Introducción a middlewares.

****

- ¿Qué es un middleware?

En una Web API, un middleware es una clase o función que se ejecuta durante el ciclo de vida de una solicitud HTTP, ya sea antes o después de que esta llegue al manejador final (por ejemplo, un controlador o endpoint).
Entonces, mediante el uso de middlewares podremos interceptar y procesar/manipular las solicitudes HTTP entrantes (requests) y/o las respuestas (responses) salientes de nuestra aplicación.

¿Cuándo se ejecuta un middleware?

Esto depende del orden en el que registremos el middleware en el pipeline del framework. Así que dependiendo de dónde esté ubicado, se ejecutará antes o después que otros middlewares. Pero más allá de esto, lo más común es registrar el middleware de manera que éste se ejecute antes de que la request llegue al controlador correspondiente.

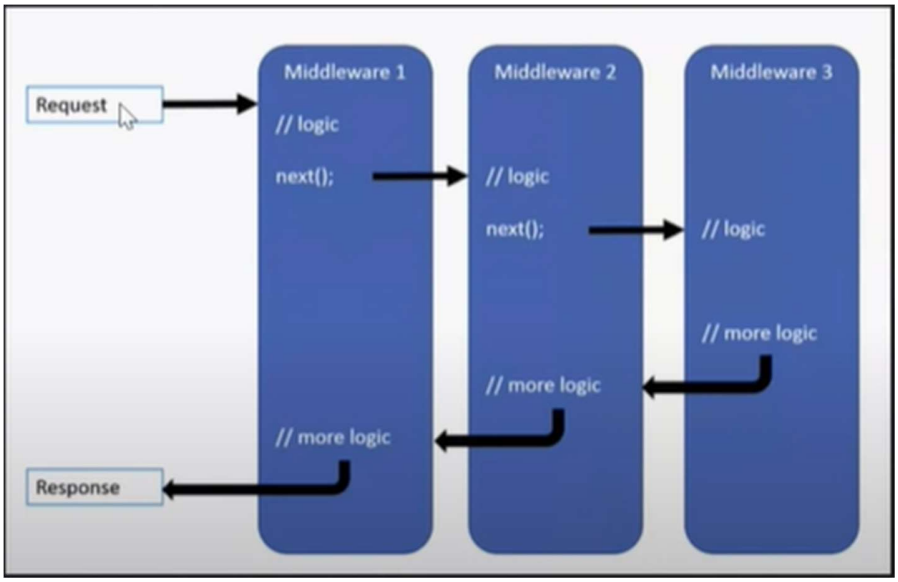
Algunos usos comunes para middlewares.

Al desarrollar una web API, independientemente del framework que se utilice, es común el uso de middlewares para tareas cómo:
- Autenticación y autorización;
- Registro de logs;
- Manejo global de errores;
- Validación de solicitudes;
- CORS (Cross-Origin Resource Sharing)

- ¿Qué es el pipeline en ASP.NET Core?

El pipeline (o tubería) se refiere a la secuencia de componentes de middleware que procesan una solicitud HTTP entrante y construyen una respuesta HTTP saliente. En esta sección de nuestra aplicación se define cómo se procesan las solicitudes HTTP de la API. Cada componente de middleware que la compone se enlaza o comunica con el siguiente en la lista conformando una secuencia o cadena de ejecución. Entonces la request entra a nuestra aplicación, pasa por todos los middlewares hasta llegar al último de la lista, y luego regresa y vuelve a recorrerlos en orden

inverso para retornar la response.



Dependiendo de la plantilla de proyecto que utilicemos, puede ser común ver que en el pipeline de ASP.NET ya vienen algunos middlewares registrados que el framework incorpora, pero nosotros también podemos crear nuestros propios middlewares personalizados (o custom middlewares) y registrarlos en el pipeline mediante la instrucción app.UseMiddleware<T>();
(NOTA: deberás reemplazar T por el nombre de la clase de tu custom middleware)

```
37
38      var app = builder.Build();
39
40      app.UseMiddleware<MyCustomMiddleware>();
41
42      // Configure the HTTP request pipeline.
```

Para crear nuestro propio custom middleware debemos crear una clase de C# que implemente la interfaz IMiddleware que provee el framework ASP.NET, y especificar las instrucciones de código que queremos que se ejecuten dentro de la función InvokeAsync().
Y luego registramos dicha clase en el pipeline con la instrucción que se menciona en el párrafo anterior,y en la ubicación que consideremos necesaria de acuerdo con la funcionalidad que provea nuestro middleware.

```
public class MyCustomMiddleware : IMiddleware
{
    //puedes agregar alguna prop privada y constructor si necesitas inyectar alguna dependencia

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        Console.WriteLine("Hola desde al nuevo middleware antes del controlador");
        await next(context);
        Console.WriteLine("Hola desde al nuevo middleware después del controlador");
    }
}
```

Versión donde inyectamos y utilizamos el Logger de .NET en lugar de Console.WriteLine():

```
public class MyCustomMiddleware : IMiddleware
{
    private readonly ILogger _logger;

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

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        _logger.LogInformation("Hola desde al nuevo middleware antes del controlador");
        await next(context);
        _logger.LogInformation("Hola desde al nuevo middleware antes del controlador");
    }
}
```

Y además recuerda agregar la clase del nuevo middleware al contenedor de servicios en la clase program:

```
27      #region Services
28      builder.Services.AddScoped<MyCustomMiddleware>();
```

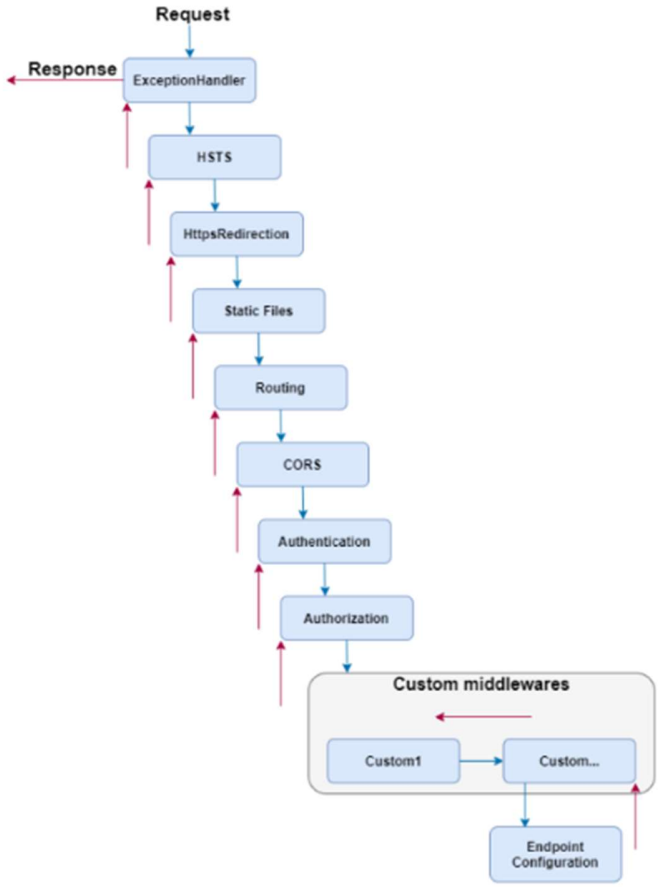
Orden de registro de un Middleware en el pipeline.

Al agregar algunos middlewares en el pipeline se conforma una cadena de funcionalidades que se ejecuta en orden. Cada middleware de la cadena podría realizar tareas como:
- Leer y procesar la solicitud entrante (HttpContext.Request);
- Pasar la request/response al siguiente middleware de la cadena (await next);
- Detener el flujo si fuese necesario (por ejemplo si la autenticación falla);
- Procesar la respuesta antes de devolverla al cliente (HttpContext.Response);

Los componentes de middleware se ejecutan en el orden en que los registramos y regresan en

orden inverso si llaman a next.
Así que es muy importante que pongamos atención al orden en el que registramos los middlewares en el pipeline dependiendo de la funcionalidad que aporte cada uno. Ya que si dicho orden no es correcto nuestra aplicación podría no funcionar correctamente. (por ejemplo, el middleware de autenticación debe ir siempre antes del middleware de autorización).

A continuación mostramos una secuencia recomendada para el orden de registración de los middlewares más comunes en el pipeline de ASP.NET en una API estándar:



Algunos middlewares incorporados en ASP.NET Core.

El IAplicationBuilder (objeto app) tiene algunos métodos que simplifican el registro de middlewares que ya vienen integrados con el framework, y que podemos utilizar para ciertas funcionalidades comunes de una Web API. Algunos de ellos son:

- UseAuthentication() -> Registra el middleware que valida el token de autenticación y establece el HttpContext.User
- UseAuthorization() -> Verifica si el usuario tiene permisos para acceder al recurso solicitado

(políticas, roles, etc).
- MapControllers() -> Mapea los controladores a rutas para que el framework sepa qué controlador usar según la URL.

Si bien el framework brinda estos métodos o helpers, no siempre vienen registrados en el pipeline. Por ejemplo, si creas una Minimal API, al no utilizar controladores, tendrás que agregar los controllers a mano y registrar estos métodos manualmente en el pipeline, mientras que si utilizas alguna otra plantilla de proyecto, puede que alguno ya venga registrado por defecto. Notarás que en la plantilla de Web Api con controladores estos métodos ya vienen registrados en el pipeline.

- ¿Qué es el objeto HttpContext?

Es un objeto que representa toda la información de la solicitud HTTP actual y la respuesta que se va a enviar al cliente. (Por eso el nombre, ya que representa el contexto de la solicitud actual)
Este objeto contiene todo lo que necesitamos saber sobre una petición HTTP, como por ejemplo la URL, los encabezados, el cuerpo, y también lo que vas a devolver como respuesta, como el código de estado y el contenido.

¿Por qué es importante este objeto?

Debemos conocer de su existencia, ya que se pasa implícitamente por todo el pipeline de middlewares y es la forma estándar en ASP.NET Core de acceder a la información de la petición y la respuesta. Así que podríamos leerlo y modificarlo desde cualquier parte del flujo para acceder o agregar información como headers, statusCodes, etc.

¿Cómo podemos acceder al objeto HttpContext?

- desde un middleware: normalmente los middlewares lo reciben por parámetro para poder manipularlo, y por convención se llama context.
- desde un controlador: mediante this.HttpContext

Se puede acceder a sus propiedades mediante notación de punto como cualquier otro objeto en C#.

Algunas propiedades importantes del objeto HttpContext.

- Request: Representa la solicitud HTTP entrante (contiene el método, headers, query string, body, etc);
- Response: Representa la respuesta HTTP que se va a enviar al cliente (contiene statusCode, body, etc);
- User: Información del usuario autenticado (ClaimsPrincipal);
- TraceIdentifier: es el identificador único de la solicitud (útil para logs);

Por ejemplo, podríamos acceder a algunos datos de la request de la siguiente manera:

```
var method = context.Request.Method;
var path = context.Request.Path;
var query = context.Request.Query["id"];
var headers = context.Request.Headers["User-Agent"];
```

Y también modificar la response así:

```
context.Response.StatusCode = 200;
await context.Response.WriteAsync("Hola mundo");
```

EXTRA:

Uso de Use(), Map() y Run() en el pipeline de ASP.NET:

Además de agregar middleares personalizados con app.UseMiddleware<>(); como ya vimos anteriormente, para construir el pipeline también se pueden utilizar métodos de extensión como “Use()”, “Map()”, y “Run()”, y en lugar de definir nuestro customMiddleware en una clase se puede poblar el pipeline con estos métodos y simples funciones dentro.

Los analizaremos a continuación:

#) app.Use():

El propósito de este método es agregar un componente de middleware al pipeline y permitirle pasar la solicitud al siguiente componente si se llama a await next().

Así que normalmente se utiliza para agregar lógica que debe ejecutarse tanto antes como después del siguiente middleware (por ej: autenticación, logs, CORS, etc).

Ejemplo:

```
var app = builder.Build();

app.Use(async (context, next) =>
{
    Console.WriteLine("Antes del siguiente middleware");
    await next.Invoke();
    Console.WriteLine("Después del siguiente middleware");
});
```

#) app.Map():

Se utiliza para dividir el pipeline en ramas según una ruta específica. Así que podemos establecer ramificaciones o bifurcaciones en el flujo de ejecución de nuestra aplicación mediante el uso de Map() en el pipeline.

Por ejemplo, podría usarse para tener rutas específicas con sus respectivos middlewares (/admin, /api, etc):

Ejemplo:

```
app.Map("/admin", adminApp =>
{
    adminApp.Run(async context =>
    {
        await context.Response.WriteAsync("Área de administración");
    });
});
```

#) app.Run():

Tiene como propósito definir un componente de middleware como terminal (último). Esto quiere decir que no se puede llamar a next(), por lo que corta el pipeline en este punto. Así que si en el pipeline colocas app.Run() entes de otros middlewares, éstos no se ejecutarán.

Ejemplo:

```
var app = builder.Build();

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

#) app.MapGet(), app.MapPost(), etc.:

Se utilizan para definer endpoints HTTP directamente en Minimal APIs. (muy convenientes cuando se trabaja sin controladores y se quiere manejar rutas específicas asociadas a un verbo HTTP determinado).

Ejemplo:

```
44    app.MapGet("/saludo", () => "¡Hola mundo!");
```

Resumen visual

Método	¿Pasa al siguiente?	¿Basado en ruta?	¿Basado en verbo HTTP?	¿Terminal?
`app.Use()`	✅	❌	❌	❌
`app.Run()`	❌	❌	❌	✅
`app.Map()`	✅ (si defines más)	✅	❌	Depende
`app.MapGet()`	❌ (maneja la ruta)	✅	✅ (`GET`)	✅