que se aloja en nuestra carpeta AppData del sistema operativo

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <UserSecretsId>a253dcd0-8f57-4ad1-9e1d-5886384e4cd3</UserSecretsId>
  </PropertyGroup>
</Project>
```



Project NetCore Estructura

Variables de Entorno

```
builder.Configuration.AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.json", optional: false, reloadOnChange: true)
    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: false)
    .AddUserSecrets<Program>()
    .AddEnvironmentVariables();
```

- Las variables de entorno, son las variables que utilizaremos para cambiar nuestra aplicación dependiendo del Entorno que queramos
 - Así con una poquita configuración y sin tocar código podemos desplegar en entornos diferentes
- Se editan en el launchsetting
 - Como ASPNETCORE_ENVIRONMENT
- Se pueden sobrescribir en el Appsettings

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:62566",
      "sslPort": 44301
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "ApiStructure": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "Version": "3.2"
      },
      "applicationUrl": "https://localhost:9009;http://localhost:5000"
    }
  }
}
```

App

**MapGet, MapPut, MapPost,
MapDelete**

```
app.MapGet("/students", () => students);
```

```
app.MapPut("/students/{id}", (int id) => //  
Change student);
```

```
app.MapPost("/students", () => // Create  
student);
```

```
App.MapDelete("/student/{id}", (int id) => //  
Delete a student)
```

Ejercicios Program

Ejercicios de Program dentro del módulo de Fundamentos de NetCore



Inyección de dependencias

Inyección de Dependencias. Es una técnica para desacopar los servicios que usamos en nuestro sistema

- En Net Core podemos registrar un servicio por interfaz
 - Esto tiene la ventaja de trabajar con interfaces cumpliendo la I de SOLID
- Los servicios registrados luego pueden inyectarse en cualquier constructor de clase.

```
2 references
public class StudentsService : IStudentService
{
    private readonly List<Student> _students;
    private readonly ITimeServiceScoped timeServiceScoped;

    0 references
    public StudentsService(ITimeServiceScoped timeService)
    {
        this._students = new List<Student>();
        this.timeServiceScoped = timeService;
    }
}
```

```
7 builder.Services.AddSingleton<IStudentService, StudentsService>();
8 builder.Services.AddSingleton<ITimeServiceSingleton, TimeService>();
9 builder.Services.AddScoped<ITimeServiceScoped, TimeService>();
10 builder.Services.AddTransient<ITimeServiceTransient, TimeService>();
```

```
app.MapGet("/students/{id}", (HttpContext http, int id, IStudentService studentService) =>
{
    var student = studentService.GetStudent(id);
    if (student != null)
    {
        return student;
    }
    else
    {
        http.Response.StatusCode = 404;
        return null;
    }
});
```

Inyección de dependencias

Inyección de Dependencias. Los servicios se pueden registrar de tres formas diferentes

- **Transient:** Servicio creado siempre.
- **Scoped:** Servicio creado por petición.
- **Singleton:** Servicio creado una vez, y su instancia se mantiene durante toda la ejecución.

```
Llamada1:          singleton: 12/12/2021 06:45:28; scoped: 12/12/2021 06:45:50; transient: 12/12/2021 06:45:50  
Llamada 2 seg latencia: singleton: 12/12/2021 06:45:28; scoped: 12/12/2021 06:45:50; transient: 12/12/2021 06:45:52
```

Ejercicios Inyección Dependencias

Ejercicios de Inyección Dependencias dentro del módulo de Fundamentos de NetCore



IOptions, IOptionsSnapshot

AppSettings con inyección de dependencias

- Para usar los interfaces IOptions se puede configurar un objeto como settings de la app

```
builder.Services.Configure<Settings>(builder.Configuration);  
builder.Services.AddSingleton<IStudentService>, StudentService();
```

- Luego podemos inyectar los interfaces IOptions para usarlos
 - Con IOptionsSnapshot tenemos la opción de poder cambiar las settings en caliente

```
app.MapGet("/secretupdate", (IOptionsSnapshot<Settings> options) => options.Value.Secret);
```

Patrón Builder

El patron de diseño Builder separa la creación de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear representaciones diferentes.

- Es típico usarlo en las configuraciones de Net Core, ya que al final queremos una configuración que pase por diferentes pasos, configurando cada uno de ellos y el orden en el que van.

```

1 class Program
2 {
3     static void Main(string[] args)
4     {
5         var pizzaPersonalizada = PizzaFluentBuilder.Crear()
6             .ConMasaSuave()
7             .ConSalsaRoquefort()
8             .AñadirMozzarella()
9             .AñadirParmesano()
10            .Cocinar();
11     }
12 }

```

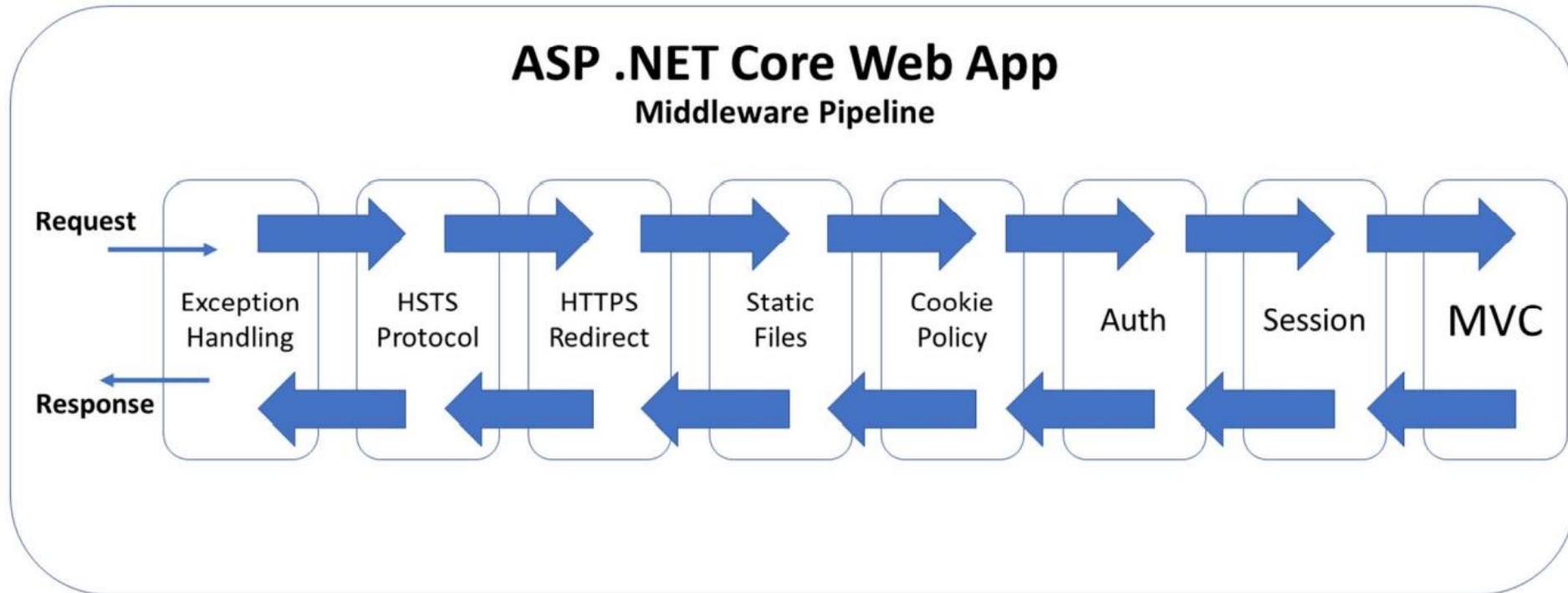
```

1 public class PizzaFluentBuilder
2 {
3     private readonly Pizza _pizza;
4
5     public static PizzaFluentBuilder Crear()
6     {
7         return new PizzaFluentBuilder();
8     }
9
10    private PizzaFluentBuilder()
11    {
12        _pizza = new Pizza();
13    }
14
15    public PizzaFluentBuilder ConMasaSuave()
16    {
17        _pizza.Masa = "Suave";
18        return this;
19    }
20
21    public PizzaFluentBuilder ConMasaCocida()
22    {
23        _pizza.Masa = "Cocido";
24        return this;
25    }
26
27    public PizzaFluentBuilder ConSalsaRoquefort()
28    {
29        _pizza.Salsa = "Roquefort";
30        return this;
31    }
32
33    public PizzaFluentBuilder ConSalsaPicante()
34    {
35        _pizza.Salsa = "Picante";
36        return this;
37    }
38 }

```

Middleware

Middleware



Middlewares

Middlewares

- El Middleware es un a pieza que se ensambla en una pipeline de una aplicación para controlar las solicitudes y las respuestas.
 - Decidir si pasa la ejecución al siguiente Middleware con la función next().
 - Realizar tareas antes y después de la ejecución de los demás Middleware.

```
app.Use(async (context, next) => {
    var log = context.RequestServices.GetService<ILogger>();

    log.LogDebug("Before the call");
    await next();
    log.LogDebug("After the call");
})
.UseMiddleware<CustomMiddlewares>();

app.Run();
```

```
1 reference
public class CustomMiddlewares
{
    private readonly RequestDelegate _next;

    0 references
    public CustomMiddlewares(RequestDelegate next)
    {
        _next = next;
    }

    0 references
    public async Task Invoke(HttpContext context, IFormatLanguage formatLanguage)
    {
        if (context.Request.Headers.ContainsKey("x-format"))
        {
            var headerValue = context.Request.Headers["x-format"];
            formatLanguage.SetLanguage(headerValue);
        }

        await _next.Invoke(context);
    }
}
```

ASP.NET Controladores

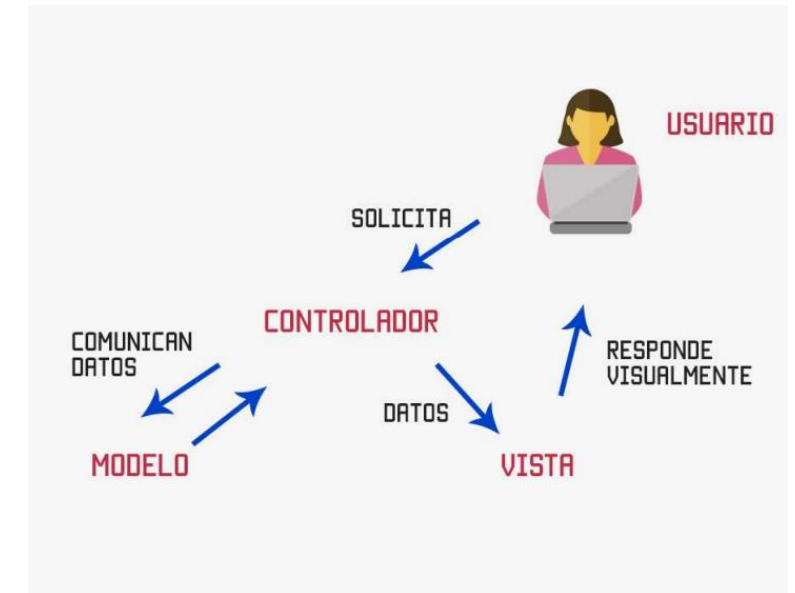
ASP.NET Core –
Controladores,
Patrón MVC



Patrón MVC

Patrón MVC: Su fundamento es la separación del Código en tres capas diferentes, acotadas por su responsabilidad

- M: Modelo. Es la capa donde se definen los datos
- V: Vista. Es la visualización de nuestra respuesta.
- C: Controlador. Contiene el código necesario para responder a las acciones que se solicitan en la aplicación



Controladores

Controladores: Son la C del modelo MVC. Son la entrada de nuestra API. Para añadirlos al pipeline hay que añadir en servicios y config:

Program.cs

```
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddControllers();  
  
var app = builder.Build();  
  
app.MapControllers();  
  
app.Run();
```

Controladores

Controladores. Reglas básicas

- Deben heredar de la clase ControllerBase
- Tienen que tener el atributo Route, donde será la ruta base de todas las llamadas
- Cada método del controlador tendrá un atributo `Http<Action>("<route>")` para indicar la ruta de cada método.
- Las rutas de los métodos junto con su verbo deben ser únicas.

```
[Route("products")]  
0 references  
public class ProductsController : ControllerBase  
{  
    [HttpGet]  
    0 references | 0 requests | 0 exceptions  
    public async Task<ActionResult> GetProduct()  
    {  
        return Ok( new Product  
        {  
            Id = 1,  
            Name = "Producto1",  
            Price = 15  
        });  
    }  
}
```

Controladores

Controladores. Respuestas

- Todos los métodos deben devolver un IActionResult
- Para devolver el código de estado correspondiente junto con el objeto que queremos enviar en la Response, podemos emplear los métodos heredados de ControllerBase como
 - Ok()
 - NotFound()
 - ServerError()
 - Created()

```
[Route("/api/students")]
0 references
public class StudentsController : ControllerBase
{
    [HttpGet("{productId}")]
    0 references
    public async Task<IActionResult> Get([FromRoute] int productId)
    {
        if (productId <= 0)
        {
            return NotFound();
        }
        return Ok(new Product(productId, "Prueba", 12));
    }
}
```

Controladores

Controladores. Enrutado

- La ruta principal de un controlador se pone en el atributo Route encima de la clase
- La ruta de cada uno de los métodos va en el atributo Http<verb> con la ruta en el propio atributo
 - HttpGet("/students")
- Los parámetros que recibe una ruta pueden ser:
 - Parámetros de ruta: Se usa el atributo [FromRoute]
 - Podemos ponerles validaciones
 - Parámetros del QueryString: Atributo [FromQueryString]
 - Parámetros del Body: Atributo [FromBody]
 - Incluso si necesitamos el valor de una cabecera podemos usar el atributo [FromHeader]

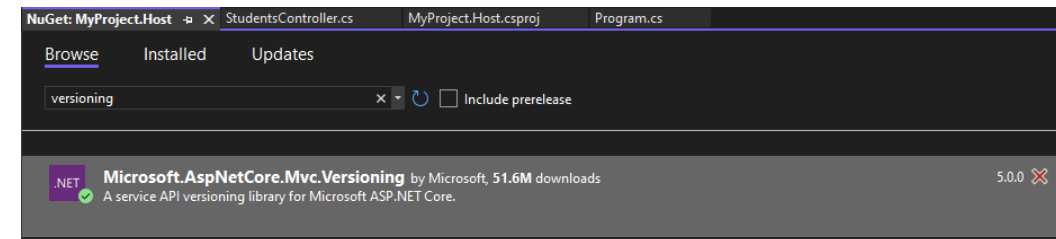
```
[Route("/api/students")]
public class StudentsController : ControllerBase
{
    [HttpGet("{productId:int}")]
    public async Task<IActionResult> Get([FromRoute] int productId)
    {
        if (productId <= 0)
        {
            return NotFound();
        }

        return Ok(new Product(productId, "Prueba", 12));
    }
}
```

Controladores

Controladores. Versionado. Para añadir el versionado tenemos que instalar el paquete Nuget: Microsoft.AspNetCore.Mvc.Versioning

- Después como cualquier servicio, debemos registrarlo.
- Por último, el controlador puede añadir el versionado correspondiente.
 - Puede existir un controlador por versión
 - Hay muchas estrategias de versionado, en el ejemplo lo hemos hecho por ruta, pero se podría hacer por QueryString o en una cabecera.

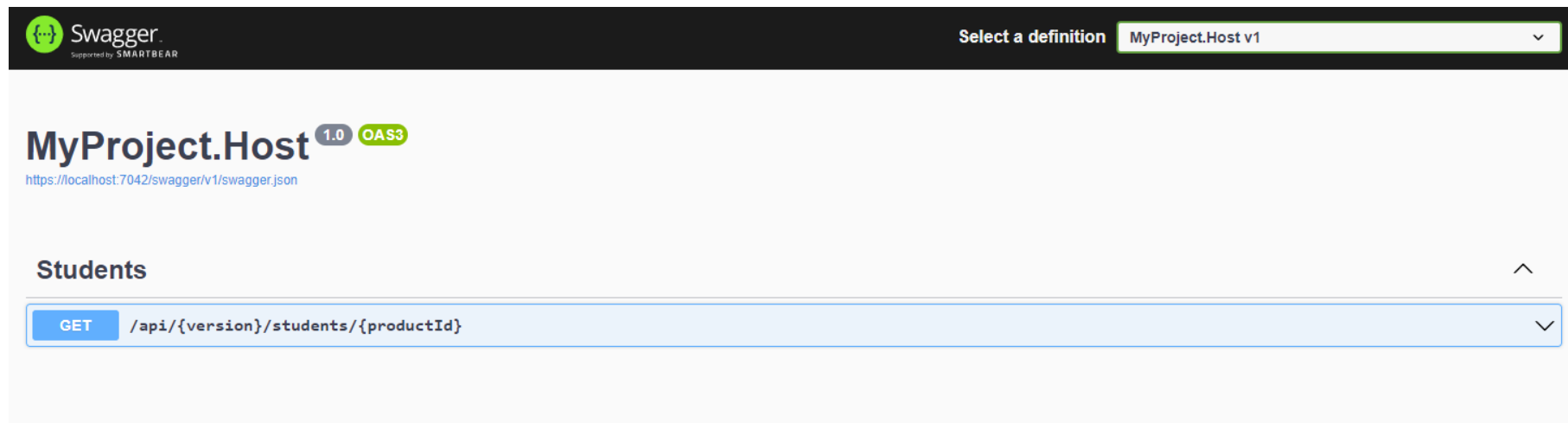


```
builder.Services.AddApiVersioning(options => { options.ReportApiVersions = true; })  
                .AddControllers();
```

```
[Route("/api/{version:apiVersion}/students")]  
[ApiVersion("1.0")]  
0 references  
public class StudentsController : ControllerBase  
{  
    [HttpGet("{productId:int}")]  
    0 references  
    public async Task<IActionResult> Get([FromRoute] int productId)  
    {  
        if (productId <= 0)  
        {  
            return NotFound();  
        }  
  
        return Ok(new Product(productId, "Prueba", 12));  
    }  
}
```

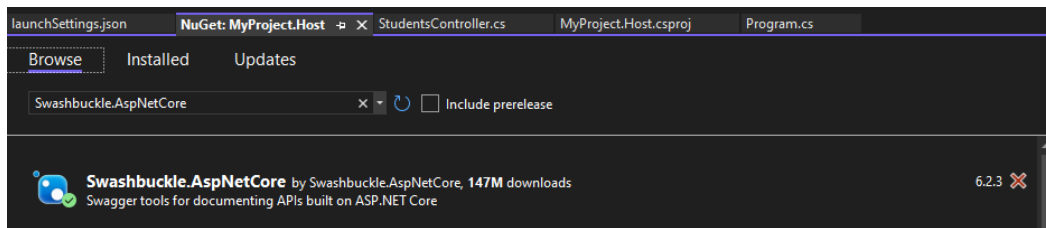
Swagger

Swagger es una herramienta que nos permite ver el API y poder consumirlo directamente desde la web. Sirve para documentar APIs



Swagger

Para configurar **Swagger**, como todos los servicios, tenemos que añadir sus Servicios y su Configuración en el Pipeline, instalando antes el paquete Swashbuckle.AspNetCore



```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddApiVersioning(options => { options.ReportApiVersions = true; })
                .AddSwaggerGen()
                .AddControllers();

var app = builder.Build();

app.UseSwagger()
    .UseSwaggerUI();

app.MapControllers();

app.Run();
```


Swagger

En los controladores se puede configurar el tipo de salida que pueden tener, para que Swagger lo añada a su documentación, con el atributo `ProducesResponseType`, a la que le podemos pasar el tipo devuelto y su Código de estado

```
[Route("/api/{version:apiVersion}/students")]
[ApiVersion("1.0")]
0 references
public class StudentsController : ControllerBase
{
    [HttpGet("/{productId:int}")]
    [ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]
    [ProducesResponseType((int)HttpStatusCode.NotFound)]
    0 references
    public async Task<IActionResult> Get([FromRoute] int productId)
    {
        if (productId <= 0)
        {
            return NotFound();
        }

        return Ok(new Product(productId, "Prueba", 12));
    }
}
```

Responses		
Code	Description	Links
200	Success	No links
	Media type text/plain	
	Controls Accept header.	
	Example Value Schema	
	<pre>{ "id": 0, "name": "string", "categoryId": 0 }</pre>	
404	Not Found	No links

Ejercicios Controladores

Ejercicios de Controladores dentro del módulo de ASPNET - Controladores



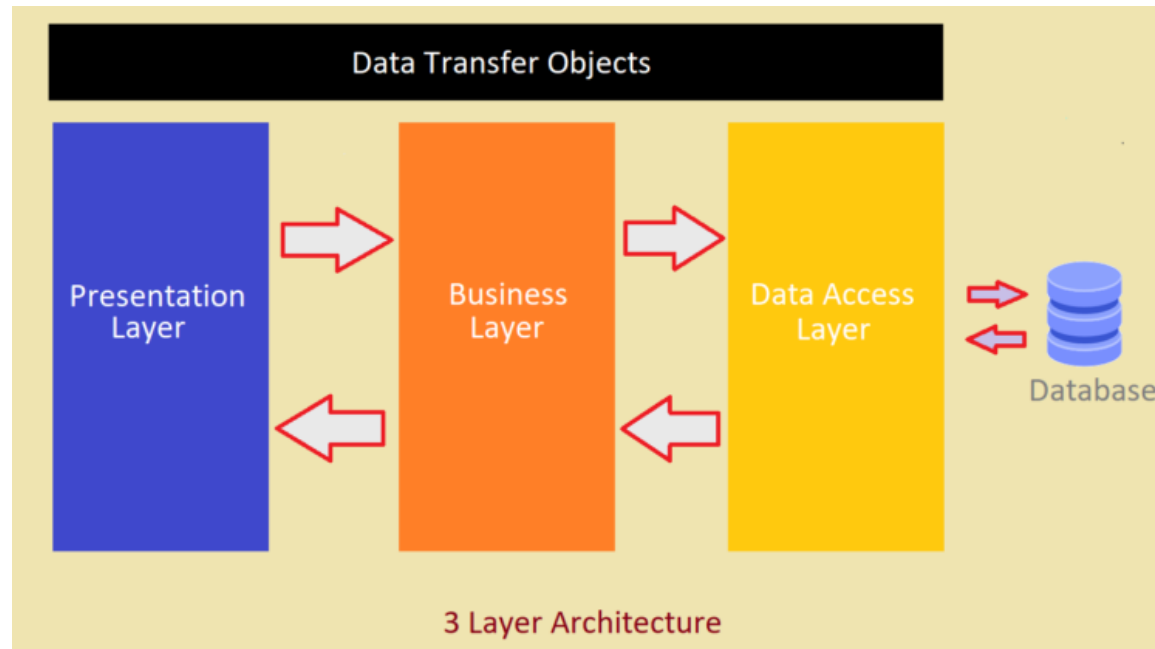
Arquitectura de proyecto

ASP.NET Core –
Arquitectura de
proyectos.



Arquitectura por capas

La **arquitectura por capas** implica dividir nuestro código en proyectos con las determinadas capas.



Arquitectura por capas

La **capa de presentación** corresponde al API, donde se encuentran nuestros controladores.

- Es el interfaz con el que interactúa el usuario
 - Conjunto de endpoints
- Devuelve lo que exponemos hacia fuera
 - Resultado en json + Código de respuesta

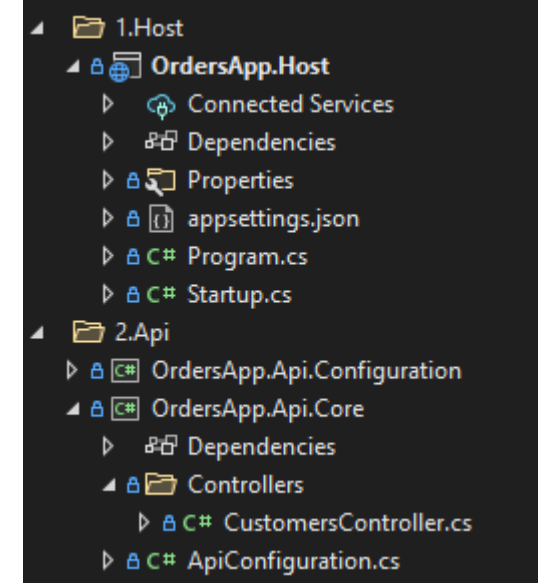
Book		▼
GET	/api/app/book/{id}	
PUT	/api/app/book/{id}	
DELETE	/api/app/book/{id}	
GET	/api/app/book	
POST	/api/app/book	

Reestructuración HOST-API

HOST: Es todo lo relativo a la ejecución de nuestro proyecto

API: El conjunto de endpoints que tiene nuestro proyecto

- HOST – contiene toda la configuración de Program.cs (y Startup en caso de que esté)
- API – es el Proyecto que tiene los controladores (y su config tiene que estar en este proyecto)



Arquitectura por capas

La **capa de negocio** es la que tiene todos los servicios que tienen implementado el problema de negocio que estamos resolviendo.

- Los controladores tienen que tener el Código mínimo y llamar a un servicio de esta capa
- El core de la implementación del negocio debe estar en las entidades de dominio

Modelos vs Entidades Dominio

Los Modelos son los recursos que se devuelven o se reciben por parte del API
Los Objetos de Dominio son los objetos internos de nuestro sistema

- Modelos (capa presentación)
 - Request
 - Response
 - DTO (Data Transfer Object)
- Entidades de Dominio (capa negocio)
 - Clases que contienen el Código del negocio del problema que estamos resolviendo
 - Son el core de nuestra solución.

Modelos vs Entidades Dominio

```

10 references | 0 changes | 0 authors, 0 changes
public class Customer
{
    private const int OVER_AGE = 18;

    4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int Id { get; private set; }
    4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string Name { get; private set; }
    4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string Surname { get; private set; }
    4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int Age { get; private set; }
    4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string Email { get; private set; }
    3 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Gender Gender { get; private set; }
    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public DateTime LastUpdate { get; private set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public ICollection<Order> Orders { get; set; }

    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Customer(string name, string surname, int age, string email, Gender gender)
    {
        1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
        Name = name;
        Surname = surname;
        Age = age;
        Email = email;
        Gender = gender;
        LastUpdate = DateTime.Now;
    }

    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public bool IsOverAge() { return Age >= OVER_AGE; }
}

```

```

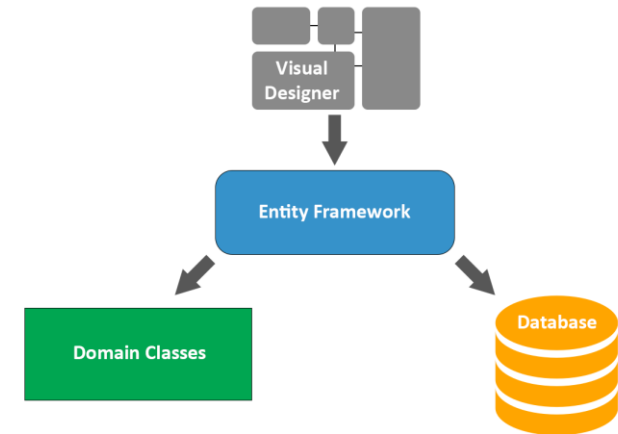
17 references | 0 changes | 0 authors, 0 changes
public class CustomerDto
{
    1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int Id { get; set; }
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string Name { get; set; }
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string Surname { get; set; }
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int Age { get; set; }
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string Email { get; set; }
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Gender Gender { get; set; }
}

```

Arquitectura por capas

La **capa de datos** es la encargada de persistir la información en el sistema de almacenaje que tengamos, normalmente una base de datos.

- Esta capa de suele dividir en:
 - Querys (consultas)
 - Commands (actualizaciones)
- Aquí entra en juego la configuración con Entity Framework y los ORMs



Test en NetCore con XUnit

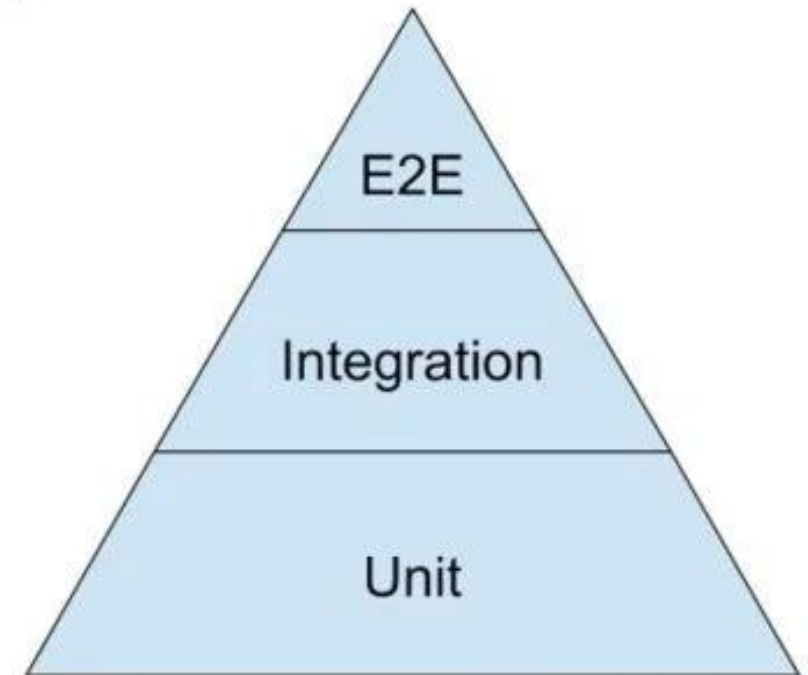
Test con Net Core

xUnit.net

Tipos de Tests

Piramide de Cohn

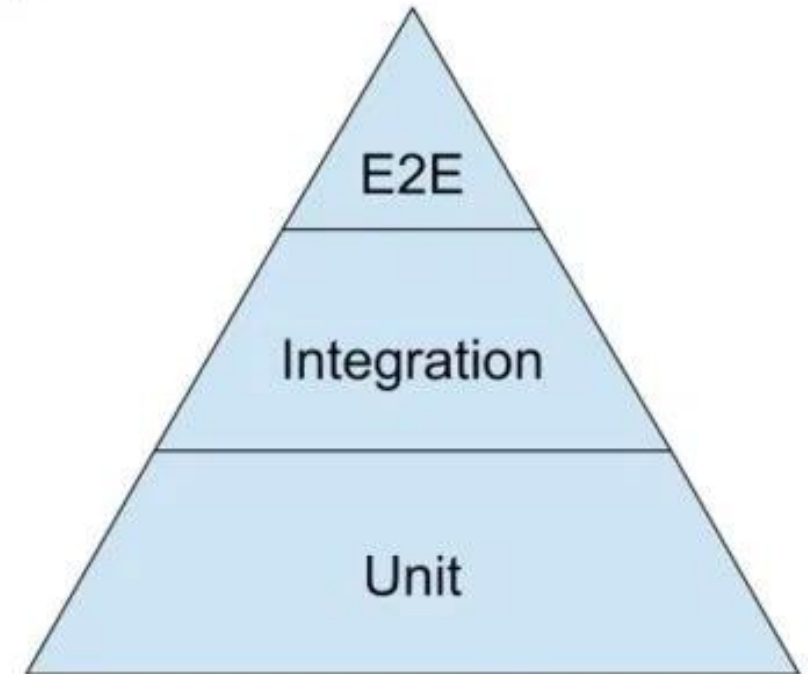
- **Unitarios:** se hacen a una clase concreta
- **Integración:** se hacen a un Sistema, con dependencias externas.
- **E2E:** end to end, a la aplicación completa
 - Backend + frontend



Tipos de Tests

Piramide de Cohn

- Grueso del Proyecto => Test Unitarios
- Seguido de Test de Integración
- Poner menos esfuerzo en los test E2E



Tipos de Tests

El trofeo del Testing

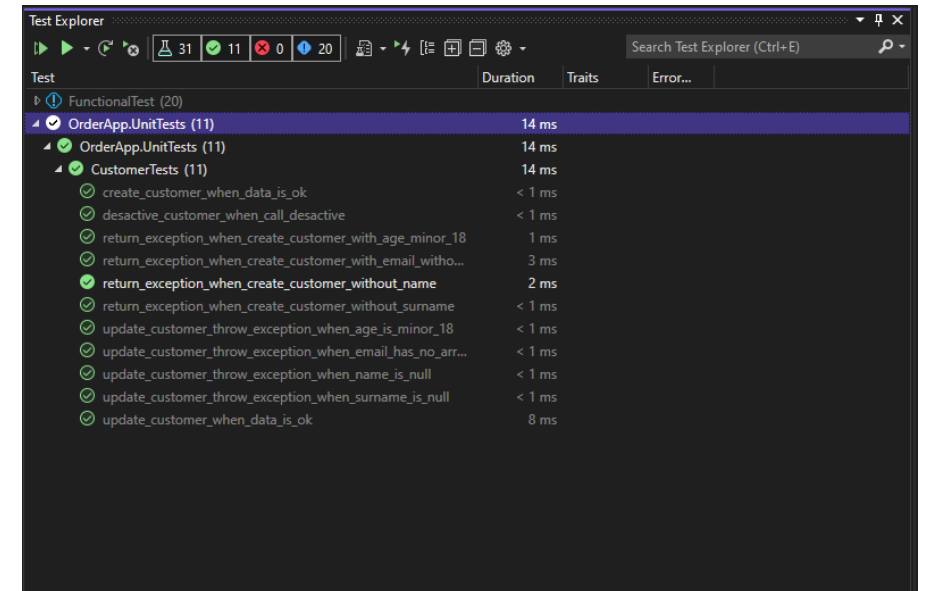
- **Static:** linter o similar.
- Escriba **pruebas unitarias** efectivas que apunten al comportamiento crítico y la funcionalidad de su aplicación.
- Desarrolle **pruebas de integración** para auditar su aplicación de manera integral y asegurarse de que todo funcione correctamente en armonía.
- Cree pruebas funcionales de extremo a extremo (e2e) para realizar pruebas de clics automatizadas



Test Unitarios

Test Unitarios => Los tests unitarios deberíamos hacerlo a la capa de dominio de nuestra aplicación: Objetos de dominio y servicios

- Los servicios de dominio son buenos candidatos para los tests unitarios
- Los objetos de dominio deben tener tests unitarios



Test Unitarios

Test Unitarios => Regla AAA

- Arrange: Preparar
- Act: Actuar
- Assert: Afirmar

Unit Testing

Arrange

Act

Assert

Test Unitarios

Test Unitarios => Regla AAA

- Arrange: Preparar
- Act: Actuar
- Assert: Afirmar

```
[Fact]
0 references
public void create_customer_when_data_is_ok()
{
    // Arrange
    var id = 1;
    var name = "Pepe";
    string surname = "Lopez";
    var age = 23;
    var email = "email@email.com";

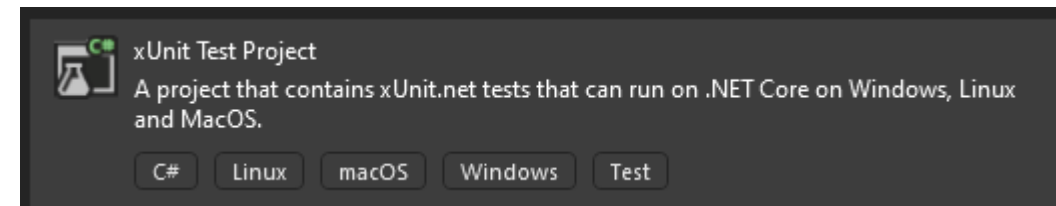
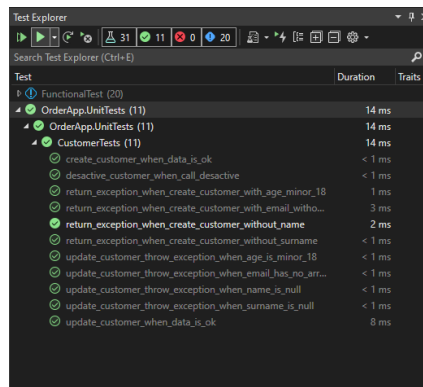
    // Act
    var customer = new Customer(id, name, surname, age, email);

    // Assert
    customer.Id.Should().Be(id);
    customer.Name.Should().Be(name);
    customer.Surname.Should().Be(surname);
    customer.Age.Should().Be(age);
    customer.Email.Should().Be(email);
    customer.Active.Should().Be(true);
}
```

Crear Proyecto Test

xUnit: framework para poder hacer test con NetCore

- Hay que marcar el método con el Atributo [Fact]
 - Con esto VS ya nos muestra los test en la Ventana TestExplorer



```
0 references
public class CustomerTests
{
    [Fact]
    public void return_exception_when_create_customer_without_name()
    {
        var id = 1;
        string name = null;
        var surname = "Lopez";
        var age = 23;
        var email = "email@email.com";

        Action createCustomer = () => new Customer(id, name, surname, age, email);

        Assert.Throws<CustomerConfigurationException>(createCustomer);
    }
}
```

Ejercicios TestUnitarios

Ejercicios de TestUnitarios dentro del módulo de ASP Net Core Tests.



Test de Integración

Fixture: Una de las tareas que consumen más tiempo a la hora de crear tests es escribir el código que configura el sistema en un estado conocido. Este estado conocido se le llama fixture

- La separación entre los proyectos de Host y API nos va a permitir configurar un Host para Test, añadiendo los servicios del API
 - Para ello nos ayudaremos de la clase TestServer



Test de Integración

Program.cs de test

- Configuramos un Program para test, eliminando todo lo que no nos hace falta para los tests
 - Servicios como Swagger no nos van a hacer falta
 - En cambio hay que añadir la configuración de la API, que tenemos en el proyecto de API.

```
using Microsoft.AspNetCore.Builder;
using OrderApp.Api;

var builder = WebApplication.CreateBuilder(args);

builder.Services.ConfigureServices();

var app = builder.Build();

app.UseApi()
    .UseRouting()
    .UseEndpoints(endpoints => endpoints.MapApiEndpoints());

app.Run();

2 references
public partial class Program { }
```

Test de Integración

TestServer

- Creamos nuestro TestServer basándonos en el Program de Test que hemos creado

```
namespace FunctionalTest
{
    2 references
    public class TestServerBuilder
    {
        2 references
        public TestServer Build()
        {
            var application = new WebApplicationFactory<Program>();

            return application.Server;
        }
    }
}
```

Test de Integración

Fixture

- Finalmente creamos el Fixture, que tiene que cumplir IDisposable.
 - En el Fixture creamos el Server que vamos a usar después en los Tests.
 - Para usar este Fixture en xUnit hay que añadirlo a una Colección
 - Usando esta colección en la clase de los test, tendremos accesible el Fixture
 - El Fixture se puede inyectar en el Test, una vez puesto el atributo de Collection en la clase.



Test de Integración

Fixture + Collection + TestClass

```

12 references
public class TestHostFixture : IDisposable
{
    private static IServiceProvider Services;

    26 references | 0/20 passing
    public TestServer Server { get; set; }

    0 references
    public TestHostFixture()
    {
        Server = new TestServerBuilder().Build();

        Services = Server.Services;
    }

    0 references
    public void Reset()
    {
        Server = new TestServerBuilder().Build();
    }

    1 reference
    public static void ResetCustomersService()
    {
        // Reset Customer Service
        var customerService = Services.GetService<ICustomerService>();
        var customersTask = customerService.GetAll();
        customersTask.Wait();
        var customer = customersTask.Result;
        customer.ToList().ForEach(customer => customerService.Delete(customer.Id));
    }

    0 references
    public void Dispose()
    {
        Server.Dispose();
    }
}

```

```

[CollectionDefinition(TestConstants.TestCollection)]
0 references
public class OrderAppCollection : ICollectionFixture<TestHostFixture>
{
}

```

```

[Collection(TestConstants.TestCollection)]

public class CreateCustomerScenario
{
    private readonly TestHostFixture _host;
    private readonly string _url;
    private readonly Fixture _autofixture;

    public CreateCustomerScenario(TestHostFixture host)
    {
        this._host = host;
        this._url = $"/api/customers";
        this._autofixture = new Fixture();
    }
}

```


Ejercicios TestIntegración

Ejercicios de TestIntegración dentro del módulo de ASP Net Core Tests.





Thank you

@plainconcepts

www.plainconcepts.com