

**.NET CORE**

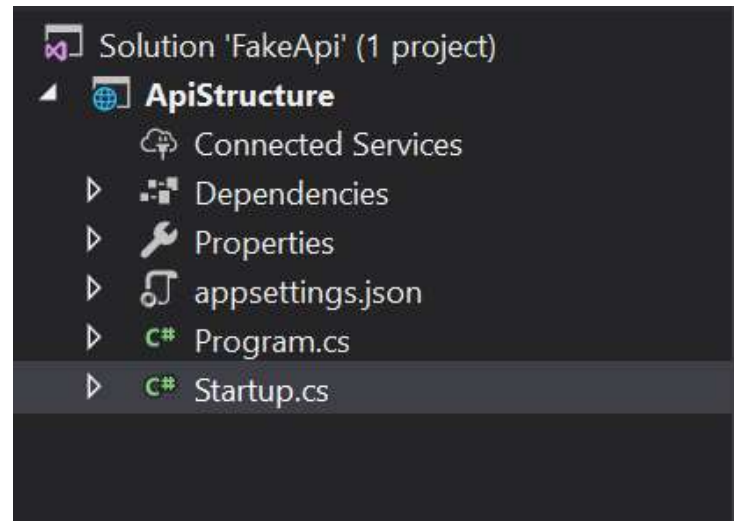
## **4. Fundamentos .NET Core**

- **Project structure**

# Configuracion .Net Core

## Project Structure

- Nuget Packages
- launchSettings.json
- Program.cs => Clean
- Appsettings.json
- Startup.cs



# Configuración .Net Core

## launchsetting.json

- Fichero de configuración de la ejecución en Visual Studio.
- Se pueden crear varios perfiles y elegir el Puerto de la aplicación.
- También se pueden setear las variables de entorno como por ej: "ASPNETCORE\_ENVIRONMENT": "Development"

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:57028",
      "sslPort": 44309
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "FakeApi": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

# Configuracion .Net Core

## Program.cs

- Al igual que en el Proyecto de consola, el punto de entrada de .Net Core es la función **Main** de la clase **Program**.
- La función `WebHost.CreateDefaultBuilder(args)` nos crea la aplicación ASP.NET MVC
- Con ***UseStartup*** le decimos cual es la **clase de configuración** de nuestra aplicación web

```
0 references
public class Program
{
    0 references | 0 exceptions
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    1 reference | 0 exceptions
    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

# Configuración .Net Core

## Startup.cs

- Es donde se va a configurar nuestra aplicación web
- Tiene dos métodos principales:
  - **ConfigureServices**: donde se configuran los servicios que va a usar nuestra aplicación
  - **Configure**: donde se configura el pipeline de la aplicación.
- Esta clase solo se ejecuta una vez para configurar nuestra aplicación al instanciarla.
- Tiene acceso a la Configuración y al Entorno de ejecución de nuestra app.

```
2 references
public class Startup
{
    1 reference
    public IHostingEnvironment Environment { get; private set; }
    2 references
    public IConfiguration Configuration { get; private set; }

    0 references
    public Startup(IConfiguration configuration, IHostingEnvironment env)
    {
        0 references
        ConfigureServices(configuration, env);
    }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        var serviceProvider = services.BuildServiceProvider();

        services.AddScoped<IFakeService, FakeService>();
        services.AddTransient<IRandomNumberService, RandomNumberService>();
        services.Configure<Configuration>(Configuration);
    }

    0 references
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

# Configuración .Net Core

## Map y MapWhen

- Map y MapWhen se pueden usar para registrar los diferentes pipelines dependiendo de la url donde se llama.
  - Con **Map** registramos directamente la acción que queremos ejecutar.
  - Con **MapWhen** podemos añadir condicionales a la acción que pongamos.

```
0 references | 0 exceptions
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    app.Map("/customers",
        builder => {
            builder.Run(async (context) =>
            {
                await context.Response.WriteAsync($"Hello customer!");
            });
        })
    .MapWhen(context => context.Request.Path.StartsWithSegments("/api"),
        builder => {
            builder.Run(async (context) =>
            {
                await context.Response.WriteAsync($"Hello api!");
            });
        })
    .Run(async (context) =>
    {
        logger.LogWarning("Se ha llamado al API");
        await context.Response.WriteAsync($"Hello world!");
    });
}
```

# Configuración .Net Core

## Inyección de Dependencias

- La Inyección de Dependencias es una técnica para desacoplar los objetos de los servicios que usamos en nuestro sistema.
- Para ello se registra un servicio por interfaz que cumpla, y así nuestras clases tirarán de interfaces en lugar de implementaciones.
- Los servicios luego pueden inyectarse en cualquier constructor de las clases de nuestro Sistema.

```
1 reference
public class Service
{
    private readonly IFakeService _fakeService;
    private readonly string author;

    0 references | 0 exceptions
    public Service(IFakeService fakeService)
    {
        _fakeService = fakeService;
    }

    0 references | 0 exceptions
    public string RandomNumber()
    {
        return $"Random: {_fakeService.GetRandomNumber()}.";
    }
}
```

```
6 references
public interface IFakeService
{
    3 references | 0 exceptions
    int GetRandomNumber();
}

2 references
public class FakeService : IFakeService
{
    private int randomNumber;
    0 references | 0 exceptions
    public FakeService()
    {
        var rnd = new Random();
        this.randomNumber = rnd.Next(100);
    }

    3 references | 0 exceptions
    public int GetRandomNumber()
    {
        return this.randomNumber;
    }
}
```

```
0 references | 0 exceptions
public Startup(IConfiguration configuration, IHostingEnvironment env)
{
    Environment = env;
    Configuration = configuration;
}

0 references | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{
    var serviceProvider = services.BuildServiceProvider();

    services.AddScoped<IFakeService, FakeService>();
    services.AddTransient<IRandomNumberService, RandomNumberService>();
    services.Configure<Configuration>(Configuration);
}
```

# Configuración .Net Core

## Inyección de Dependencias

- Los servicios se pueden registrar de tres formas:
  - **Transient:** Servicio creado siempre.
  - **Scoped:** Uno por petición.
  - **Singleton:** Uno creado una vez y su estancia se mantendrá en todo el ciclo de ejecución de la aplicación.

LLADAMA 1 con números random:

```
1 TrasientNumber: 48.  
2 TrasientNumber2: 19.  
3 ScopedNumber1: 94.  
4 ScopedNumber2: 94  
5 SingletonNumber1: 65.  
6 SingletonNumber2: 65
```

LLADAMA 2 con números random:

```
1 TrasientNumber: 60.  
2 TrasientNumber2: 22.  
3 ScopedNumber1: 31.  
4 ScopedNumber2: 31  
5 SingletonNumber1: 65.  
6 SingletonNumber2: 65
```



# Configuración .Net Core

## AppSettings

- El fichero **appsettings.json** es la configuración de nuestro entorno (antiguo Web.config). Puede haber tantos por entorno como queramos, renombrándolo a appsettings.development.json por ejemplo.
- Podemos mapear el resultado de este fichero a un objeto para poder usarlo después.
- También podemos configurar si es obligatorio y si se carga al instante aunque la aplicación este ejecutandose. Aunque por defecto en .NET Core 2.2 ya viene configurado.

```
0 references
public class Program
{
    0 references | 0 exceptions
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    1 reference | 0 exceptions
    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .ConfigureAppConfiguration((hostingContext, configuration) =>
            {
                configuration
                    .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
                    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
                    .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.EnvironmentName}.json", optional: true, reloadOnChange: true);
            })
            .Build();
}
```

# Configuración .Net Core

## AppSettings

- Para configurar AppSettings contra un objeto, hay que crear una estructura de clases igual que el json.
- Luego hay que añadirlo a los servicios con la función Configure y la Configuración que Podemos coger del Startup

```
Schema: http://json.schemastore.org/appsettings
1  {
2    "Version": "2.2",
3    "AppInfo": {
4      {
5        "Company": "plainconcepts",
6        "Author": "aclopez"
7      }
8    }
9  }
```

```
2 references
public class Configuration
{
    0 references | 0 exceptions
    public string Version { get; set; }
    0 references | 0 exceptions
    public AppInfo AppInfo { get; set; }
}

1 reference
public class AppInfo
{
    0 references | 0 exceptions
    public string Author { get; set; }
    0 references | 0 exceptions
    public string Company { get; set; }
}
```

```
3 references
public class Startup
{
    1 reference | 0 exceptions
    public IHostingEnvironment Environment { get; private set; }
    2 references | 0 exceptions
    public IConfiguration Configuration { get; private set; }

    0 references | 0 exceptions
    public Startup(IConfiguration configuration, IHostingEnvironment env)
    {
        Environment = env;
        Configuration = configuration;
    }

    0 references | 0 exceptions
    public void ConfigureServices(IServiceCollection services)
    {
        var serviceProvider = services.BuildServiceProvider();

        services.AddTransient<IFakeService, FakeService>();
        services.AddScoped<IRandomNumberService, RandomNumberService>();
        services.AddSingleton<IService, Service>();
        services.Configure<Configuration>(Configuration);
    }
}
```

# Configuración .Net Core

## AppSettings

- Para usarlo podemos inyectar el interface IOptions. Si usamos IOptionsSnapshot habilitaremos el recargo del appsettings en caliente.

```
3 references
public class Startup
{
    1 reference | 0 exceptions
    public IHostingEnvironment Environment { get; private set; }
    3 references | 0 exceptions
    public IConfiguration Configuration { get; private set; }

    0 references | 0 exceptions
    public Startup(IConfiguration configuration, IHostingEnvironment env)
    {
        Environment = env;
        Configuration = configuration;
    }

    0 references | 0 exceptions
    public void ConfigureServices(IServiceCollection services)
    {
        var serviceProvider = services.BuildServiceProvider();

        services.AddTransient<IFakeService, FakeService>();
        services.AddScoped<IRandomNumberService, RandomNumberService>();
        services.AddSingleton<IService, Service>();
        services.Configure<Configuration>(Configuration);
    }

    0 references | 0 exceptions
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILogger<Startup> logger, IOptions<Configuration> options)
    {
        app.Run(async (context) =>
        {
            var version = Configuration.GetValue<string>("Version");

            await context.Response.WriteAsync($"El autor es: {options.Value.AppInfo.Author}. La version es {version}");
        });
    }
}
```

# Configuración .Net Core

## UserSecrets

- Para no subir configuración indebida a nuestro sistema de gestión de versiones se han creado las UserSecrets.
- Las UserSecrets no son más que una configuración que se guarda en nuestro equipo para poderlas usar en lugar del fichero appsettings correspondientes.
- Es un json más que sobrescriben nuestras appsettings.

```
1 reference | 0 exceptions
public class Program
{
    References | Exceptions
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

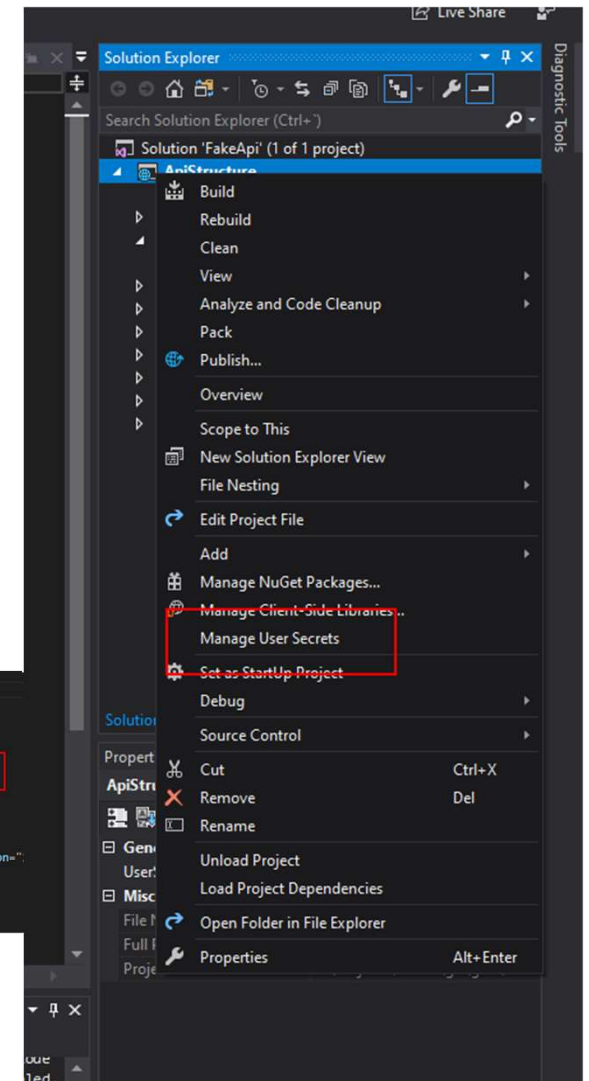
    1 reference | 0 exceptions
    public static IWebHost BuildWebHost(string[] args) =>
    {
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .ConfigureAppConfiguration((hostingContext, configuration) =>
            {
                configuration
                    .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
                    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
                    .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.EnvironmentName}.json", optional: true)
                    .AddUserSecrets<Program>();
            })
            .Build();
    }
}
```

```
Projects > C:\Users\> git > git2 > startup > FakeApi > ApiStructure > ApiStructure.csproj
[Project Sdk="Microsoft.NET.Sdk.Web"]

<PropertyGroup>
  <TargetFramework>netcoreapp2.2</TargetFramework>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  <UserSecretsId>cf86e264-5709-451f-92fe-ebdd96c2d6f2</UserSecretsId>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
  <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" />
</ItemGroup>

</Project>
```



# Configuración .Net Core

## Variables de Entorno

- Las Variables de Entorno son las variables que podemos usar en nuestra solución para por ejemplo indicar en que modo estamos. En el launchsettings teníamos seteada la variable de entorno ASPNETCORE\_ENVIRONMENT con el valor Development.
- Estas variables pueden sobrescribir nuestro appsettings también

```
1 reference
public class Program
{
    0 references | 0 exceptions
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    1 reference | 0 exceptions
    public static IWebHost BuildWebHost(string[] args) =>
    {
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .ConfigureAppConfiguration((hostingContext, configuration) =>
            {
                configuration
                    .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
                    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
                    .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.EnvironmentName}.json", optional: true)
                    .AddUserSecrets<Program>()
                    .AddEnvironmentVariables();
            })
            .Build();
    }
}
```

```
{
  "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"
    }
  }
}
```



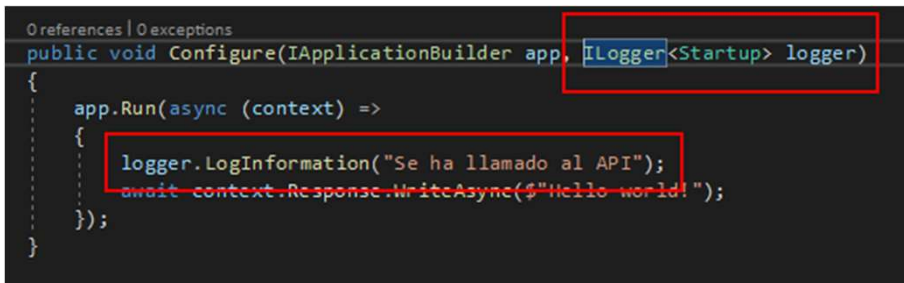
# Configuración .Net Core

## Logging

NET Core es compatible con una API de registro que funciona con una gran variedad de proveedores de registro integrados y de terceros:

- Debug
- Console
- EventSource
- EventLog
- TraceSource
- Azure App Service
- ...

<https://github.com/aspnet/logging>



```
0 references | 0 exceptions
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    app.Run(async (context) =>
    {
        logger.LogInformation("Se ha llamado al API");
        await context.Response.WriteAsync($"Hello world!");
    });
}
```

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                    optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            // Requires `using Microsoft.Extensions.Logging;`
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

# Configuración .Net Core

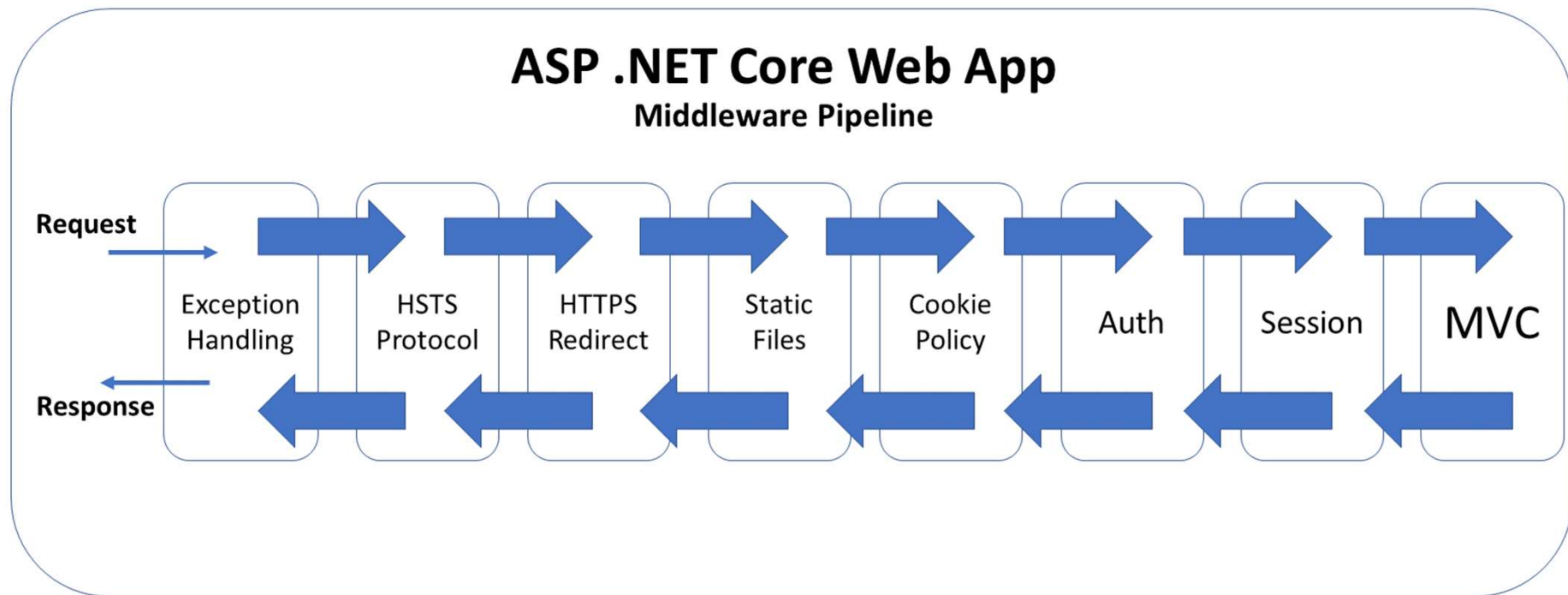
## Patrón Builder

- El **patrón 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





# Configuración .Net Core

## Middleware

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

```
0 references | 0 exceptions
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILogger<Startup> logger)
{
    app.Use(async (context, next) =>
    {
        logger.LogDebug("Use: antes de la llamada");
        await next();
        logger.LogDebug("Use: después de la llamada");
    })
    .UseMiddleware<CustomMiddleware>()
    .Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

```
2 references
public class CustomMiddleware
{
    private readonly RequestDelegate _next;
    0 references | 0 exceptions
    public CustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    0 references | 0 exceptions
    public async Task Invoke(HttpContext context, IFormatService formatService)
    {
        if (context.Request.Headers.ContainsKey("x-format"))
        {
            var headerValue = context.Request.Headers["x-format"];
            formatService.SetLanguage(headerValue);
        }

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