¿Qué es .NET?

.NET es una **plataforma de desarrollo gratuita, de código abierto y multiplataforma**, diseñada y mantenida por Microsoft junto con una comunidad global de desarrolladores. Permite la creación de una amplia gama de aplicaciones, tales como aplicaciones web, de escritorio, móviles, juegos, servicios en la nube, inteligencia artificial, IoT, entre otros.

.NET se caracteriza por ser una plataforma moderna y eficiente, pensada para maximizar la productividad del desarrollador sin sacrificar rendimiento ni confiabilidad. Su lenguaje principal es **C#**, aunque también soporta otros como F# y Visual Basic.

La plataforma proporciona un entorno de ejecución de alto rendimiento, un conjunto extenso de bibliotecas y herramientas modernas para facilitar la construcción, prueba, despliegue y monitoreo de aplicaciones.

Componentes fundamentales de .NET

1. Entorno de ejecución (Runtime)

El runtime de .NET se encarga de ejecutar las aplicaciones. Entre sus responsabilidades clave se incluyen:

- **Administración automática de memoria** mediante un recolector de elementos no utilizados (GC).
- **Seguridad de tipos**, lo que evita errores comunes en tiempo de ejecución relacionados con tipos de datos incorrectos.
- **Protección de la memoria**, impidiendo accesos indebidos.
- Soporte para **interoperabilidad con código nativo**, lo que permite invocar funciones de librerías externas escritas en C/C++.
- Posibilidad de compilar código a tiempo de ejecución (JIT) o anticipadamente (AOT).

2. Bibliotecas de clases (Base Class Library)

.NET incluye un amplio conjunto de bibliotecas estandarizadas que proporcionan funcionalidades esenciales:

- Manejo de cadenas de texto, fechas, archivos y flujos.
- Serialización y deserialización (por ejemplo, JSON).
- Acceso a bases de datos, trabajo con redes, criptografía y colecciones.
- APIs para tareas comunes en desarrollo web, escritorio, móvil, etc.

Estas bibliotecas están altamente optimizadas y expuestas a través de un modelo coherente, facilitando su aprendizaje y reutilización.

3. Compiladores

.NET incluye compiladores para múltiples lenguajes. El más relevante es **Roslyn**, el compilador de C#, que no solo compila el código fuente sino que también expone APIs para análisis, refactorización y otras herramientas avanzadas.

El código fuente se compila a un lenguaje intermedio (IL, Intermediate Language) que luego es ejecutado por el runtime mediante compilación JIT o AOT.

4. SDK y herramientas

El SDK de .NET proporciona herramientas modernas para el ciclo de vida completo del desarrollo:

- Creación de proyectos (`dotnet new`).
- Compilación y ejecución (`dotnet build`, `dotnet run`).
- Administración de dependencias con NuGet (`dotnet add package`).
- Pruebas automatizadas (`dotnet test`).
- Publicación (`dotnet publish`).

Estas herramientas funcionan de manera uniforme en todas las plataformas compatibles (Windows, Linux y macOS).

5. Pilas de aplicaciones

Sobre la base del runtime, las bibliotecas y los lenguajes, .NET ofrece stacks específicos para distintos tipos de aplicaciones:

- **ASP.NET Core**: Desarrollo web, APIs REST, aplicaciones en tiempo real (SignalR).

- **Windows Forms y WPF**: Interfaces gráficas en entornos Windows.
- **MAUI y Xamarin**: Aplicaciones móviles multiplataforma.
- **Blazor**: Aplicaciones web interactivas con C# en lugar de JavaScript.
- **Unity (con C#)**: Desarrollo de videojuegos.

Características clave de diseño

Productividad

.NET está pensado para maximizar la productividad del desarrollador. Esto se logra mediante:

- Herramientas integradas (IDE, CLI, depuración).
- Lenguaje expresivo (C#), que facilita escribir código conciso y legible.
- Amplio ecosistema de bibliotecas y documentación.

Rendimiento

.NET es utilizado en escenarios de producción a gran escala debido a su alto rendimiento:

- Uso eficiente de CPU y memoria.
- Recolector de basura moderno, adaptativo y configurable.
- Optimizaciones de código específicas para plataformas (x64, ARM, etc.).
- Compilación nativa con .NET Native y AOT en determinados escenarios.

Seguridad y confiabilidad

.NET incorpora múltiples mecanismos de seguridad:

- Seguridad de tipos estricta, evitando errores comunes.
- Administración automática de memoria para prevenir fugas y corrupción.
- Modelo robusto de manejo de excepciones que permite separar el código funcional del control de errores.
- APIs criptográficas modernas y fáciles de usar.

Programación asincrónica y concurrencia

.NET ofrece soporte de primera clase para tareas asincrónicas mediante el patrón **async/await**. Esto permite escribir código que realiza múltiples operaciones en paralelo de forma sencilla, sin bloquear el hilo principal.

El sistema de tareas (`Task`, `Task<T>`) y las estructuras como `ValueTask` permiten un control eficiente del paralelismo, crucial en aplicaciones que deben manejar muchas solicitudes simultáneas o procesamiento en segundo plano.

Sistema de tipos y orientación a objetos

.NET está basado en un sistema de tipos robusto que soporta:

- Programación orientada a objetos completa (herencia, polimorfismo, encapsulamiento).
- Interfaces con implementación por defecto.
- Tipos genéricos ampliamente usados y optimizados.
- Tipos de valor y tipos de referencia.
- Compatibilidad con reflexión (capacidad de inspeccionar tipos en tiempo de ejecución).

Ecosistema y variantes de .NET

A lo largo de los años, se han desarrollado distintas implementaciones de .NET:

- **.NET Framework**: Implementación original, diseñada exclusivamente para Windows. Se utiliza aún en muchas aplicaciones empresariales legadas. Se encuentra en mantenimiento, no en evolución activa.
- **Mono**: Iniciativa de código abierto que permitió llevar .NET a Linux, macOS, Android, iOS y WebAssembly. Se utiliza en Xamarin y es clave en el desarrollo móvil y de videojuegos.
- **.NET (antes .NET Core)**: Implementación moderna, modular, multiplataforma, y actualmente la versión principal de .NET. Su diseño está enfocado en la nube,

contenedores, microservicios y multiplataforma. Se actualiza de forma regular y es el futuro de la plataforma.

Administración de paquetes: NuGet

NuGet es el **sistema de gestión de paquetes oficial de .NET**. Permite compartir, consumir y actualizar librerías de terceros de manera sencilla. Contiene una galería en línea mantenida por Microsoft con cientos de miles de paquetes reutilizables.
A través del CLI o Visual Studio, se pueden instalar y gestionar dependencias fácilmente sin necesidad de configurar manualmente rutas o versiones.

Licenciamiento y comunidad

.NET es un proyecto de **código abierto** bajo licencia **MIT**, lo que garantiza su uso libre tanto comercial como académico. Está supervisado por la **.NET Foundation**, una organización sin fines de lucro que promueve su desarrollo abierto y sostenible.
La plataforma se desarrolla de forma colaborativa en GitHub, con múltiples repositorios gestionados por Microsoft y la comunidad.

Actualizaciones y soporte

Microsoft publica una **nueva versión mayor de .NET cada noviembre**, siguiendo un calendario predecible. Además, se realizan actualizaciones menores de seguridad y corrección de errores el **segundo martes de cada mes** (“martes de parches”).
Existen versiones con **soporte a largo plazo (LTS)**, que reciben actualizaciones durante varios años, ideales para entornos productivos donde la estabilidad es prioritaria.

Conclusión

.NET es una plataforma sólida, moderna y madura que permite desarrollar aplicaciones en una amplia variedad de dominios tecnológicos. Su enfoque en la productividad, rendimiento y seguridad, sumado a su capacidad multiplataforma y su fuerte comunidad, lo convierte en una de las mejores opciones para desarrolladores tanto en el ámbito empresarial como independiente.
Su arquitectura modular y escalable, junto con el soporte para patrones actuales como microservicios, contenedores y nube, garantiza su relevancia en los próximos años.
Además, el uso de C# como lenguaje principal, con sus características avanzadas como async/await, LINQ, y orientación a objetos, ofrece una experiencia de desarrollo potente y eficiente.

C#

C# es un lenguaje multiplataforma de uso general que hace que los desarrolladores sean productivos al escribir código de alto rendimiento. Con millones de desarrolladores, C# es el lenguaje .NET más popular. C# tiene una amplia compatibilidad con el ecosistema y todas las cargas de trabajo de .NET. Basado en principios orientados a objetos, incorpora muchas características de otros paradigmas, en particular de la programación funcional. Las características de bajo nivel admiten escenarios de alta eficiencia sin necesidad de escribir código no seguro. La mayor parte del entorno de ejecución y las bibliotecas de .NET se escriben en C#, y los avances en C# suelen beneficiar a todos los desarrolladores de .NET.

Implementaciones de .NET

Una aplicación desarrollada con .NET se basa en alguna de las **implementaciones existentes de la plataforma**, cada una con características propias, enfoques específicos y entornos de ejecución distintos. Estas implementaciones comparten una arquitectura general pero pueden variar en sus capacidades, plataformas soportadas y herramientas asociadas.

Componentes comunes en toda implementación de .NET

Cada implementación de .NET incluye, al menos, los siguientes componentes fundamentales:

- **Entorno de ejecución (Runtime)**: Es el motor responsable de ejecutar las aplicaciones, administrar la memoria, controlar la seguridad de tipos y realizar otras tareas críticas. Ejemplos: el CLR (Common Language Runtime) en .NET Framework y .NET 8.
- **Biblioteca de clases base (BCL, Base Class Library)**: Conjunto de APIs estándar que ofrecen funcionalidades básicas como manipulación de cadenas, acceso a archivos, colecciones, concurrencia, redes, criptografía, etc.
- **Marcos de trabajo opcionales**: Permiten construir aplicaciones específicas. Ejemplos: ASP.NET para aplicaciones web, Windows Forms y WPF para aplicaciones gráficas en Windows.
- **Herramientas de desarrollo**: Conjunto de utilidades que facilitan el trabajo del desarrollador, tales como compiladores, depuradores, gestores de paquetes y editores. Algunas herramientas son compartidas entre implementaciones (por ejemplo, Visual Studio y la CLI de .NET).

Principales implementaciones de .NET

Microsoft reconoce oficialmente **cuatro implementaciones principales de .NET**:

1. .NET 5 y versiones posteriores (incluido .NET 8)

Esta es la **implementación principal y moderna de .NET**, sucesora directa de .NET Core. Fue pensada desde sus inicios como una plataforma unificada, modular, **multiplataforma**, de alto rendimiento y abierta a la comunidad.

Características:

- Soporta **Windows, Linux y macOS**.
- Es completamente de **código abierto** y mantiene un desarrollo activo bajo la supervisión de la .NET Foundation.
- Diseñada para cargas de trabajo modernas: **aplicaciones web, APIs, microservicios, contenedores (Docker), servicios cloud, y aplicaciones de consola**.
- Compatible con varios entornos de ejecución: WebAssembly (Blazor), nativo (AOT), y clásico (JIT).
- Utiliza una única **base de código compartida** para todas las plataformas.
- Compatible con .NET Standard, lo que facilita la reutilización de bibliotecas existentes.

Marcos de trabajo disponibles en esta implementación:

- **ASP.NET Core**
- **Windows Forms** (solo en Windows)
- **WPF** (solo en Windows)
- **Blazor** (para web interactiva y WebAssembly)
- **MAUI** (para desarrollo móvil y de escritorio multiplataforma)

Última versión estable: .NET 8

.NET 8 representa una evolución significativa en términos de rendimiento, soporte nativo para inteligencia artificial, mejoras en MAUI y Blazor, y optimizaciones para entornos de nube y microservicios.

2. .NET Framework

Es la **implementación original de .NET**, lanzada en 2002, y centrada exclusivamente en el ecosistema Windows. Se trata de una plataforma madura, ampliamente adoptada en aplicaciones empresariales y entornos corporativos.

Características:

- Funciona **solo en sistemas operativos Windows**.
- Integra marcos tradicionales como **Windows Forms, WPF y ASP.NET (no Core)**.
- Incluye **API específicas de Windows** no disponibles en otras implementaciones, como soporte completo para COM, Windows Services y tecnologías heredadas.
- Compatible con versiones de .NET Standard desde la 1.0 hasta la 2.0 (dependiendo de la versión del framework).

Estado actual:

- Se encuentra **en modo de mantenimiento**, es decir, no se agregan nuevas funcionalidades; solo se actualiza por motivos de seguridad y correcciones críticas.
- Aún se usa ampliamente para aplicaciones legadas que no pueden migrar fácilmente a .NET moderno.

3. Mono

Mono es una implementación de .NET desarrollada inicialmente por la comunidad como un esfuerzo de **portar .NET a sistemas Unix y móviles**, antes de la aparición de .NET Core.

Características:

- Compatible con **Android, iOS, macOS, Linux, WebAssembly y otras plataformas específicas**.

- Utiliza un entorno de ejecución liviano, ideal para dispositivos con recursos limitados.
- Motor de ejecución de aplicaciones en **Xamarin** (actualmente parte de .NET MAUI).
- Soporta **compilación Just-In-Time (JIT)** y **Ahead-Of-Time (AOT)**, esencial para plataformas como iOS donde no se permite ejecución dinámica.
- Incluye herramientas específicas para videojuegos mediante su integración con **Unity**, el motor de desarrollo de juegos que utiliza C#.

Uso actual:

- Fundamental en entornos móviles y videojuegos.
- Aunque en proceso de integración con .NET MAUI, Mono sigue siendo relevante en proyectos que requieren footprint reducido o compilación nativa estricta.

4. UWP (Plataforma Universal de Windows)

UWP fue diseñado para permitir la creación de **aplicaciones modernas y responsivas** para el ecosistema de Windows, desde PC hasta Xbox, tabletas, y dispositivos IoT.

Características:

- Utiliza el modelo de ejecución seguro llamado **AppContainer**.
- Accede a las **APIs modernas de Windows (WinRT)** en lugar de las clásicas Win32.
- Soporta múltiples lenguajes de desarrollo: **C#, Visual Basic, C++, y JavaScript**.
- Proporciona integración directa con la **Microsoft Store** para distribución de aplicaciones.

Estado actual:

- Si bien fue un intento de unificar las aplicaciones para todos los dispositivos Windows, en la actualidad ha sido reemplazada progresivamente por **Windows App SDK (Project Reunion)** y tecnologías como **WinUI y MAUI**.

Resumen comparativo de las implementaciones

Implementación	Plataforma	Estado Actual	Multiplataforma	Casos de uso más comunes
.NET 5+ (.NET 8)	Windows, Linux, macOS	En evolución	Sí	Web, escritorio, nube, microservicios, MAUI
.NET Framework	Solo Windows	Mantenimiento	No	Aplicaciones legadas, escritorio clásico
Mono	Móvil, Unix, Unity	En uso activo	Sí	Apps móviles (Xamarin), videojuegos (Unity)

UWP	Dispositivos Windows	Decreciente uso	Parcial	Aplicaciones modernas de Windows, Xbox, IoT

Glosario

.NET

Término paraguas que engloba .NET Standard y todas las implementaciones y cargas de trabajo de la plataforma. Más comúnmente identifica la implementación moderna, multiplataforma y de código abierto (lo que era .NET Core), desde .NET 5 en adelante

assembly (ensamblado)

Archivo `.dll` o `.exe` que contiene un conjunto de API y tipos (clases, interfaces, delegados). Representa la unidad de compilación y despliegue en .NET

BCL (Base Class Library)

Conjunto de bibliotecas básicas (`System.*`) que forman la base funcional del runtime. Incluye componentes como colecciones, IO, threading, criptografía y más.

CLR (Common Language Runtime)

Entorno de ejecución que gestiona:

- Asignación y liberación de memoria (con GC).
- Seguridad de tipos.
- Compilación JIT de IL a código nativo.
- Manejo de excepciones y threading

Core CLR

CLR específico de la implementación moderna .NET (5+), originalmente derivado de Silverlight, y construido sobre la misma base que el CLR clásico, pero multiplataforma

CoreRT

Runtime alternativo enfocado en AOT (compilación previa) que prescinde del JIT. Incluye GC, reflexión limitada y es útil para apps nativas autopublicadas

cross-platform (multiplataforma)

Capacidad de desarrollar y ejecutar aplicaciones en **Windows, Linux, macOS**, sin cambios en el código fuente

ecosystem (ecosistema)

Conjunto formado por runtimes, herramientas, IDEs, paquetes y recursos comunitarios que rodean a .NET

framework (marco de trabajo)

Conjunto de APIs especializadas para distintos tipos de aplicaciones: ej. **ASP.NET Core**, **Windows Forms**, **WPF**

GC (Garbage Collector)

Recolector automático de memoria que libera objetos no referenciados, sin intervención del desarrollador.

IL (Intermediate Language)

Lenguaje intermedio al que se compila código en C#, F#, VB, MSIL/CIL. Es convertido a código nativo por el runtime

JIT (Just-In-Time compiler)
Compilador en tiempo de ejecución que traduce IL a código máquina en el host, optimizando para el hardware específico

implementation of .NET (implementación de .NET)
Conjunto que incluye:

- Uno o varios runtimes (CLR, CoreRT).
- Una biblioteca de clases compatible con .NET Standard.
- Opcionalmente, frameworks de aplicación y herramientas.
  Ejemplos: .NET Framework, .NET (Core), Mono, UWP

Mono
Implementación ligera, multiplataforma y de código abierto. Utilizada en Xamarin y Unity. Soporta JIT y AOT, y es compatible con .NET Standard .

Native AOT
Modo de publicación donde la aplicación se compila completamente a código nativo, sin depender del JIT ni de un runtime instalado

.NET Standard
Especificación de APIs que garantiza compatibilidad entre implementaciones. Una vez versionado (hasta 2.1), sirve como contrato común para bibliotecas reutilizables .
¿Por qué usar *namespaces*?
Cuando desarrollás programas más grandes en C#, comienzan a aparecer dos problemas:

1. **Mayor dificultad para entender y mantener el código**: Cuanto más grande el programa, más compleja se vuelve su estructura, y más difícil es leerlo y modificarlo sin cometer errores.
2. **Conflictos de nombres**: A medida que agregás más clases y métodos, aumentan las chances de que dos elementos se llamen igual (por ejemplo, dos clases llamadas `Usuario`). Esto se agrava si también estás usando librerías de terceros, donde pueden existir nombres similares o idénticos.

Solución tradicional (no recomendada):
Usar prefijos en los nombres, como `MiEmpresa_Usuario` o `Proyecto1_Usuario`.
El problema de esto es que **no escala bien**: los nombres se hacen demasiado largos, repetitivos y difíciles de leer.

¿Qué son los *namespaces* y cómo ayudan?
Un **namespace** es como una caja o contenedor para clases, interfaces, métodos, etc.
Permite **organizar el código y evitar conflictos de nombres**.
Ejemplo:

```
namespace Proyecto1 {
    class Usuario {
        ...
    }
}
namespace Proyecto2 {
    class Usuario {
        ...
    }
}
```

Aunque ambas clases se llaman `Usuario`, no entran en conflicto porque están en *namespaces* distintos:

- `Proyecto1.Usuario`
- `Proyecto2.Usuario`

Esto también permite que múltiples desarrolladores trabajen en un mismo proyecto sin que sus nombres de clases se pisen entre sí.

¿Qué es una directiva `using`?
Una línea como esta:
`using System;`
Es una **directiva `using`** que indica al compilador que vas a usar elementos del *namespace* `System` sin tener que escribir su nombre completo todo el tiempo.

Ejemplo sin `using`:
```
System.Console.WriteLine("Hola");
```

Ejemplo con `using System;`:
```
Console.WriteLine("Hola");
```

Esto mejora la legibilidad y reduce la repetición innecesaria.
Visual Studio automáticamente agrega ciertos `using` cuando creás un nuevo proyecto (por ejemplo, `using System;`, `using System.Collections.Generic`, etc.).

¿Qué es un *assembly*?
Un **assembly** es un archivo que contiene código compilado. Puede tener extensión `.dll` o `.exe`.
Diferencias:

- `.dll`: librerías de clases (no ejecutables).
- `.exe`: aplicaciones ejecutables.

Un assembly puede contener muchas clases y namespaces.
Por ejemplo, la clase `System.Console` está en el archivo `System.Console.dll`.
Los assemblies permiten distribuir partes del sistema de forma modular. Microsoft, por ejemplo, divide su inmensa biblioteca de clases en muchos assemblies diferentes, agrupados por funcionalidad: consola, bases de datos, interfaces gráficas, etc.

¿Cómo se relacionan *namespaces* y *assemblies*?

- Un **namespace** puede estar **repartido en varios assemblies**.
- Un **assembly** puede contener **varios namespaces**.

Por eso **no hay una correspondencia uno a uno entre namespace y assembly**.
Por ejemplo, `System.Collections` puede estar en varios `.dll` distintos, y `System.Console.dll` puede incluir más de un *namespace*.

Agregar referencias a assemblies
Para usar clases de un assembly externo, primero hay que **agregar una referencia** al proyecto. Visual Studio lo hace automáticamente según el tipo de proyecto. Si no está incluida, podés:

- Usar el **Administrador de paquetes NuGet** para instalar librerías.
- Agregar manualmente referencias a archivos `.dll`.

Diferencias entre .NET Core y .NET Framework en cuanto a assemblies

- `.NET Core` usa `System.Runtime.dll` como base común para tipos esenciales.

- `.NET Framework` usa `mscorlib.dll`, que es **específica de Windows**.
- No son intercambiables.
- En .NET Core **no se debe usar mscorlib**. Si necesitás APIs específicas de Windows, usá proyectos .NET Framework.

Microsoft planea unificar ambos entornos en una única plataforma llamada simplemente **.NET**, como ya ocurre desde .NET 5 y .NET 8.

Comentarios en C#
Sirven para **documentar** o **deshabilitar temporalmente** partes del código:
Comentario de una línea:

```
// Este es un comentario de una línea
```

Comentario de varias líneas:

```
/*
   Este es un comentario
   de múltiples líneas
*/
```

Identificadores en C#
Cuando programás en C#, uno de los primeros elementos con los que vas a interactuar son los **identificadores**. Un identificador es simplemente el nombre que le das a distintos componentes de tu programa, como variables, métodos, clases, espacios de nombres (namespaces), entre otros. En otras palabras, son las etiquetas que utilizás para referirte a diferentes cosas dentro del código.
Para que un identificador sea válido, tiene que cumplir con ciertas reglas de sintaxis establecidas por el lenguaje:

1. Puede contener únicamente letras (mayúsculas o minúsculas), números y el carácter de guion bajo (_).
2. No puede comenzar con un número. Siempre debe iniciar con una letra o un guion bajo.
3. No puede contener símbolos como %, $, #, etc.

Por ejemplo, los siguientes son identificadores válidos: `resultado`, `_puntos`, `equipoFutbol`, `plan9`. En cambio, `9plan`, `total%` o `nombre$` no lo son porque violan alguna de las reglas anteriores.
Un detalle muy importante es que **C# es un lenguaje sensible a mayúsculas y minúsculas**. Esto significa que `edad`, `Edad` y `EDAD` se consideran tres identificadores distintos. Esto puede jugar a favor cuando querés distinguir cosas específicas, pero también puede ser una fuente de errores si no tenés cuidado.

Palabras clave (keywords) del lenguaje
Además de las reglas de sintaxis, hay ciertas palabras que directamente **no podés usar como identificadores** porque están **reservadas por el lenguaje**. Estas palabras son conocidas como **palabras clave (keywords)**. C# reserva estas palabras porque tienen un significado especial para el compilador. Por ejemplo, palabras como `class`, `namespace`, `public`, `if`, `return` y `using` ya tienen una función específica en la estructura del lenguaje, por lo que no se pueden reutilizar con otro propósito.
En total, hay 77 palabras clave reservadas. Si intentás usar alguna de ellas como nombre para una variable, el compilador marcará un error. En los editores como Visual Studio, estas palabras suelen mostrarse automáticamente en azul, lo que te ayuda a identificarlas visualmente.
Además, hay otro conjunto de palabras que **no están reservadas** oficialmente pero que forman parte de ciertas expresiones del lenguaje, como `var`, `from`, `select`, `await`,

`yield`, entre otras. Técnicamente podés usarlas como nombres de variables, pero no es recomendable hacerlo, ya que puede causar confusión, tanto para vos como para otros programadores que lean tu código.

Uso y declaración de variables

Una **variable** en C# representa una **ubicación de memoria** que puede contener un valor temporal. Pensalo como una caja con una etiqueta (el nombre de la variable) que guarda un dato. A través del programa, podés consultar, modificar o reemplazar ese dato.
Para poder usar una variable, primero tenés que **declararla**, indicando qué tipo de dato va a almacenar. Esto se hace especificando primero el tipo y luego el nombre de la variable. Por ejemplo:

```
int edad;
```

En este caso, `int` indica que la variable `edad` almacenará números enteros. Luego, podés asignarle un valor así:

```
edad = 42;
```

También podés hacer ambas cosas en una sola línea:

```
int edad = 42;
```

Una vez que la variable tiene un valor, podés usarla, por ejemplo, para mostrarla en pantalla:

```
Console.WriteLine(edad);
```

Una característica importante de C# es que **no permite el uso de variables sin valor asignado**. Si declarás una variable pero no le asignás un valor antes de usarla, el compilador generará un error. Esto se conoce como la regla de **asignación definitiva**, y su objetivo es evitar errores difíciles de detectar en tiempo de ejecución.

Convenciones para nombrar variables

Aunque técnicamente podés ponerle muchos nombres distintos a tus variables siempre que respetes la sintaxis, existen algunas buenas prácticas que te van a ayudar a escribir código más limpio, legible y mantenible. Especialmente si trabajás en equipo, adoptar una convención de nombres clara es fundamental.
Estas son algunas recomendaciones comunes:

- **No comiences identificadores con guion bajo (_)**. Aunque es válido, puede generar problemas de compatibilidad con otros lenguajes como Visual Basic.
- **Evitá nombres que se diferencien solo por mayúsculas/minúsculas**, como `montoTotal` y `MontoTotal`. Aunque C# los reconoce como distintos, esto puede causar confusiones.
- **Comenzá siempre con una letra minúscula**.
- **Usá camelCase** para nombres de varias palabras: la primera palabra va en minúscula y las siguientes empiezan con mayúscula. Ejemplo: `montoTotal`, `nombreUsuario`, `esActivo`.
- **No uses notación húngara**, es decir, no antepongas el tipo de dato al nombre (`strNombre`, `intEdad`). Esta práctica está obsoleta.

Tipos de datos y asignación de valores

Cada variable que declares debe tener un tipo de dato. C# tiene varios **tipos primitivos** que cubren distintos tipos de información. Algunos de los más comunes son:

- `int`: para números enteros. Ejemplo: `int cantidad = 10;`
- `long`: para enteros de gran tamaño. Se recomienda usar el sufijo `L` para indicar que es un long. Ejemplo: `long poblacion = 1000000L;`

- `float`: para números decimales de precisión simple. Se usa el sufijo `F`. Ejemplo: `float peso = 70.5F;`
- `double`: para números decimales de doble precisión. Es el tipo por defecto cuando usás un número con punto decimal. Ejemplo: `double altura = 1.75;`
- `decimal`: para valores monetarios o que requieren mucha precisión decimal. Se usa el sufijo `M`. Ejemplo: `decimal precio = 199.99M;`

- `string`: para cadenas de texto. Ejemplo: `string nombre = "Agustín";`
- `char`: para un único carácter. Ejemplo: `char letra = 'A';`
- `bool`: para valores booleanos (verdadero o falso). Ejemplo: `bool esEstudiante = true;`

Es importante tener en cuenta que si asignás un valor con un tipo incompatible al tipo declarado, puede ocurrir un error o una conversión implícita no deseada. Por ejemplo, si intentás guardar un número decimal en una variable `int`, el compilador no te va a dejar. También hay diferencias entre usar un `float` o un `double`. Los cálculos en `float` son más rápidos y consumen menos memoria, pero tienen menos precisión. Por eso, si vas a trabajar con valores que requieren precisión exacta (como dinero), conviene usar `decimal`.

Operadores Aritméticos en C#

C# soporta los operadores básicos de aritmética: suma (+), resta (–), multiplicación (*) y división (/). Estos operadores actúan sobre valores llamados operandos para producir un nuevo resultado. Por ejemplo, si quieres calcular cuánto se paga a un consultor que cobra 750 por día durante 20 días, usarías:

```
long moneyPaidToConsultant = 750 * 20;
```

Aquí `*` es el operador y `750` y `20` son los operandos.

Operadores y Tipos de Datos

No todos los operadores funcionan con todos los tipos de datos. Por ejemplo, los operadores aritméticos funcionan con tipos numéricos como `int`, `long`, `float`, `double` y `decimal`. Sin embargo, no puedes usar estos operadores con `bool`, y con `string` sólo puedes usar el operador `+` para concatenar cadenas, no para restar o multiplicar. Por ejemplo:

```
// Error de compilación:
Console.WriteLine("Gillingham" - "Forest Green Rovers");

// Concatenación válida:
Console.WriteLine("43" + "1");  // Imprime "431"
```

Si necesitas hacer operaciones con números en cadenas, debes convertirlos con métodos como `Int32.Parse`.

Interpolación de Cadenas

Para combinar texto con variables, C# ofrece la interpolación de cadenas, que es más eficiente y legible que usar `+`. Por ejemplo:

```
string username = "John";
string message = $"Hello {username}";
```

El símbolo `$` indica que las expresiones dentro de `{}` se evaluarán y reemplazarán con su valor.

Tipos de Resultado en Operaciones Aritméticas

El tipo del resultado depende de los tipos de los operandos. Por ejemplo, `5.0 / 2.0` da `2.5` porque ambos son `double`, mientras que `5 / 2` da `2` porque ambos son enteros y la división entera trunca el resultado. Si mezclas tipos, como `5 / 2.0`, el entero se convierte en `double` y el resultado es un `double` (2.5), aunque mezclar tipos no es buena práctica.

Operador Módulo (%)

El operador `%` devuelve el resto de una división entera. Por ejemplo, `9 % 2` es 1 porque 9 dividido 2 da 4 con resto 1. En C# también puede usarse con números de punto flotante, por ejemplo, `7.0 % 2.4` da 2.2.

Valores Especiales: Infinity y NaN

Dividir un entero por cero (`5 / 0`) causa error, pero con `double` o `float`, `5.0 / 0.0` produce `Infinity`. En cambio, `0.0 / 0.0` es una indeterminación que da `NaN` (Not a Number). Estos valores se propagan en cálculos: cualquier operación con `NaN` produce `NaN`, y con `Infinity` normalmente da `Infinity`, salvo casos especiales.

Qué es un Método

Un método es una secuencia nombrada de instrucciones que realiza una tarea específica. Si tienes experiencia en otros lenguajes como C, C++ o Visual Basic, un método es similar a una función o subrutina. Cada método tiene un nombre descriptivo que indica su propósito, un cuerpo con las instrucciones que ejecuta, puede recibir datos para procesar y puede devolver un resultado.

Declaración de un Método

La sintaxis básica para declarar un método en C# es:

csharp
CopiarEditar

```
returnType methodName(parameterList)
{
    // cuerpo del método
}
```

- **returnType**: indica el tipo de dato que el método devolverá (por ejemplo, `int`, `string`). Si el método no devuelve nada, se usa `void`.
- **methodName**: nombre del método, que debe seguir las reglas de identificadores y preferiblemente usar camelCase (como `calculateTax`).
- **parameterList**: opcional, lista de parámetros con tipo y nombre separados por comas, que el método puede usar para recibir datos.
- **cuerpo del método**: bloque de código entre `{ }` que define lo que hace el método.

Todos los métodos en C# deben estar dentro de una clase; no existen métodos globales.

Ejemplos Básicos

Método que suma dos números y devuelve el resultado:

```
int addValues(int leftHandSide, int rightHandSide)
{
    return leftHandSide + rightHandSide;
}
```

Método que recibe un número pero no devuelve nada (`void`), solo muestra un mensaje:

```
void showResult(int answer)
{
```

```
    Console.WriteLine($"The answer is {answer}");
}
```

Retornar Valores y Uso de `return`

Para devolver un valor, el método debe usar una sentencia `return` que especifique el valor a devolver, cuyo tipo debe coincidir con el declarado. El `return` también termina la ejecución del método, por lo que cualquier código después de esta línea no se ejecuta. En métodos `void` puedes usar simplemente `return;` para salir anticipadamente, aunque si no se usa, el método termina al llegar al final del cuerpo.

Métodos con Cuerpo de Expresión (Expression-bodied Methods)

Para métodos simples que realizan una sola operación o expresión, C# permite una sintaxis más corta usando `=>`:

```
int addValues(int leftHandSide, int rightHandSide) => leftHandSide + rightHandSide;

void showResult(int answer) => Console.WriteLine($"The answer is {answer}");
```

Esta forma es solo sintaxis abreviada y funciona igual que los métodos normales, pero hace el código más limpio y fácil de leer.

Dividir un programa en métodos pequeños hace que el código sea más organizado, fácil de entender y mantener.

Llamar a un Método

Los métodos existen para ser llamados y así ejecutar la tarea que realizan. Para llamar un método, usas su nombre seguido de paréntesis, y dentro de esos paréntesis, si el método requiere datos (parámetros), debes proporcionar los argumentos correspondientes.

Sintaxis para Llamar a un Método

La forma básica de llamar un método en C# es:

```
result = methodName(argumentList);
```

- **methodName**: debe coincidir exactamente con el nombre del método, respetando mayúsculas y minúsculas (C# es case-sensitive).
- **result =**: esta parte es opcional. Se usa cuando el método devuelve un valor y quieres almacenarlo en una variable. Si el método es `void` (no devuelve nada), no debes poner esta parte.
- **argumentList**: los valores que se pasan a los parámetros del método. Debes enviar uno por cada parámetro definido, y el tipo de cada argumento debe ser compatible con el parámetro correspondiente. Si hay varios, se separan con comas.

Consideraciones Importantes

- Siempre debes incluir los paréntesis `()` en la llamada al método, incluso si el método no recibe argumentos. Por ejemplo:

```
showMessage();
```

- Si el método devuelve un valor pero no lo guardás (es decir, no usás `result =`), el valor se calcula pero se descarta.

Devolver Múltiples Valores desde un Método

A veces es necesario que un método entregue más de un valor. Por ejemplo, si quieres que un método devuelva tanto el resultado de una división como el resto (módulo) de esa

operación, puedes usar una **tupla**.
Una **tupla** es una colección pequeña de valores agrupados, y aunque originalmente las tuplas tenían solo dos valores, en C# pueden contener varios.
Para indicar que un método devuelve una tupla, se especifican los tipos de cada valor que se retornará en la declaración del método, separándolos con comas.

Ejemplo Simple de Método que Devuelve una Tupla

```
// Método que devuelve el resultado de la división y el resto
(int quotient, int remainder) DivideAndRemainder(int dividend, int
divisor)
{
    int quotient = dividend / divisor;
    int remainder = dividend % divisor;
    return (quotient, remainder);
}
```

Al llamar este método, puedes recibir ambos valores así:

```
var result = DivideAndRemainder(10, 3);
Console.WriteLine($"Cociente: {result.quotient}, Resto:
{result.remainder}");
```

Aplicando Scope (Ámbito)

El **scope** determina dónde puede usarse una variable o método en un programa.

Scope Local

Las variables definidas dentro de un método solo existen y pueden usarse dentro de ese método. Cuando el método termina, la variable desaparece.

Ejemplo:

```
private void calcularClick(object sender, RoutedEventArgs e)
{
    int calculatedValue = 0;  // Variable con scope local
    // ...
}
```

La variable `calculatedValue` solo está disponible dentro de `calcularClick`. Si intentás usarla fuera, dará error.

Scope de Clase

Las variables definidas dentro de una clase, pero fuera de los métodos, se llaman **campos** (fields). Estas variables tienen **scope de clase** y pueden usarse en cualquier método dentro de esa clase.

Ejemplo:

```
class Ejemplo
{
    int myField = 0;  // Campo con scope de clase
    void firstMethod()
    {
        myField = 42; // válido
    }
    void anotherMethod()
    {
        myField++; // válido
    }
}
```

Un método puede usar un campo antes o después de su declaración en la clase, porque el compilador se encarga de eso. Sin embargo, es buena práctica declarar los campos al principio de la clase.

Sobrecarga de Métodos (Overloading)

La **sobrecarga** ocurre cuando dos métodos tienen el mismo nombre pero diferente lista de parámetros (número o tipo). Esto permite usar un mismo nombre para realizar operaciones similares en distintos contextos.

- No se puede sobrecargar un método solo cambiando el tipo de retorno.
- La sobrecarga facilita escribir métodos con el mismo nombre que acepten distintos tipos o cantidades de argumentos.

Ejemplo con la clase `Console` y su método `WriteLine`:

```
static void Main()
{
    Console.WriteLine("The answer is "); // Sin parámetros
    Console.WriteLine(42);               // Parámetro int
}
```

Cada llamada usa una versión distinta de `WriteLine` según el tipo y número de argumentos.

Nesting Methods (Métodos Anidados)

Cuando tenés un método muy grande que realiza un proceso complejo, es buena práctica dividirlo en partes más pequeñas, cada una implementada en un **helper method** (método auxiliar). Esto hace que:

- Sea más fácil probar cada parte por separado, verificando que cada paso funcione correctamente.
- El método principal (el método grande) se vuelve más legible y más fácil de mantener, porque no tiene que contener todo el código en un solo bloque.

Aunque **métodos grandes** y **métodos auxiliares** no son términos oficiales en C#, los usamos para diferenciar el método principal que coordina el proceso y los pequeños métodos que hacen las tareas específicas.

Por defecto, todos los métodos definidos en una clase son accesibles para cualquier otro método dentro de esa misma clase. Pero si un helper method sólo sirve para un método grande específico, es mejor **mantener ese helper method local** dentro del método grande, si el lenguaje o la implementación lo permite.

¿Por qué? Porque así:

- Evitamos que otro método lo use accidentalmente, ya que el helper method está pensado para un contexto muy específico.
- Mejoramos la **encapsulación**: ocultamos detalles de implementación del método grande, para que no afecten ni sean afectados por otras partes del código.
- Reducimos dependencias entre métodos grandes, lo que facilita modificar o reemplazar un método grande sin temor a romper otras cosas.

Parámetros Opcionales y Argumentos Nombrados

Por qué existen

En C#, podés definir varios métodos con el mismo nombre pero diferente lista de parámetros, a esto se lo llama **sobrecarga**. Por ejemplo:

```
void DoWorkWithData(int a, float b, int c) { ... }
void DoWorkWithData(int a, float b) { ... }
```

El compilador sabe cuál usar según la cantidad y tipos de parámetros en la llamada.

Pero hay limitaciones. Por ejemplo, no podés tener dos métodos con la misma cantidad y tipo de parámetros porque generaría confusión:

```
void DoWorkWithData(int a) { ... }
void DoWorkWithData(int c) { ... } // ERROR, los parámetros tienen
el mismo tipo y cantidad
```

Esto da error porque el compilador no puede distinguir cuál método llamar cuando le pasás un `int`.

Para resolver estos casos y para facilitar la interoperabilidad con tecnologías como **COM** (que no soporta sobrecarga), C# permite definir **parámetros opcionales**, que tienen un valor por defecto.

Parámetros Opcionales

Cuando definís un método, podés asignar valores por defecto a algunos parámetros, convirtiéndolos en opcionales. Por ejemplo:

```
void optMethod(int first, double second = 0.0, string third =
"Hello") {
  // código
}
```

Aquí:

- `first` es obligatorio porque no tiene valor por defecto.
- `second` y `third` son opcionales, si no se les pasa un argumento usan el valor asignado (0.0 y "Hello").

Esto te permite llamar a `optMethod` con diferente cantidad de argumentos:

```
optMethod(99, 123.45, "World");  // usa los 3 parámetros
optMethod(100, 54.321);           // usa 1º y 2º, 3º toma valor
"Hello"
optMethod(101);                   // usa solo 1º, 2º y 3º toman
valores por defecto
```

Una regla importante: todos los parámetros obligatorios deben ir antes que los opcionales.

Argumentos Nombrados (Named Arguments)

Normalmente, cuando llamás a un método, los argumentos se asignan a los parámetros según el orden en que aparecen:

```
optMethod(99, 123.45, "World");  // first=99, second=123.45,
third="World"
```

Pero con **argumentos nombrados**, podés especificar qué argumento corresponde a qué parámetro usando el nombre del parámetro seguido de dos puntos `:`:

```
optMethod(first: 99, third: "World", second: 123.45);
```

Esto tiene varias ventajas:

- Podés pasar los argumentos en cualquier orden.
- Podés omitir algunos argumentos y pasar solo los que te interesan.
- Podés mezclar argumentos posicionales (por orden) y nombrados (por nombre), pero los posicionales siempre deben ir primero.

Ejemplo de mezcla:

```
optMethod(99, third: "World");  // first=99 (posicional),
third="World" (nombrado), second toma valor por defecto
```

Ambigüedades con Parámetros Opcionales y Sobrecarga

Cuando combinás sobrecarga con parámetros opcionales puede surgir un problema: el compilador puede no saber cuál método llamar si la llamada es ambigua.

Ejemplo con dos métodos:

```
void optMethod(int first, double second = 0.0, string third =
```

```
"Hello") { ... }
void optMethod(int first, double second = 1.0, string third =
"Goodbye", int fourth = 100) { ... }
```
Ambos métodos son legales porque tienen diferentes listas de parámetros.
Llamada clara, sin ambigüedad:

csharp
CopiarEditar
```
optMethod(1, 2.5, "World");  // llama a la versión de 3 parámetros
optMethod(1, fourth: 101);   // llama a la versión de 4 parámetros
```
- Llamada ambigua, sin solución clara para el compilador:

```
optMethod(1, 2.5);
optMethod(1);
optMethod(1, third: "World");
```
- El compilador no sabe a qué versión llamar, porque ambas coinciden parcialmente, y da error de compilación.

Uso de Sentencias `switch` en C#

Cuando tenés una serie de condiciones que evalúan la misma expresión pero comparan contra distintos valores, en vez de usar muchos `if`/`else if` anidados, es más eficiente y legible usar una sentencia `switch`.

Ejemplo clásico con `if (Lo que hacen en las grandes empresas)`

Supongamos que querés determinar el nombre de un día según un número `day` (0 = domingo, 1 = lunes, etc.):

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    // ...
}
else
{
    dayName = "Unknown";
}
```
Este código puede hacerse más claro con un `switch`.

Sintaxis básica de un `switch`

```
switch (expresiónDeControl)
{
```

```
    case valorConstante1:
        // instrucciones
        break;
    case valorConstante2:
        // instrucciones
        break;
    // ...
    default:
        // instrucciones por defecto
        break;
}
```

- La `expresiónDeControl` se evalúa una sola vez.
- Se compara con cada `case` (cada etiqueta o valor constante).
- Si coincide con un `case`, se ejecutan las instrucciones bajo ese `case` hasta que aparece un `break`.
- El `break` termina el `switch` y continúa la ejecución luego del `switch`.
- Si ningún `case` coincide, se ejecuta el bloque `default` (opcional).

Ejemplo equivalente con `switch`
csharp
CopiarEditar

```
switch (day)
{
    case 0:
        dayName = "Sunday";
        break;
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    // ...
    default:
        dayName = "Unknown";
        break;
}
```

Reglas importantes para usar `switch`

1. **Tipos permitidos en `switch`:** Podés usar `switch` con tipos como `int`, `char`, `string` (y otros tipos numéricos enteros como `long` o `decimal`).
   Aunque técnicamente se puede usar con `float` o `double`, no es recomendable porque estos tipos representan aproximaciones y no valores exactos, lo que puede causar errores difíciles de detectar.
2. **Las etiquetas (`case`) deben ser expresiones constantes:** No podés usar variables o resultados calculados en tiempo de ejecución como etiquetas. Ejemplo válido: `case 42:`, `case 'A':`, `case "Hello":`.
3. **Cada etiqueta debe ser única:** No puede haber dos `case` con el mismo valor, porque causaría conflicto sobre qué bloque ejecutar.

Agrupamiento de casos con mismo código:
Si querés que varios valores ejecuten el mismo bloque, podés listar varios `case` consecutivos sin instrucciones intermedias, y luego poner el código una sola vez. Por ejemplo:

```
switch (suit)
{
    case "Hearts":
    case "Diamonds":  // No hay código entre estos casos
        color = "Red";  // Se ejecuta para Hearts y Diamonds
        break;
    case "Clubs":
        color = "Black";
        break;
    case "Spades":
        color = "Black";
        break;
}
```

4. **Importante:** No podés poner código entre etiquetas y luego seguir con otro `case` sin `break`. Eso genera error de compilación porque C# no permite caída automática ("fall-through") entre casos como en C o C++.
5. **Uso de `break`:** Cada caso debe terminar con un `break` para evitar continuar la ejecución hacia el siguiente caso. Alternativamente, podés usar `return` o `throw` para salir del método o lanzar una excepción.

Sobre la "caída" (fall-through) en C# vs C/C++

- En C/C++ la ejecución pasa de un `case` al siguiente si no hay `break`, lo que puede causar errores.
- En C# **no está permitido fall-through automático**, salvo en casos donde no hay código entre etiquetas (`case` agrupados).
- Si realmente necesitás hacer fall-through, podés usar `goto case`, pero no es recomendable porque dificulta la legibilidad.

Aplicación práctica: procesamiento de caracteres especiales en XML
En XML, ciertos caracteres (`<`, `>`, `&`, `'`, `"`) tienen significado especial y deben ser reemplazados por entidades (por ejemplo, `<` se convierte en `<`).
Con un `switch` podés detectar estos caracteres y reemplazarlos, por ejemplo:

```
switch (c)
{
    case '<':
        // reemplazar por "&lt;"
        break;
    case '>':
        // reemplazar por "&gt;"
        break;
    case '&':
        // reemplazar por "&amp;"
        break;
    case '\'':
        // reemplazar por "&apos;"
```

```
            break;
        case '"':
            // reemplazar por "&quot;"
            break;
        default:
            // copiar el carácter tal cual
            break;
}
```

Switch Expressions con Pattern Matching en C#

¿Qué es un switch expression?

Un **switch expression** es similar a un `switch` statement tradicional, pero en lugar de ejecutar instrucciones, **devuelve un valor**. Esto permite asignar directamente el resultado a una variable o usarlo dentro de expresiones más complejas.

Ventajas sobre switch statements

- Es más compacto y legible.
- No requiere `break`.
- Usa **pattern matching** para comparar rangos o condiciones complejas.
- Obliga a manejar todos los casos posibles o poner un patrón "catchall" para evitar errores.

Ejemplo: Categorizar un número en rangos

Imaginemos que tenemos un número `measurement` y queremos saber si es:

- negativo (< 0)
- cero (0)
- de un solo dígito (entre 1 y 9)
- de dos dígitos (entre 10 y 99)
- grande (100 o más)

Con un switch expression, podemos escribir:

```
int measurement = ...;
string range = measurement switch
{
    < 0 => "negative",
    0 => "zero",
    >= 1 and <= 9 => "singledigit",
    >= 10 and <= 99 => "doubledigit",
    >= 100 => "large"
};
```

Detalles de sintaxis importantes

- El operador `switch` funciona como una expresión que recibe un valor y un conjunto de patrones.
- Cada línea dentro de las llaves `{}` tiene:
    - **Un patrón a la izquierda** (ejemplo: `< 0`, `0`, `>= 1 and <= 9`).
    - El operador `=>` que separa el patrón de la expresión o valor a devolver.
- Las diferentes líneas se separan por comas, no hay `break`.
- La expresión a la derecha de `=>` es la que se retorna si el patrón coincide.

Exhaustividad

El switch expression **debe cubrir todos los posibles valores** del tipo evaluado (ser "exhaustivo"). Si queda algún valor sin patrón, el compilador da error.

Para manejar casos no cubiertos, se usa el patrón `_` (guion bajo), que es un "comodín" que coincide con cualquier valor.

Ejemplo con `_`:

```
string range = measurement switch
{
    < 0 => "negative",
    0 => "zero",
    >= 1 and <= 9 => "singledigit",
    >= 10 and <= 99 => "doubledigit",
    _ => "large"  // Cubre cualquier otro valor que no haya
coincidido antes
};
```

¿Para qué es útil?

- Validaciones complejas.
- Clasificación de datos según rangos o condiciones múltiples.
- Simplificar lógica que con if-else sería muy larga y difícil de seguir.

Operadores de Asignación Compuesta (Compound Assignment Operators)

¿Qué son?

Son operadores que combinan una operación aritmética con una asignación. Simplifican el código que actualiza el valor de una variable.

Ejemplo básico sin operador compuesto

```
answer = answer + 42;
```

Esto suma 42 al valor actual de `answer` y luego asigna el resultado a `answer`.

Ejemplo con operador compuesto (shorthand)

```
answer += 42;
```

Equivale a la línea anterior pero es más corto y claro.

Otros operadores compuestos

No escribas...	Escribe...
`variable = variable * 5;`	`variable *= 5;`
`variable = variable / 2;`	`variable /= 2;`
`variable = variable % 3;`	`variable %= 3;`
`variable = variable + 10;`	`variable += 10;`
`variable = variable -`	`variable -=`

```
4;                    4;
```

Nota importante

- Estos operadores tienen la **misma precedencia** que el operador `=`.
- Para incrementar o decrementar en 1, se recomienda usar `++` o `--`:

```
count++;
// en vez de
count += 1;
```

Aplicación en cadenas (strings)

El operador `+=` también funciona con strings, para concatenarlos:

```
string name = "John";
string greeting = "Hello ";
greeting += name;  // greeting ahora es "Hello John"
Console.WriteLine(greeting);
```

Los demás operadores compuestos no funcionan con strings.

Sentencias while en C#

¿Para qué sirve?

Para ejecutar repetidamente un bloque de código mientras se cumpla una condición.

Sintaxis

```
while (condiciónBooleana)
{
    // Bloque de código que se ejecuta mientras la condición sea
verdadera
}
```

- La condición se evalúa antes de cada iteración.
- Si la condición es falsa al principio, el bloque no se ejecuta ni una vez.
- Si la condición es verdadera, el bloque se ejecuta y luego se vuelve a evaluar la condición.

Ejemplo que imprime números del 0 al 9

```
int i = 0;
while (i < 10)
{
    Console.WriteLine(i);
    i++;  // Incrementa i para evitar un ciclo infinito
}
```

Consejos y advertencias

- Siempre usa llaves `{}` para delimitar el bloque aunque tengas solo una línea, para evitar errores difíciles de detectar.
- No olvides modificar alguna variable dentro del ciclo para que la condición eventualmente sea falsa y el ciclo termine (evitar ciclos infinitos).

Error común

Este código **no** hace lo que se espera:

```
int i = 0;
while (i < 10)
    Console.WriteLine(i);
    i++;  // NO está dentro del while, se ejecuta una vez al final,
ciclo infinito
```

Como `i++` no está dentro del bloque del while, `i` nunca cambia dentro del ciclo y el ciclo se vuelve infinito.

Estructura for en C#

¿Qué es?

El `for` es una estructura de repetición que combina en una sola línea la **inicialización**, la **condición** y la **actualización** de la variable que controla el ciclo.

Sintaxis

```
for (inicialización; condición booleana; actualización)
{
    // cuerpo del ciclo
}
```

Ejemplo: imprimir números del 0 al 9

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

- La **inicialización** se ejecuta una sola vez al inicio.
- La **condición** se evalúa antes de cada iteración.
- El **cuerpo** del ciclo se ejecuta si la condición es verdadera.
- Luego, se ejecuta la **actualización**.
- El ciclo continúa hasta que la condición sea falsa.

Características y recomendaciones

- Siempre usar llaves `{}` para agrupar el cuerpo, incluso si es una línea.

Puedes omitir cualquiera de las tres partes del for. Por ejemplo:

```
for (int i = 0; ; i++)  // condición omitida = true, ciclo infinito
{
    Console.WriteLine("Ciclo infinito");
}
```

- Puedes usar múltiples inicializaciones y actualizaciones separadas por comas:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
    Console.WriteLine($"i={i}, j={j}");
}
```

- Las variables declaradas en la inicialización están **en alcance sólo dentro del ciclo**:

```
for (int i = 0; i < 10; i++)
{
    // usar i aquí
}
// Console.WriteLine(i); // Error: i no está en alcance aquí
```

- Puedes reutilizar el mismo nombre de variable en diferentes ciclos `for` porque cada

uno tiene su propio alcance.

Estructura do-while en C#
¿Qué es?
El `do-while` ejecuta el cuerpo al menos una vez antes de evaluar la condición. La condición se evalúa **al final** de cada iteración.
Sintaxis

```
do
{
    // cuerpo del ciclo
} while (condición booleana);
```

Ejemplo: imprimir números del 0 al 9

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
} while (i < 10);
```

Sentencias break y continue
break

- Sale inmediatamente del ciclo (`for`, `while` o `do-while`).
- El código después del ciclo continúa ejecutándose.

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
        break;  // termina el ciclo cuando i es 5
    Console.WriteLine(i);
}
```

continue

- Salta a la siguiente iteración del ciclo, omitiendo el resto del código que queda en el cuerpo para esa iteración.
- En un `for`, la actualización ocurre antes de la siguiente evaluación.

```
for (int i = 0; i < 10; i++)
{
    if (i % 2 == 0)
        continue;  // salta la impresión para números pares
    Console.WriteLine(i);
}
```

Ejemplo avanzado: conversión decimal a octal con do-while

```
int dec = 999;
string octal = "";
do
{
    int remainder = dec % 8;
    dec = dec / 8;
    octal = remainder.ToString() + octal;
} while (dec != 0);
Console.WriteLine(octal);  // Salida: 1747
```

Explicación:

- Se divide el número decimal entre 8.

- Se guarda el resto.
- Se repite hasta que el cociente sea 0.
- Se concatena el resultado en orden inverso para obtener la representación octal.

Manejo de Errores y Excepciones en C#
Objetivos principales:

- Manejar excepciones con `try`, `catch` y `finally`.
- Controlar desbordamientos con `checked` y `unchecked`.
- Lanzar excepciones propias con `throw`.
- Asegurar ejecución de código con `finally`.

1. ¿Por qué manejar excepciones?
Los errores pueden ocurrir en cualquier momento por causas externas (hardware, usuario, red, etc.). Los lenguajes modernos como C# usan excepciones para detectar y manejar errores de forma elegante, en vez de usar variables globales o chequeos manuales constantes.

2. Uso de `try`, `catch` y `finally`

- **`try`**: Bloque donde colocas el código que puede generar excepciones.
- **`catch`**: Bloque que "atrapa" una excepción específica para manejarla.
- **`finally`**: Bloque que siempre se ejecuta, haya ocurrido o no una excepción (usualmente para liberar recursos).
  Ejemplo:

```
try
{
    int num = int.Parse(userInput);
    // más código...
}
catch (FormatException ex)
{
    Console.WriteLine("Error: Formato inválido.");
}
finally
{
    Console.WriteLine("Operación terminada.");
}
```

3. Excepciones específicas y múltiples `catch`
Puedes tener varios `catch` para diferentes tipos de excepciones:

```
try
{
    // código que puede fallar
}
catch (FormatException fEx)
{
    // manejar formato inválido
}
catch (OverflowException oEx)
{
    // manejar desbordamiento
}
```

El primer `catch` que coincida con la excepción atrapará el error.

4. Excepción general: `Exception`

Si quieres atrapar cualquier excepción no manejada específicamente, puedes usar:

```
catch (Exception ex)
{
    // manejar cualquier excepción
}
```

Nota: No es recomendable usar `catch` sin parámetros porque pierdes acceso al objeto excepción y su información.

5. Jerarquía de excepciones

- `FormatException` y `OverflowException` heredan de `SystemException`.
- `SystemException` hereda de `Exception`.
- Por eso, un `catch (Exception ex)` atrapa todas las excepciones.

6. Orden de los `catch`

Coloca primero los más específicos y luego los más generales. Si colocas primero el `catch (Exception)`, los más específicos nunca se ejecutarán.

7. Filtrado de excepciones con `when`

Puedes usar un filtro booleano para decidir cuándo un `catch` debe ejecutarse:

```
catch (Exception ex) when (someCondition)
{
    // solo manejar cuando someCondition es true
}
```

8. Excepciones no manejadas

Si una excepción no es atrapada en ningún `catch`, la aplicación terminará con un error. Durante el desarrollo en Visual Studio, el depurador detiene la ejecución en el punto de excepción para facilitar el análisis.

Lanzar Excepciones con throw

Cuando un método detecta que no puede continuar normalmente (por ejemplo, porque recibió un argumento inválido), la mejor práctica es **no devolver un valor erróneo** ni intentar manejar el error silenciosamente, sino **lanzar una excepción**.

Cómo funciona `throw`

- `throw` lanza un objeto de excepción (una instancia de una clase derivada de `Exception`).
- Esa excepción puede ser capturada luego por un bloque `try-catch` más arriba en la pila de llamadas.
- El objeto excepción debe contener información sobre el error, como un mensaje descriptivo.

Ejemplo práctico: método `monthName`

```
public string monthName(int month) => month switch {
    1 => "January",
    2 => "February",
    3 => "March",
    4 => "April",
    5 => "May",
    6 => "June",
    7 => "July",
    8 => "August",
```

```
    9 => "September",
    10 => "October",
    11 => "November",
    12 => "December",
    _ => throw new ArgumentOutOfRangeException("Bad month") //
Lanzamos excepción si el mes no está en 1-12
};
```

Explicación:

- Usamos un `switch expression` para mapear el número a nombre del mes.
- Si el argumento `month` no está entre 1 y 12, el caso `_` (default) lanza la excepción `ArgumentOutOfRangeException`.
- El constructor de esta excepción recibe un mensaje para describir el problema.

Conceptos importantes

- `ArgumentOutOfRangeException` es una excepción estándar de .NET para argumentos fuera de rango.
- Puedes usar otras excepciones según el error (ej. `ArgumentNullException` si un parámetro es `null` cuando no debería).
- También puedes definir tus propias clases de excepción (pero eso lo verás más adelante).

Uso del bloque `finally` en C#

Cuando lanzamos una excepción, el flujo normal del programa se interrumpe. Por eso, **no podemos asegurar que una línea de código se ejecute si ocurre una excepción antes**. Pero hay casos donde es crucial que una línea **se ejecute siempre**, por ejemplo, para liberar recursos como archivos abiertos, conexiones a bases de datos, etc.

¿Qué es el bloque `finally`?

- Es un bloque que va **después de un bloque `try` o después del último `catch`**.
- Se ejecuta **siempre**, haya ocurrido o no una excepción.
- Si ocurre una excepción y hay un `catch` que la maneja, se ejecuta primero el `catch` y luego el `finally`.
- Si la excepción no se maneja en el método actual, se ejecuta el `finally` antes de propagar la excepción.

Ejemplo práctico: liberar un archivo abierto

```
TextReader reader = ...;  // Abrimos un archivo para lectura
try
{
    string line = reader.ReadLine();
    while (line is not null)
    {
        // Procesar línea...
        line = reader.ReadLine();
    }
}
finally
{
    if (reader is not null)
    {
        reader.Dispose();  // Siempre liberar el recurso
    }
```

```
}
```

¿Por qué usar `finally`?

- Sin el `finally`, si ocurre una excepción dentro del `try`, la instrucción `reader.Dispose()` **no se ejecutaría**.
- Esto puede provocar fugas de recursos, como que se queden archivos abiertos o conexiones a bases de datos sin cerrar.
- Usar `finally` garantiza la limpieza y el correcto manejo de recursos.

Nota extra

En C# moderno existe también la instrucción `using`, que hace todo esto automáticamente para objetos que implementan `IDisposable`, pero el bloque `finally` es el mecanismo básico y fundamental para este control.

Valores y Referencias en C#

1. Tipos por valor (Value Types)

- Ejemplos: `int`, `float`, `double`, `char`, `bool`, `struct`.
- Se almacenan **directamente en la memoria** asignada a la variable.
- Cuando copias una variable por valor, se crea una **copia completa e independiente** del dato.
- Cambiar una copia **no afecta** al original.
- Tamaño fijo en memoria (por ejemplo, `int` ocupa 4 bytes).

Ejemplo:

```
int i = 42;
int copyi = i;  // Se copia el valor 42 en copyi
i++;            // i ahora es 43
Console.WriteLine(copyi);  // Imprime 42, no cambia
```

2. Tipos por referencia (Reference Types)

- Ejemplos: Clases (`class`), cadenas (`string`), arrays.
- La variable no contiene directamente el dato, sino una **referencia (dirección en memoria)** hacia donde está almacenado el objeto.
- Cuando copias una variable de tipo referencia, copias la **dirección**, por lo que **ambas variables apuntan al mismo objeto**.
- Cambiar el objeto a través de una referencia afecta a todas las referencias que apuntan a ese objeto.
- El objeto en sí se crea en el **heap** (memoria dinámica) usando `new`.

Ejemplo:

```
var c = new Circle(42);  // c apunta a un objeto Circle en memoria
Circle refc = c;         // refc apunta al mismo objeto que c
refc.Radius = 50;        // Cambia el objeto apuntado
Console.WriteLine(c.Radius);  // Imprime 50, porque c y refc
refieren al mismo objeto
```

3. Caso especial: `string`

- `string` es una clase, por lo que es tipo referencia.
- Sin embargo, los strings son **inmutables** (no pueden modificarse después de creados).
- Cuando haces una operación que parece modificar un string, en realidad se crea un nuevo objeto string.

4. ¿Por qué es importante esta diferencia?

- **Pasar variables a métodos:**

    - Si pasas un tipo por valor, el método recibe una copia y no puede modificar la variable original.
    - Si pasas un tipo por referencia, el método puede modificar el objeto al que apunta.

- **Asignaciones y copias:**
    - Con tipos por valor, tienes datos independientes.
    - Con tipos por referencia, varias variables pueden apuntar al mismo objeto.
- **Administración de memoria:**
    - Los tipos por valor están en la pila (stack), con gestión rápida y automática.
    - Los tipos por referencia están en el heap, con gestión más compleja y recolección de basura.

5. Terminología extra que usarás después

- **Boxing:** Convertir un valor (tipo por valor) a tipo referencia (object).
- **Unboxing:** Extraer el valor original desde el objeto boxed.
- **`ref` y `out`:** Palabras clave para pasar parámetros por referencia explícitamente.

Copiar objetos de tipos por referencia y privacidad de datos

1. Copiar solo la referencia no es suficiente

Cuando haces esto:

```
Circle refc = c;
```

- Solo copias la **referencia** (la dirección en memoria) del objeto `c` a `refc`.
- Ambos variables apuntan al mismo objeto.
- Cambiar `refc.radius` o `c.radius` afecta al mismo objeto.

2. ¿Y si quiero copiar los datos reales?

Para crear un nuevo objeto con los mismos datos, tienes que:

- Crear un **nuevo objeto**.
- Copiar los **datos campo por campo** desde el objeto original al nuevo.

```
var refc = new Circle();
refc.radius = c.radius;  // Si radius es público, esto funciona
```

3. ¿Qué pasa si los campos son privados?

- No puedes acceder directamente a los campos privados desde fuera de la clase.
- Para copiar datos privados necesitas **exponerlos mediante propiedades públicas (getters/setters)**.

Ejemplo simple (te lo van a explicar mejor en el capítulo de propiedades):

```
class Circle
{
    private int radius;
    public int Radius  // propiedad para acceder al campo privado
    {
        get { return radius; }
        set { radius = value; }
    }
}
```

Con esto, puedes copiar así:

```
var refc = new Circle();
refc.Radius = c.Radius;
```

4. Método Clone para copiar objetos

Una mejor práctica es que la clase implemente un método `Clone` que:

- Cree una nueva instancia.
- Copie los datos privados internamente (porque está dentro de la clase, puede acceder a sus privados).

- Devuelva la copia.

Ejemplo:

```
class Circle
{
    private int radius;
    public Circle Clone()
    {
        Circle clone = new Circle();
        clone.radius = this.radius;  // acceso permitido porque es
dentro de la misma clase
        return clone;
    }
}
```

5. Shallow copy vs Deep copy

- **Shallow copy (copia superficial):** El método `Clone` copia solo los campos primitivos o referencias, pero **no crea copias nuevas de los objetos referenciados**.
  Si `Circle` tuviera un campo `Point position` (que es un objeto), entonces la copia superficial copiaría solo la referencia a `position`. Cambiar la posición en uno afectaría al otro.
- **Deep copy (copia profunda):** El método `Clone` no solo copia los campos primitivos, sino que también clona **todos los objetos referenciados** dentro de la clase, creando una estructura independiente completa.

Para lograr deep copy:

```
public Circle Clone()
{
    Circle clone = new Circle();
    clone.radius = this.radius;
    clone.position = this.position.Clone(); // suponiendo que Point
también tiene Clone()
    return clone;
}
```

6. ¿Qué tan "privados" son los campos privados?

- La palabra clave `private` impide que otros objetos **fuera de la clase** accedan a ese campo.
- Pero dentro del código de la clase, **cualquier instancia puede acceder a los campos privados de otra instancia**.
- Esto permite, por ejemplo, que un método `Clone` copie campos privados directamente de otro objeto de la misma clase.

Valores nulos y tipos anulables en C#

¿Qué es `null`?

- `null` es un valor especial que significa "no apunta a ningún objeto" o "sin valor".
- Sólo los tipos por referencia (clases, interfaces, arrays, delegados) pueden ser `null`.
- Los tipos por valor (como `int`, `bool`, `double`) **no pueden ser null** por defecto.

Tipos por referencia y null

```
string texto = null; // Es válido, string es tipo referencia
if (texto == null)
{
```

```
    Console.WriteLine("No hay texto");
}
```
Aquí `texto` no apunta a ningún dato, así que es nulo.

Tipos por valor no pueden ser null

```
int numero;
// numero = null; // Esto da error porque int no puede ser null
```

Tipos anulables (nullable)

- Para que un tipo por valor acepte null, usamos `?`:

```
int? numeroNullable = null;
if (numeroNullable == null)
{
    Console.WriteLine("El número es nulo");
}
numeroNullable = 10;
Console.WriteLine(numeroNullable); // Imprime 10
```

- Esto crea un **tipo anulable**, que es una forma especial de tipo por valor que puede contener un valor normal o `null`.

Propiedades importantes de los tipos anulables

- `.HasValue`: dice si tiene un valor o es null.
- `.Value`: obtiene el valor (sólo si `.HasValue` es true, si no lanza error).

```
int? n = 5;
if (n.HasValue)
{
    Console.WriteLine(n.Value); // Imprime 5
}
else
{
    Console.WriteLine("n es null");
}
```

Operador `??` (coalescencia nula)

- Permite asignar un valor por defecto si la variable es null.

```
int? numeroNullable = null;
int numeroNormal = numeroNullable ?? 100; // Si numeroNullable es
null, asigna 100
Console.WriteLine(numeroNormal); // Imprime 100
```

Operador `?.` (acceso condicional)

- Para evitar excepciones al usar métodos o propiedades en objetos que pueden ser null.

```
string texto = null;
int? longitud = texto?.Length; // No lanza excepción, longitud es
null
Console.WriteLine(longitud);   // Imprime ""
```

2. Parámetros `ref` y `out`

Parámetro `ref`

- Pasa una variable **por referencia** al método.
- El método puede modificar el valor original.
- La variable debe estar inicializada antes de llamar.

```
void Doblar(ref int x)
{
    x = x * 2;
}
int valor = 5;
Doblar(ref valor);
Console.WriteLine(valor); // Imprime 10
```

Parámetro `out`

- También pasa por referencia.
- Se usa para que un método **devuelva valores a través de parámetros**.
- La variable **no necesita estar inicializada** antes de llamar.
- El método está obligado a asignar un valor.

```
void ObtenerDatos(out int x)
{
    x = 42; // Obligatorio asignar
}
int valor;
ObtenerDatos(out valor);
Console.WriteLine(valor); // Imprime 42
```

Diferencias entre `ref` y `out`

Característica	`ref`	`out`
Variable debe estar inicializada	Sí	No
Método debe asignar valor	No es obligatorio (pero puede)	Sí, es obligatorio
Usos comunes	Modificar valor existente	Devolver valores desde el método

Trabajando con herencia en C#

¿Qué es la herencia?

La herencia es un concepto fundamental en la programación orientada a objetos. Permite crear nuevas clases basadas en clases existentes para **reutilizar código y evitar repetición**. Por ejemplo:

- Si tienes varias clases que comparten características comunes (como empleados en una fábrica), puedes crear una clase base común que tenga esas características, y las clases específicas (como gerentes o trabajadores manuales) heredan esas características comunes y además tienen sus propias particularidades.

Concepto clave con ejemplo: Mammal (mamífero)

- Si tienes varias clases que representan tipos de mamíferos (Caballo, Ballena, Humano, etc.), todas comparten cosas en común (respiran, alimentan a sus crías) pero también tienen comportamientos únicos (Caballo trota, Ballena nada).
- Crearías una clase `Mammal` con métodos comunes (`Breathe()`, `SuckleYoung()`)

y luego clases derivadas que hereden de Mammal y tengan sus propios métodos (`Horse` con `Trot()`, `Whale` con `Swim()`).

Sintaxis básica de herencia en C#

```
class DerivedClass : BaseClass
{
    // Código de la clase derivada
}
```

- La clase derivada hereda todos los miembros públicos y protegidos de la clase base.
- En C#, solo se puede heredar de una clase base (herencia simple).
- Puedes crear jerarquías de herencia, por ejemplo:

```
class DerivedSubClass : DerivedClass
{
    // Código
}
```

Ejemplo completo:

```
class Mammal
{
    public void Breathe()
    {
        Console.WriteLine("Respirando");
    }
    public void SuckleYoung()
    {
        Console.WriteLine("Alimentando a la cría");
    }
}
class Horse : Mammal
{
    public void Trot()
    {
        Console.WriteLine("El caballo trota");
    }
}
class Whale : Mammal
{
    public void Swim()
    {
        Console.WriteLine("La ballena nada");
    }
}
```

Uso:

```
Horse myHorse = new Horse();
myHorse.Trot();          // Método propio
myHorse.Breathe();       // Método heredado
myHorse.SuckleYoung();   // Método heredado
Whale myWhale = new Whale();
myWhale.Swim();          // Método propio
```

```
myWhale.Breathe();        // Método heredado
myWhale.SuckleYoung();    // Método heredado
```

Detalles importantes

- **Herencia sólo aplica a clases, no a estructuras (`struct`).**
- Todas las clases en C# derivan de forma implícita de `System.Object`, que es la clase raíz.
- Esto significa que todas las clases heredan métodos como `ToString()`, `Equals()`, etc.
- Los constructores de la clase base **se pueden llamar desde el constructor de la clase derivada** usando `base(...)`.

Ejemplo de constructor:

```
class Mammal
{
    public Mammal(string name)
    {
        Console.WriteLine($"Mammal creado: {name}");
    }
}
class Horse : Mammal
{
    public Horse(string name) : base(name)
    {
        Console.WriteLine($"Horse creado: {name}");
    }
}
```

- Si no llamas explícitamente a un constructor de la clase base, el compilador intenta llamar al constructor por defecto (`base()`).
- Si la clase base no tiene constructor por defecto, **debes llamar explícitamente a un constructor existente de la clase base** para evitar error de compilación.

Asignación entre clases en una jerarquía de herencia

- Puedes asignar un objeto a una variable cuyo tipo es una **clase base** de la clase del objeto, porque un objeto de una clase derivada **es un tipo** de la clase base.

```
class Mammal { }
class Horse : Mammal { }
class Whale : Mammal { }

Horse myHorse = new Horse();
Mammal myMammal = myHorse;  // Legal, porque Horse es un Mammal
```

- No puedes asignar un objeto de la clase base a una variable de la clase derivada **sin comprobar el tipo** porque no todos los Mammals son Horses:

```
Mammal myMammal = new Mammal();
Horse myHorse = myMammal; // Error de compilación
```

- Para hacer esa asignación debes validar el tipo usando `is`, `as` o un cast:

```
Horse myHorseAgain = myMammal as Horse;  // Devuelve null si no es
Horse
if (myHorseAgain != null) {
   // Es seguro usar myHorseAgain como Horse
}
```

Métodos que se ocultan (hiding) con new

- Si una clase derivada declara un método con la **misma firma** que uno en la clase base sin usar `override`, el método de la clase derivada **oculta** (no sobreescribe) al de la clase base.
- El compilador muestra una advertencia, que puedes silenciar con el modificador `new`:

```
class Mammal {
    public void Talk() { Console.WriteLine("Mammal talks"); }
}

class Horse : Mammal {
    new public void Talk() { Console.WriteLine("Horse talks"); }
}
```

- Aquí `Horse.Talk()` oculta a `Mammal.Talk()`, pero no es polimórfico. Si accedes a `Talk` desde una referencia de tipo `Mammal`, se llamará al método de `Mammal`.

Métodos virtuales y overriding (sobrescritura)

- Un método marcado como `virtual` en la clase base puede ser **sobrescrito** en la clase derivada con `override`, para proporcionar una implementación específica.
- Esto permite **polimorfismo**: la llamada al método se resuelve en tiempo de ejecución según el tipo real del objeto.
- Ejemplo con `ToString()` (que es virtual en `System.Object`):

```
class Mammal {
    public virtual string Talk() {
        return "Mammal sound";
    }
}
class Horse : Mammal {
    public override string Talk() {
        return "Horse neigh";
    }
}
```

- Si tienes:

```
Mammal m = new Horse();
Console.WriteLine(m.Talk());  // Llama a Horse.Talk() gracias al
override
```

- La implementación del método en la clase base puede ser llamada desde la clase derivada con la palabra clave `base`:

```
public override string Talk() {
    string baseResult = base.Talk();
    return baseResult + " and Horse neigh";
}
```

Reglas importantes sobre virtual y override

- Un método `virtual` **no puede ser privado** porque debe ser accesible para la herencia.
- El método que sobrescribe (`override`) debe tener la **misma firma** que el método virtual (nombre, parámetros, tipo de retorno).

- Solo puedes sobrescribir métodos que hayan sido declarados `virtual` o `abstract` en la clase base.
- Si un método con la misma firma en la clase derivada no usa `override`, sino que solo tiene el mismo nombre, se considera ocultamiento (`new`), no sobrescritura.
- Un método `override` es implícitamente virtual y puede ser sobrescrito por clases derivadas aún más abajo en la jerarquía, pero no se debe volver a marcar como `virtual` explícitamente.

Métodos virtuales y polimorfismo en C#
¿Qué son métodos virtuales?

- Un método declarado como `virtual` en una clase base permite que las clases derivadas lo **sobrescriban** usando la palabra clave `override`.
- En tiempo de ejecución, la versión del método que se ejecuta depende del **tipo real del objeto**, no del tipo de la variable.

Ejemplo con jerarquía Mammal

```
class Mammal
{
    public virtual string GetTypeName()
    {
        return "This is a mammal";
    }
}
class Horse : Mammal
{
    public override string GetTypeName()
    {
        return "This is a horse";
    }
}
class Whale : Mammal
{
    public override string GetTypeName()
    {
        return "This is a whale";
    }
}
class Aardvark : Mammal
{
    // No sobrescribe GetTypeName
}
```

Código que ilustra el polimorfismo:

```
Mammal myMammal;
Horse myHorse = new Horse();
Whale myWhale = new Whale();
Aardvark myAardvark = new Aardvark();
myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName()); // Imprime: "This is a
```

```
horse"
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // Imprime: "This is a
whale"
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // Imprime: "This is a
mammal"
```

Explicación:

- Aunque la variable `myMammal` es de tipo base `Mammal`, el método que se ejecuta depende del tipo concreto del objeto que referencia.
- En `Horse` y `Whale` el método está sobrescrito, por eso se ejecutan sus versiones.
- En `Aardvark` no hay sobrescritura, por lo que se llama la versión base.

Entendiendo el modificador `protected`
Comparación de accesos en C#

Modificador	Accesibilidad
`public`	Accesible desde cualquier lugar
`private`	Accesible solo dentro de la clase
`protected`	Accesible en la clase y sus derivadas

- `protected` permite a las clases derivadas acceder a miembros que están ocultos para otras clases.
- Esto mantiene la encapsulación pero permite a las subclases interactuar con ciertos detalles internos.
- Se recomienda mantener campos **privados** y solo usar `protected` cuando sea necesario.
- Miembros `protected` son accesibles también en clases que heredan de las clases derivadas (herencia en cadena).

Métodos de Extensión (Extension Methods) en C#
¿Qué son?

- Permiten **agregar nuevos métodos a tipos existentes** (clases o estructuras) sin necesidad de modificar su código original o usar herencia.
- Se definen como **métodos estáticos dentro de una clase estática**.
- El primer parámetro del método tiene la palabra clave `this` delante, y especifica el tipo que será extendido.

¿Por qué usarlos?

- No siempre es posible o recomendable usar herencia (por ejemplo, no se puede heredar de estructuras como `int`).
- Permite agregar funcionalidad a tipos ya existentes, incluso a tipos integrados del sistema como `int`, `string`, etc.
- Los métodos de extensión pueden usarse con la sintaxis de llamada a métodos de instancia, haciendo que el código sea más limpio y legible.

Ejemplo práctico: extender `int` con un método `Negate`

```
static class Utils
{
    public static int Negate(this int i)
    {
        return -i;
    }
}
```

- Aquí `Negate` es un método de extensión que “extiende” el tipo `int`.
- El parámetro `this int i` indica que `Negate` será invocable sobre cualquier `int`.
- Para usarlo, simplemente haces:

```
 int x = 591;
Console.WriteLine($"x.Negate {x.Negate()}");  // Imprime: x.Negate -
591
```

- No necesitas llamar explícitamente a `Utils.Negate(x)`, aunque también puedes:

```
Console.WriteLine($"x.Negate {Utils.Negate(x)}");
```

Limitaciones de la herencia para extender tipos integrados

- No puedes heredar de estructuras (`struct`) como `System.Int32`.
- Si creas una nueva clase que hereda de `int`, tendrías que cambiar todo tu código para usar esa clase en vez de `int`.
- Los métodos de extensión solucionan estos problemas al permitir extender `int` sin herencia ni cambios en el código que usa `int`.

Resumen rápido de la sintaxis

```
// Clase estática que contiene métodos de extensión
public static class MiClaseExtensiones
{
    // Método estático, primer parámetro con 'this' + tipo a
extender
    public static TipoRetorno MetodoExtension(this TipoAExtender
obj, ...)
    {
        // Implementación
    }
}
```

Qué son las propiedades en C#?

Una **propiedad** es un miembro de una clase o estructura que permite controlar el acceso a los campos (variables internas), manteniendo una sintaxis similar al acceso directo, como si fueran campos públicos, pero sin perder el control que ofrece la encapsulación. Esto significa que una propiedad combina las ventajas de un **campo** (acceso directo con una sintaxis clara) y de un **método** (capacidad de validar, calcular o restringir valores).
En lugar de acceder a los datos internos de una clase directamente (lo cual puede provocar errores o inconsistencias si se hace sin validación), se usan propiedades que permiten definir cómo se obtienen o modifican esos datos mediante dos componentes principales:

- `get`: se ejecuta cuando se accede al valor de la propiedad.
- `set`: se ejecuta cuando se intenta modificar el valor de la propiedad. Recibe un valor especial implícito llamado `value`, que es el dato que se quiere asignar.

¿Por qué usar propiedades en lugar de campos públicos?

Supongamos que una estructura representa una posición en pantalla con coordenadas X e Y. Si esos campos son públicos, cualquier parte del programa puede modificar sus valores sin ninguna restricción. Esto puede llevar a que se asignen valores inválidos (por ejemplo, una posición Y negativa o fuera del tamaño permitido del monitor).

```
ScreenPosition p = new ScreenPosition(100, 100);
p.Y = -100; // Esto es posible si Y es un campo público.
```

Para evitar esto, la solución clásica es hacer los campos privados y luego crear métodos públicos que permitan acceder o modificar su valor, validando en el proceso.

Pero esta solución introduce un problema de legibilidad: en lugar de usar `p.X = 10`, hay que llamar a métodos como `SetX(10)` o `GetX()`, lo cual es más verboso y menos natural.

Las propiedades solucionan este problema, ya que permiten seguir escribiendo `p.X = 10` (como si X fuera público), pero internamente llaman al código del método `set`, donde se puede validar el valor.

Sintaxis básica de una propiedad

```
private int _x;
public int X
{
    get { return _x; }
    set { _x = ValidarX(value); }
}
```

Esto significa:

- Cuando se accede a `X`, se ejecuta el código dentro de `get`, devolviendo el valor almacenado en `_x`.
- Cuando se asigna un valor a `X`, se ejecuta el `set`, que primero valida el valor antes de almacenarlo.

Campos y nombres: la convención `_x`

Una convención común en C# es que:

- Las **propiedades públicas** se escriban con **mayúscula inicial** (ej. `X`).
- Los **campos privados** se escriban con minúscula y, opcionalmente, con un guion bajo al inicio (ej. `_x`).

Esta convención evita confusiones entre el nombre del campo y el de la propiedad, ya que de otra forma podrías tener `int x` y `int X`, lo cual puede inducir a errores de lectura.

Aunque en general se recomienda no usar el guion bajo al inicio de los nombres, se hace una excepción en estos casos para facilitar la distinción entre campos privados y propiedades públicas.

Propiedades con expresión simplificada (expression-bodied)

Para propiedades sencillas, se puede usar una sintaxis más compacta:

```
public int X
{
    get => _x;
    set => _x = ValidarX(value);
}
```

Es exactamente lo mismo que la forma larga, pero más directa. No requiere la palabra `return` ni llaves si la operación es simple.

Importancia de las propiedades

Las propiedades permiten:

- Validar valores antes de asignarlos.
- Ocultar detalles internos de implementación (el consumidor de la clase no necesita saber si el dato está almacenado o calculado).
- Proteger la integridad de los datos.
- Mantener una sintaxis limpia y clara al trabajar con clases y estructuras.

USO DE PROPIEDADES EN C#

1. Contexto de lectura y escritura (read/write context)

Cuando usás una propiedad como `origin.X`, hay dos formas posibles de utilizarla:

Lectura (read context): cuando simplemente querés obtener su valor.

```
int xpos = origin.X; // Internamente llama a get
```

- **Escritura (write context)**: cuando querés cambiar su valor.

  ```
  origin.X = 40; // Internamente llama a set con value = 40
  ```

- La ventaja de las propiedades es que usás una **sintaxis idéntica a la de un campo**, pero por detrás el compilador traduce ese uso en una **llamada a los métodos `get` o `set`**. Esto significa que podés controlar y validar el acceso sin cambiar la manera en la que se escribe el código.

También existen **usos mixtos (read/write)**, por ejemplo:

```
origin.X += 10; // Internamente: llama a get, suma 10, luego llama a set
```

2. Propiedades estáticas

Del mismo modo que los campos y métodos pueden ser `static`, también lo pueden ser las propiedades. Las accedés directamente desde el tipo, sin necesidad de crear una instancia.

```
int altura = Pantalla.MaxAltura;
```

3. Propiedades de solo lectura (`get` solamente)

Podés declarar una propiedad que **solo permite lectura**. Esto es útil cuando querés que el valor esté disponible, pero no sea modificable desde fuera de la clase o estructura.

```
public int X { get => _x; }
```

Intentar asignar un valor causará un error en tiempo de compilación:

```
origin.X = 140; // Error: no tiene set
```

4. Propiedades de solo escritura (`set` solamente)

Aunque son menos comunes, también es posible definir una propiedad que **solo permita escritura**. En este caso, no podés leer el valor, pero sí asignarlo. Son útiles, por ejemplo, para contraseñas.

```
public string Password
{
    set => _password = Encriptar(value);
}
```

Intentar leerla causará un error:

```
Console.WriteLine(user.Password); // Error: no tiene get
```

5. Control de accesibilidad (`get` público, `set` privado)

Una propiedad puede tener acceso público en general, pero diferenciar el nivel de acceso de sus métodos `get` y `set`.

Por ejemplo:

```
public int X
{
    get => _x;
    private set => _x = Validar(value);
}
```

Esto permite **leer el valor desde fuera** de la clase, pero **evita que pueda modificarse** directamente desde afuera. Solo métodos internos de la clase podrán asignar el valor.

6. Restricciones importantes al usar propiedades

Hay algunas limitaciones específicas en el uso de propiedades:

No se pueden usar como `ref` o `out` en métodos

```
MyMethod(ref origin.X); // Error
```

- Esto se debe a que una propiedad es un método, no una dirección de memoria directa como un campo.
- **No se puede usar `const` con propiedades**. Las propiedades no pueden tener un valor constante porque son métodos, no valores fijos.
- **No se puede declarar más de un `get` o más de un `set` por propiedad**.
- **No se pueden pasar parámetros a `get` o `set`**. Solo reciben (implícitamente) `value` en el `set`.

7. Inicialización obligatoria antes de uso

En estructuras, si usás propiedades, debés inicializar el objeto antes de acceder a ellas. De lo contrario, obtenés un error de compilación:

```
ScreenPosition p;      // sin inicializar
p.X = 20;              // Error: no fue creado con `new`
```

Este tipo de restricciones no se aplica si usás directamente campos, lo que demuestra que usar propiedades agrega seguridad y control, pero también requiere cumplir reglas adicionales.

8. Buen diseño orientado a objetos: evitar "setters" innecesarios

Un punto fundamental es que **usar propiedades no significa automáticamente tener buen diseño**. La encapsulación no es solo ocultar los campos, sino **modelar el comportamiento** correctamente.

Un mal diseño sería esto:

```
public decimal Balance { get; set; }
```

Porque en el mundo real, no tiene sentido que alguien pueda modificar el saldo de su cuenta sin realizar una operación específica. Es mejor modelar esto con métodos:

```
public decimal Balance { get; private set; }
public void Deposit(decimal amount) { ... }
public bool Withdraw(decimal amount) { ... }
```

Esto es mucho más representativo del dominio (modelo del problema): la clase no es un simple contenedor de datos, sino una **unidad lógica con comportamientos bien definidos**.

Propiedades automáticas en C#

¿Por qué usar propiedades en lugar de campos públicos?

1. Compatibilidad con aplicaciones

- Campos y propiedades se compilan diferente.
- Si se expone un campo como público y luego se cambia a propiedad, se rompe la compatibilidad con programas ya compilados que usan esa clase.
- Las propiedades permiten agregar lógica interna más adelante sin afectar a quienes usan la clase.

2. Compatibilidad con interfaces

- Si una interfaz define una propiedad, no se puede implementar con un campo.
- Es obligatorio usar una propiedad que cumpla con el contrato de la interfaz.

Propiedades automáticas

Sintaxis:

```
public Tipo Nombre { get; set; }
```

Ejemplo:

```
class Circle
{
    public int Radius { get; set; }
}
```

Equivale a:

```
class Circle
{
    private int _radius;
    public int Radius
    {
        get { return _radius; }
        set { _radius = value; }
    }
}
```

Propiedades de solo lectura

Definición:

```
public DateTime FechaCreacion { get; }
```

Inicialización:

En el constructor:

```
public Circle()
{
    FechaCreacion = DateTime.Now;
}
```

1. En la declaración:

```
public DateTime FechaCreacion { get; } = DateTime.Now;
```

Nota: Si se usa ambos métodos, el valor asignado en el constructor sobrescribe al de la declaración.

Restricciones

No se pueden crear propiedades automáticas de solo escritura (sin `get`).

```
public int Edad { set; } // Error de compilación
```

Inicialización de objetos con propiedades

Alternativa a múltiples constructores:

```
var tri1 = new Triangle { Side1Length = 10, Side3Length = 20 };
```

- C# primero ejecuta el constructor por defecto y luego asigna los valores de las propiedades.
- Permite inicializar combinaciones sin definir muchos constructores con parámetros.

Propiedades `init` (C# 9.0+)

Uso:

```
class Grade
{
    public int StudentID { get; init; }
```

```
    public string Subject { get; init; }
    public char SubjectGrade { get; init; }
}
```

Inicialización:

```
var grade1 = new Grade { StudentID = 1, Subject = "Math",
SubjectGrade = 'A' };
```

No se puede modificar después:

```
grade1.SubjectGrade = 'B'; // Error de compilación
```

- Útil para mantener objetos inmutables después de ser creados.

Ejemplo de clase `Polygon`

```
class Polygon
{
    public int NumSides { get; set; }
    public double SideLength { get; set; }

    public Polygon()
    {
        this.NumSides = 4;
        this.SideLength = 10.0;
    }
}
```

- Las propiedades automáticas facilitan el código y permiten evolución futura sin romper compatibilidad.
- El uso de `init` permite crear propiedades inmutables pero inicializables.
- La sintaxis de inicialización con propiedades mejora la claridad y evita sobrecargar constructores.

Uso de Records con Propiedades para Implementar Estructuras Livianas

¿Qué es un *record*?

Un **record** en C# (introducido en la versión 9.0) es un tipo de dato **inmutable y liviano**, diseñado para representar entidades cuyos valores **no cambian** después de su creación. Permiten una sintaxis concisa y útil para objetos centrados en **datos** más que en **comportamiento**.

Ventajas de los records

1. Inmutabilidad incorporada: Las propiedades definidas en un record son automáticamente de solo lectura (`init`), lo que permite asignarlas una única vez, en el momento de la creación.

2. Sintaxis concisa: No es necesario escribir constructores ni métodos `Equals` y `GetHashCode` manualmente. El compilador los genera automáticamente.

3. Comparación por valor: A diferencia de las clases (comparadas por referencia), los records se comparan **por valor**. Dos records con los mismos valores son considerados iguales.

4. Eficiencia: Comparar por valor en estructuras comunes usa `Equals` de `ValueType`, que recurre a *reflection*, lo que puede ser ineficiente. Los records generan implementaciones optimizadas.

Ejemplo básico de record

```
public record Student(int Id, string Name, char Grade);
```

Este record define un tipo `Student` con tres propiedades de solo lectura, inicializables mediante un constructor implícito.

Uso:

```
var student1 = new Student(1, "Ana", 'A');
var student2 = new Student(1, "Ana", 'A');
Console.WriteLine(student1 == student2); // True (comparación por
valor)
```

Equivalente a clase inmutable tradicional

Este record:

```
public record Student(int Id, string Name, char Grade);
```

Es equivalente a:

```
public class Student
{
    public int Id { get; init; }
    public string Name { get; init; }
    public char Grade { get; init; }
    public Student(int id, string name, char grade)
    {
        Id = id;
        Name = name;
        Grade = grade;
    }
    public override bool Equals(object? obj) { ... } // Comparación
por valor
    public override int GetHashCode() { ... }        // Hash por
contenido
}
```

Consideraciones sobre igualdad

- Las clases comparan referencias: `object.ReferenceEquals(a, b)`
- Los records comparan valores: `a == b` si todas las propiedades son iguales.
- En estructuras, `Equals` es implementado por `ValueType` y usa *reflection*, lo cual es general pero lento.
- Si sobreescribís `Equals`, también **deberías sobreescribir `GetHashCode`**, para mantener coherencia.

Nota sobre UWP

- En **Universal Windows Platform (UWP)**, las estructuras no se basan en `ValueType`.
- El *runtime* de UWP genera automáticamente las implementaciones necesarias para `Equals` y `GetHashCode`.

Conclusión

Los records:

- Proveen una forma **compacta y eficiente** de crear tipos inmutables.

- Son ideales para representar **modelos de datos**.
- Evitan errores comunes en la implementación manual de comparación y hash.
- Mejoran el rendimiento y la claridad del código.

Creación de Tipos de Valor con Enumeraciones y Estructuras
Conceptos generales
En C#, existen dos tipos fundamentales:
- **Tipos de valor**: almacenan el dato directamente en la pila (stack).
- **Tipos de referencia**: almacenan una referencia a un objeto en el heap.

Las clases son tipos de referencia. Las **enumeraciones (enum)** y **estructuras (struct)** son **tipos de valor**.

Trabajando con Enumeraciones
¿Qué es una enumeración?
Una **enumeración** es un tipo que representa un conjunto finito de valores con nombres simbólicos. Sirve para reemplazar constantes numéricas poco claras con identificadores legibles.
Ejemplo básico:
```
enum Season { Spring, Summer, Fall, Winter }
```
Este enum define cuatro valores posibles: `Spring`, `Summer`, `Fall`, `Winter`.

Uso de una enumeración
Podés usar `Season` como cualquier otro tipo:
```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // Imprime "Fall"
```

Declaración en clase:
```
class Example
{
    private Season currentSeason;
    public void Method(Season parameter)
    {
        Season localVariable;
        ...
    }
}
```

Enumeraciones anulables
Podés permitir que una variable de enumeración almacene `null`:
```
Season? colorful = null;
```

Conversión a string
Todos los enums implementan `ToString()`:
```
string name = colorful.ToString();
Console.WriteLine(name); // "Fall"
```

Comparación y operaciones
Podés comparar valores de un enum:
```
if (colorful == Season.Fall) { ... }
```
También podés hacer aritmética (aunque no siempre tenga sentido).

Valores subyacentes

Cada elemento de un enum tiene un **valor entero** subyacente (por defecto, tipo `int`).

```
enum Season { Spring, Summer, Fall, Winter }

Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // Imprime 2
```

Podés asignar valores explícitos:

```
enum Season { Spring = 1, Summer, Fall, Winter }
// Ahora: Spring=1, Summer=2, Fall=3, Winter=4
```

Podés reutilizar valores:

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

Cambiar el tipo base del enum

Por defecto, el tipo base es `int`, pero podés especificar otro tipo entero:

```
enum Season : short { Spring, Summer, Fall, Winter }
```

Podés usar: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`.

Motivos para cambiar el tipo base:

- Optimización de memoria.
- Control del rango de valores posibles.

Uso de Arrays

¿Por qué usar arrays?

Las variables que ya conoces almacenan un **único dato** (como un número entero, un valor decimal, un objeto `Circle` o `Date`). Pero, ¿qué pasa si necesitas trabajar con un **conjunto de elementos**? Imaginate que tenés que guardar las edades de 100 personas o una lista de contraseñas. Crear una variable para cada una (`edad1`, `edad2`, `edad3`, etc.) es poco práctico porque:

- No sabrías de antemano cuántas variables crear.
- Tendrías nombres repetitivos y difíciles de manejar.
- Realizar la misma operación en cada variable (como aumentar todas las edades) requeriría un **código muy repetitivo**.
- A menudo, no sabes la cantidad exacta de elementos que necesitarás al momento de escribir tu programa (por ejemplo, al leer datos de una base de datos).

Los **arrays** resuelven estos problemas.

¿Qué es un array?

Un **array** es una **secuencia ordenada de elementos** que son **todos del mismo tipo**. A diferencia de los campos en una estructura o clase, que se acceden por su nombre y pueden ser de tipos diferentes, los elementos de un array se guardan en un bloque contiguo de memoria y se acceden usando un **índice numérico**.

Declarar variables array

Para declarar una variable que contendrá un array, se especifica el **tipo de los elementos**, seguido de un par de **corchetes** `[ ]`, y luego el **nombre de la variable**. Los corchetes indican que la variable es un array.

```
int[] pins; // Un array de números enteros llamado pins (para PINs)
Date[] dates; // Un array de estructuras Date
```

Consejo: Es común usar nombres en **plural** para las variables de array, como `personas` (donde cada elemento es una `Persona`) o `horarios` (donde cada elemento es un `Horario`).

Crear una instancia de array

Los arrays son **tipos por referencia**, lo que significa que la variable del array solo almacena una referencia a un bloque de memoria que contiene los elementos, y ese bloque se guarda en el **heap**. Declarar el array no asigna memoria para los elementos; solo reserva espacio para la referencia en la stack.

Para **crear la instancia del array** y asignarle el espacio real en memoria, usas la palabra clave `new` seguida del tipo de elemento y el **tamaño** deseado entre corchetes:

```
pins = new int[4]; // Crea un array de 4 enteros.
```

Cuando se crea un array con `new`, todos sus elementos se **inicializan automáticamente** con sus valores por defecto (por ejemplo, `0` para números, `null` para referencias, `false` para booleanos).

El **tamaño del array puede ser dinámico**; no tiene por qué ser una constante, lo que es muy útil:

```
int size = int.Parse(Console.ReadLine()); // Lee el tamaño desde la consola
int[] dynamicPins = new int[size]; // Crea un array con el tamaño leído
```

Incluso puedes crear un array con **tamaño 0**. No es un array nulo, sino uno que simplemente no contiene elementos, lo que puede ser útil en escenarios donde el tamaño se determina en tiempo de ejecución y podría no haber elementos.

Inicializar y usar arrays

Puedes **asignar valores específicos** a los elementos de un array directamente al momento de crearlo, usando una lista de valores separados por comas entre llaves `{}`:

```
int[] pins = new int[4] { 9, 3, 7, 2 }; // Inicializa el array con estos valores
```

Los valores dentro de las llaves no tienen que ser constantes; pueden ser resultados de cálculos:

```
Random r = new Random();
int[] randomPins = new int[4] { r.Next() % 10, r.Next() % 10, r.Next() % 10, r.Next() % 10 };
```

Es importante que la **cantidad de valores en la lista de inicialización coincida exactamente con el tamaño** del array que estás creando, de lo contrario, obtendrás un error en tiempo de compilación.

Si proporcionas una lista de inicializadores, puedes **omitir la expresión `new` y el tamaño**; el compilador de C# inferirá el tamaño automáticamente:

```
int[] pins = { 9, 3, 7, 2 }; // El compilador crea un array de 4 enteros
```

Si tus elementos son estructuras u objetos, puedes llamar a sus constructores directamente en la inicialización:

```
Time[] schedule = { new Time(12, 30), new Time(5, 30) };
```

Arrays implícitamente tipados

Puedes dejar que el compilador de C# **infiera el tipo de los elementos del array** por ti, usando la palabra clave `var` y `new[]`:

```
var names = new[] { "John", "Diana", "James", "Francesca" }; // C# deduce que es un array de strings
```

Con esta sintaxis, no incluyes los corchetes `[]` en la declaración de `var`, pero sí debes usar `new[]` antes de la lista de inicializadores. Es fundamental que **todos los inicializadores sean del mismo tipo**, o que C# pueda convertirlos de forma compatible (por ejemplo, puede convertir enteros a `double` si hay valores `double` en la lista).

Los arrays implícitamente tipados son muy útiles cuando trabajas con **tipos anónimos**:

```
var familyMembers = new[]
{
```

```
    new { Name = "Juan", Age = 57 },
    new { Name = "Maria", Age = 55 },
    new { Name = "Pedro", Age = 30 }
};
```

Acceder a elementos individuales

Para acceder a un elemento específico de un array, debes proporcionar un **índice** que indique su posición. Los índices de los arrays en C# son **cero-basados**, lo que significa que el primer elemento está en el **índice `0`**, el segundo en el `1`, y así sucesivamente.

```
int myPin;
myPin = pins[2]; // Lee el valor del tercer elemento (índice 2)
pins[2] = 1645; // Cambia el valor del tercer elemento
```

C# siempre **verifica los límites del array**. Si intentas usar un índice negativo o uno que sea igual o mayor que la longitud del array, el programa lanzará una excepción `IndexOutOfRangeException`, lo que te ayuda a detectar errores.

Acceder a series de elementos (rangos)

C# te permite extraer un **subconjunto contiguo de elementos** de un array usando una sintaxis de rango `x..y`. Esta secuencia incluye los elementos desde el índice `x` hasta el `y-1`. El resultado es un nuevo array con los elementos seleccionados.

```
var names = new[] { "John", "Diana", "James", "Francesca" };
var subset = names[0..2]; // 'subset' contendrá "John" y "Diana" (elementos en índice 0 y 1)
```

También puedes especificar índices contando desde el final del array usando el operador `^`. Por ejemplo, `^1` es el último elemento, `^2` el penúltimo, etc.

```
var subsetFromEnd = names[^3..]; // 'subsetFromEnd' contendrá los últimos tres elementos
```

Iterando a Través de un Array

Los arrays en C# son instancias de la clase `System.Array`, que ofrece propiedades y métodos útiles.

Usando un bucle `for`

Puedes usar la propiedad `Length` de un array para saber cuántos elementos contiene. Luego, un bucle `for` te permite recorrer todos los elementos, accediendo a cada uno por su índice.

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++) // Recorre desde el índice 0 hasta Length-1
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```

Es importante recordar que los arrays empiezan en el índice `0`, y el último elemento está en el índice `Length - 1`.

Usando `foreach` (la forma preferida)

Para evitar preocupaciones sobre los índices y la longitud del array, C# ofrece la instrucción `foreach`. Esta es la forma más común y recomendada para iterar por todos los elementos de un array.

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins) // 'pin' tomará el valor de cada elemento en el array 'pins'
{
    Console.WriteLine(pin);
}
```

La variable de iteración (`int pin` en este caso) automáticamente toma el valor de cada elemento. El tipo de esta variable debe coincidir con el tipo de los elementos del array.

¿Cuándo usar `for` en lugar de `foreach`? Aunque `foreach` es preferido, hay situaciones donde `for` es necesario:

- Si solo quieres recorrer **una parte** del array o **saltar elementos** (ej., cada tercer elemento).
- Si necesitas iterar en **orden inverso** o en una secuencia específica no lineal.
- Si necesitas conocer el **índice** de cada elemento además de su valor.
- Si necesitas **modificar los elementos del array** dentro del bucle. La variable de iteración de `foreach` es una **copia de solo lectura** del elemento.

Nota: Es seguro usar `foreach` en un array de tamaño cero; simplemente no ejecutará el cuerpo del bucle ninguna vez.

Puedes usar `var` con `foreach` si no sabes el tipo exacto de los elementos, o si trabajas con tipos anónimos:

```
var familyMembers = new[]
{
   new { Name = "John", Age = 57 },
   new { Name = "Diana", Age = 57 }
};
foreach (var familyMember in familyMembers)
{
   Console.WriteLine($"Name: {familyMember.Name}, Age: {familyMember.Age}");
}
```

Pasar Arrays como Parámetros o Valores de Retorno

Puedes definir métodos que acepten arrays como argumentos o que devuelvan arrays.

Arrays como parámetros

La sintaxis para pasar un array como parámetro es similar a su declaración:C#

```
public void ProcessData(int[] data) // El método espera un array de enteros
{
   foreach (int i in data)
   {
      // ... procesar cada elemento ...
   }
}
```

Dado que los arrays son tipos por referencia, si modificas el contenido de un array dentro de un método, esos cambios serán visibles en todas las referencias al array, incluida la original que se pasó como argumento.

Arrays como valores de retorno

Para que un método devuelva un array, especificas el tipo del array como el tipo de retorno del método. Dentro del método, creas y llenas el array, y luego lo retornas.

```
public int[] ReadData() // El método retornará un array de enteros
{
   Console.WriteLine("¿Cuántos elementos?");
   string reply = Console.ReadLine();
   int numElements = int.Parse(reply);

   int[] data = new int[numElements]; // Crea el array
   for (int i = 0; i < numElements; i++)
   {
      Console.WriteLine($"Ingresa dato para el elemento {i}");
      reply = Console.ReadLine();
```

```
        int elementData = int.Parse(reply);
        data[i] = elementData; // Llena el array
    }
    return data; // Retorna el array
}
// Así se llamaría al método:
int[] myData = ReadData();
```

Parámetros de array en el método `Main`

Probablemente hayas notado que el método `Main` de una aplicación de C# siempre tiene un array de strings como parámetro:

```
static void Main(string[] args)
{
    // ...
}
```

Cuando inicias una aplicación desde la línea de comandos, puedes pasar argumentos adicionales (separados por espacios). El sistema operativo y el CLR (Common Language Runtime) pasan estos argumentos al método `Main` a través de este array `string[] args`. Esto permite que el usuario proporcione información al iniciar el programa.

```
// Ejemplo de uso de argumentos de línea de comandos en Main
static void Main(string[] args)
{
    foreach (string filename in args)
    {
        // ProcessFile(filename); // Un método que procesaría cada archivo
    }
}
// Un usuario podría ejecutarlo así: MiUtilDeArchivos archivo1.txt archivo2.log
```

Copiar Arrays

Dado que los arrays son tipos por referencia, al "copiar" una variable de array, lo que realmente haces es crear una **nueva referencia al mismo array original**.

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // 'alias' y 'pins' ahora apuntan al MISMO array en memoria.
// Si modificas pins[1], alias[1] también reflejará el cambio.
```

Si necesitas una **copia independiente del contenido del array** (una copia real de los datos), debes:

1. Crear una **nueva instancia de array** del mismo tipo y tamaño.
2. **Copiar los datos** elemento por elemento del array original al nuevo array.

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length]; // 1. Crea un nuevo array
for (int i = 0; i < pins.Length; i++)
{
    copy[i] = pins[i]; // 2. Copia los elementos
}
```

La clase `System.Array` ofrece métodos útiles para copiar arrays de forma más sencilla:

`CopyTo` (método de instancia): Copia los elementos del array actual a otro array, a partir de un índice de destino especificado.

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0); // Copia 'pins' a 'copy' desde el índice 0
```

- **`Array.Copy` (método estático):** Copia un número específico de elementos de un array de origen a un array de destino. El array de destino debe estar inicializado.

```
int[] pins = { 9, 3, 7, 2 };
```

```
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length); // Copia 'pins' a 'copy' toda su longitud
```

- **`Clone` (método de instancia):** Crea una nueva instancia de array que es una copia superficial del array actual. Necesita un "cast" (conversión explícita) al tipo de array correcto.

```
 int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone(); // Crea y copia en un solo paso
```

- **Importante:** `CopyTo`, `Array.Copy` y `Clone` realizan una **copia superficial**. Si los elementos del array son tipos por referencia (objetos), solo se copian las referencias a esos objetos, no los objetos en sí. Ambas arrays apuntarán a los mismos objetos. Para una "copia profunda" (donde también se copien los objetos referenciados), deberás hacer un bucle y clonar cada objeto individualmente.

Usando Arrays Multidimensionales

Hasta ahora, hemos visto arrays de una sola dimensión (como listas). Pero puedes crear arrays con más de una dimensión.

Arrays "rectangulares"

Son arrays donde cada dimensión tiene una forma regular, como una tabla (filas y columnas) o un cubo.

Array bidimensional (matriz): Se declara con comas dentro de los corchetes para indicar las dimensiones. El acceso a los elementos requiere dos índices.

```
int[,] items = new int[4, 6]; // Un array de 4 filas y 6 columnas (24 elementos)
items[2, 3] = 99; // Accede al elemento en la fila 2, columna 3
items[2, 4] = items[2, 3];
items[2, 4]++;
```

Arrays con más dimensiones: No hay límite en el número de dimensiones, aunque el uso práctico rara vez excede tres.

```
int[,,] cube = new int[5, 5, 5]; // Un array tridimensional (5x5x5 = 125 elementos)
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

Advertencia: Los arrays multidimensionales pueden consumir **mucha memoria**. Un array con pocas dimensiones puede tener muchísimos elementos. Siempre considera manejar posibles excepciones `OutOfMemoryException` cuando uses arrays grandes o con muchas dimensiones.

Arrays "escalonados" (`Jagged Arrays`)

A diferencia de los arrays rectangulares (donde todas las "filas" tienen la misma longitud de "columnas"), un array escalonado es un **array de arrays**, donde cada sub-array puede tener una longitud diferente. Esto es útil para optimizar la memoria si no todas las "filas" necesitan el mismo número de "columnas".

```
int[][] items = new int[4][]; // Un array de 4 elementos, donde cada elemento será un array de
int
// Ahora inicializas cada uno de esos 4 elementos con arrays de diferentes longitudes
items[0] = new int[3];    // La primera 'fila' tiene 3 elementos
items[1] = new int[10];   // La segunda 'fila' tiene 10 elementos
items[2] = new int[40];   // La tercera 'fila' tiene 40 elementos
items[3] = new int[25];   // La cuarta 'fila' tiene 25 elementos
// Para acceder a un elemento:
items[0][1] = 99; // Accede al segundo elemento de la primera 'fila'
```

La declaración `int[][] items;` significa "un array de arrays de enteros".

Estos arrays escalonados son más flexibles y pueden ahorrar memoria si tus datos no encajan perfectamente en una estructura rectangular. Puedes extender esta idea a "arrays de arrays de arrays", y así sucesivamente, para crear estructuras más complejas y eficientes en el uso de la memoria.
Claro, aquí tienes el resumen sobre los arrays de parámetros en C#, con el formato solicitado:

Entendiendo los Arrays de Parámetros
¿Por qué usar arrays de parámetros?
A veces, necesitas escribir métodos que puedan aceptar **cualquier número de argumentos**, e incluso que esos argumentos sean de **diferentes tipos**. Si bien la sobrecarga de métodos (definir varias versiones de un método con el mismo nombre pero diferentes parámetros) puede ayudar, no siempre es la mejor solución, especialmente cuando el número de parámetros puede variar mucho o cuando los tipos son completamente impredecibles. Los **arrays de parámetros** (`params`) resuelven este problema.

Repaso de la sobrecarga de métodos
La **sobrecarga** consiste en definir dos o más métodos con el mismo nombre en el mismo ámbito, pero con diferentes listas de parámetros (diferente número o tipo de argumentos). Un ejemplo clásico es el método `Console.WriteLine` en C#, que está sobrecargado para aceptar y mostrar valores de diversos tipos (enteros, decimales, booleanos, cadenas, etc.).

```
class Console
{
   public static void WriteLine(Int32 value) { /* ... */ }
   public static void WriteLine(Double value) { /* ... */ }
   public static void WriteLine(Boolean value) { /* ... */ }
   public static void WriteLine(String value) { /* ... */ }
   // ... y muchas más versiones
}
```

Aunque la sobrecarga es muy útil, no maneja bien la situación en la que el **número de parámetros varía continuamente**, incluso si el tipo es el mismo. Por ejemplo, si quisieras un `WriteLine` que tomara 2, 3, 4 o más cadenas, necesitarías un sinfín de sobrecargas, lo cual sería tedioso y redundante.

Usando argumentos de array
Una alternativa a la sobrecarga es hacer que un método acepte un **único array como parámetro**. Por ejemplo, un método `Min` que encuentre el valor mínimo en un conjunto de enteros:

```
class Util
{
   public static int Min(int[] paramList) // Acepta un array de enteros
   {
      if (paramList == null || paramList.Length == 0)
      {
         throw new ArgumentException("La lista no puede estar vacía.");
      }
      int currentMin = paramList[0];
      foreach (int i in paramList)
      {
         if (i < currentMin)
         {
            currentMin = i;
         }
```

```
        }
        return currentMin;
    }
}
```

Para usar este método, tendrías que crear y llenar un array explícitamente cada vez que lo llames:

```
int first = 10, second = 5;
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array); // Se pasa el array explícitamente

// O usando un array anónimo (más conciso, pero sigue siendo un array):
int min2 = Util.Min(new int[] { first, second, 3, 8 });
```

Esta solución evita muchas sobrecargas, pero introduce la "molestia" de tener que crear el array para cada llamada.

Declarando un array de parámetros (`params`)

La palabra clave **`params`** permite que un método acepte un **número variable de argumentos** de una manera más limpia. El compilador de C# se encarga de crear el array por ti.

Para usarlo, pones `params` antes del tipo de array en la declaración del parámetro del método:

```
class Util
{
    public static int Min(params int[] paramList) // ¡Ahora con 'params'!
    {
        // El código del método es exactamente el mismo que antes.
        // El compilador se encarga de construir el array.
    }
}
```

Ahora, puedes llamar al método `Min` con cualquier número de argumentos enteros directamente, sin crear el array manualmente:

```
int first = 10, second = 5, third = 12;
int min1 = Util.Min(first, second);      // Se ve como si pasaras dos enteros
int min2 = Util.Min(first, second, third); // O tres enteros
int min3 = Util.Min();                   // ¡Incluso cero enteros (pasaría un array vacío)!
```

El compilador transformará estas llamadas en código que crea y llena un array, y luego llama al método `Min` con ese array, tal como lo harías manualmente.

Puntos importantes sobre los arrays `params`:

- **No se pueden usar con arrays multidimensionales** (ej. `params int[,] table` no compilará).
- El keyword `params` **no forma parte de la firma del método** para sobrecarga. No puedes tener un método `Min(int[] paramList)` y otro `Min(params int[] paramList)`; el compilador no podría distinguirlos.
- **No se puede usar `ref` o `out`** con arrays `params`.
- Un array `params` **debe ser el último parámetro** en la lista de parámetros de un método. Solo puede haber **un array `params` por método**.
- Si existe una **sobrecarga sin `params`** que coincide exactamente con los argumentos, esa **tendrá prioridad**. Esto puede ser una optimización para casos comunes (ej., tener `Min(int a, int b)` y `Min(params int[] list)`).

Usando `params object[]` (para cualquier tipo de argumento)

Si necesitas que el número de argumentos varíe Y también sus **tipos**, puedes usar un array de parámetros de tipo `object` (`params object[]`). Esto es posible porque `object` es el tipo base de todos los demás tipos en C# (y el "boxing" permite tratar los tipos de valor como objetos).

```
class BlackHole
{
    public static void LogAll(params object[] paramList) // Acepta cualquier tipo de argumento
    {
        foreach (object item in paramList)
        {
            Console.WriteLine(item);
        }
    }
}
```

Este método puede recibir:

- **Ningún argumento:** `BlackHole.LogAll();` (el compilador pasaría un `object[]` vacío).
- **`null`:** `BlackHole.LogAll(null);` (un array nulo).

Un array explícito:

```
object[] data = new object[2];
data[0] = "Hola";
data[1] = 123;
BlackHole.LogAll(data);
```

- **Argumentos de diferentes tipos:**

```
 BlackHole.LogAll("Mensaje", 42, true, new DateTime(2025, 1, 1));
// Esto se convierte a: BlackHole.LogAll(new object[]{"Mensaje", 42, true, new
DateTime(2025, 1, 1)})
```

- El `params object[]` es una herramienta muy flexible cuando la cantidad y los tipos de los argumentos son totalmente variables.

Creando Interfaces y Definiendo Clases Abstractas

¿Qué es una interfaz?

Las interfaces son un pilar fundamental de la programación orientada a objetos en C#. A diferencia de las clases, una interfaz **no contiene ningún código ni datos de implementación**. Su propósito es definir un **contrato**: especifica qué métodos y propiedades una clase o estructura *debe* proporcionar si quiere "implementar" esa interfaz. Esto permite **separar completamente lo "qué" se debe hacer de lo "cómo" se hace**. La interfaz solo describe la funcionalidad que una clase debe ofrecer, sin preocuparse por los detalles de su implementación.

Ejemplo de necesidad: Imagina que quieres crear una colección que pueda ordenar objetos. Si los objetos son cadenas, los ordenarías alfabéticamente; si son números, numéricamente. ¿Cómo hacer que la colección ordene objetos de tipos que no conoces de antemano? La solución es requerir que esos objetos implementen un método `CompareTo` que la colección pueda usar para compararlos. Una interfaz define este requisito.

Definiendo una interfaz

Definir una interfaz es similar a definir una clase, pero usas la palabra clave `interface`. Dentro de la interfaz, declaras los métodos y propiedades, pero **nunca especificas un modificador de acceso** (como `public`, `private`); todos son implícitamente públicos. Además, los métodos de una interfaz **no tienen implementación**; solo son declaraciones y terminan con un punto y coma.

```
interface IComparable
{
    int CompareTo(object obj); // Declaración de método sin cuerpo
}
```

Convención de nombres: La documentación de .NET recomienda prefijar los nombres de las interfaces con la letra `I` mayúscula (ej., `IComparable`, `ILandBound`).
Restricción importante: Una interfaz **no puede contener datos (campos)**, ni siquiera campos privados. Tampoco puede tener constructores ni destructores, ya que estos son detalles de implementación de clases o estructuras.

Implementando una interfaz

Para implementar una interfaz, una clase o estructura debe **heredar de ella** y **proporcionar una implementación para cada método y propiedad** especificado en la interfaz.

```
interface ILandBound // Interfaz para mamíferos terrestres
{
    int NumberOfLegs();
}

class Horse : ILandBound // La clase Horse implementa ILandBound
{
    // ... otros miembros de Horse ...

    public int NumberOfLegs() // Implementación del método de la interfaz
    {
        return 4;
    }
}
```

Reglas de implementación:

- El nombre del método y el tipo de retorno deben **coincidir exactamente**.
- Los parámetros (incluyendo `ref` y `out`) deben **coincidir exactamente**.
- Los métodos que implementan una interfaz deben ser **públicamente accesibles**, a menos que uses una implementación explícita (ver más abajo).

Una clase puede heredar de otra clase **Y** al mismo tiempo implementar una o varias interfaces. La clase base se nombra primero, seguida de las interfaces separadas por comas.

```
class Mammal { /* ... */ }

class Horse : Mammal, ILandBound // Hereda de Mammal e implementa ILandBound
{
    // ...
}
```

Extensión de interfaces: Una interfaz puede heredar de otra interfaz. Una clase que implementa la interfaz "hija" deberá implementar todos los métodos de ambas interfaces.

Referenciando una clase a través de su interfaz

De forma similar a la herencia de clases, puedes referenciar un objeto usando una variable del tipo de una interfaz que su clase implementa.

```
Horse myHorse = new Horse();
ILandBound iMyHorse = myHorse; // Legal: un objeto Horse puede ser referenciado como ILandBound
```

Esto es muy útil porque permite escribir métodos que acepten **cualquier tipo de objeto siempre y cuando implemente una interfaz específica**:

```
int FindLandSpeed(ILandBound landBoundMammal) // Acepta cualquier cosa que sea ILandBound
{
    // ...
}
```

Puedes verificar si un objeto implementa una interfaz usando el operador `is`:

```
if (myObject is ILandBound)
{
    ILandBound landAnimal = myObject as ILandBound; // Convertir al tipo de interfaz
    // ...
}
```

Nota: Al referenciar un objeto a través de una interfaz, solo puedes invocar los métodos y propiedades definidos en esa interfaz.

Trabajando con múltiples interfaces

Una clase puede implementar **un número ilimitado de interfaces**, aunque solo puede heredar de una única clase base. La clase debe implementar todos los métodos de todas las interfaces que declara.

Cuando una clase implementa varias interfaces, se listan separadas por comas después de la clase base (si la hay):

```
interface IGrazable { void ChewGrass(); } // Nueva interfaz

class Horse : Mammal, ILandBound, IGrazable // Implementa múltiples interfaces
{
    // ... debe implementar NumberOfLegs() y ChewGrass() ...
}
```

Implementación explícita de una interfaz

Si dos interfaces diferentes tienen un método con el mismo nombre y firma, y tu clase implementa ambas, puede haber ambigüedad sobre qué método se está implementando. La **implementación explícita de interfaz** resuelve esto.

Para implementar un método de interfaz explícitamente, prefijas el nombre del método con el nombre de la interfaz y **no le pones un modificador de acceso** (porque los métodos explícitos son siempre privados para la clase, solo visibles a través de la interfaz).

```
interface IJourney { int NumberOfLegs(); } // Otra interfaz con el mismo nombre de método

class Horse : ILandBound, IJourney
{
    // ...
    int ILandBound.NumberOfLegs() // Implementación explícita para ILandBound
    {
        return 4; // Las 4 patas del caballo
    }

    int IJourney.NumberOfLegs() // Implementación explícita para IJourney
    {
        return 3; // Las 3 "etapas" o "patas" del viaje
    }
}
```

Cuando un método de interfaz se implementa explícitamente, **no puede ser invocado directamente a través de la variable de la clase**. Debes referenciar el objeto a través de la interfaz correspondiente:

```
Horse horse = new Horse();
IJourney journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs(); // Llama al método de IJourney

ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs(); // Llama al método de ILandBound
```

Recomendación: Es buena práctica implementar interfaces explícitamente siempre que sea posible, para mayor claridad y para evitar conflictos de nombres.

Manejo de versiones con interfaces

Históricamente, modificar una interfaz existente (ej., añadir un nuevo método) rompía todas las clases que ya la implementaban, obligándolas a actualizarse.

C# 9+ aborda esto permitiendo **implementaciones por defecto de métodos en interfaces**:

```
public interface IMyInterface
{
    public void DoSomeWork();
    public void DoAdditionalWork() // Nueva característica con implementación por defecto
    {
        throw new NotImplementedException(); // Una implementación simple por defecto
    }
}
```

Si una clase implementa `IMyInterface` pero no define `DoAdditionalWork`, automáticamente usará esta implementación por defecto. Si una nueva clase quiere proporcionar su propia implementación, simplemente la define (sin `override`):

```
public class myNewClass : IMyInterface
{
    void IMyInterface.DoSomeWork() { /* ... */ }
    void IMyInterface.DoAdditionalWork() { /* Mi propia implementación */ }
}
```

Esto permite evolucionar las interfaces sin romper la compatibilidad con el código existente.

Restricciones de las interfaces

Las interfaces son para definir "formas", no implementaciones completas. Por lo tanto, tienen las siguientes restricciones:

- **No pueden definir campos** (ni siquiera `static`).
- **No pueden definir constructores**.
- **No pueden definir destructores**.
- **No puedes especificar un modificador de acceso** para los métodos; todos son implícitamente `public`.
- **No pueden anidar otros tipos** (como enumeraciones, estructuras, clases o interfaces).
- Una interfaz **no puede heredar de una estructura, registro o clase**, solo de otra interfaz.

Clases Abstractas

Cuando trabajas con interfaces y herencia, es común que varias clases derivadas compartan una **implementación común** para ciertos métodos o características. Si simplemente copias y pegas este código, terminas con **duplicación de código**, lo que dificulta el mantenimiento y la evolución de tu aplicación.

Ejemplo de duplicación: Si tienes clases `Horse` y `Sheep` que implementan una interfaz `IGrazable`, es probable que la lógica para `ChewGrass()` sea idéntica en ambas:

```
class Horse : Mammal, ILandBound, IGrazable
```

```
{
    // ...
    void IGrazable.ChewGrass()
    {
        Console.WriteLine("Masticando pasto"); // Código duplicado
    }
}

class Sheep : Mammal, ILandBound, IGrazable
{
    // ...
    void IGrazable.ChewGrass()
    {
        Console.WriteLine("Masticando pasto"); // Mismo código
    }
}
```

Una solución a esto es introducir una **nueva clase en la jerarquía** que encapsule esa lógica común.

```
class GrazingMammal : Mammal, IGrazable // Nueva clase para mamíferos que pastan
{
    // ...
    void IGrazable.ChewGrass()
    {
        Console.WriteLine("Masticando pasto"); // Implementación común aquí
    }
}
class Horse : GrazingMammal, ILandBound // Ahora Horse hereda de GrazingMammal
{
    // ...
}

class Sheep : GrazingMammal, ILandBound // Y Sheep también
{
    // ...
}
```

Esto reduce la duplicación, pero surge un nuevo problema: `GrazingMammal` (y `Mammal`) ahora son clases de las que podrías crear instancias. Sin embargo, su propósito es ser una **abstracción**, un punto para compartir funcionalidad, no una entidad que tenga sentido instanciar directamente.

Declarando Clases Abstractas

Para indicar que una clase está diseñada solo para ser heredada y que **no se pueden crear instancias directamente de ella**, se la declara como `abstract` usando la palabra clave `abstract`.

```
abstract class GrazingMammal : Mammal, IGrazable // ¡Ahora es abstracta!
{
    // ...
}
```

Si intentas crear un objeto de una clase abstracta, el código **no compilará**:

```
GrazingMammal myGrazingMammal = new GrazingMammal(); // ¡ILEGAL! Error de compilación
```

Métodos Abstractos

Una clase abstracta puede contener **métodos abstractos**. Un método abstracto es similar a un método virtual (que las clases derivadas pueden sobrescribir), pero con una diferencia clave: **no tiene un cuerpo de implementación** en la clase abstracta.

Una clase derivada de una clase abstracta **debe sobrescribir e implementar todos los métodos abstractos** definidos en la clase base. Un método abstracto no puede ser privado.

```
abstract class GrazingMammal : Mammal, IGrazable
{
    public abstract void DigestGrass(); // Método abstracto: sin cuerpo, debe ser
implementado por derivados
    // ...
}
```

Un método abstracto es útil cuando no tiene sentido proporcionar una implementación por defecto en la clase base abstracta, pero quieres **asegurarte** de que cualquier clase que herede de ella proporcione su propia implementación específica de ese método.

Diferencia clave con interfaces con implementaciones por defecto: Aunque una clase abstracta y una interfaz con implementaciones por defecto pueden parecer similares, son fundamentalmente diferentes:

- Una **clase abstracta** puede contener **campos, constructores, un destructor, métodos privados** y otros miembros comunes de una clase.
- Una **interfaz** es solo una **especificación** de lo que una clase *debería* parecer; no contiene detalles de implementación a nivel de datos.

Clases Selladas (`Sealed Classes`)

Diseñar una jerarquía de herencia es complejo. Si creas una interfaz o una clase abstracta, lo haces con la intención de que otras clases hereden de ellas. Sin embargo, no siempre querrás que todas tus clases puedan ser usadas como clases base.

La palabra clave **`sealed`** se usa para **evitar que una clase pueda ser utilizada como clase base**. Si declaras una clase como `sealed`, ninguna otra clase podrá heredar de ella, y cualquier intento generará un error de compilación.

```
sealed class Horse : GrazingMammal, ILandBound // La clase Horse es sellada
{
    // ...
}
```

Notas importantes sobre las clases selladas:

- Una clase sellada **no puede declarar ningún método virtual**.
- Una clase `abstract` **no puede ser `sealed`** (porque su propósito es ser heredada).

Métodos Sellados (`Sealed Methods`)

También puedes usar la palabra clave `sealed` para **sellar un método individual** dentro de una clase que *no es* sellada. Esto significa que las clases derivadas no podrán sobrescribir ese método específico.

Un método solo puede ser sellado si ha sido declarado con la palabra clave **`override`**. Se declara como `sealed override`.

Puedes entender la relación entre estas palabras clave así:

- Una **interfaz** introduce el nombre de un método.
- Un método **`virtual`** es la **primera implementación** de un método que puede ser sobrescrito.

- Un método **`override`** es **otra implementación** de un método virtual de una clase base.
- Un método **`sealed`** (usado con `override`) es la **implementación final** de un método virtual; ninguna clase más abajo en la jerarquía podrá sobrescribirlo.

Implementando y Usando Clases Abstractas

Las clases abstractas son muy útiles para **racionalizar el código y evitar duplicación** entre clases relacionadas, capturando detalles de implementación comunes. Por ejemplo, si tienes clases `Square` y `Circle` que comparten mucha lógica de dibujo, puedes mover ese código común a una clase abstracta `DrawingShape`, mejorando la mantenibilidad.

Introducción a los Genéricos

¿Cuál es el propósito de los genéricos?

Anteriormente, aprendiste a usar el tipo `object` para referirte a instancias de cualquier clase, lo que te daba mucha flexibilidad al pasar y devolver valores de diferentes tipos. Sin embargo, esta flexibilidad tiene un costo: recae en el programador recordar qué tipo de datos se está utilizando, lo que puede llevar a **errores en tiempo de ejecución** (`InvalidCastException`) y la necesidad de realizar **conversiones (`casting`) explícitas**. Además, el uso de `object` con tipos de valor (como `int` o `float`) puede generar **sobrecarga de rendimiento** debido a las operaciones de "boxing" y "unboxing". Los **genéricos** son una característica de C# diseñada para **prevenir estos errores en tiempo de ejecución**, mejorar la **seguridad de tipos**, **reducir el boxing** y facilitar la creación de clases y métodos generalizados, permitiendo escribir código más robusto y eficiente.

El problema: Problemas con el tipo `object`

Para entender los genéricos, veamos un ejemplo con una clase `Queue` (cola, una estructura FIFO: primero en entrar, primero en salir):

```
// Versión inicial de Queue que solo maneja 'int'
class Queue
{
    private int[] data; // Solo puede almacenar enteros
    // ... métodos Enqueue(int item) y Dequeue() que devuelve int ...
}
```

Esta `Queue` funciona bien para enteros, pero si quieres una cola de `string`, `float`, o tus propios objetos como `Circle` o `Horse`, tendrías que crear una clase `Queue` diferente para cada tipo, o modificarla para que use `object`:

```
// Versión de Queue que usa 'object'
class Queue
{
    private object[] data; // Ahora puede almacenar cualquier cosa
    // ...
    public void Enqueue(object item) { /* ... */ }
    public object Dequeue() { /* ... */ return this.data[this.tail]; }
}
```

Con la versión de `object`, puedes encolar cualquier tipo:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Legal: Horse es un objeto
```

Sin embargo, al desencolar, **debes hacer un `cast` explícito** para recuperar el tipo original:

```
Horse dequeuedHorse = (Horse)queue.Dequeue(); // Necesitas hacer el casting
```

Si olvidas el `cast` o lo haces incorrectamente, obtendrás errores:

- `Cannot implicitly convert type 'object' to 'Horse'` (error de compilación si no haces el cast).
- `System.InvalidCastException` (error en tiempo de ejecución si intentas convertir a un tipo incorrecto, ej., `(Circle)queue.Dequeue()` cuando en realidad es un `Horse`). El compilador no puede detectar este error porque solo ve `object`.

Además, para tipos de valor (`int`, `float`), el uso de `object` implica **boxing** (empaquetar el tipo de valor en un objeto) al encolar y **unboxing** (desempaquetar el objeto de nuevo en un tipo de valor) al desencolar. Estas operaciones, aunque transparentes, consumen memoria y tiempo de procesador, lo que puede afectar el rendimiento en gran escala.

La solución: Genéricos

Los genéricos resuelven estos problemas al permitirte definir clases y métodos que operan sobre un **tipo de datos generalizado**, especificado como un **parámetro de tipo** en el momento de la instanciación.

En C#, se indica que una clase es genérica añadiendo un **parámetro de tipo entre corchetes angulares `< >`** después del nombre de la clase. Comúnmente se usa `T` (de "Type") como marcador de posición.

```
class Queue<T> // La 'T' es un marcador de posición para el tipo
{
    private T[] data; // El array ahora es de tipo 'T'
    // ...

    public Queue()
    {
        this.data = new T[DEFAULTQUEUESIZE]; // Usa 'T' para crear el array
    }

    public void Enqueue(T item) // El parámetro del método es de tipo 'T'
    {
        // ...
    }

    public T Dequeue() // El valor de retorno es de tipo 'T'
    {
        // ...
        T queueItem = this.data[this.tail];
        return queueItem;
    }
}
```

Cuando instanciamos la clase `Queue<T>`, **sustituimos `T` por un tipo real** (ej., `int`, `Horse`, `Person`). El compilador genera una nueva versión de la clase específica para ese tipo.

```
Queue<int> intQueue = new Queue<int>();       // Una cola de enteros
Queue<Horse> horseQueue = new Queue<Horse>(); // Una cola de objetos Horse
```

Beneficios clave de los genéricos:

Seguridad de tipos en tiempo de compilación: El compilador ahora sabe el tipo exacto que la cola contendrá.

```
intQueue.Enqueue(99);
int myInt = intQueue.Dequeue();        // ¡No se necesita casting!
Horse myHorse = intQueue.Dequeue();    // ¡Error de compilación! El compilador detecta el error.
```

- **Reducción de `boxing`/`unboxing`:** Para tipos de valor, el compilador genera código específico que evita estas operaciones, mejorando el rendimiento.
- **Código más limpio y legible:** No hay necesidad de `casts` constantes.

Importante: La sustitución del parámetro de tipo `T` no es solo textual. El compilador realiza una **sustitución semántica completa**, generando una nueva clase `construida` y optimizada para el tipo especificado. Así, `Queue<int>` y `Queue<Horse>` son tipos **completamente diferentes**, aunque compartan la misma lógica definida por la clase genérica `Queue<T>`.
Puedes usar **múltiples parámetros de tipo** (ej., `Dictionary<TKey, TValue>`) y definir **estructuras e interfaces genéricas** con la misma sintaxis.

Genéricos vs. Clases Generalizadas
Una **clase generalizada** (como la `Queue` basada en `object` anterior) tiene una **única implementación** que maneja todos los tipos a través del tipo `object`, requiriendo `casts` y boxing/unboxing.
Una **clase genérica** (`Queue<T>`) define una **plantilla**. Cada vez que la usas con un parámetro de tipo diferente (ej., `Queue<int>`, `Queue<Horse>`), el compilador **genera una clase completamente nueva y específica para ese tipo**. Son tipos distintos pero con el mismo comportamiento.

Genéricos y Restricciones (`Constraints`)
A veces, necesitas asegurarte de que el tipo usado para un parámetro genérico cumpla ciertos requisitos (ej., que implemente una interfaz específica o que sea un tipo de valor/referencia). Para esto, usas **restricciones (`constraints`)** con la palabra clave `where`.

```
// Restricción de interfaz: 'T' debe implementar IPrintable
public class PrintableCollection<T> where T : IPrintable
{
    // ...
}

// Restricción de tipo de valor: 'T' debe ser una estructura (struct)
public class StructCollection<T> where T : struct
{
    // ...
}

// Restricción de tipo de referencia: 'T' debe ser una clase (class)
public class ClassCollection<T> where T : class
{
    // ...
}
```

El compilador verificará estas restricciones en tiempo de compilación. Si el tipo usado para `T` no cumple la restricción, se generará un error de compilación, lo que refuerza la seguridad de tipos.

Creando una Clase Genérica: El Árbol Binario
A continuación, se introduce un ejemplo práctico para demostrar los genéricos: la implementación de un **árbol binario genérico**.
Teoría de árboles binarios: Un árbol binario es una estructura de datos recursiva (se auto-referencia) que puede estar vacía o contener:

- Un **nodo** (el dato).
- Dos **sub-árboles** (izquierdo y derecho), que también son árboles binarios.

Son muy eficientes para **ordenar y buscar datos**. Para insertar un elemento `I` en un árbol ordenado `B`:

- Si `B` está vacío, `I` se convierte en el nodo raíz.
- Si `I` es menor que el nodo actual `N`, se inserta en el sub-árbol izquierdo.
- Si `I` es mayor o igual que `N`, se inserta en el sub-árbol derecho. Este algoritmo es **recursivo**.

Una vez construido un árbol binario ordenado, puedes recorrerlo para mostrar sus contenidos en secuencia (ej., de forma ascendente). El algoritmo para mostrar un árbol ordenado también es recursivo:

1. Mostrar el sub-árbol izquierdo.
2. Mostrar el valor del nodo actual.

Mostrar el sub-árbol derecho.

Construyendo una clase de árbol binario genérica: El ejercicio propone crear una clase de árbol binario que pueda contener **casi cualquier tipo de dato**. La única restricción es que el tipo de dato **debe proporcionar un mecanismo para comparar valores** (como la interfaz `IComparable` que vimos antes).

Esta clase se implementará como una **biblioteca de clases (`.dll`)**, lo que permitirá reutilizarla en diferentes proyectos sin necesidad de copiar y recompilar el código fuente.

Creando un Método Genérico

Además de definir clases genéricas, C# permite crear **métodos genéricos**. Un método genérico te permite especificar los tipos de sus parámetros y su tipo de retorno utilizando un **parámetro de tipo**, similar a cómo se definen las clases genéricas.

Propósito de los Métodos Genéricos

Los métodos genéricos son una herramienta poderosa para:

- Definir algoritmos generalizados que pueden operar con **cualquier tipo de dato**.
- Garantizar la **seguridad de tipos** en tiempo de compilación.
- **Evitar la necesidad de `casting`** (y `boxing` en algunos casos con tipos de valor), lo que lleva a un código más limpio y eficiente.

Frecuentemente, los métodos genéricos se usan en conjunto con clases genéricas, especialmente cuando necesitan tomar o devolver tipos genéricos.

Sintaxis y Uso

Para definir un método genérico, se usa la misma sintaxis de parámetro de tipo (corchetes angulares `< >`) que se utiliza para las clases genéricas. También puedes especificar **restricciones** para el parámetro de tipo, igual que en las clases.

Ejemplo: Método `Swap` genérico Considera un método que intercambia los valores de dos variables. Esta funcionalidad es útil para cualquier tipo de dato.

```
static void Swap<T>(ref T first, ref T second) // '<T>' indica que es un método genérico
{
    T temp = first; // 'T' es el tipo que se usará en tiempo de ejecución
    first = second;
    second = temp;
}
```

Para invocar este método, simplemente especificas el tipo apropiado para su parámetro de tipo. El compilador inferirá el tipo si es obvio por el contexto.

```
int a = 1, b = 2;
Swap<int>(ref a, ref b); // Se llama a Swap para enteros
// O el compilador puede inferirlo: Swap(ref a, ref b);

string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2); // Se llama a Swap para cadenas
```

```
// O el compilador puede inferirlo: Swap(ref s1, ref s2);
```

Importante: Al igual que con las clases genéricas, cada uso distinto de un método genérico con diferentes parámetros de tipo hace que el compilador genere una **versión diferente del método**. `Swap<int>` no es el mismo método que `Swap<string>`. Ambos se generaron a partir de la misma plantilla genérica y, por lo tanto, exhiben el mismo comportamiento, pero operan sobre tipos distintos y específicos.

Definiendo un Método Genérico para Construir un Árbol Binario

En un ejercicio anterior, se creó una clase `Tree<T>` genérica para implementar un árbol binario. Esta clase probablemente tiene un método `Insert` para añadir elementos de datos uno por uno.

Sin embargo, insertar un gran número de elementos llamando repetidamente a `Insert` puede ser inconveniente. Por lo tanto, el siguiente ejercicio propone definir un **método genérico** llamado `InsertIntoTree` que permita insertar una **lista completa de elementos de datos** en un árbol binario con una sola llamada al método.

Este método genérico tomará una lista de elementos (probablemente un `IEnumerable<T>` o un `T[]`) y los insertará en un `Tree<T>` dado. Se probará insertando una lista de caracteres en un árbol de caracteres.

Uso de Colecciones

¿Qué son las clases de colección?

Los arrays (vistos en el Capítulo 10) son útiles para almacenar conjuntos de datos, pero tienen limitaciones: son de tamaño fijo (difícil de redimensionar), no ofrecen funcionalidad integrada para ordenar o buscar, y solo permiten el acceso a datos mediante índices enteros.

Aquí es donde las **clases de colección** resultan útiles. .NET proporciona varias clases que agrupan elementos de manera que una aplicación puede acceder a ellos de formas especializadas. Estas colecciones residen principalmente en el espacio de nombres `System.Collections.Generic`.

Como su nombre lo indica, estas colecciones son **tipos genéricos**. Requieren que proporciones un **parámetro de tipo** (`T`) que indica el tipo de datos que almacenarán. Cada clase de colección está optimizada para una forma particular de almacenamiento y acceso de datos, y cada una proporciona métodos especializados para esa funcionalidad.

Tipos de Colección Comúnmente Usados

Colección	Descripción
`List<T>`	Una lista de objetos a la que se puede acceder por índice (como un array), pero con métodos adicionales para buscar, ordenar, insertar y eliminar elementos dinámicamente.
`Queue<T>`	Una estructura de datos **primero en entrar, primero en salir (FIFO)**. Permite añadir elementos al final (`Enqueue`) y retirarlos del principio (`Dequeue`). También se puede examinar el primer elemento sin eliminarlo.

`PriorityQueue<TElement, TPriority>`	Extiende `Queue`. Asocia una prioridad a cada elemento encolado. Los elementos con mayor prioridad se desencolan antes. Los elementos con la misma prioridad se procesan en orden FIFO.
`Stack<T>`	Una estructura de datos **último en entrar, primero en salir (LIFO)**. Permite añadir elementos a la cima (`Push`) y retirarlos de la cima (`Pop`). También se puede examinar el elemento en la cima sin eliminarlo.
`LinkedList<T>`	Una lista doblemente enlazada, optimizada para inserciones y eliminaciones rápidas en cualquier punto (principio, final, antes/después de un nodo). Puede actuar como cola o pila, y también permite acceso secuencial.
`HashSet<T>`	Un conjunto desordenado de valores optimizado para una **recuperación rápida** de datos y operaciones de conjuntos (unión, intersección, diferencia, subconjunto/superconjunto). Los elementos no tienen un orden específico y no se permiten duplicados.
`Dictionary<TKey, TValue>`	Una colección de valores (`TValue`) que se identifican y recuperan utilizando **claves** (`TKey`) en lugar de índices enteros. Implementa un mapeo clave-valor eficiente.
`SortedList<TKey, TValue>`	Similar a `Dictionary`, pero los pares clave/valor se mantienen **siempre ordenados por la clave**. La inserción puede ser más lenta, pero la recuperación es a menudo más rápida y consume menos memoria que `SortedDictionary`. Las claves deben implementar `IComparable`.

Tipos de Colección de la Biblioteca de Clases .NET (Información Adicional)

- **`System.Collections` (Colecciones no genéricas):** Contienen colecciones diseñadas antes de que C# soportara tipos genéricos (pre-.NET 2.0). Almacenan referencias a `object`, lo que requiere `casts` y boxing/unboxing. **No se recomienda su uso en nuevas soluciones** y no están disponibles para apps UWP (Universal Windows Platform), salvo `BitArray`.
- **`BitArray`:** Excepción en `System.Collections`, implementa un array compacto de valores booleanos (cada bit representa `true` o `false`). Disponible para apps UWP.
- **`System.Collections.Concurrent`:** Contiene clases de colección **seguras para subprocesos (thread-safe)**, diseñadas para aplicaciones multiproceso. Se describen con más detalle en el Capítulo 24.

La Clase de Colección `List<T>`

`List<T>` es la más simple y versátil. Se usa mucho como un array dinámico.

Ventajas sobre los arrays:

- **Tamaño dinámico:** Puede crecer y encoger automáticamente. No necesitas especificar una capacidad inicial, aunque puedes hacerlo para optimizar el rendimiento.
- **Adición de elementos:** Usa el método `Add()` para añadir un elemento al final.
- **Inserción de elementos:** Usa `Insert(index, item)` para insertar en una posición específica, desplazando los demás elementos.
- **Eliminación de elementos:** Usa `Remove(item)` para eliminar la primera aparición de un valor, o `RemoveAt(index)` para eliminar en una posición específica. La lista se reordena automáticamente.
- **Ordenación:** El método `Sort()` permite ordenar fácilmente los elementos.
- **Acceso por índice:** Puedes acceder a los elementos existentes usando la notación de corchetes `[ ]` (como en los arrays), pero no para añadir nuevos elementos.

Consideraciones:

- **`Count` vs. `Length`:** Para `List<T>`, se usa la propiedad `Count` para obtener el número de elementos. Para arrays, se usa `Length`.
- **Iteración `foreach`:** No puedes modificar la colección (añadir, eliminar, insertar) dentro de un bucle `foreach` que la recorre; esto lanzará una `InvalidOperationException`.

Ejemplo de uso de `List<int>`:

```
using System.Collections.Generic;

List<int> numbers = new List<int>();

// Llenar la lista
foreach (int number in new int[]{10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1})
{
    numbers.Add(number);
}

// Insertar un elemento
numbers.Insert(numbers.Count - 1, 99); // Inserta 99 antes del último elemento

// Eliminar elementos
numbers.Remove(7);      // Elimina la primera ocurrencia de 7
numbers.RemoveAt(6);    // Elimina el elemento en el índice 6

// Iterar y mostrar (con for y foreach)
// (Output: 10, 9, 8, 7, 6, 5, 10, 99, 4, 3, 2)
```

La Clase de Colección `LinkedList<T>`

`LinkedList<T>` implementa una **lista doblemente enlazada**. Cada elemento (un nodo `LinkedListNode<T>`) contiene su valor y referencias al elemento siguiente (`Next`) y al anterior (`Previous`).

Características:

- **No soporta notación de array `[ ]`** para insertar o acceder a elementos.
- **Inserción y Eliminación:** Optimizada para operaciones en los extremos y en el medio:
    - `AddFirst(item)`: Añade al principio.
    - `AddLast(item)`: Añade al final.
    - `AddBefore(node, item)`: Inserta antes de un nodo específico.
    - `AddAfter(node, item)`: Inserta después de un nodo específico.

    - `Remove(item)`, `RemoveFirst()`, `RemoveLast()`.
- **Acceso:**
    - `First`: Propiedad que devuelve el primer nodo.
    - `Last`: Propiedad que devuelve el último nodo.
- **Iteración:**
    - Puedes recorrerla manualmente usando `node = node.Next` (o `node.Previous`) desde `First` (o `Last`) hasta encontrar `null`.
    - Puedes usar un bucle `foreach` para iterar hacia adelante automáticamente.

Ejemplo de uso de `LinkedList<int>`:

```
using System.Collections.Generic;

LinkedList<int> numbers = new LinkedList<int>();

// Llenar la lista (añade al principio, el orden final será inverso al de inserción)
foreach (int number in new int[] { 10, 8, 6, 4, 2 })
{
    numbers.AddFirst(number); // (2, 4, 6, 8, 10)
}

// Iterar hacia adelante (manual y foreach)
// (Output: 2, 4, 6, 8, 10)

// Iterar hacia atrás
for (LinkedListNode<int> node = numbers.Last; node is not null; node = node.Previous)
{
    Console.WriteLine(node.Value); // (Output: 10, 8, 6, 4, 2)
}
```

La Clase de Colección `Queue<T>`

Implementa un mecanismo **FIFO (First-In, First-Out)**.

- **`Enqueue(item)`:** Inserta un elemento al final de la cola.
- **`Dequeue()`:** Elimina y devuelve el elemento del principio de la cola.
- **`Peek()`:** Devuelve el elemento del principio sin eliminarlo.
- `Count`: Número de elementos en la cola.

Ejemplo de uso de `Queue<int>`:

```
using System.Collections.Generic;

Queue<int> numbers = new Queue<int>();

// Llenar la cola
foreach (int number in new int[]{9, 3, 7, 2})
{
    numbers.Enqueue(number); // Encola: 9, 3, 7, 2
}

// Iterar a través de la cola (foreach muestra en orden FIFO)
// (Output: 9, 3, 7, 2)

// Vaciar la cola
while (numbers.Count > 0)
{
    int number = numbers.Dequeue(); // Desencola en orden FIFO
    Console.WriteLine($"{number} ha salido de la cola");
```

```
    // (Output: 9, 3, 7, 2 en orden)
}
```

La Clase de Colección `PriorityQueue<TElement, TPriority>`

Extiende la idea de `Queue` al asociar una **prioridad** con cada elemento.

- **`Enqueue(element, priority)`:** Añade un elemento con su prioridad.
- **`TryDequeue(out TElement item, out TPriority priority)`:** Intenta desencolar el elemento con la prioridad más alta. Si hay elementos con la misma prioridad, se desencolan en el orden en que fueron encolados (FIFO).
- **Prioridad:** Generalmente un `int`, donde un valor de prioridad más bajo suele indicar una prioridad más alta (ej., prioridad 1 es mayor que prioridad 2).

Ejemplo de uso de `PriorityQueue<string, int>`:

```
using System.Collections.Generic;

PriorityQueue<string, int> messages = new PriorityQueue<string, int>();

// Añadir mensajes con diferentes prioridades
messages.Enqueue("Twas", 1);
messages.Enqueue("Brillig", 1);
messages.Enqueue("and", 2);
messages.Enqueue("the", 3);
messages.Enqueue("Slithy", 2);
messages.Enqueue("Toves", 3);

// Recuperar mensajes por prioridad y luego por orden de cola
while (messages.TryDequeue(out string item, out int priority))
{
    Console.WriteLine($"Elemento Desencolado: {item} Prioridad: {priority}");
    // (Output: "Twas", "Brillig" (prioridad 1), "Slithy", "and" (prioridad 2), "the", "Toves"
(prioridad 3))
}
```

La Clase de Colección `Stack<T>`

Implementa un mecanismo **LIFO (Last-In, First-Out)**.

- **`Push(item)`:** Inserta un elemento en la cima de la pila.
- **`Pop()`:** Elimina y devuelve el elemento de la cima de la pila.
- **`Peek()`:** Devuelve el elemento de la cima sin eliminarlo.
- `Count`: Número de elementos en la pila.

Ejemplo de uso de `Stack<int>`:

```
using System.Collections.Generic;

Stack<int> numbers = new Stack<int>();

// Llenar la pila
foreach (int number in new int[]{9, 3, 7, 2})
{
    numbers.Push(number); // Empila: 9, luego 3, luego 7, luego 2 (2 en la cima)
}

// Iterar a través de la pila (foreach muestra de la cima a la base)
// (Output: 2, 7, 3, 9)

// Vaciar la pila
```

```
while (numbers.Count > 0)
{
    int number = numbers.Pop(); // Desempila en orden LIFO
    Console.WriteLine($"{number} ha salido de la pila");
    // (Output: 2, 7, 3, 9 en orden)
}
```

La Clase de Colección `Dictionary<TKey, TValue>`

Implementa un **array asociativo** o **mapa**. Mapea **claves (`TKey`) a valores (`TValue`)**.

Características:

- **No puede contener claves duplicadas.**
  - `Add(key, value)`: Lanzará una excepción si la clave ya existe.
  - `dictionary[key] = value`: Si la clave no existe, la añade. Si ya existe, **sobrescribe** el valor existente. Es más seguro para añadir/actualizar.
- **Eficiencia:** Internamente es una tabla hash, lo que permite una **recuperación rápida** de valores por clave. Sin embargo, puede consumir bastante memoria para operar eficientemente con grandes volúmenes de datos.
- **`ContainsKey(key)`:** Permite verificar si una clave ya existe.
- **Iteración `foreach`:** Devuelve objetos `KeyValuePair<TKey, TValue>`, que son estructuras que contienen las propiedades `Key` y `Value`. Estas propiedades son de solo lectura.

Ejemplo de uso de `Dictionary<string, int>`:

```
using System.Collections.Generic;

Dictionary<string, int> ages = new Dictionary<string, int>();

// Llenar el diccionario
ages.Add("John", 57);
ages.Add("Diana", 57);
ages["James"] = 30;     // Usando notación de array
ages["Francesca"] = 27;

// Iterar y mostrar
foreach (KeyValuePair<string, int> element in ages)
{
    Console.WriteLine($"Nombre: {element.Key}, Edad: {element.Value}");
    // (Output: Orden no garantizado, pero mostrará las cuatro entradas)
}
```

Nota: `System.Collections.Generic` también incluye `SortedDictionary<TKey, TValue>`, que mantiene la colección ordenada por las claves.

La Clase de Colección `SortedList<TKey, TValue>`

Similar a `Dictionary`, también asocia claves con valores, pero con una diferencia crucial: los pares clave/valor se mantienen **siempre ordenados por la clave**.

Características:

- **Ordenación:** Los elementos se insertan en el array de claves en el índice correcto para mantenerlo ordenado. El array de valores se sincroniza.
- **Rendimiento:** La inserción de datos suele ser más lenta que en `SortedDictionary` (porque implica movimientos de elementos), pero la recuperación es a menudo igual o más rápida, y `SortedList` usa menos memoria.
- **No puede contener claves duplicadas.**

- **Iteración `foreach`:** Devuelve `KeyValuePair<TKey, TValue>`, y los elementos se devuelven **ordenados por la propiedad `Key`**.

Ejemplo de uso de `SortedList<string, int>`:

```
using System.Collections.Generic;

SortedList<string, int> ages = new SortedList<string, int>();

// Llenar la lista ordenada (el orden de inserción no importa para el orden final)
ages.Add("John", 57);
ages.Add("Diana", 57);
ages["James"] = 30;
ages["Francesca"] = 27;

// Iterar y mostrar (Output estará ordenado alfabéticamente por nombre)
foreach (KeyValuePair<string, int> element in ages)
{
    Console.WriteLine($"Nombre: {element.Key}, Edad: {element.Value}");
    // (Output: Diana, Francesca, James, John - ordenado por nombre)
}
```

La Clase de Colección `HashSet<T>`

`HashSet<T>` está optimizada para **operaciones de conjuntos** (matemáticos), como determinar la pertenencia a un conjunto y generar uniones e intersecciones.

Características:

- **No mantiene un orden específico** de los elementos.
- **No permite elementos duplicados.**
- **Operaciones:**
  - `Add(item)`: Añade un elemento.
  - `Remove(item)`: Elimina un elemento.
  - `IntersectWith(otherSet)`: Modifica el conjunto actual para contener solo los elementos comunes con `otherSet`. (Destructiva)
  - `UnionWith(otherSet)`: Modifica el conjunto actual para contener todos los elementos únicos de ambos conjuntos. (Destructiva)
  - `ExceptWith(otherSet)`: Modifica el conjunto actual para contener solo los elementos que no están en `otherSet`. (Destructiva)
  - `IsSubsetOf(otherSet)`, `IsSupersetOf(otherSet)`, `IsProperSubsetOf(otherSet)`, `IsProperSupersetOf(otherSet)`: Métodos no destructivos que devuelven `bool`.
- **Eficiencia:** Internamente se usa una tabla hash, lo que permite una **búsqueda de elementos muy rápida**. Un `HashSet` grande puede requerir una cantidad significativa de memoria.

Ejemplo de uso de `HashSet<string>`:

```
using System.Collections.Generic;

HashSet<string> employees = new HashSet<string>(new string[] {"Fred", "Bert", "Harry",
"John"});
HashSet<string> customers = new HashSet<string>(new string[] {"John", "Sid", "Harry",
"Diana"});

employees.Add("James");
customers.Add("Francesca");

Console.WriteLine("Empleados:");
```

```
foreach (string name in employees) Console.WriteLine(name);

Console.WriteLine("\nClientes:");
foreach (string name in customers) Console.WriteLine(name);

Console.WriteLine("\nClientes que también son empleados (Intersección):");
customers.IntersectWith(employees); // Modifica el HashSet 'customers'
foreach (string name in customers)
{
    Console.WriteLine(name);
    // (Output: Harry, John - el orden puede variar)
}
```

La Colección `SortedSet<T>` (Nota Adicional)

El espacio de nombres `System.Collections.Generic` también ofrece el tipo de colección **`SortedSet<T>`**. Esta clase funciona de manera similar a `HashSet<T>` (optimizada para operaciones de conjuntos y búsqueda rápida), con la principal diferencia de que los datos se mantienen siempre en **orden ordenado**. `SortedSet` y `HashSet` son interoperables, lo que significa que puedes, por ejemplo, obtener la unión de un `SortedSet` y un `HashSet`.

Inicializadores de Colección

Ya sabes cómo añadir elementos individualmente a una colección usando métodos como `Add()`, `Enqueue()`, `Push()`, etc. C# también permite **inicializar algunas colecciones directamente al declararlas**, utilizando una sintaxis similar a la de los arrays.

Inicialización de `List<T>` (y similares): Para colecciones que soportan el método `Add` (como `List<T>`, `HashSet<T>`, `SortedSet<T>`), puedes listar los elementos entre llaves `{}`:

```
List<int> numbers = new List<int>() { 10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1 };
```

Internamente, el compilador C# convierte esta sintaxis en una serie de llamadas al método `Add`. Por lo tanto, esta sintaxis **solo funciona para colecciones que tienen un método `Add`** público (por ejemplo, `Stack<T>` y `Queue<T>` no lo tienen y, por lo tanto, no pueden usar este tipo de inicializador).

Inicialización de colecciones clave/valor (`Dictionary<TKey, TValue>`, `SortedList<TKey, TValue>`, etc.): Para colecciones que manejan pares clave/valor, tienes dos opciones:

Notación de indexador (recomendada por el autor): Permite especificar el valor para cada clave directamente. Es muy legible.

```
Dictionary<string, int> ages = new Dictionary<string, int>()
{
    ["John"] = 53,
    ["Diana"] = 53,
    ["James"] = 26,
    ["Francesca"] = 23
};
```

- **Pares anónimos (opcional):** Especificas cada par clave/valor como un tipo anónimo entre llaves.

```
Dictionary<string, int> ages = new Dictionary<string, int>()
{
    {"John", 53},
    {"Diana", 53},
```

```
    {"James", 26},
    {"Francesca", 23}
};
```

- En esta forma, el primer ítem del par es la clave y el segundo es el valor. La notación de indexador es generalmente preferible por su claridad.

Métodos `Find`, Predicados y Expresiones Lambda

Las colecciones orientadas a diccionarios (`Dictionary`, `SortedDictionary`, `SortedList`) permiten encontrar rápidamente un valor especificando su clave. Sin embargo, otras colecciones que no usan claves (`List`, `LinkedList`) no soportan la notación de array para buscar y, en su lugar, ofrecen métodos como **`Find`** para localizar un elemento.

Predicados: El argumento al método `Find` es un **predicado**. Un predicado es esencialmente un método que **examina cada elemento** en la colección y devuelve un valor `Boolean` (`true` o `false`) indicando si el elemento coincide con el criterio de búsqueda. El método `Find` devuelve la primera coincidencia que encuentra.

Las clases `List<T>` y `LinkedList<T>` también soportan otros métodos de búsqueda:

- **`FindLast`**: Devuelve el último objeto que coincide.
- **`FindAll`** (solo en `List<T>`): Devuelve una `List<T>` con todos los objetos que coinciden.

Expresiones Lambda: La forma más sencilla y común de especificar un predicado es usando una **expresión lambda**. Una expresión lambda es una forma concisa de definir un método anónimo (un método sin nombre).

Las expresiones lambda contienen dos elementos clave:

- Una **lista de parámetros** (opcionalmente entre paréntesis).
- Un **cuerpo del método**.

El nombre del método y el tipo de retorno (si lo hay) se infieren del contexto. Para el método `Find`, el predicado procesa cada elemento de la colección, y el cuerpo del predicado debe examinar el elemento y devolver `true` o `false` según si coincide con el criterio de búsqueda.

Ejemplo de uso de `Find` con una expresión lambda: Considera una estructura `Person`:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

// Crear y poblar la lista de personas
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 53 },
    new Person() { ID = 2, Name = "Fred", Age = 28 },
    new Person() { ID = 3, Name = "Bill", Age = 34 },
    new Person() { ID = 4, Name = "Sven", Age = 22 },
};

// Encontrar la persona con ID = 3 usando Find y una expresión lambda
Person match = personnel.Find((Person p) => { return p.ID == 3; });

Console.WriteLine($"ID: {match.ID}\nName: {match.Name}\nAge: {match.Age}");
// Output:
// ID: 3
```

```
// Name: Bill
// Age: 34
```

Análisis de la expresión lambda `(Person p) => { return p.ID == 3; }`:

- **`(Person p)`**: La lista de parámetros. `p` es el nombre del parámetro (la `Person` actual de la colección que se está examinando). Si no hay parámetros, solo se usan paréntesis vacíos `()`.
- **`=>` operador**: Indica al compilador C# que esto es una expresión lambda.
- **`{ return p.ID == 3; }`**: El cuerpo del método. En este caso, es una expresión simple que devuelve un valor booleano. Puede contener múltiples sentencias.

Formas simplificadas de expresiones lambda: Las expresiones lambda pueden ser aún más concisas:

- Si el cuerpo es una **expresión única**, puedes omitir las llaves `{}` y la palabra clave `return`.
- Si la expresión toma un **único parámetro**, puedes omitir los paréntesis `()` que lo rodean.
- En muchos casos, puedes **omitir el tipo del parámetro** porque el compilador puede inferirlo del contexto.

Así, la expresión anterior se simplifica a:

```
Person match = personnel.Find(p => p.ID == 3);
```

Esto es mucho más legible y común en la práctica.

Colecciones sin soporte de búsqueda directa: Las clases de colección **`Stack<T>`**, **`Queue<T>`** y **`HashSet<T>`** **no soportan métodos de búsqueda** como `Find`. Sin embargo, puedes usar el método `Contains()` para verificar la membresía de un elemento en un `HashSet`.

Las Formas de las Expresiones Lambda

Las expresiones lambda son construcciones muy poderosas y versátiles. Aquí hay más ejemplos de sus diferentes formas:

- `x => x * x`
  - Una expresión simple que devuelve el cuadrado de su parámetro.
  - El tipo de `x` se infiere del contexto.
- `x => { return x * x; }`
  - Semánticamente igual que la anterior, pero usando un bloque de sentencias C# como cuerpo.
- `(int x) => x / 2`
  - Una expresión simple que devuelve el valor del parámetro dividido por 2.
  - El tipo del parámetro `x` se declara explícitamente.
- `() => folder.StopFolding()`
  - Llama a un método.
  - La expresión no toma parámetros (paréntesis vacíos).
  - Puede o no devolver un valor, dependiendo del método `StopFolding`.
- `(x, y) => { x++; return x / y; }`
  - Múltiples parámetros; el compilador infiere los tipos.
  - El parámetro `x` se pasa por valor, por lo que el efecto de la operación `++` es local a la expresión.
- `(ref int x, int y) => { x++; return x / y; }`
  - Múltiples parámetros con tipos explícitos.

- El parámetro `x` se pasa por referencia (`ref`), por lo que el efecto de la operación `++` es permanente fuera de la expresión. (¡Ten mucho cuidado con esto, no es recomendado generalmente!)

Resumen de características de las expresiones lambda:

- **Parámetros:** Se especifican entre paréntesis a la izquierda del operador `=>`. Sus tipos pueden inferirse o declararse explícitamente. Se pueden pasar por referencia (`ref`), pero no es la práctica habitual.
- **Valores de retorno:** Pueden devolver valores, pero el tipo de retorno debe coincidir con el delegado correspondiente al que se asignan.
- **Cuerpo:** Puede ser una expresión simple o un bloque de código C# con múltiples sentencias, llamadas a métodos, definiciones de variables, etc.
- **Alcance de variables locales:** Las variables definidas dentro del cuerpo de una lambda tienen un alcance local y desaparecen cuando la expresión termina.
- **Captura de variables externas:** Una expresión lambda puede acceder y **modificar todas las variables fuera de la expresión que están en el ámbito** cuando la lambda se define. ¡Usa esta característica con mucha precaución!

Expresiones Lambda y Métodos Anónimos

Las **expresiones lambda** se introdujeron en C# 3.0. En C# 2.0 se introdujeron los **métodos anónimos**, que cumplen una función similar pero son menos flexibles.

Métodos Anónimos: Permiten definir delegados sin crear un método con nombre, proporcionando la definición del cuerpo del método directamente:

```
this.stopMachinery += delegate { folder.StopFolding(); };
```

Para pasar un método anónimo como parámetro en lugar de un delegado:

```
control.Add(delegate { folder.StopFolding(); });
```

Siempre debes prefijar un método anónimo con la palabra clave **`delegate`**. Los parámetros se especifican entre paréntesis después de `delegate`.

```
control.Add(delegate(int param1, string param2)
{
    /* código que usa param1 y param2 */
});
```

Las **expresiones lambda** ofrecen una sintaxis más concisa y natural que los métodos anónimos y son la forma preferida de definir funciones anónimas en C# moderno.

Entendiendo los Delegados

Un **delegado** en C# es fundamentalmente una **referencia a un método**. Se les llama "delegados" porque "delegan" el procesamiento al método al que hacen referencia cuando son invocados. Este es un concepto simple pero con implicaciones muy poderosas en el diseño de software.

¿Cómo funcionan los delegados?

Normalmente, cuando invocas un método, especificas su nombre (y el objeto al que pertenece). Es claro qué método se está ejecutando.

```
Processor p = new Processor();
p.performCalculation(); // Invoca directamente el método
```

Con un delegado, puedes **almacenar una referencia a un método** en una variable de delegado, de manera similar a como asignas un valor a una variable.

```
Processor p = new Processor();
```

```
// (Declaración del delegado omitida por ahora para enfocarse en el concepto)
// delegate ... performCalculationDelegate ...;

// Asignación de la referencia del método al delegado
performCalculationDelegate = p.performCalculation;
```

Nota importante: La línea `performCalculationDelegate = p.performCalculation;` **no ejecuta el método**. Simplemente **almacena su dirección** o referencia. No hay paréntesis ni parámetros después del nombre del método.
Una vez que la referencia del método está almacenada en el delegado, puedes **invocar el método a través del delegado**:

```
performCalculationDelegate(); // Invoca el método a través del delegado
```

Esto parece una llamada a un método normal, pero el Common Language Runtime (CLR) sabe que es un delegado. Recupera el método al que el delegado hace referencia y lo ejecuta.

Poder de los delegados:

- **Flexibilidad:** Un delegado puede referirse a **diferentes métodos** en distintos momentos.
- **Multidifusión:** Un delegado puede referirse a **más de un método a la vez**. Cuando invocas el delegado, todos los métodos a los que hace referencia se ejecutarán en secuencia.

Comparación con C++: Si estás familiarizado con C++, un delegado es similar a un puntero a función, pero con una diferencia crucial: los delegados en C# son **completamente seguros en cuanto a tipos**. Solo puedes hacer que un delegado se refiera a un método que coincida exactamente con la **firma del delegado** (tipo de retorno y parámetros), y no puedes invocar un delegado que no haga referencia a un método válido.

Ejemplos de Delegados en la Biblioteca de Clases .NET

La biblioteca de clases de Microsoft .NET hace un uso extensivo de los delegados. Ejemplos de esto se encuentran en el Capítulo 18, "Uso de colecciones", con los métodos `Find` y `Exists` de la clase `List<T>`.

Cuando se diseñaron estos métodos, los creadores de la clase `List` no podían saber qué criterios de búsqueda específicos necesitarías. Por lo tanto, permitieron que el programador definiera esos criterios proporcionando su propio código en forma de un **predicado**. Un predicado es simplemente un delegado que devuelve un valor booleano.

Ejemplo con `Find` (recordatorio):

```
struct Person { /* ... propiedades ID, Name, Age ... */ }
List<Person> personnel = new List<Person>() { /* ... */ };

// El argumento es un predicado (un delegado)
Person match = personnel.Find(p => p.ID == 3); // 'p => p.ID == 3' es una expresión lambda que define el delegado
```

Otros métodos de `List<T>` que usan delegados son `Average`, `Max`, `Min`, `Count` y `Sum`. Estos métodos toman un delegado `Func` como parámetro. Un delegado `Func` se refiere a un método que devuelve un valor (una función).

Ejemplos con `Func`:

```
double averageAge = personnel.Average(p => p.Age); // Calcula el promedio de la propiedad Age
Console.WriteLine($"Edad promedio es {averageAge}");

int id = personnel.Max(p => p.ID); // Encuentra el ID máximo
Console.WriteLine($"Persona con ID más alto es {id}");
```

```
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39); // Cuenta personas en el rango de edad
Console.WriteLine($"Número de personal entre los treinta es {thirties}");
```

Tipos de Delegados `Func` y `Action`

El espacio de nombres `System` en .NET define una familia de tipos de delegados genéricos predefinidos, lo que simplifica la creación de delegados comunes:

- **`Func<TResult>` y `Func<T1, TResult>`, `Func<T1, T2, TResult>`, ... hasta `Func<T1, ..., T16, TResult>`:**
  - Se utilizan para referenciar métodos (funciones) que **devuelven un valor**.
  - Los parámetros de tipo (`T1`, `T2`, etc.) especifican los tipos de los parámetros de entrada del método.
  - `TResult` especifica el tipo de retorno del método.
  - **Ejemplo:** En `personnel.Count(p => p.Age >= 30 && p.Age <= 39)`, el método `Count` espera un delegado de tipo `Func<Person, bool>` (toma un `Person` y devuelve un `bool`). El compilador infiere esto automáticamente.
- **`Action` y `Action<T1>`, `Action<T1, T2>`, ... hasta `Action<T1, ..., T16>`:**
  - Se utilizan para referenciar métodos que **realizan una acción** pero **no devuelven ningún valor** (`void` methods).
  - Los parámetros de tipo (`T1`, `T2`, etc.) especifican los tipos de los parámetros de entrada del método.
  - **Ejemplo:** Un delegado `Action` sin parámetros se usaría para un método `void DoSomething()`. Un `Action<string>` para un método `void LogMessage(string message)`.

Cuando te encuentres creando tus propios tipos de delegados que coincidan con estos patrones, es una buena práctica usar los tipos `Func` o `Action` predefinidos en lugar de declarar un nuevo tipo de delegado personalizado.

Escenario de Fábrica Automatizada (Implementación de Delegados)

Considera un sistema de control para una fábrica automatizada con varias máquinas (plegadora, soldadora, pintora), cada una con su propio método para apagarse de forma segura (`StopFolding()`, `FinishWelding()`, `PaintOff()`).

Enfoque sin delegados (acoplamiento fuerte):

```
class Controller
{
   private FoldingMachine folder;
   private WeldingMachine welder;
   private PaintingMachine painter;

   public void ShutDown()
   {
      folder.StopFolding();
      welder.FinishWelding();
      painter.PaintOff();
   }
}
```

Este enfoque es **poco extensible y flexible**. Si se añade o quita una máquina, la clase `Controller` debe modificarse, lo que indica un **acoplamiento fuerte**.

Enfoque usando delegados (desacoplamiento): Observa que, aunque los nombres de los métodos son diferentes, todos tienen la misma "forma": no toman parámetros y no devuelven un valor (`void methodName()`). Aquí es donde un delegado es útil.

Declarar el tipo de delegado: Defines un tipo de delegado que coincida con la firma de los métodos que quieres referenciar.

```
delegate void stopMachineryDelegate(); // Un delegado que apunta a métodos void sin parámetros
```

- Se usa la palabra clave `delegate`.
- Se especifica el tipo de retorno (`void`), un nombre para el delegado (`stopMachineryDelegate`), y cualquier parámetro (ninguno aquí).

Crear una instancia del delegado y añadir métodos: Puedes crear una instancia del delegado y usar el operador de asignación compuesta `+=` para hacer que se refiera a los métodos que coinciden.

```
class Controller
{
   private stopMachineryDelegate stopMachinery; // Una instancia del delegado

   public Controller()
   {
      // Nota: Aquí se asumen que folder, welder, painter son inicializados antes.
      this.stopMachinery += folder.StopFolding;      // Añade referencia al método
      this.stopMachinery += welder.FinishWelding;    // Añade otra referencia
      this.stopMachinery += painter.PaintOff;        // Añade una tercera
   }
   // ...
}
```

- El operador `+=` en delegados tiene el significado de **añadir una referencia de método** a la lista de invocación del delegado.
- Simplemente se especifica el nombre del método, **sin paréntesis ni parámetros**.
- Es seguro usar `+=` en un delegado no inicializado; se inicializará automáticamente.

Invocar los métodos a través del delegado:

```
public void ShutDown()
{
   this.stopMachinery(); // Invoca el delegado, que a su vez llama a todos los métodos añadidos
}
```

- La misma sintaxis que una llamada a método normal.
- Si el delegado no está inicializado o no refiere a ningún método, invocarlo lanzará una `NullReferenceException`.

Ventaja clave: Al invocar `this.stopMachinery()`, se llaman automáticamente todos los métodos a los que el delegado hace referencia. La clase `Controller` ya no necesita saber cuántas máquinas hay ni cuáles son los nombres de sus métodos de apagado. Esto logra el **desacoplamiento**.

Eliminar métodos de un delegado: Se usa el operador `-=` para eliminar una referencia de método:

```
this.stopMachinery -= folder.StopFolding;
```

Manejo del delegado para desacoplamiento total: Para que la clase `Controller` sea totalmente independiente de las máquinas específicas, se necesita una forma de permitir que clases externas añadan métodos al delegado. Varias opciones:

Hacer la variable del delegado pública:

```
public stopMachineryDelegate stopMachinery;
```

1. **Crear una propiedad de lectura/escritura para el delegado:**

```
private stopMachineryDelegate stopMachinery;
public stopMachineryDelegate StopMachinery
{
   get => this.stopMachinery;
   set => this.stopMachinery = value;
}
```

2. **Encapsulación completa con métodos `Add` y `Remove`:** (Preferido por puristas OO)

```
Fragmento de código
public void Add(stopMachineryDelegate stopMethod) => this.stopMachinery += stopMethod;
public void Remove(stopMachineryDelegate stopMethod) => this.stopMachinery -=
stopMethod;
```

Independientemente de la técnica, el código que añade los métodos al delegado debe moverse fuera del constructor del `Controller`.

Ejemplo de uso desacoplado (con `Add`/`Remove`):

```
Controller control = new Controller();
FoldingMachine folder = new FoldingMachine();
WeldingMachine welder = new WeldingMachine();
PaintingMachine painter = new PaintingMachine();

control.Add(folder.StopFolding);       // Añadir métodos al delegado
control.Add(welder.FinishWelding);
control.Add(painter.PaintOff);

control.ShutDown(); // Invoca todos los métodos de apagado registrados
```

Expresiones Lambda y Delegados

Ya has visto cómo se añade un método a un delegado usando el nombre del método, por ejemplo: `this.stopMachinery += folder.StopFolding;`. Esto funciona bien si hay un método con una firma que coincide exactamente con la del delegado.

El problema de las firmas de métodos que no coinciden: ¿Qué pasa si el método `StopFolding` tiene una firma diferente, como `void StopFolding(int shutDownTime)`? Ahora, su firma no coincide con la de otros métodos como `FinishWelding` y `PaintOff` (que no toman parámetros), lo que significa que no puedes usar el mismo delegado (`stopMachineryDelegate`, que no espera parámetros) para manejar los tres.

Solución 1: El Patrón Adapter (Método Adaptador): Una forma de solucionar esto es crear un nuevo método "adaptador" que no tome parámetros y que, a su vez, llame al método original con la firma diferente, pasándole un valor por defecto.

```
void FinishFolding()
{
   folder.StopFolding(0); // Apagado inmediato
}
```

Luego, podrías añadir `FinishFolding` al delegado:

```
this.stopMachinery += folder.FinishFolding;
```

Cuando `stopMachinery` se invoca, llama a `FinishFolding`, que a su vez llama a `StopFolding(0)`. Este método `FinishFolding` es un clásico ejemplo del patrón **Adapter**, un método que "adapta" la firma de otro método. Aunque útil, estos pequeños métodos adaptadores pueden ser difíciles de manejar en clases grandes.

Solución 2: Expresiones Lambda (la forma preferida en C#): C# proporciona las **expresiones lambda** para estas situaciones. Las lambdas te permiten definir el "adaptador" de forma concisa y en línea, sin necesidad de crear un método nombrado separado.
En el escenario de la fábrica, podrías usar la siguiente expresión lambda:

```
this.stopMachinery += (() => folder.StopFolding(0));
```

Cuando el delegado `stopMachinery` es invocado, ejecuta el código definido por la expresión lambda, que a su vez llama al método `StopFolding` con el parámetro `0`. Esto mantiene tu código más limpio y directo al evitar métodos adaptadores triviales.

Habilitando Notificaciones Usando Eventos

Has aprendido a declarar un tipo de delegado, llamar a un delegado y crear instancias de delegados. Sin embargo, esto solo cubre la mitad de la historia: aún tienes que invocar explícitamente el delegado.
En muchos casos, sería útil que un delegado se ejecutara automáticamente cuando ocurre algo significativo. Por ejemplo, en la fábrica, sería crucial detener el equipo si se detecta un sobrecalentamiento de la máquina.
Las bibliotecas .NET proporcionan **eventos**, que se utilizan para definir y capturar acciones significativas, y para que un delegado sea llamado automáticamente en respuesta a estas situaciones. Muchas clases en .NET exponen eventos (por ejemplo, los controles de interfaz de usuario de las aplicaciones UWP usan eventos para responder a la interacción del usuario como clics de botón o entrada de texto). También puedes declarar tus propios eventos.

Declarando un Evento

Declaras un evento en una clase que actuará como **fuente del evento**. Una fuente de evento es típicamente una clase que monitorea su entorno y **lanza un evento** cuando sucede algo relevante.
En el escenario de la fábrica, una clase `TemperatureMonitor` podría ser una fuente de eventos. Si detecta que una máquina se ha sobrecalentado, podría lanzar un evento "MachineOverheating".
Un evento mantiene una lista de métodos a los que llamar cuando se lanza. A estos métodos se les conoce como **suscriptores**. Volviendo al ejemplo de la fábrica, estos métodos suscriptores deberían estar preparados para manejar el evento de sobrecalentamiento y tomar la acción correctiva necesaria (por ejemplo, apagar las máquinas).
Declarar un evento es similar a declarar un campo, pero con algunas diferencias clave:

- El tipo de un evento **debe ser un delegado**.
- Debes anteponer la declaración con la palabra clave **`event`**.

La sintaxis general para declarar un evento es:

```
event delegateTypeName eventName;
```

Ejemplo en `TemperatureMonitor`: Si reubicamos el delegado `StopMachineryDelegate` a la clase `TemperatureMonitor`:

```
class TemperatureMonitor
{
   public delegate void StopMachineryDelegate(); // Tipo de delegado
   public event StopMachineryDelegate MachineOverheating; // Declaración del evento
}
```

La lógica (no mostrada) dentro de `TemperatureMonitor` se encargaría de lanzar el evento `MachineOverheating` cuando sea necesario. Es importante notar que se **añaden**

métodos al evento (un proceso llamado **suscribirse al evento**) en lugar de añadirlos directamente al delegado subyacente.

Suscribirse a un Evento

Al igual que los delegados, los eventos vienen con un operador `+=` predefinido, y lo usas para suscribirte a un evento. En el ejemplo de la fábrica:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
}

TemperatureMonitor tempMonitor = new TemperatureMonitor();

// Suscribiendo métodos al evento MachineOverheating
tempMonitor.MachineOverheating += (() => { folder.StopFolding(0); }); // Usando una lambda
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

La sintaxis es idéntica a la de añadir un método a un delegado. Puedes incluso suscribirte usando una expresión lambda, lo cual es muy común. Cuando el evento `tempMonitor.MachineOverheating` se lanza, llamará a todos los métodos suscriptores, apagando las máquinas.

Cancelar la Suscripción a un Evento

Así como usas `+=` para adjuntar un delegado a un evento, usas el operador `-=` para desadjuntar un delegado de un evento. Esta acción elimina el método de la colección interna de delegados del evento y se conoce como **cancelar la suscripción al evento**.

Lanzar un Evento (Disparar un Evento)

Puedes lanzar un evento llamándolo como si fuera un método. Cuando un evento se lanza, todos los delegados adjuntos son llamados en secuencia.

Ejemplo en `TemperatureMonitor`:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;

    private void Notify() // Un método privado que lanza el evento
    {
        // Es crucial verificar si el evento no es null antes de invocarlo
        if (this.MachineOverheating is not null)
        {
            this.MachineOverheating(); // Lanza el evento
        }
    }
}
```

La comprobación `if (this.MachineOverheating is not null)` es necesaria porque un campo de evento es implícitamente `null` hasta que al menos un método se suscribe a él con `+=`. Si intentas lanzar un evento `null`, obtendrás una `NullReferenceException`. Si el delegado que define el evento espera parámetros, los argumentos apropiados deben proporcionarse al lanzar el evento.

Característica de seguridad importante: Un evento `public` (como `MachineOverheating`) **solo puede ser lanzado por métodos dentro de la clase que lo define** (en este caso, `TemperatureMonitor`). Cualquier intento de lanzar el evento fuera de esta clase resultará en un error de compilación.

¿Qué es LINQ? (Language-Integrated Query)

LINQ (Language-Integrated Query) es un conjunto de características introducidas en el Microsoft .NET Framework que **abstrae el mecanismo de consulta de datos del código de la aplicación**. Su objetivo principal es simplificar el procesamiento de datos y reducir el acoplamiento entre el código y la estructura de los datos que maneja.

Históricamente, las aplicaciones a menudo incluían lógica personalizada para el procesamiento de datos, lo que llevaba a un fuerte acoplamiento. Si la estructura de los datos cambiaba, se requerían modificaciones significativas en el código. LINQ aborda este problema.

Inspiración y Concepto

Los creadores de LINQ se inspiraron en cómo los sistemas de gestión de bases de datos relacionales (como Microsoft SQL Server) separan el lenguaje de consulta (SQL) del formato interno de los datos. En SQL, los desarrolladores escriben sentencias de alto nivel para describir los datos que desean recuperar, sin especificar cómo la base de datos debe recuperarlos físicamente. Esto permite que la estructura de almacenamiento de la base de datos cambie sin afectar las sentencias SQL de la aplicación.

LINQ aplica un principio similar, proporcionando una sintaxis y semántica que recuerdan a SQL, con ventajas parecidas: puedes **cambiar la estructura subyacente de los datos que se consultan sin necesidad de modificar el código de las consultas**.

Flexibilidad de LINQ

Aunque similar a SQL, LINQ es **mucho más flexible** y puede manejar una variedad más amplia de estructuras de datos lógicas. Por ejemplo, LINQ puede trabajar con datos organizados jerárquicamente, como los que se encuentran en un documento XML, además de los datos relacionales.

Uso de LINQ en una Aplicación C#

LINQ opera sobre cualquier estructura de datos que implemente la interfaz `IEnumerable<T>` o `IEnumerable`. Esto significa que puedes usar LINQ con arrays, `HashSet<T>`, `Queue<T>`, `List<T>`, o cualquier otra colección personalizada siempre que sea enumerable.

El texto proporciona ejemplos utilizando dos arrays de tipos anónimos para `customers` (clientes) y `addresses` (direcciones) para ilustrar las consultas.

```
var customers = new[] {
    new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie", CompanyName =
"Alpine Ski House" },
    // ... más clientes
};

var addresses = new[] {
    new { CompanyName = "Alpine Ski House", City = "Berne", Country = "Switzerland"},
    // ... más direcciones
};
```

LINQ ofrece dos formas principales de escribir consultas:

1. **Métodos de Extensión (Sintaxis de Método):** Utiliza los métodos de extensión de la clase `Enumerable` (en el espacio de nombres `System.Linq`), como `Select`,

`Where`, `OrderBy`, `GroupBy`, `Join`, etc. Estos métodos toman expresiones lambda como parámetros para definir la lógica de la consulta.

2. **Operadores de Consulta (Sintaxis de Consulta):** Proporciona una sintaxis más declarativa y similar a SQL, utilizando palabras clave como `from`, `select`, `where`, `orderby`, `group`, `join`. El compilador de C# traduce estas expresiones de consulta a las llamadas a métodos de extensión correspondientes en tiempo de compilación.

Funcionalidades Básicas de LINQ (a través de Métodos de Extensión):

1. Selección de Datos (`Select`)

Permite seleccionar o "proyectar" campos específicos de una colección. El método `Select` toma una expresión lambda que define cómo transformar cada elemento de la colección de origen en el resultado deseado.

```
IEnumerable<string> customerFirstNames = customers.Select(cust => cust.FirstName);
// Resultado: Una colección enumerable de solo los nombres de pila.

// Seleccionar múltiples campos (usando tipos anónimos o tipos definidos)
var customerFullNames = customers.Select(cust => new { cust.FirstName, cust.LastName });
// Resultado: Una colección enumerable de objetos anónimos con FirstName y LastName.
```

Puntos clave del `Select`:

- El parámetro de la lambda (`cust` en el ejemplo) actúa como un alias para cada elemento en la colección de origen.
- `Select` no recupera los datos inmediatamente; devuelve un objeto enumerable que **evalúa la consulta de forma diferida** cuando se itera sobre ella (`foreach`).
- `Select` es un **método de extensión** de la clase `Enumerable`, no del tipo `Array` o `List`.

2. Filtrado de Datos (`Where`): Permite restringir las filas (elementos) que la colección resultante contendrá, basándose en criterios específicos. El método `Where` toma una expresión lambda que devuelve un `bool` (un predicado).

```
IEnumerable<string> usCompanies = addresses
    .Where(addr => String.Equals(addr.Country, "United States")) // Filtra por país
    .Select(usComp => usComp.CompanyName);                       // Proyecta solo el nombre de la compañía
// Resultado: Nombres de compañías en Estados Unidos.
```

La secuencia de operaciones es importante: `Where` se aplica primero para filtrar, luego `Select` para proyectar.

3. Ordenamiento, Agrupamiento y Agregación

- **Ordenamiento (`OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`):** Permite recuperar datos en un orden específico. `OrderBy` ordena de forma ascendente; `OrderByDescending` de forma descendente. `ThenBy` y `ThenByDescending` se usan para un ordenamiento secundario.

  ```
  IEnumerable<string> companyNames = addresses
      .OrderBy(addr => addr.CompanyName) // Ordena por nombre de compañía
      .Select(comp => comp.CompanyName);
  ```

  -
- **Agrupamiento (`GroupBy`):** Permite agrupar datos según valores comunes en uno o más campos. El método `GroupBy` devuelve un conjunto enumerable de grupos, donde cada grupo es a su vez un conjunto enumerable de filas. Cada grupo tiene

una propiedad `Key` que representa el valor por el que se agrupó.

```
var companiesGroupedByCountry = addresses.GroupBy(addrs => addrs.Country);
// Para acceder a los resultados:
// foreach (var companiesPerCountry in companiesGroupedByCountry)
// { Console.WriteLine($"Country: {companiesPerCountry.Key}"); // Accede al valor de agrupamiento
//   foreach (var company in companiesPerCountry)
//   { Console.WriteLine($"\t{company.CompanyName}"); } }
```

- **Agregación (`Count`, `Max`, `Min`, `Sum`, `Average`):** Permite calcular valores de resumen. Estos métodos devuelven un único valor escalar (no una colección enumerable).

```
int numberOfCompanies = addresses.Select(addr => addr.CompanyName).Count();
// Cuenta el número de nombres de compañía proyectados.
// Para contar elementos distintos, se usa `Distinct()`:
int numberOfDistinctCountries = addresses.Select(addr =>
addr.Country).Distinct().Count();
```

`Count()` directamente sobre la colección (sin `Select`) cuenta el número total de filas: `addresses.Count()`.

4. Unir Datos (`Join`)

Permite combinar múltiples conjuntos de datos basándose en uno o más campos clave comunes (equi-joins).

```
var companiesAndCustomers = customers
    .Select(c => new { c.FirstName, c.LastName, c.CompanyName }) // Proyecta campos para la unión
    .Join(addresses,                                    // Colección a unir
        custs => custs.CompanyName,                     // Clave de la primera colección
        addrs => addrs.CompanyName,                     // Clave de la segunda colección
        (custs, addrs) => new { custs.FirstName, custs.LastName, addrs.Country }); // Proyección del resultado
// Resultado: Clientes con su país de ubicación.
```

Nota: LINQ solo soporta **equi-joins** (basados en igualdad).

Más Allá de lo Básico

LINQ ofrece muchas otras funcionalidades, como:

- **Operaciones de conjuntos:** `Intersect` (intersección), `Union` (unión).
- **Cuantificadores:** `Any` (verifica si al menos un elemento coincide con un predicado), `All` (verifica si todos los elementos coinciden).
- **Particionamiento:** `Take` (toma los primeros N elementos), `Skip` (salta los primeros N elementos).

En resumen, LINQ es una característica poderosa en C# que proporciona una forma unificada y expresiva de consultar datos de diversas fuentes, reduciendo el acoplamiento y mejorando la legibilidad del código.

LINQ y la Evaluación Diferida

Cuando defines una colección enumerable utilizando consultas LINQ (ya sea con los métodos de extensión o los operadores de consulta), es crucial entender que **la aplicación no construye la colección en el momento en que se define la consulta LINQ**. En su lugar, la colección se enumera (o se evalúa) **solo cuando la iteras** (por ejemplo, en un bucle `foreach`).

Esto implica que el estado de los datos en la colección original **puede cambiar entre el momento en que defines la consulta LINQ y el momento en que se recuperan los datos identificados por la consulta**. Sin embargo, la ventaja es que **siempre obtendrás los datos más actualizados** al momento de la iteración.

Ejemplo de Evaluación Diferida:
Considera la siguiente consulta LINQ que define una colección enumerable de empresas ubicadas en Estados Unidos:

```
var usCompanies = from a in addresses
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

En este punto, cuando se ejecuta esta línea de código, **los datos en el array `addresses` no se recuperan ni se evalúan las condiciones del filtro `Where`**. La consulta simplemente se ha *definido*.
La evaluación real de la consulta y la recuperación de los datos ocurren solo cuando iteras a través de la colección `usCompanies`:

```
foreach (string name in usCompanies)
{
   Console.WriteLine(name);
}
```

Si modificas los datos en el array `addresses` (por ejemplo, añadiendo una nueva empresa con sede en Estados Unidos) **entre la definición de `usCompanies` y el bucle `foreach`, verás estos nuevos datos** cuando se ejecute el `foreach`. Esta estrategia se conoce como **evaluación diferida** (o "ejecución diferida").

Forzar la Evaluación Inmediata (Colecciones en Caché)
A veces, es posible que desees que la consulta LINQ se evalúe en el momento de su definición y que genere una **colección estática y en caché**. Esta colección será una **copia de los datos originales** en ese instante y **no cambiará** si los datos en la colección original se modifican posteriormente.
LINQ proporciona métodos para esto:

- **`ToList()`**: Para construir un objeto `List<T>` estático que contiene una copia en caché de los datos.
- **`ToArray()`**: Para almacenar la colección en caché como un array.

Ejemplo de Evaluación Inmediata con `ToList()`:

```
var usCompanies = (from a in addresses.ToList() // Aplica ToList() a la fuente antes de la consulta
                   where String.Equals(a.Country, "United States")
                   select a.CompanyName);
```

O, de forma más común, se suele aplicar al final de la consulta:

```
var usCompaniesList = (from a in addresses
                       where String.Equals(a.Country, "United States")
                       select a.CompanyName).ToList(); // El ToList() al final fuerza la ejecución
```

En este caso, la lista de empresas se **fija** cuando creas la consulta. Si añades más empresas de Estados Unidos al array `addresses` **después** de esta línea, **no las verás** cuando iteres a través de la colección `usCompaniesList`, porque ya es una copia estática.

La elección entre evaluación diferida y evaluación inmediata depende de los requisitos específicos de tu aplicación y de cómo esperas que se manejen los cambios en los datos subyacentes.

El Modelo de Programación Asincrónica de Tareas (TAP) en C#

El **Modelo de Programación Asincrónica de Tareas (TAP)** en C# es una abstracción que facilita la escritura de código asincrónico. Con TAP, puedes escribir tu código como una secuencia de instrucciones normales, pero el compilador realiza transformaciones internas para que estas instrucciones se ejecuten de manera más compleja y eficiente, basándose en la disponibilidad de recursos externos y la finalización de tareas. El objetivo es que el código se lea de forma secuencial, pero se ejecute asincrónicamente.

Analogía del Desayuno: Sincrónico vs. Asincrónico

Para entender TAP, el texto usa la analogía de preparar el desayuno.

Instrucciones Sincrónicas (como una computadora "antigua" las interpretaría):

1. Vierte una taza de café.
2. Calienta una sartén y fríe dos huevos.
3. Fríe tres rebanadas de tocino.
4. Tuesta dos rebanadas de pan.
5. Unta mantequilla y mermelada en la tostada.
6. Vierte un vaso de jugo de naranja.

Si sigues estas instrucciones **sincrónicamente**, una tras otra, el desayuno tardaría mucho. Por ejemplo, no empezarías a tostar el pan hasta que los huevos y el tocino estén completamente listos. Esto es ineficiente y puede resultar en comida fría.

El código C# inicial ilustra este comportamiento **sincrónico**:

```
// Código Sincrónico de Ejemplo
static void Main(string[] args)
{
   Coffee cup = PourCoffee(); // Bloquea hasta que el café esté listo
   Console.WriteLine("coffee is ready");

   Egg eggs = FryEggs(2); // Bloquea hasta que los huevos estén fritos
   Console.WriteLine("eggs are ready");

   // ... y así sucesivamente para cada paso
}
```

En este modelo, cada método (como `FryEggs` o `ToastBread`) usa `Task.Delay(...).Wait();`. El método `Wait()` es un **bloqueo síncrono**, lo que significa que el hilo principal se detiene y no hace nada más hasta que esa tarea específica se complete. Esto es inaceptable para aplicaciones modernas, especialmente para interfaces de usuario que deben permanecer responsivas o servidores que manejan múltiples solicitudes.

Instrucciones Asincrónicas (como una persona eficiente o TAP las interpreta):

Una persona eficiente prepararía el desayuno **asincrónicamente**:

- Comienza a calentar la sartén para los huevos.
- Mientras se calienta, empieza a freír el tocino.
- Pone el pan en la tostadora.
- Mientras el tocino y el pan se cocinan, atiende los huevos.
- En cada paso, inicia una tarea y, luego, pasa a otras tareas que están listas para su atención.

Esto permite que varias cosas progresen simultáneamente, reduciendo el tiempo total de preparación. Es importante destacar que esto es **asincronía**, no necesariamente

paralelismo. Una sola persona (un solo hilo) puede manejar todas las tareas, pero no bloquea su atención en una sola tarea hasta que termine completamente.

La Importancia de `async` y `await`

La escritura de código asincrónico era compleja antes de `async` y `await`, a menudo involucrando *callbacks* o *eventos de finalización* que oscurecían la lógica. Las palabras clave `async` y `await` facilitan la lectura y la comprensión del código asincrónico, permitiéndote razonar sobre él como si fuera secuencial.

`await`: Es la clave para la programación asincrónica sin bloqueo. Cuando aplicas `await` a una `Task`, el compilador suspende la ejecución del método en ese punto y libera el hilo actual para que haga otro trabajo. Cuando la `Task` se completa, el resto del método se reanuda en el punto donde se detuvo.

`async`: Se usa para marcar un método que contiene operaciones asincrónicas y que, por lo tanto, puede usar la palabra clave `await`. Los métodos `async` suelen devolver `Task` o `Task<TResult>`.

Primera Mejora (No Bloquear, Esperar en su Lugar):

```
static async Task Main(string[] args) // Main ahora es async Task
{
   Coffee cup = PourCoffee();
   Console.WriteLine("coffee is ready");

   // Los métodos de preparación de alimentos ahora son asincrónicos
   Egg eggs = await FryEggsAsync(2); // await FryEggsAsync
   Console.WriteLine("eggs are ready");

   Bacon bacon = await FryBaconAsync(3); // await FryBaconAsync
   Console.WriteLine("bacon is ready");

   Toast toast = await ToastBreadAsync(2); // await ToastBreadAsync
   ApplyButter(toast);
   ApplyJam(toast);
   Console.WriteLine("toast is ready");

   Juice oj = PourOJ();
   Console.WriteLine("oj is ready");
   Console.WriteLine("Breakfast is ready!");
}
```

En esta versión, los métodos de preparación (`FryEggsAsync`, etc.) ahora devuelven `Task<TResult>` y usan `await Task.Delay()` internamente en lugar de `Wait()`. Aunque `Main` ahora usa `await`, **el desayuno todavía tarda el mismo tiempo en prepararse**. ¿Por qué? Porque el programa aún espera a que cada tarea termine completamente antes de iniciar la siguiente. A pesar de que el hilo no se bloquea, la ejecución sigue siendo secuencial en `Main`.

Iniciar Tareas Simultáneamente

Para aprovechar la asincronía y reducir el tiempo total, necesitas **iniciar varias tareas independientes inmediatamente** y solo `await` (esperar su resultado) cuando realmente lo necesites.

Segunda Mejora (Lanzar y Esperar Más Tarde):

```
static async Task Main(string[] args)
{
   Coffee cup = PourCoffee();
```

```
    Console.WriteLine("Coffee is ready");

    Task<Egg> eggsTask = FryEggsAsync(2);      // Inicia la tarea de los huevos
    Task<Bacon> baconTask = FryBaconAsync(3);  // Inicia la tarea del tocino
    Task<Toast> toastTask = MakeToastWithButterAndJamAsync(2); // Inicia la tarea de la
tostada

    // Solo esperamos cuando necesitamos los resultados
    Toast toast = await toastTask; // Esperamos la tostada porque la necesitamos para untar
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("Toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("Oj is ready");

    Egg eggs = await eggsTask; // Esperamos los huevos (ahora que la tostada está lista)
    Console.WriteLine("Eggs are ready");

    Bacon bacon = await baconTask; // Esperamos el tocino
    Console.WriteLine("Bacon is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

Con esta revisión, el desayuno se prepara más rápido (unos 20 minutos). Se inician `eggsTask`, `baconTask` y `toastTask` casi al mismo tiempo. El programa solo `await` sobre `toastTask` para aplicar la mantequilla y la mermelada, y luego `await` sobre `eggsTask` y `baconTask` más tarde, cuando sus resultados son necesarios para completar el desayuno. Este es el principio de **componer tareas**.

Composición de Tareas Asincrónicas

Un concepto importante es que **si alguna parte de una operación es asincrónica, toda la operación es asincrónica**. El ejemplo de la tostada lo demuestra: tostar el pan (`ToastBreadAsync`) es asincrónico, pero untar mantequilla y mermelada son operaciones sincrónicas. Sin embargo, la combinación de estas, `MakeToastWithButterAndJamAsync`, sigue siendo una operación asincrónica porque una de sus partes es asincrónica.

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number); // Espera la operación asincrónica
    ApplyButter(toast); // Operación sincrónica
    ApplyJam(toast);    // Operación sincrónica

    return toast;
}
```

Este método encapsula la secuencia de operaciones relacionadas con la tostada en una única tarea `Task<Toast>`. Esto mejora la modularidad y permite un mejor control sobre cuándo esperar la finalización de esta tarea compuesta.

Control de Excepciones Asincrónicas

Los métodos asincrónicos pueden lanzar excepciones, al igual que los sincrónicos. Cuando una tarea asincrónica falla, la excepción se almacena en la propiedad `Task.Exception`

del objeto `Task`. Esta propiedad es de tipo `System.AggregateException` porque una tarea puede fallar con múltiples excepciones.
Cuando aplicas `await` a una `Task` que ha fallado, el compilador automáticamente "desempaqueta" la `AggregateException` y **relanza la primera excepción interna** de la colección `AggregateException.InnerExceptions`. Esto hace que el manejo de errores en código asincrónico sea muy similar al sincrónico, donde puedes usar bloques `try-catch` estándar.
Ejemplo de Excepción (Tostadora en llamas):
Si `ToastBreadAsync` lanza una `InvalidOperationException`:

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    // ...
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire"); // Excepción lanzada
    // ...
}
```

Al ejecutar el código, verás que otras tareas continúan ejecutándose por un tiempo hasta que la línea donde se `await` sobre `toastTask` (en `Main`) es alcanzada. En ese momento, la `InvalidOperationException` se relanza y el programa termina con una excepción no controlada.

Aplicar Eficientemente `await` a las Tareas
Para un control más sofisticado y eficiente sobre múltiples tareas asincrónicas, la clase `Task` ofrece métodos útiles:
`Task.WhenAll(task1, task2, ...)`: Este método devuelve una única `Task` que se completa **cuando todas las tareas proporcionadas como argumentos han terminado**. Es ideal cuando necesitas que múltiples operaciones asincrónicas finalicen antes de continuar.

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

- Esto garantiza que todas las preparaciones del desayuno estén listas antes de imprimir los mensajes de "ready".

`Task.WhenAny(task1, task2, ...)`: Este método devuelve una `Task<Task>` que se completa **cuando cualquiera de las tareas proporcionadas como argumentos ha terminado**. Es útil cuando quieres procesar el resultado de la primera tarea que finalice. Debes `await` la tarea devuelta por `WhenAny` para saber cuál completó, y luego `await` esa tarea completada para recuperar su resultado o propagar cualquier excepción.

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks); // Espera a que CUALQUIER tarea termine
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("eggs are ready");
    }
    else if (finishedTask == baconTask)
```

```
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    await finishedTask; // Vuelve a esperar para propagar excepciones o obtener resultados
    breakfastTasks.Remove(finishedTask);
}
```

- Este enfoque es muy similar a cómo una persona real manejaría el desayuno: se ocupa de la primera cosa que está lista, y luego continúa con las demás. El `await finishedTask;` es crucial para asegurar que cualquier excepción de la tarea completada sea manejada.

Claro, aquí tienes un resumen del modelo de programación asincrónica de tareas (TAP) en C#, basado en el texto que proporcionaste.

Modelo de Programación Asincrónica de Tareas (TAP) en C#

La programación asincrónica es fundamental para mejorar la capacidad de respuesta y evitar cuellos de botella de rendimiento en las aplicaciones modernas. Tradicionalmente, escribir código asincrónico era complejo y difícil de depurar y mantener. C# simplifica esto con el **Modelo de Programación Asincrónica de Tareas (TAP)**, que aprovecha el soporte asincrónico del entorno de ejecución de .NET. El compilador realiza el trabajo pesado, permitiendo que el desarrollador escriba código con una estructura lógica similar al código sincrónico, pero obteniendo todos los beneficios de la asincronía.

Async Mejora la Capacidad de Respuesta

La asincronía es crucial para actividades que pueden bloquear el hilo de ejecución, como el acceso a la web, la manipulación de archivos o el procesamiento de imágenes. Si estas actividades se bloquean en un proceso sincrónico, toda la aplicación debe esperar, volviéndose no responsiva. En un proceso asincrónico, la aplicación puede continuar con otro trabajo que no depende de la tarea potencialmente bloqueadora hasta que esta finalice. Esto es especialmente valioso para las **aplicaciones con interfaz de usuario (UI)**, ya que toda la actividad de la UI suele compartir un único hilo. Si un proceso se bloquea en una aplicación sincrónica, la UI se congela, haciendo que la aplicación parezca "colgada". Con métodos asincrónicos, la aplicación sigue siendo responsiva, permitiendo al usuario redimensionar, minimizar o incluso cerrar la ventana mientras las tareas asincrónicas se ejecutan en segundo plano.

Áreas típicas donde la asincronía mejora la respuesta:

- **Acceso web:** `HttpClient`
- **Trabajo con archivos:** `JsonSerializer`, `StreamReader`, `StreamWriter`, `XmlReader`, `StorageFile`
- **Trabajo con imágenes:** `MediaCapture`, `BitmapEncoder`, `BitmapDecoder`
- **Programación WCF**

Los Métodos Asincrónicos Son Fáciles de Escribir

Las palabras clave **`async`** y **`await`** son el corazón de la programación asincrónica en C#. Permiten crear métodos asincrónicos casi tan fácilmente como los sincrónicos. Un método asincrónico se define utilizando la palabra clave `async`.

Ejemplo de Método Asincrónico:

```
public async Task<int> GetUrlContentLengthAsync()
{
    using var client = new HttpClient();
```

```
    // Inicia la descarga de contenido; no espera aquí
    Task<string> getStringTask = client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork(); // Realiza trabajo que no depende del resultado de getStringTask

    // Suspende GetUrlContentLengthAsync hasta que getStringTask se complete
    string contents = await getStringTask;

    return contents.Length; // Devuelve la longitud del contenido descargado
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Prácticas clave del ejemplo:

- **`async` en la firma:** El modificador `async` en la firma del método indica que este método puede contener expresiones `await`.
- **Convención de nomenclatura:** Los nombres de los métodos asincrónicos terminan con el sufijo "Async" (ej. `GetUrlContentLengthAsync`).
- **Tipo de valor devuelto:**
  - **`Task<TResult>`**: Si el método tiene una instrucción `return` con un operando de tipo `TResult`. (En el ejemplo, `Task<int>` porque devuelve `contents.Length`, que es un `int`).
  - **`Task`**: Si el método no tiene una instrucción `return` o tiene una instrucción `return` sin operando (similar a `void` en métodos sincrónicos).
  - **`void`**: Principalmente para controladores de eventos asincrónicos, donde el tipo de retorno `void` es requerido. Sin embargo, no se puede `await` un método `void` asincrónico, y las excepciones lanzadas por estos no pueden ser capturadas por el llamador.
  - Cualquier otro tipo que tenga un método `GetAwaiter` (ej. `ValueTask<TResult>`).
- **Operador `await`**: Marca un punto en el que el método asincrónico se suspende.
  - El método actual (`GetUrlContentLengthAsync`) no puede continuar hasta que la tarea esperada (`getStringTask`) se complete.
  - Mientras el método está suspendido, el control **regresa al autor de la llamada** de `GetUrlContentLengthAsync`. Esto significa que el hilo no se bloquea.
  - Cuando la tarea esperada se completa, el operador `await` recupera su resultado (el `string contents` en este caso) y el método `GetUrlContentLengthAsync` se reanuda desde ese punto.
  - Si no hay trabajo independiente entre iniciar una tarea y esperarla, se puede simplificar el código: `string contents = await client.GetStringAsync(...)`.

Los métodos asincrónicos simplifican el manejo de rutinas complejas como bucles y control de excepciones en código asincrónico, permitiendo escribirlos de manera similar a como se haría en una solución sincrónica.

¿Qué Ocurre en un Método Asincrónico? (Flujo de Control)

Entender cómo se mueve el control entre métodos es clave en la programación asincrónica:

1. Un **método de llamada** invoca y `await`s el método asincrónico (`GetUrlContentLengthAsync`).
2. `GetUrlContentLengthAsync` llama a otro método asincrónico (`GetStringAsync`) que devuelve una `Task<string>`.
3. `GetStringAsync` encuentra una operación de larga duración (ej. descarga web) y **suspende su progreso**, cediendo el control a su llamador (`GetUrlContentLengthAsync`). `GetStringAsync` devuelve una `Task<string>` (promesa de un resultado futuro).
4. `GetUrlContentLengthAsync` puede realizar **trabajo independiente** (`DoIndependentWork()`) que no depende del resultado de `GetStringAsync`.
5. `GetUrlContentLengthAsync` se queda sin trabajo independiente y necesita el resultado de `getStringTask`. Utiliza un operador `await` para **suspender su propio progreso**, cediendo el control al método que lo llamó originalmente. `GetUrlContentLengthAsync` devuelve una `Task<int>` (promesa de su propio resultado).
6. El método de llamada original puede hacer más trabajo o `await` el `Task<int>` devuelto por `GetUrlContentLengthAsync`.
7. Finalmente, `GetStringAsync` **completa su trabajo** y produce el resultado de la cadena, que se almacena en `getStringTask`.
8. El operador `await` en `GetUrlContentLengthAsync` recupera este resultado. `GetUrlContentLengthAsync` ahora puede completar su trabajo (calcular la longitud de la cadena) y marcar su propia `Task<int>` como completada.
9. El operador `await` en el método de llamada original recupera el resultado `int` de la tarea de `GetUrlContentLengthAsync` y el programa continúa.

Diferencia clave: Un método sincrónico devuelve cuando su trabajo se completa. Un método asincrónico devuelve una **tarea** cuando su trabajo se suspende, y marca esa tarea como completada cuando su trabajo finaliza.

Métodos Asincrónicos de API

Las API que soportan programación asincrónica se reconocen típicamente por el sufijo "Async" en sus nombres y porque devuelven `Task` o `Task<TResult>`. Ejemplos en .NET incluyen `CopyToAsync`, `ReadAsync`, `WriteAsync` en `System.IO.Stream`. Las APIs de Windows Runtime también usan tipos de retorno similares (ej. `IAsyncOperation<TResult>`, `IAsyncAction`).

Hilos

Es crucial entender que `async` y `await` **no crean hilos adicionales**. Un método asincrónico no se ejecuta en su propio hilo; se ejecuta en el contexto de sincronización actual y solo utiliza tiempo del hilo cuando está activo. La expresión `await` simplemente registra una "continuación" (el resto del método) y devuelve el control al llamador, sin bloquear el hilo.

Puedes usar `Task.Run` para mover trabajo intensivo de CPU a un hilo de fondo, pero esto no es necesario para operaciones enlazadas a E/S (lectura de red, disco), ya que estas esperan activamente por una respuesta, no realizan cálculos intensivos. Para operaciones enlazadas a E/S, `async`/`await` es generalmente superior a `BackgroundWorker` o enfoques multihilo manuales, ya que simplifica el código y evita condiciones de carrera.

`async` y `await` (Detalle)

- **`async`**:
    - Habilita el uso de `await` dentro del método.

    - Permite que el método sea `await`ed por otros métodos.
    - Si un método `async` no contiene ninguna expresión `await`, se ejecuta sincrónicamente y el compilador emite una advertencia.
- **`await`**:
    - Designa un punto de suspensión en el método asincrónico.
    - Indica al compilador que el método no puede continuar hasta que la operación asincrónica esperada se complete.
    - Mientras se espera, el control vuelve al llamador del método asincrónico.
    - La suspensión **no significa que el método haya finalizado**, y los bloques `finally` no se ejecutarán hasta que el método realmente termine su ejecución.

Ambas son palabras clave contextuales.

Tipos de Valor Devuelto y Parámetros

Los métodos asincrónicos suelen devolver `Task<TResult>` o `Task`:

- `Task<TResult>`: Para métodos que devuelven un valor de tipo `TResult` (similar a `return TResult;`).
- `Task`: Para métodos que no devuelven un valor (similar a `void`).

Un método asincrónico marcado con `async void` se usa principalmente para **controladores de eventos**. No se puede `await` un método `async void` y las excepciones que lance no pueden ser capturadas por el llamador.

Los métodos asincrónicos **no pueden declarar parámetros `in`, `ref` o `out`**, ni pueden devolver un valor por referencia, aunque pueden llamar a métodos que sí los usen.

Convención de Nomenclatura

Por convención, los métodos que devuelven tipos esperables (`Task`, `Task<T>`, `ValueTask`, etc.) deben terminar con el sufijo "Async" (ej. `DownloadDataAsync`). Los métodos que inician una operación asincrónica pero no devuelven un tipo esperable no deben usar "Async", pero pueden comenzar con "Begin" o "Start". Esta convención puede omitirse si un evento, clase base o interfaz sugieren un nombre diferente (ej. `OnButtonClick`).

Tipos de Valor Devueltos Asincrónicos en C#

Los métodos asincrónicos en C# están diseñados para realizar operaciones que pueden tardar mucho tiempo sin bloquear el hilo de ejecución. El tipo de valor devuelto de un método asincrónico es crucial, ya que indica si el método devuelve un valor, si es solo para notificar eventos, o si produce una secuencia de datos.

Los métodos asincrónicos pueden tener los siguientes tipos de valor devuelto:

1. **`Task`**
2. **`Task<TResult>`**
3. **`void`**
4. Cualquier tipo que tenga un método **`GetAwaiter`** accesible (incluido **`ValueTask<TResult>`**)
5. **`IAsyncEnumerable<T>`**

1. Tipo de Valor Devuelto: `Task`

Los métodos asincrónicos que realizan una operación pero **no devuelven ningún valor** (similar a los métodos sincrónicos `void`) deben tener un tipo de valor devuelto de **`Task`**.

- **Propósito:** Permite que un método de llamada use el operador `await` para suspender su propia ejecución hasta que el método asincrónico llamado (`Task`) finalice.

- **Sin `Result`:** Un `Task` no tiene una propiedad `Result` porque no devuelve ningún valor.
- **Uso del `await`:** Cuando el operando derecho de un `await` es un `Task`, la expresión `await` y su operando actúan como una declaración; no producen un valor.

Ejemplo:

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync(); // Espera a que WaitAndApologizeAsync termine

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000); // Espera 2 segundos de forma asincrónica

    Console.WriteLine("Sorry for the delay...\n");
}
```

En este ejemplo, `WaitAndApologizeAsync` no devuelve nada, por lo que su tipo de retorno es `Task`. `DisplayCurrentInfoAsync` lo `await`s, asegurando que el mensaje de disculpa se muestre antes de imprimir la información de la fecha y hora.

2. Tipo de Valor Devuelto: `Task<TResult>`

Los métodos asincrónicos que **devuelven un valor** de un tipo específico (`TResult`) deben tener un tipo de valor devuelto de **`Task<TResult>`**.

- **Propósito:** Encapsula el valor `TResult` que será el resultado de la operación asincrónica. Permite al llamador `await` la tarea y recuperar el valor `TResult` cuando la tarea se completa.
- **Propiedad `Result`:** Un `Task<TResult>` tiene una propiedad `Result` de tipo `TResult`.

Ejemplo:

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}"; // await recupera el int de la tarea

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek); // Simula una
operación asincrónica

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5; // Lógica para calcular horas de ocio
```

```
    return leisureHours; // Devuelve un int, por lo que el método es Task<int>
}
```

Cuando `GetLeisureHoursAsync` se `await`s en `ShowTodaysInfoAsync`, la expresión `await` recupera el valor entero (`leisureHours`) que `GetLeisureHoursAsync` devolvió. **Importante sobre `Result`:** Aunque `Task<TResult>` tiene una propiedad `Result`, **acceder directamente a ella (`myTask.Result`) es una práctica de bloqueo**. Esto significa que el hilo actual se detendrá hasta que la tarea termine y el valor esté disponible. En la mayoría de los casos, es preferible usar `await` para acceder al valor de forma no bloqueante.

3. Tipo de Valor Devuelto: `void`

El tipo de retorno **`void`** en métodos asincrónicos debe usarse **casi exclusivamente para los controladores de eventos asincrónicos**.

- **Propósito:** Cumplir con los requisitos de firma de los delegados de eventos, que a menudo esperan un método `void`.
- **Limitaciones:**
    - No se puede **`await`** un método asincrónico que devuelva `void`. Esto significa que el llamador no puede esperar a que el método finalice y continuará su ejecución independientemente de la operación asincrónica.
    - **Las excepciones lanzadas por un método asincrónico `void` no pueden ser capturadas por el llamador**. Estas excepciones probablemente provocarán que la aplicación falle, ya que no hay una `Task` a la que la excepción pueda ser asociada y propagada. Para cualquier método que pueda lanzar una excepción, se recomienda devolver `Task` o `Task<TResult>`.

Ejemplo (Controlador de eventos asincrónico):

```
public class NaiveButton
{
    public event EventHandler? Clicked;
    public void Click() { Clicked?.Invoke(this, EventArgs.Empty); }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task; // Para saber cuándo el handler 2
termina

        var button = new NaiveButton();
        button.Clicked += OnButtonClicked1;      // Sincrónico void
        button.Clicked += OnButtonClicked2Async; // Asincrónico void (ejemplo)
        button.Clicked += OnButtonClicked3;      // Sincrónico void

        Console.WriteLine("Before button.Click() is called...");
        button.Click(); // Dispara los eventos
        Console.WriteLine("After button.Click() is called...");
```

```
        await secondHandlerFinished; // Espera que OnButtonClicked2Async termine
    }

    // Un controlador de eventos asincrónico con void
    private static async void OnButtonClicked2Async(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait(); // Sincrónico (bloquea)
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500); // Asincrónico (no bloquea)
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true); // Señala que ha terminado
    }
}
```

En este ejemplo, `OnButtonClicked2Async` es un `async void` que se ejecuta de forma asincrónica. El resto de los controladores de eventos (`OnButtonClicked1`, `OnButtonClicked3`) se ejecutan y finalizan, y el código después de `button.Click()` puede continuar, mientras `OnButtonClicked2Async` sigue ejecutándose en segundo plano (simulado por `Task.Delay`).

4. Tipos de Retorno Asincrónicos Generalizados y `ValueTask<TResult>`

C# permite que un método asincrónico devuelva **cualquier tipo que tenga un método `GetAwaiter` accesible** (que a su vez devuelve una instancia de un "tipo awaiter" que implementa `ICriticalNotifyCompletion`). Esta característica, combinada con el atributo `AsyncMethodBuilderAttribute`, permite al compilador generar métodos `async` que devuelven tipos personalizados.

Esto es una característica avanzada orientada a la **optimización del rendimiento**, especialmente en escenarios donde `Task` y `Task<TResult>` (que son tipos de referencia) pueden causar un overhead de asignación de memoria significativo en bucles ajustados.

`System.Threading.Tasks.ValueTask<TResult>` es una implementación ligera de un valor de retorno de tarea generalizado que es una **estructura (tipo de valor)** en lugar de una clase (tipo de referencia). Esto puede reducir las asignaciones de memoria y mejorar el rendimiento en ciertos casos.

Ejemplo con `ValueTask<int>`:

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}"); // Espera un
ValueTask<int>

    static async ValueTask<int> GetDiceRollAsync() // Método devuelve ValueTask<int>
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync(); // Espera un ValueTask<int>
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }
```

```
    static async ValueTask<int> RollAsync() // Método devuelve ValueTask<int>
    {
        await Task.Delay(500); // Simula operación asincrónica

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
```

`ValueTask<TResult>` es útil cuando la mayoría de las veces el resultado de la tarea ya está disponible, o cuando el costo de asignar un objeto `Task` es significativo. Es un escenario avanzado; para la mayoría de los casos, `Task` y `Task<TResult>` son suficientes.

5. Secuencias Asincrónicas con `IAsyncEnumerable<T>`

Un método asincrónico también puede devolver una **secuencia asincrónica**, representada por **`IAsyncEnumerable<T>`**. Esto permite producir elementos de forma asincrónica, uno a uno, a medida que están disponibles.

- **Propósito:** Ideal para escenarios donde los elementos se generan en fragmentos a lo largo del tiempo, y cada fragmento requiere una operación asincrónica para obtener el siguiente conjunto de datos.
- **`yield return` con `await`:** Permite usar `yield return` dentro de un método `async` que también contiene `await`.
- **Consumo:** Se consume utilizando la instrucción **`await foreach`**.

Ejemplo:

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
          Here is the second line of text.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync(); // Primera lectura asincrónica
    while (line != null)
    {
        foreach (string word in line.Split(' ', StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word; // Emite cada palabra sincrónicamente
        }

        line = await readStream.ReadLineAsync(); // Siguiente lectura asincrónica
    }
}
// Cómo consumir:
// await foreach (var word in ReadWordsFromStreamAsync())
// {
//     Console.WriteLine(word);
// }
```

Este método lee líneas de texto asincrónicamente y luego divide cada línea en palabras, emitiéndolas una por una. El `await foreach` en el llamador esperará asincrónicamente cada vez que el productor necesite realizar una operación de E/S (`ReadLineAsync`).

Tipos de Retorno Específicos de Windows Runtime

Además de los tipos de .NET estándar, las aplicaciones para la Plataforma Universal de Windows (UWP) y otros escenarios de Windows tienen tipos de retorno asincrónicos específicos:

- **`DispatcherOperation`**: Para operaciones asincrónicas limitadas a Windows (históricamente en WPF).
- **`IAsyncAction`**: Para acciones asincrónicas en UWP que no devuelven un valor.
- **`IAsyncActionWithProgress<TProgress>`**: Para acciones asincrónicas en UWP que notifican progreso pero no devuelven un valor.
- **`IAsyncOperation<TResult>`**: Para operaciones asincrónicas en UWP que devuelven un valor.
- **`IAsyncOperationWithProgress<TResult,TProgress>`**: Para operaciones asincrónicas en UWP que notifican progreso y devuelven un valor.

La elección del tipo de valor devuelto asincrónico adecuado depende de si tu método necesita devolver un valor, si es un controlador de eventos, si requiere optimizaciones de rendimiento con tipos de valor, o si está generando una secuencia de elementos asincrónicamente. La mayoría de los escenarios se cubrirán con `Task` y `Task<TResult>`.

Interfaces, Tipos y Roles: Conceptos Clave en UML

El texto profundiza en la definición y representación de **interfaces** dentro del modelado de sistemas con UML, diferenciándolas de **tipos** y **roles**.

1. Definiciones Fundamentales

- **Interfaz:** Es una **colección de operaciones** que se utiliza para especificar un **servicio** que una clase o un componente promete proveer. Es un contrato que define *qué* se puede hacer, pero no *cómo* se hace.
- **Tipo:** Es un **estereotipo de una clase** utilizado para especificar un **dominio de objetos** y las operaciones aplicables a ellos. A diferencia de las interfaces, no implica una lista de servicios específicos.
- **Rol:** Representa el **comportamiento** de una entidad cuando participa en un **contexto particular**. Define cómo una entidad actúa o se espera que actúe en una situación específica.

2. Representación Gráfica y Nomenclatura de Interfaces

Gráficamente, una interfaz se representa como una **clase estereotipada** con el término `«interface»` encima de su nombre. También puede mostrar sus operaciones y otras propiedades.

Nombres de Interfaces:

- Cada interfaz debe tener un **nombre único** dentro del paquete que la contiene.
- Puede ser un **nombre simple** (ej., `IUnknown`, `ISensor`).
- Puede ser un **nombre calificado** si se incluye el nombre del paquete donde se encuentra (ej., `Red::IRouter`, `Sensores::IDestino`).
- Los nombres suelen ser **sustantivos cortos o expresiones nominales** que describen el servicio.

Ejemplos de Nombres (Figura 11.2):

«interface» IUnknown
«interface» ISensor
«interface» IOrtografía
«interface» Red::IRouter

«interface» Sensores::IDestino

3. Operaciones de una Interfaz

Una interfaz es fundamentalmente una colección de **operaciones nombradas**.

- **No especifican implementación:** A diferencia de las clases, las interfaces **no incluyen métodos** porque estos proporcionan la implementación. Las interfaces solo declaran las operaciones (qué hacer), no cómo hacerlas.
- **Propiedades:** Las operaciones pueden tener propiedades como visibilidad, concurrencia, estereotipos, valores etiquetados y restricciones.
- **Representación:** Al declararse, las operaciones se listan en un compartimento dentro de la clase estereotipada de la interfaz. Pueden mostrarse solo con su nombre o con su signatura completa y otras propiedades.

4. Relaciones de las Interfaces

Las interfaces, al igual que las clases, pueden participar en relaciones de:

- **Generalización:** Una interfaz puede heredar de otra interfaz.
- **Asociación:** Relaciones entre interfaces.
- **Dependencia:** Una interfaz puede depender de otra.

Además, y de manera crucial, las interfaces participan en la **relación de realización**.

Realización:

- Es una relación semántica donde un clasificador (como una clase o un componente) **especifica un contrato** que otro clasificador **garantiza cumplir**.
- Una clase o componente puede **realizar múltiples interfaces**, comprometiéndose a implementar todas las operaciones definidas en ellas. Este conjunto de servicios prometidos se conoce como **interfaz proporcionada**.
- De manera inversa, una clase o componente puede **depender de varias interfaces**, lo que significa que espera que otros componentes cumplan esos contratos. Este conjunto de servicios que una clase espera de otra se denomina **interfaz requerida**.
- Las interfaces actúan como una **línea de separación** en un sistema: el cliente y el proveedor pueden cambiar independientemente, siempre que ambos cumplan el contrato de la interfaz.

Representación Gráfica de Realizaciones (Figura 11.4): La realización se puede mostrar de dos formas:

1. **Forma Sencilla (Notación "Piruleta"):**
    - Una interfaz **proporcionada** se dibuja como un pequeño círculo conectado a un lado de la clase o componente. Es útil para mostrar las líneas de separación del sistema de forma concisa.
    - Una interfaz **requerida** se dibuja como una semiesfera (o "socket") conectada a un lado de la clase o componente.
2. **Forma Expandida:**
    - La interfaz se representa como una clase estereotipada completa (mostrando operaciones).
    - Se dibuja una **relación de realización** (línea discontinua con una flecha triangular vacía) desde la clase que implementa hacia la interfaz (para interfaces proporcionadas).
    - Se dibuja una **relación de dependencia** (línea discontinua con flecha normal) desde la clase que usa hacia la interfaz (para interfaces requeridas).

Nota importante: Las interfaces son similares a las **clases abstractas** en que ninguna puede ser instanciada directamente. Sin embargo, las clases abstractas pueden tener implementaciones concretas para algunas de sus operaciones, mientras que **todas las operaciones de una interfaz son implícitamente abstractas** (no tienen implementación).

5. Comprender y Enriquecer una Interfaz

Inicialmente, una interfaz es un conjunto de operaciones. Para comprender plenamente su **semántica** y cómo deben usarse sus operaciones, se puede añadir más información en UML:

- **Pre y Post-condiciones e Invariantes:** Asociar estas condiciones a cada operación y a la clase/componente que la realiza. Esto ayuda a un cliente a entender qué hace

la interfaz sin ver la implementación. Se puede usar **OCL (Object Constraint Language)** para especificarlas formalmente.
- **Máquina de Estados:** Asociar una máquina de estados a una interfaz para especificar el **orden parcial legal de las operaciones**. Esto define la secuencia o los estados válidos en los que se pueden llamar las operaciones.
- **Colaboraciones:** Asociar colaboraciones a la interfaz para especificar su **comportamiento esperado** a través de diagramas de interacción. Esto muestra cómo las operaciones trabajan juntas para lograr un comportamiento.

Técnicas Comunes de Modelado en UML: Interfaces y Tipos

Modelar un sistema eficazmente implica identificar sus componentes y las **líneas de separación** que permiten que diferentes partes evolucionen de forma independiente. Las **interfaces** son la herramienta principal en UML para definir estas demarcaciones, estableciendo contratos de servicio entre componentes.

1. Identificación y Modelado de las Líneas de Separación del Sistema

Cuando se construyen sistemas complejos, ya sea reutilizando componentes existentes o creando nuevos, es crucial definir cómo interactúan. Esto requiere comprender las **interfaces proporcionadas** (servicios que un componente ofrece) y las **interfaces requeridas** (servicios de los que un componente depende).

Las **líneas de demarcación claras** en la arquitectura, definidas por interfaces, permiten que los componentes a cada lado de la línea cambien de forma independiente, siempre y cuando cumplan con el contrato de la interfaz.

Pasos para modelar el vocabulario de un sistema (identificar interfaces):

1. **Agrupar clases y componentes:** Dibuja una línea alrededor de aquellos que están estrechamente acoplados o tienden a cambiar juntos. Estos pueden formar **colaboraciones**.
2. **Identificar operaciones y señales:** Observa qué operaciones y señales cruzan estos límites entre grupos de clases/componentes.
3. **Empaquetar como interfaces:** Agrupa los conjuntos lógicamente relacionados de estas operaciones y señales en **interfaces**.
4. **Identificar interfaces requeridas y proporcionadas:** Para cada colaboración (o componente), determina:
    - **Interfaces requeridas (importadas):** Aquellas de las que depende para su funcionamiento. Se modelan con **relaciones de dependencia**.
    - **Interfaces proporcionadas (exportadas):** Aquellas que ofrece a otros componentes. Se modelan con **relaciones de realización**.
5. **Documentar la dinámica de las interfaces:** Para cada interfaz, especifica:
    - **Pre y post-condiciones** para cada operación.
    - **Casos de uso** para ilustrar escenarios de uso.
    - **Máquinas de estados** para definir el orden legal de las operaciones.
    - (Opcional) Usar **OCL** para especificar formalmente la semántica.

Ejemplo (Figura 11.5): El componente `LibroMayor` es un buen ejemplo.

- **Proporciona (realiza)** las interfaces `IUnknown`, `ILibroMayor` e `IInformes`. Esto significa que `LibroMayor` implementa los servicios definidos por estas interfaces para que otros componentes puedan usarlos.
- **Requiere (usa)** las interfaces `IFlujo` e `ITransacción`. Esto indica que `LibroMayor` necesita servicios definidos por `IFlujo` e `ITransacción` para funcionar, y que otros componentes deben proveer implementaciones de estas interfaces.

Al definir `ITransacción` como una interfaz, se logra un **desacoplamiento**: cualquier componente que implemente `ITransacción` puede ser utilizado por `LibroMayor`, sin que `LibroMayor` tenga que saber los detalles de su implementación.

Nota: Cuando se reutilizan o compran componentes, a menudo solo se dispone de una documentación mínima. Es importante modelar las interfaces en UML para **documentar el conocimiento** sobre cómo deben usarse esos componentes, incluyendo el orden de las llamadas a las operaciones y los mecanismos subyacentes. La **introspección de componentes** (como en Eclipse o Java Beans) puede ser el primer paso para entender un componente poco documentado.

2. Modelado de Tipos Estáticos y Dinámicos

En programación orientada a objetos, el **tipo estático** de un objeto se establece en el momento de su creación. Sin embargo, un objeto puede desempeñar diferentes **roles** o adquirir diferentes "tipos" a lo largo de su vida, especialmente en sistemas de negocio con flujos de trabajo cambiantes.

- **Tipado Estático:** Se visualiza en un **diagrama de clases** estándar.
- **Tipado Dinámico:** Cuando un objeto puede ganar y perder tipos a lo largo de su ciclo de vida, es útil modelar explícitamente esta naturaleza dinámica.

Pasos para modelar un tipo dinámico:

1. **Especificar tipos posibles:** Representa cada tipo dinámico posible del objeto como una **clase** (si requiere estructura y comportamiento) o como una **interfaz** (si solo requiere comportamiento).
2. **Modelar roles:** Modela los roles que puede asumir la clase del objeto, que pueden estereotiparse como `«dynamic»` (estereotipo no predefinido en UML).
3. **Representar en diagramas de interacción:** En un **diagrama de interacción**, muestra el tipo dinámico de una instancia entre corchetes debajo del nombre del objeto, similar a un estado (ej., `miPersona:Persona [Candidato]`). Esto es un uso novedoso de la sintaxis de UML, pero consistente con la idea de estados.

Ejemplo (Figura 11.6): La clase `Persona` puede asumir diferentes roles/tipos dinámicos en un sistema de recursos humanos:

- `Persona «dynamic» Candidato`
- `Persona «dynamic» Empleado`
- `Persona «dynamic» Jubilado`

Sugerencias y Consejos para Modelar Interfaces

Cuando se dibuja una interfaz en UML, siempre debe representar una **línea de separación** en el sistema, diferenciando la especificación de la implementación.

Una **interfaz bien estructurada** es:

- **Sencilla y completa:** Provee todas las operaciones necesarias y suficientes para un único servicio.
- **Comprensible:** Contiene suficiente información para ser usada o implementada sin necesidad de examinar código existente.
- **Manejable:** Presenta la información clave de forma concisa, sin abrumar con detalles.

Elección de la notación gráfica:

- Utiliza la **notación de "piruleta" o "socket"** (forma sencilla) cuando solo necesites indicar la presencia de una línea de separación. Esto es común para componentes.
- Utiliza la **forma expandida** (clase estereotipada con operaciones listadas) cuando necesites visualizar los detalles del servicio en sí. Esto es más frecuente para especificar líneas de separación asociadas a paquetes o subsistemas.

Paquetes en UML: Organización y Vistas Arquitectónicas

Los **paquetes** son un mecanismo fundamental en UML para manejar la complejidad de los sistemas grandes. Nos permiten organizar elementos de modelado en grupos jerárquicos, facilitando la comprensión y el control del acceso a sus contenidos. Piensa en ellos como carpetas que contienen y estructuran tu modelo.

Introducción a los Paquetes

Los sistemas grandes, al igual que los edificios complejos, requieren una organización jerárquica. No podemos comprender un rascacielos simplemente como una suma de paredes y techos; necesitamos abstracciones mayores como "pisos", "zonas comerciales" o "áreas de servicio". De manera similar, los **paquetes** nos permiten agrupar abstracciones básicas (como clases) en conjuntos más grandes.

Aunque muchas agrupaciones son puramente conceptuales (no se manifiestan como objetos individuales en el sistema desplegado), son esenciales para organizar y comprender las **vistas arquitectónicas** del sistema.

- **Definición:** Un **paquete** es un mecanismo de propósito general para organizar elementos en grupos dentro de un modelo.
- **Representación Gráfica:** Se dibuja como una carpeta con una pestaña. El nombre del paquete va en la carpeta o en la pestaña (si se muestra el contenido).

Términos y Conceptos Clave de los Paquetes

Nombres

- Cada paquete debe tener un **nombre único** dentro del paquete que lo contiene.
- Puede ser un **nombre simple** (ej., `Reglas de negocio`).
- Puede ser un **nombre calificado**, que incluye el nombre del paquete contenedor, separados por `::` (ej., `Sensores::Visión`).

Elementos Contenidos

- Un paquete puede contener cualquier tipo de elemento de modelado: **clases, interfaces, componentes, nodos, colaboraciones, casos de uso, diagramas, y otros paquetes**.
- Existe una **relación de posesión:** el elemento se declara dentro del paquete. Si el paquete se destruye, el elemento también lo hace.
- Cada elemento pertenece **exclusivamente a un único paquete**.
- Los paquetes forman un **espacio de nombres**: los elementos de la misma categoría deben tener nombres únicos dentro de su paquete contenedor (ej., no dos clases `Cola` en el mismo paquete). Sin embargo, elementos de diferentes tipos pueden tener el mismo nombre (ej., una clase `Temporizador` y un componente `Temporizador`). Se recomienda evitar la duplicación de nombres para prevenir confusiones.
- Los paquetes pueden anidarse, permitiendo una **descomposición jerárquica** del modelo. Se recomienda no anidar demasiado (2 o 3 niveles es un límite manejable).

Visibilidad

Se puede controlar la visibilidad de los elementos contenidos en un paquete, similar a los atributos y operaciones de una clase.

- **Público (`+`):** Visible para los contenidos de cualquier paquete que importe al paquete contenedor. El conjunto de partes públicas de un paquete constituye su **interfaz**.
- **Protegido (`#`):** Visible solo para los paquetes que heredan del paquete contenedor.
- **Privado (`-`):** No visible fuera del paquete en el que se declara.
- **Paquete (`~`):** Visible para otras clases declaradas en el mismo paquete, pero invisible para clases en otros paquetes.

Importación y Exportación

Para manejar la complejidad en sistemas grandes, se utiliza la importación y exportación de paquetes para controlar el acceso entre ellos.

- **Importación (`«import»`):** Una relación de dependencia que significa que el paquete origen tiene acceso a los elementos públicos del paquete destino y los

añade a su propio espacio de nombres público. Esto permite referenciar los elementos importados por sus nombres simples (sin calificación).
- **Acceso (`«access»`):** Otro estereotipo de dependencia similar a `import`, pero que añade el contenido del paquete destino al **espacio de nombres privado** del origen. Esto significa que los elementos importados no pueden ser re-exportados por el paquete importador. En la práctica, **`«import»` es más común**.
- **Exportaciones:** Las partes públicas de un paquete son sus exportaciones. Solo son visibles para los paquetes que lo importan explícitamente.
- **Dependencias Transitivas:** Las dependencias de importación y acceso son transitivas. Si `Cliente` importa `Politicas` y `Politicas` importa `GUI`, entonces `Cliente` importa `GUI` de manera transitiva.

Técnicas Comunes de Modelado con Paquetes
Modelado de Grupos de Elementos
El uso más frecuente de los paquetes es organizar elementos de modelado en grupos coherentes.
- **Identificación:** Busca grupos de elementos conceptual o semánticamente cercanos dentro de una vista arquitectónica.
- **Empaquetamiento:** Envuelve cada grupo en un paquete.
- **Visibilidad:** Distingue qué elementos serán accesibles desde fuera del paquete (públicos `+`), y oculta los demás (protegidos `#` o privados `-`).
- **Conexión:** Conecta los paquetes que dependen de otros mediante **dependencias de importación**.
- **Generalización:** Para familias de paquetes, usa generalizaciones para conectar paquetes especializados con sus versiones más generales.

Distinción importante: Las **clases** son abstracciones de entidades en el problema o la solución; los **paquetes** son mecanismos de organización del modelo. Los paquetes no se manifiestan directamente en el sistema en ejecución.

Modelado de Vistas Arquitectónicas
Los paquetes también son ideales para modelar las diferentes **vistas arquitectónicas** de un sistema, que son proyecciones de su organización y estructura centradas en un aspecto particular.
- **Identificación de Vistas:** Define las vistas arquitectónicas significativas (ej., Vista de Diseño, Vista de Interacción, Vista de Implementación, Vista de Despliegue, Vista de Casos de Uso).
- **Contenido del Paquete:** Coloca todos los elementos (y diagramas) pertinentes para cada vista en su paquete correspondiente.
- **Anidamiento:** Agrupa aún más los elementos dentro de estos paquetes de vistas si es necesario.
- **Interconexión:** Generalmente, las vistas de nivel superior están "abiertas" a otras vistas al mismo nivel, lo que implica dependencias.

Componentes en UML: Modularidad, Interfaces y Estructura Interna
El desarrollo de software, al igual que el montaje de un sistema de cine en casa, puede abordarse de dos maneras: como una unidad monolítica rígida o como un sistema flexible construido a partir de **componentes** individuales. UML nos proporciona las herramientas para modelar este último enfoque, promoviendo sistemas más adaptables y mantenibles.

Introducción a los Componentes
Un sistema monolítico es como un sistema de cine en casa "todo en uno": fácil de instalar pero carente de flexibilidad. Si una parte falla o se necesita una actualización, a menudo hay que reemplazar todo el conjunto.

En contraste, un sistema basado en componentes es como construir un cine en casa pieza por pieza (pantalla, altavoces, sintonizador, reproductor de DVD), conectándolos con cables y puertos estandarizados. Esto permite:

- **Flexibilidad:** Elegir los componentes que mejor se adaptan a nuestras necesidades y presupuesto.
- **Sustitución:** Reemplazar o actualizar un solo componente sin afectar el resto del sistema.
- **Calidad:** Integrar componentes de mayor calidad en áreas específicas.

El software sigue la misma lógica. Construir aplicaciones como **componentes bien definidos** que pueden enlazarse de forma flexible y sustituirse cuando los requisitos cambian es la clave para la adaptabilidad y la reutilización.

Términos y Conceptos Clave

- **Interfaz:** Una colección de operaciones que especifica un servicio **proporcionado** o **solicitado** por una clase o componente. Actúa como el "pegamento" que une los componentes.
- **Componente:** Una **parte reemplazable** de un sistema que cumple un contrato (conforma) y proporciona la implementación de un conjunto de interfaces.
- **Puerto:** Una "ventana" específica en un **componente encapsulado** a través de la cual pasan los mensajes (aceptando entradas y enviando salidas) que cumplen con las interfaces especificadas. Los puertos tienen identidad y pueden tener multiplicidad.
- **Estructura Interna:** La implementación de un componente definida por un conjunto de **partes** conectadas de una manera específica. Permite construir componentes grandes a partir de componentes más pequeños.
- **Parte:** La especificación de un **rol** que forma parte de la implementación de un componente. Cada instancia de un componente tiene una instancia correspondiente de la parte. Las partes tienen nombre, tipo y multiplicidad.
- **Conector:** Una **relación de comunicación** entre dos partes o puertos dentro del contexto de un componente. Representa un enlace o un enlace transitorio entre instancias.

Componentes e Interfaces

La relación entre componentes e interfaces es fundamental para construir sistemas modulares.

- **Interfaces Proporcionadas:** Servicios que el componente **ofrece** a otros. Se representan como un **círculo** ("piruleta") unido al componente.
- **Interfaces Requeridas:** Servicios que el componente **necesita** de otros. Se representan como un **semicírculo** ("enchufe") unido al componente.
- Un componente puede tanto proporcionar como requerir interfaces.

Representación: Un componente se dibuja como un rectángulo con un icono de dos pestañas en la esquina superior derecha. Las interfaces se muestran con su notación abreviada (piruleta/enchufe) o expandida (clase estereotipada) con relaciones de realización (para interfaces proporcionadas) o dependencia (para interfaces requeridas).
El uso de interfaces **rompe la dependencia directa** entre componentes, permitiendo que un componente que usa una interfaz funcione con cualquier otro componente que la implemente, sin importar su localización o implementación interna.

Sustitución y Componentes

El objetivo principal de los sistemas basados en componentes es la **sustitución**.

- Los componentes son **reemplazables**: se puede sustituir uno por otro que cumpla con las mismas interfaces sin reconstruir todo el sistema. Las interfaces son la clave para lograrlo.

- Un componente es una **parte de un sistema**, que colabora con otros y existe en un contexto arquitectónico. Es una unidad cohesiva, reutilizable y fundamental para la construcción de sistemas.
- Esta definición es recursiva: un sistema a un nivel de abstracción puede ser un componente a un nivel superior.

Organización de Componentes
Los componentes se pueden organizar de varias maneras:
- **Agrupándolos en paquetes**, al igual que las clases.
- Especificando **relaciones** entre ellos (dependencia, generalización, asociación, realización).
- Construyéndolos a partir de **otros componentes** (estructura interna).

Puertos
Los **puertos** ofrecen un control más granular sobre la interacción de un componente, en comparación con las interfaces a nivel global.
- Son "ventanas" explícitas en un componente encapsulado. Todas las interacciones externas pasan a través de ellos.
- Un puerto tiene **identidad** y puede ser identificado de forma única por su nombre y el nombre del componente.
- Pueden tener **interfaces requeridas y proporcionadas** asociadas.
- También pueden tener **multiplicidad**, indicando múltiples instancias de un puerto que cumplen la misma interfaz pero pueden tener estados diferentes.

Representación (Figura 15.2): Un puerto se dibuja como un pequeño cuadrado insertado en el borde del componente. Las interfaces requeridas/proporcionadas se enlazan al puerto.
Ejemplo: Un componente `Vendedor de Entradas` podría tener puertos `ventasNormales` y `ventasPrioritarias` (ambos con interfaz `Venta de Entradas`), y puertos para `Cargar espectáculos` y `Cobros` (con interfaces requeridas).

Estructura Interna
La **estructura interna** de un componente describe cómo se implementa utilizando **partes** y **conectores**.
- **Partes:** Son las unidades de implementación que componen el componente. Tienen nombre, tipo y multiplicidad.
- **Diferencia con clases:** Una parte no es una clase. Representa un rol específico dentro del componente y puede haber múltiples partes del mismo tipo (con nombres distintos).
- **Conectores:** Son las relaciones de comunicación entre partes o puertos.
    - **Conexión directa:** Una línea entre partes o puertos. Implica un acoplamiento más fuerte.
    - **Conexión por interfaces (Junta circular):** Si dos componentes se conectan porque tienen interfaces compatibles, se usa una notación de junta circular. Implica un acoplamiento más débil y permite la sustitución fácil.
    - **Conector de delegación:** Conecta un puerto interno de una parte con un puerto externo del componente principal. Los mensajes al puerto externo se delegan al puerto interno.

Modelado de Clases Estructuradas
La estructura interna (partes y conectores) no es exclusiva de los componentes; también puede usarse para modelar **clases estructuradas**. Esto es útil para representar estructuras de datos complejas donde las partes tienen conexiones contextuales que solo aplican dentro de esa clase.
Pasos:
1. Identificar partes internas y sus tipos.

2. Dar nombres a las partes que indiquen su propósito.
3. Dibujar conectores entre partes que se comunican.
4. Se pueden usar otras clases estructuradas como tipos de partes, conectándose a través de sus puertos externos.

Modelado de una API

Las **API (Interfaces de Programación de Aplicaciones)** son las líneas de separación programáticas de un sistema, y se modelan eficazmente mediante **interfaces y componentes**.

- Una API es esencialmente una **interfaz** realizada por uno o más componentes.
- Los desarrolladores se preocupan por la interfaz en sí, mientras que la gestión de la configuración se preocupa por las realizaciones disponibles.
- Las APIs ricas semánticamente pueden tener muchas operaciones; a menudo, solo se exponen las propiedades relevantes en el diagrama.

Pasos para modelar una API:

1. Identificar las líneas de separación del sistema y modelarlas como **interfaces**, con sus atributos y operaciones de frontera.
2. Exponer solo las propiedades importantes de la interfaz; ocultar las demás pero mantenerlas referenciadas en la especificación.
3. Modelar la **realización** de la API solo si es importante para mostrar la configuración de una implementación específica.

Sugerencias y Consejos para Componentes

Un **buen componente** debe:

- Encapsular un servicio con una interfaz y frontera bien definidas.
- Tener suficiente estructura interna para justificar su descripción.
- No combinar funcionalidades no relacionadas.
- Organizar su comportamiento externo con pocas interfaces y puertos.
- Interactuar solo a través de los puertos declarados.

Al mostrar la implementación de un componente con subcomponentes:

- Utiliza un número pequeño de subcomponentes; si son muchos, usa más niveles de descomposición.
- Asegúrate de que los subcomponentes interactúan solo a través de puertos y conectores.
- Modela los subcomponentes que interactúan directamente con el exterior usando **conectores de delegación**.

Al dibujar un componente en UML:

- Dále un nombre claro a él y a sus interfaces.
- Nombra los subcomponentes y puertos si su significado no es obvio o si hay múltiples partes del mismo tipo.
- **Oculta los detalles innecesarios** de implementación en el diagrama de componentes.
- Usa **diagramas de interacción** para mostrar la dinámica y secuenciación de los mensajes dentro del componente.

Artefactos en UML: Modelado de la Dimensión Física del Sistema

Así como en la construcción de un edificio se requiere tanto un modelado lógico (planos, estructuras) como un modelado físico (ladrillos, hormigón), en el desarrollo de software necesitamos ambos. Los **artefactos** en UML son la herramienta para modelar la **dimensión física** de un sistema, representando los elementos que existen en el mundo real de los bits y que finalmente se despliegan en nodos.

Introducción a los Artefactos

El modelado lógico se enfoca en el vocabulario del dominio y cómo colaboran los elementos conceptuales. El **modelado físico**, por otro lado, se dedica a construir el sistema ejecutable, donde los elementos lógicos se convierten en entidades tangibles.

- **Definición:** Un **artefacto** es un elemento **físico y reemplazable** de un sistema que existe a nivel de la plataforma de implementación.
- **Ejemplos:** Bibliotecas de código objeto, ejecutables, componentes .NET, Enterprise Java Beans, tablas de bases de datos, archivos de datos, documentos, código fuente, etc.
- **Representación Gráfica:** Se dibuja como un rectángulo con la palabra clave `«artifact»`. Puede personalizarse con **estereotipos** para representar tipos específicos de artefactos (Figura 26.1).

Términos y Conceptos Clave de los Artefactos

Nombres

- Cada artefacto debe tener un **nombre único** dentro del nodo que lo contiene.
- Puede ser un **nombre simple** (ej., `agent.java`).
- Puede ser un **nombre calificado** (ej., `system::dialog.dll`), incluyendo el nombre del paquete en el que se encuentra.
- Los nombres suelen ser cortos, extraídos del vocabulario de implementación, y pueden incluir extensiones de archivo (ej., `.java`, `.dll`).

Artefactos y Clases

Aunque ambos son clasificadores, hay diferencias cruciales:

- **Clases:** Representan abstracciones **lógicas y conceptuales**. No residen directamente en nodos. Pueden tener atributos y operaciones.
- **Artefactos:** Representan elementos **físicos** ("bits"). Residen directamente en nodos. Implementan clases y métodos, pero no tienen atributos u operaciones propias.
- **Relación:** Un artefacto es la **implementación física** de un conjunto de elementos lógicos (clases, colaboraciones). Esta relación se modela con una **relación de manifestación** (`«manifest»`), mostrando qué clases son implementadas por un artefacto (Figura 26.3).

Tipos de Artefactos

UML distingue tres tipos principales de artefactos:

1. **Artefactos de Despliegue:** Son los necesarios y suficientes para formar un sistema ejecutable (ej., DLLs, EXEs). Cubren modelos de objetos y otras implementaciones (páginas web dinámicas, tablas de bases de datos).
2. **Artefactos Producto del Trabajo:** Son productos del proceso de desarrollo que no participan directamente en el sistema ejecutable, pero se usan para crearlo (ej., archivos de código fuente, archivos de datos).
3. **Artefactos de Ejecución:** Se crean como consecuencia de un sistema en ejecución (ej., una instancia de un objeto .NET a partir de una DLL).

Elementos Estándar

Los mecanismos de extensibilidad de UML (valores etiquetados, estereotipos) se aplican a los artefactos. UML define varios **estereotipos estándar** para artefactos:

- `«executable»`: Artefacto que puede ejecutarse en un nodo.
- `«library»`: Biblioteca de objetos (estática o dinámica).
- `«file»`: Documento que contiene código fuente o datos.
- `«document»`: Documento.

Técnicas Comunes de Modelado con Artefactos

Modelado de Ejecutables y Bibliotecas

Es crucial para sistemas compuestos por múltiples ejecutables y bibliotecas, y para la gestión de versiones y configuraciones.

Pasos:

1. **Identificar la partición física:** Considerar el impacto de aspectos técnicos, de gestión de configuraciones y de reutilización.
2. **Modelar como artefactos:** Representar ejecutables y bibliotecas usando los estereotipos estándar o nuevos estereotipos apropiados.
3. **Modelar interfaces:** Si es importante para las líneas de separación del sistema, mostrar las interfaces que algunos artefactos utilizan y otros realizan.
4. **Modelar relaciones:** Si es necesario, representar las relaciones entre artefactos y sus interfaces, comúnmente usando **dependencias** para visualizar el impacto del cambio (Figura 26.4).

- Los artefactos pueden agruparse en **paquetes** para organizar grandes modelos.
- Para sistemas distribuidos, se modela cómo se distribuyen los artefactos en **nodos**.

Modelado de Tablas, Archivos y Documentos

Además de ejecutables, los sistemas incluyen artefactos auxiliares críticos para el despliegue físico (archivos de datos, scripts, logs, documentos de ayuda).

Pasos:

1. **Identificar artefactos auxiliares:** Reconocer todos los elementos no ejecutables/bibliotecas que son parte de la implementación física.
2. **Modelar como artefactos:** Usar artefactos con estereotipos estándar o nuevos para estos elementos.
3. **Modelar relaciones:** Representar las dependencias entre estos artefactos auxiliares y los demás artefactos (Figura 26.5).

Modelado de Código Fuente

Los artefactos también se usan para modelar la **configuración de archivos de código fuente**, que son "productos del trabajo" del proceso de desarrollo.

Utilidad:

- Visualizar **dependencias de compilación** entre archivos de código fuente.
- Gestionar la división y combinación de archivos en entornos de desarrollo con control de versiones.
- Actuar como interfaz gráfica para herramientas de gestión de configuraciones.

Pasos:

1. **Modelar archivos de código fuente:** Representarlos como artefactos, incluyendo sus dependencias de compilación según las herramientas de desarrollo.
2. **Incluir valores etiquetados:** Añadir información de versión, autor, etc., si se integra con herramientas de gestión de configuraciones.
3. **Delegar a herramientas:** Permitir que las herramientas de desarrollo gestionen las relaciones entre archivos y usar UML para visualizar y documentar (Figura 26.6).

- Los archivos de código fuente también pueden agruparse en **paquetes**.
- Las relaciones entre clases, archivos de código fuente y ejecutables/bibliotecas pueden visualizarse con **relaciones de traza**, aunque no siempre es necesario tanto detalle.

Sugerencias y Consejos para Modelar Artefactos

Al modelar artefactos, siempre se está operando en la **dimensión física** del sistema. Un **artefacto bien estructurado** debería:

- Implementar directamente un conjunto de clases que colaboran eficazmente.
- Estar **débilmente acoplado** en relación con otros artefactos.

En resumen, los artefactos son esenciales para comprender, visualizar y gestionar los aspectos tangibles y desplegables de un sistema de software, cerrando la brecha entre el diseño lógico y la implementación física.

¡Excelente! Has proporcionado un texto claro y conciso sobre los **Nodos en UML** y su papel en el **modelado del despliegue**.
Aquí tienes un resumen estructurado de los puntos clave de este capítulo:

Nodos en UML: Modelado de la Topología de Hardware para el Despliegue
Así como los edificios necesitan cimientos y estructuras físicas, los sistemas de software requieren hardware sobre el cual ejecutarse. En UML, los **nodos** son los bloques de construcción para modelar la **topología del hardware** en la que se despliegan y ejecutan los **artefactos** de software.

Introducción a los Nodos
Un sistema con gran cantidad de software tiene dos dimensiones:
- **Lógica:** Clases, interfaces, colaboraciones, interacciones, máquinas de estados.
- **Física: Artefactos** (empaquetamientos físicos de elementos lógicos) y **nodos** (hardware donde se despliegan y ejecutan los artefactos).

Los **nodos** pertenecen al mundo material, son elementos físicos en tiempo de ejecución que representan un **recurso computacional**, generalmente con memoria y capacidad de procesamiento. Sirven para visualizar, especificar, construir y documentar las decisiones sobre la infraestructura de hardware.
- **Representación Gráfica:** Un nodo se representa como un **cubo** (Figura 27.1).
- **Extensibilidad:** Se pueden usar **estereotipos** para particularizar la notación y representar tipos específicos de procesadores o dispositivos (ej., `«processor»`, `«device»`).

Términos y Conceptos Clave de los Nodos
Nombres
- Cada nodo debe tener un **nombre único** dentro del paquete que lo contiene.
- Puede ser un **nombre simple** (ej., `oficina_7`).
- Puede ser un **nombre calificado**, incluyendo el nombre del paquete contenedor (ej., `servidor::backup`).
- Los nombres suelen ser cortos y descriptivos del vocabulario de la implementación.

Nodos y Artefactos
Nodos y artefactos son similares en muchos aspectos (nombres, relaciones, anidamiento, instancias, interacciones), pero se diferencian fundamentalmente en su rol:
- Los **artefactos** son los elementos que **participan en la ejecución** de un sistema (las "cosas" que se ejecutan). Representan el **empaquetamiento físico** de los elementos lógicos.
- Los **nodos** son los elementos **donde se ejecutan los artefactos**. Representan el **despliegue físico** de los artefactos.

Una clase se **manifiesta** como uno o más artefactos, y un artefacto puede **desplegarse** sobre uno o más nodos. La relación entre un nodo y los artefactos que despliega se puede mostrar:
- Mediante **anidamiento** (el artefacto dentro del nodo, Figura 27.3).
- Como parte de la especificación del nodo (ej., en una tabla).
- Un **unidad de distribución** es un conjunto de objetos o artefactos asignados a un nodo como grupo.

Nota: Los nodos pueden tener **atributos y operaciones** (ej., `velocidadDelProcesador`, `encendido()`), similar a las clases, para modelar sus propiedades y comportamientos operacionales.

Organización de Nodos
Los nodos se pueden organizar de varias maneras:
- **Agrupándolos en paquetes**, al igual que clases y artefactos.
- Especificando **relaciones** entre ellos (dependencia, generalización, asociación).

Conexiones

El tipo de relación más común entre nodos es la **asociación**, que representa una **conexión física** (ej., Ethernet, línea serie, bus compartido, enlace satelital).

- Se pueden usar **roles, multiplicidad y restricciones**.
- Las asociaciones entre nodos deben **estereotiparse** para indicar el tipo específico de conexión (ej., `«10-T Ethernet»`, `«RS-232»`) (Figura 27.4).

Técnicas Comunes de Modelado con Nodos

Modelado de Procesadores y Dispositivos

Los nodos se utilizan principalmente para modelar la topología de hardware en sistemas monolíticos, embebidos, cliente/servidor o distribuidos.

- **Procesador:** Un nodo con capacidad de procesamiento que puede ejecutar artefactos (estereotipado como `«processor»`).
- **Dispositivo:** Un nodo sin capacidad de procesamiento a este nivel de abstracción, que interactúa con el mundo real (estereotipado como `«device»`).

Pasos:

1. **Identificar elementos computacionales:** Modelar cada uno como un nodo.
2. **Estereotipar:** Usar estereotipos estándar (`«processor»`, `«device»`) o crear nuevos estereotipos con iconos significativos para el vocabulario del dominio (Figura 27.5).
3. **Considerar atributos y operaciones:** Definir propiedades y comportamientos relevantes para el nodo.

Modelado de la Distribución de Artefactos

Visualizar la ubicación física de los artefactos en los nodos es crucial para entender la configuración de un sistema.

Pasos:

1. **Ubicar artefactos:** Asignar cada artefacto significativo a un nodo determinado.
2. **Considerar duplicación:** Es común que el mismo tipo de artefacto resida en múltiples nodos.
3. **Representar la localización de una de estas tres formas:**
    - **No visible:** Dejarlo solo en la especificación del modelo (ej., en la especificación de cada nodo).
    - **Relaciones de dependencia:** Conectar cada nodo con el artefacto que despliega.
    - **Compartimento adicional:** Listar los artefactos desplegados dentro de un compartimento especial del nodo (puede ser una lista textual o símbolos anidados de artefactos) (Figura 27.6). Esto es común en diagramas de objetos que muestran instancias específicas de nodos.

Sugerencias y Consejos para Modelar Nodos

Al modelar nodos, se está trabajando directamente con la **dimensión física** del sistema. Un **nodo bien estructurado** debe:

- Proporcionar una abstracción clara del hardware en el dominio de la solución.
- Descomponerse solo al nivel necesario para la comunicación (no sobredetallar).
- Mostrar solo atributos y operaciones relevantes para el dominio modelado.
- Desplegar directamente un conjunto de artefactos que residen en él.
- Estar conectado a otros nodos de forma que refleje fielmente la topología del sistema real.

Al dibujar un nodo en UML:

- Define un conjunto de **estereotipos con iconos apropiados** a nivel de proyecto u organización para mejorar la claridad visual para los lectores.
- Muestra solo los **atributos y operaciones esenciales** para comprender el nodo en el contexto dado.

Los nodos son, por lo tanto, la base para los **diagramas de despliegue**, que son vitales para planificar y documentar cómo el software se ejecuta en la infraestructura de hardware.

Principios SOLID: Fundamentos para un Diseño de Software Robusto

Los **principios SOLID** son un conjunto de cinco principios de diseño de software introducidos por Robert C. Martin, diseñados para crear sistemas más **comprensibles, flexibles y fáciles de mantener**. Es crucial aplicarlos con **pragmatismo**, ya que una aplicación descuidada puede introducir una complejidad innecesaria. El objetivo es aspirar a ellos, no aplicarlos ciegamente en cada escenario.

S: Principio de Responsabilidad Única (Single Responsibility Principle - SRP)

Definición: "Una clase sólo debe tener una razón para cambiar."

Este principio busca que cada clase sea **responsable de una única porción de funcionalidad** dentro del software, encapsulando completamente esa responsabilidad.

Objetivo Principal

Reducir la **complejidad** del código. A medida que un programa crece, las clases monolíticas se vuelven difíciles de manejar, recordar detalles, navegar y modificar sin riesgo.

Problemas de Violación

- **Aumento de la complejidad:** Clases grandes e inmanejables.
- **Alto acoplamiento:** Un cambio en una de las responsabilidades de la clase puede forzar cambios inesperados o romper otras funcionalidades no relacionadas dentro de la misma clase.
- **Dificultad de mantenimiento:** Es difícil enfocarse en un aspecto específico del programa sin afectar otros.

Ejemplo

- **ANTES:** Una clase `Empleado` maneja tanto la información del empleado como la impresión de informes de horas. Si el formato del informe cambia, la clase `Empleado` debe modificarse, lo que introduce un riesgo innecesario.
- **DESPUÉS:** Se extrae el comportamiento relacionado con la impresión de informes de horas a una **clase separada** (ej., `GeneradorDeInformesDeHoras`). Ahora, `Empleado` solo se encarga de la información del empleado, y `GeneradorDeInformesDeHoras` de los informes. Cada clase tiene una única razón para cambiar.

O: Principio Abierto/Cerrado (Open/Closed Principle - OCP)

Definición: "Las clases deben estar abiertas a la extensión pero cerradas a la modificación."

La idea es poder **añadir nuevas funcionalidades** a una clase (abierta a la extensión) **sin alterar su código existente** (cerrada a la modificación).

¿Qué significa?

- **Abierta a la extensión:** Se puede extender una clase (crear una subclase) para añadir nuevos métodos, campos o sobrescribir comportamiento.
- **Cerrada a la modificación:** Una vez que una clase ha sido desarrollada, probada y puesta en uso, su interfaz y su código interno no deben cambiarse directamente para evitar romper el código cliente existente. En su lugar, se extiende su comportamiento mediante herencia o composición.

Cuándo aplicar

Este principio es crucial cuando se trabaja con **código existente que ya está en producción** o es utilizado por otros clientes. Si hay un error, se debe corregir directamente; el OCP se aplica para nuevas características.

Ejemplo

- **ANTES:** Una clase `Pedido` en una aplicación de comercio electrónico calcula los costos de envío con todos los métodos de envío incrustados. Añadir un nuevo

método de envío requiere modificar la clase `Pedido`, arriesgándose a romper la funcionalidad existente.

- **DESPUÉS:** Se aplica el **patrón Strategy**. Los métodos de envío se extraen a **clases separadas** que implementan una **interfaz común** (ej., `InterfazEnvio`).
    - Ahora, para añadir un nuevo método de envío, solo se crea una **nueva clase** que implementa `InterfazEnvio` sin tocar la clase `Pedido`.
    - La clase `Pedido` (o su cliente) simplemente se vincula con un objeto de envío a través de la interfaz. Este diseño también facilita el SRP al mover el cálculo del tiempo de entrega a las clases de envío relevantes.

L: Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)

Definición: "Al extender una clase, recuerda que debes tener la capacidad de pasar objetos de las subclases en lugar de objetos de la clase padre, sin descomponer el código cliente." Este principio asegura que las **subclases sean compatibles con el comportamiento de sus superclases**. Al sobrescribir un método, el comportamiento debe ser una **extensión**, no un reemplazo que altere las expectativas del código cliente.

Requisitos Formales para Subclases y Métodos

El LSP es menos abierto a la interpretación y establece reglas específicas, especialmente importantes en el desarrollo de bibliotecas y frameworks:

- **Parámetros de Métodos:** Los tipos de parámetros en el método de una subclase deben coincidir o ser **más abstractos** (contravarianza) que los de la superclase.
    - **BIEN:** Si `alimentar(Gato c)` en la superclase, la subclase puede tener `alimentar(Animal c)` (donde `Animal` es superclase de `Gato`).
    - **MAL:** Si la subclase tiene `alimentar(GatoDeBengala c)` (donde `GatoDeBengala` es subclase de `Gato`), el código cliente que espera `Gato` fallará.
- **Tipo de Retorno de Métodos:** El tipo de retorno en el método de una subclase debe coincidir o ser un **subtipo** (covarianza) del tipo de retorno de la superclase.
    - **BIEN:** Si `comprarGato(): Gato` en la superclase, la subclase puede tener `comprarGato(): GatoDeBengala`.
    - **MAL:** Si la subclase tiene `comprarGato(): Animal`, el cliente que espera un `Gato` se descompondrá.
- **Excepciones:** Un método de una subclase no debe arrojar tipos de excepciones que no se esperen del método base. Las excepciones deben coincidir o ser subtipos. Esto evita que el código cliente (con sus bloques `try-catch`) falle inesperadamente. (En muchos lenguajes modernos de tipado estático, estas reglas son impuestas por el compilador).
- **Precondiciones:** Una subclase **no debe fortalecer las condiciones previas** de un método. No puede exigir más de lo que exige la superclase. (Ej., si el padre acepta un número, el hijo no puede exigir que sea positivo).
- **Postcondiciones:** Una subclase **no debe debilitar las condiciones posteriores** de un método. El comportamiento prometido por el padre (ej., cerrar conexiones de BD) debe ser mantenido o extendido por el hijo.
- **Invariantes:** Los invariantes de una superclase deben **preservarse**. Las condiciones bajo las cuales un objeto tiene sentido deben mantenerse. Es la regla más informal y difícil de garantizar si no se comprenden todos los invariantes implícitos.
- **Campos Privados:** Una subclase no debe cambiar los valores de campos privados de la superclase (aunque algunos lenguajes lo permitan a través de reflexión o falta de protección).

Ejemplo

- **ANTES:** Una jerarquía de documentos donde `DocumentosDeSoloLectura` hereda de `Documento` e intenta "desactivar" el método `guardar()` lanzando una excepción. El código cliente que espera un `Documento` no está preparado para esta excepción y fallará. También viola el OCP al forzar al cliente a verificar el tipo de documento antes de guardar.
- **DESPUÉS:** Se rediseña la jerarquía. La clase `DocumentosDeSoloLectura` se convierte en la **clase base**, y `DocumentoDeEscritura` (que añade el comportamiento de guardar) hereda de ella. Así, la subclase extiende el comportamiento de la superclase en lugar de contradecirlo.

I: Principio de Segregación de la Interfaz (Interface Segregation Principle - ISP)

Definición: "No se debe forzar a los clientes a depender de métodos que no utilizan." Este principio aboga por crear **interfaces más pequeñas y específicas** en lugar de interfaces "gruesas" o monolíticas.

Objetivo

Evitar que las clases cliente tengan que implementar o depender de comportamientos que realmente no necesitan o no les conciernen.

Problemas de Violación

- Si una interfaz es demasiado grande, un cambio en uno de sus métodos (incluso si solo afecta a una pequeña parte de su funcionalidad) puede **descomponer a todos los clientes** que implementan esa interfaz, aunque no usen el método cambiado.
- Fuerza a las clases a implementar métodos "vacíos" o "simulados" (`mock`) sin sentido, lo que lleva a un código menos limpio y confuso.

Solución

Desintegrar las interfaces "gruesas" en varias interfaces más detalladas y específicas. Una clase puede implementar múltiples interfaces refinadas si es necesario, pero las clases cliente solo implementarán aquellas que sean relevantes para ellas.

Ejemplo

- **ANTES:** Una biblioteca para la integración con proveedores de computación en la nube tiene una interfaz `CloudProvider` muy "abotargada" que asume que todos los proveedores tienen el mismo conjunto de funciones que Amazon Cloud. Cuando se intenta integrar un nuevo proveedor con menos funciones, este se ve forzado a implementar métodos que no le son aplicables, resultando en un código ineficiente o con "maquetas".
- **DESPUÉS:** La interfaz `CloudProvider` se **divide en un grupo de interfaces más específicas** (ej., `CloudStorage`, `CloudCompute`, `CloudMessaging`). Ahora, cada proveedor puede implementar solo las interfaces que corresponden a los servicios que realmente ofrece, sin depender de métodos que no utiliza.

D: Principio de Inversión de la Dependencia (Dependency Inversion Principle - DIP)

Definición: "Las clases de alto nivel no deben depender de clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones."

Este principio busca invertir la dirección tradicional de las dependencias.

Niveles de Clases

- **Clases de bajo nivel:** Implementan operaciones básicas y detalles de infraestructura (ej., trabajar con disco, red, base de datos).
- **Clases de alto nivel:** Contienen la lógica de negocio compleja y coordinan las operaciones de bajo nivel.

Tradicionalmente, las clases de alto nivel dependen directamente de las de bajo nivel.

Inversión de la Dependencia

El DIP sugiere cambiar esta dirección:

1. **Definir interfaces (abstracciones) para las operaciones de bajo nivel:** Estas interfaces deben ser descritas en términos de la lógica de negocio de alto nivel (ej., `abrirInforme(archivo)` en lugar de `abrirArchivo(x)`, `leerBytes(n)`).
2. **Las clases de alto nivel dependen de estas interfaces:** Esto crea una dependencia mucho más débil y flexible.
3. **Las clases de bajo nivel implementan estas interfaces:** Al hacerlo, las clases de bajo nivel se vuelven dependientes de las abstracciones de alto nivel, **invirtiendo la dirección de la dependencia original**.

El DIP a menudo trabaja de la mano con el **Principio Abierto/Cerrado (OCP)**, permitiendo extender las clases de bajo nivel para usarlas con diferentes lógicas de negocio sin modificar las clases existentes.

Ejemplo

- **ANTES:** Una clase de alto nivel `InformesPresupuestarios` (lógica de negocio) depende directamente de una clase de bajo nivel `BaseDeDatos` (detalles de almacenamiento). Cualquier cambio en `BaseDeDatos` (ej., nueva versión de servidor) afecta a `InformesPresupuestarios`, aunque no debería conocer esos detalles.
- **DESPUÉS:** Se crea una **interfaz de alto nivel** (abstracción) para operaciones de lectura/escritura (ej., `InterfazAlmacenamiento`).
    - La clase `InformesPresupuestarios` ahora utiliza `InterfazAlmacenamiento`.
    - La clase `BaseDeDatos` (de bajo nivel) **implementa** `InterfazAlmacenamiento`.
    - La dirección de la dependencia se invierte: la clase de bajo nivel `BaseDeDatos` ahora depende de la abstracción de alto nivel `InterfazAlmacenamiento`.

¿Qué es un Patrón de Diseño?

Un **patrón de diseño** es una **solución general y reutilizable** para un problema común que surge repetidamente en el diseño de software. Imagínalos como **planos o plantillas prefabricadas** que puedes adaptar y personalizar para resolver situaciones recurrentes en tu código.

Es crucial entender que un patrón de diseño **no es una porción de código lista para copiar y pegar** como una función o una biblioteca. En cambio, es un **concepto de alto nivel** que describe una estrategia para resolver un problema específico. Puedes seguir las directrices del patrón e implementarlo de una manera que se ajuste a las particularidades de tu propio programa. Esto significa que el código de un mismo patrón aplicado en dos programas diferentes puede variar.

A menudo, los patrones de diseño se confunden con los **algoritmos**, ya que ambos ofrecen soluciones a problemas conocidos. Sin embargo, la diferencia clave radica en su nivel de abstracción:

- Un **algoritmo** es como una **receta de cocina**: define un conjunto de pasos claros y específicos para lograr un objetivo determinado. No hay mucha libertad en el orden o los detalles de la ejecución.
- Un **patrón de diseño** es más parecido a un **plano arquitectónico**: te muestra cómo se verá el resultado final y cómo funcionarán sus partes, pero el orden exacto de implementación y los detalles específicos dependen de ti y del contexto de tu proyecto.

¿En qué Consiste la Descripción de un Patrón?

Para facilitar su reproducción y comprensión en diversos contextos, la mayoría de los patrones de diseño se describen de manera formal, incluyendo las siguientes secciones clave:

- **Propósito:** Una breve explicación del problema que el patrón intenta resolver y la esencia de su solución.
- **Motivación:** Una descripción más detallada del problema y la justificación de por qué el patrón ofrece una solución eficaz.
- **Estructura:** Un diagrama (a menudo en UML) que ilustra las diferentes partes del patrón (clases, interfaces, etc.) y cómo se relacionan entre sí.
- **Ejemplo de Código:** Una implementación práctica del patrón en uno de los lenguajes de programación populares, que ayuda a asimilar la idea subyacente.

Patrón de Diseño: Factory Method

El **Factory Method** (también conocido como Método Fábrica o Constructor Virtual) es un **patrón de diseño creacional** que aborda el problema de crear objetos de forma flexible, permitiendo a las subclases decidir qué tipo de objetos instanciar.

Propósito

Proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases modifiquen el tipo exacto de objetos que se crearán.

Problema: Acoplamiento Fuerte con Clases Concretas

Imagina una aplicación de logística inicial que solo maneja el transporte por camión. Gran parte del código está directamente acoplado a la clase `Camión`. Si más tarde necesitas añadir transporte marítimo (Barco), te enfrentarás a un problema de **acoplamiento fuerte**.

- **Necesidad de cambios masivos:** Añadir `Barco` implicaría modificar una gran parte de la base de código existente que usa `Camión`.
- **Código sucio y condicionales:** Si sigues añadiendo tipos de transporte (avión, tren), el código se llenará de sentencias condicionales (`if/else` o `switch`) para determinar qué clase de transporte instanciar, lo que lo hace difícil de mantener y extender.

Solución: Desacoplar la Creación de Objetos

El patrón Factory Method propone una solución elegante:

1. **Introduce un "método fábrica" especial:** En lugar de llamar directamente al operador `new` para crear objetos, invoca un método específico (el "método fábrica") para esta tarea.
2. **Los "productos" siguen una interfaz común:** Los objetos creados por el método fábrica (conocidos como **productos**) deben implementar una interfaz o heredar de una clase base común. Por ejemplo, `Camión` y `Barco` implementarían la interfaz `Transporte`.
3. **Las subclases del "creador" sobrescriben el método fábrica:** Las subclases de la clase que contiene el método fábrica pueden sobrescribir este método para devolver diferentes tipos de productos concretos.

¿Cómo funciona? En el ejemplo de logística:

- Una interfaz `Transporte` declara un método `entrega()`.
- `Camión` y `Barco` son **Productos Concretos** que implementan `Transporte` de manera específica.
- Una clase `Logística` (el **Creador**) declara un método fábrica abstracto `createTransporte(): Transporte`.
- `LogísticaTerrestre` (un **Creador Concreto**) hereda de `Logística` y sobrescribe `createTransporte()` para devolver un `new Camión()`.

- `LogísticaMarítima` (otro **Creador Concreto**) hereda de `Logística` y sobrescribe `createTransporte()` para devolver un `new Barco()`.

El **código cliente** que utiliza el método fábrica solo interactúa con la interfaz `Transporte`. No necesita saber la clase concreta del producto que recibe, lo que lo hace flexible y extensible.

Estructura del Patrón Factory Method

El patrón Factory Method consta de cuatro componentes principales:

1. **Producto (Product):**
   - Declara la **interfaz común** que todos los objetos creados (productos concretos) deben seguir.
   - Ejemplo: `interface Transporte`.
2. **Producto Concreto (Concrete Product):**
   - Son las **distintas implementaciones** de la interfaz del producto.
   - Ejemplo: `class Camión implements Transporte`, `class Barco implements Transporte`.
3. **Creador (Creator):**
   - Declara el **método fábrica** que devuelve nuevos objetos de producto.
   - El tipo de retorno de este método debe coincidir con la interfaz del producto.
   - Puede ser una clase abstracta con un método fábrica abstracto (forzando a las subclases a implementarlo) o puede proporcionar una implementación por defecto.
   - **Importante:** La creación del producto no es su responsabilidad principal. Contiene la lógica de negocio central que depende de los productos devueltos por el método fábrica, y el patrón ayuda a desacoplar esta lógica de las clases de producto concretas.
   - Ejemplo: `abstract class Dialog` (en el pseudocódigo), `abstract class Logística`.
4. **Creador Concreto (Concrete Creator):**
   - **Sobrescribe** el método fábrica base para devolver un tipo diferente de producto concreto.
   - Ejemplo: `class WindowsDialog extends Dialog`, `class WebDialog extends Dialog`, `class LogísticaTerrestre extends Logística`, `class LogísticaMarítima extends Logística`.

Pseudocódigo de Ejemplo: Diálogo Multiplataforma

Este ejemplo ilustra cómo se puede utilizar el Factory Method para crear elementos de UI multiplataforma sin acoplar el código cliente a clases UI concretas:

```
// Interfaz de Producto
interface Button is
    method render()
    method onClick(f)

// Productos Concretos
class WindowsButton implements Button is
    method render(a, b) is
        // Renderiza un botón en estilo Windows.
    method onClick(f) is
        // Vincula un evento clic de OS nativo.

class HTMLButton implements Button is
    method render(a, b) is
        // Devuelve una representación HTML de un botón.
```

```
    method onClick(f) is
        // Vincula un evento clic de navegador web.

// Clase Creadora
class Dialog is
    abstract method createButton():Button // Método fábrica

    method render() is
        Button okButton = createButton() // Usa el método fábrica para obtener el producto
        okButton.onClick(closeDialog)
        okButton.render()

// Creadores Concretos
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()

// Código Cliente (Application)
class Application is
    field dialog: Dialog

    method initialize() is
        config = readApplicationConfigFile()
        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    method main() is
        this.initialize()
        dialog.render() // El diálogo renderiza los botones a través de su método fábrica
```

En este ejemplo, la `Application` no necesita saber si está usando un `WindowsButton` o un `HTMLButton`. Solo interactúa con el `Dialog` genérico, que a su vez se encarga de crear el tipo de botón apropiado a través de su `createButton()` método fábrica.

Aplicabilidad del Factory Method

Utiliza el patrón Factory Method cuando:

- **No conozcas de antemano los tipos exactos de objetos** que tu código debe crear. El patrón desacopla el código de construcción del producto del código que lo utiliza, facilitando la extensión.
- **Necesites ofrecer una forma de extender los componentes internos** de una biblioteca o framework. Permite a los usuarios sobrescribir el método fábrica y devolver sus propias subclases de productos personalizados. Por ejemplo, un framework de UI podría permitirte crear botones con una forma personalizada (`BotónRedondo`) sin modificar el código base del framework.
- **Quieras reutilizar objetos existentes** para ahorrar recursos del sistema (ej., conexiones a bases de datos, recursos de red). El método fábrica puede buscar en

una caché o agrupación de objetos existentes antes de crear uno nuevo, lo cual no es posible con un constructor tradicional.

Cómo Implementar el Factory Method

1. **Define la Interfaz de Producto:** Crea una interfaz o clase base común que todos los productos deben seguir, declarando los métodos que tienen sentido para todos ellos.
2. **Añade el Método Fábrica al Creador:** En la clase creadora, añade un método fábrica (puede ser vacío o abstracto) cuyo tipo de retorno coincida con la interfaz de producto.
3. **Sustituye llamadas al constructor:** Reemplaza todas las llamadas directas a los constructores de productos (`new ConcreteProduct()`) en el código del creador por invocaciones al método fábrica. Mueve el código de creación al método fábrica. Puedes añadir un parámetro temporal al método fábrica si necesitas controlar el tipo de producto inicialmente.
4. **Crea Subclases Creadoras Concretas:** Para cada tipo de producto, crea una subclase del creador. En cada una, sobrescribe el método fábrica y extrae la lógica de creación del producto específico.
5. **Refina el Método Fábrica Base:** Si el método fábrica base queda vacío después de las extracciones, hazlo abstracto. Si aún tiene lógica, puede servir como un comportamiento por defecto.

Pros y Contras

Ventajas (Pros):

- **Desacoplamiento:** Elimina el acoplamiento fuerte entre el creador y los productos concretos.
- **Principio de Responsabilidad Única (SRP):** Concentra el código de creación de productos en un solo lugar, facilitando su mantenimiento.
- **Principio Abierto/Cerrado (OCP):** Permite añadir nuevos tipos de productos al programa sin modificar el código cliente existente.

Desventajas (Contras):

- **Aumento de complejidad:** Puede hacer que el código sea más complicado al introducir una multitud de nuevas subclases para implementar el patrón, especialmente si no se inserta en una jerarquía de clases existente.

Relaciones con Otros Patrones

- **Abstract Factory, Prototype, Builder:** Muchos diseños comienzan con Factory Method y evolucionan hacia estos patrones más complejos y flexibles cuando se requiere una mayor sofisticación en la creación de objetos.
- **Abstract Factory:** A menudo se compone de un grupo de métodos de fábrica.
- **Iterator:** Puede utilizarse junto con Factory Method para permitir que las subclases de una colección devuelvan diferentes tipos de iteradores compatibles.
- **Prototype:** Factory Method se basa en la herencia, mientras que Prototype no. Prototype requiere inicialización compleja del objeto clonado, a diferencia de Factory Method.
- **Template Method:** Factory Method es una especialización de Template Method. Al mismo tiempo, un Factory Method puede ser un paso dentro de un Template Method más grande.

Patrón de Diseño: Singleton

El **Singleton** (también conocido como Instancia única) es un **patrón de diseño creacional** que asegura que una clase tenga una **única instancia**, a la vez que proporciona un **punto de acceso global** a esa instancia.

Propósito

Garantiza que una clase tenga una sola instancia y proporciona un método para acceder a ella desde cualquier parte del programa.

Problema: Control de Instancias y Acceso Global

El patrón Singleton resuelve dos problemas fundamentales, aunque al hacerlo **viola el Principio de Responsabilidad Única (SRP)**, ya que se encarga de la creación del objeto y de la lógica de negocio.

1. **Garantizar una instancia única por clase:**
    - **Motivación:** Controlar el acceso a un **recurso compartido** (ej., una conexión a una base de datos, un gestor de configuración, un sistema de archivos).
    - **Comportamiento:** Si intentas crear un objeto de esta clase cuando ya existe una instancia, no se creará una nueva; en su lugar, se te devolverá la instancia ya existente.
    - **Limitación del constructor normal:** Los constructores normales siempre devuelven un nuevo objeto, lo que hace imposible implementar este comportamiento directamente. Los clientes pueden no darse cuenta de que siempre están trabajando con el mismo objeto.
2. **Proporcionar un punto de acceso global a dicha instancia:**
    - **Motivación:** Acceder a un objeto esencial desde cualquier parte del programa sin usar variables globales inseguras (que pueden ser sobrescritas por cualquier código).
    - **Ventaja:** Permite el acceso global pero **evita que la instancia sea sobrescrita** por código externo.
    - **Organización:** Centraliza la lógica para controlar la instancia única dentro de la propia clase, en lugar de dispersarla por todo el código.

Solución: Constructor Privado y Método de Creación Estático

Todas las implementaciones del patrón Singleton comparten dos pasos clave:

1. **Hacer el constructor por defecto `privado`:** Esto impide que otros objetos utilicen el operador `new` directamente con la clase Singleton, forzando a los clientes a usar el método de creación especial.
2. **Crear un método de creación `estático` (ej., `obtenerInstancia()` o `getInstance()`):**
    - Este método actúa como el "constructor" público de la clase.
    - En su primera llamada, invoca al constructor privado para crear la instancia del Singleton y la guarda en un **campo estático** dentro de la propia clase.
    - En llamadas posteriores, simplemente devuelve la instancia que ya está almacenada en ese campo estático.

Así, cualquier código que necesite el objeto Singleton invoca su método estático y siempre recibe la misma instancia.

Analogía en el Mundo Real

El **gobierno de un país** es una analogía perfecta. Un país solo tiene un gobierno oficial. Aunque las personas que lo componen cambien, el "Gobierno de X" es un punto de acceso global y único que representa a la autoridad en turno.

Estructura

- La clase **Singleton** tiene un **método estático** `obtenerInstancia()` (o `getInstance()`) que es el único punto de acceso para obtener la instancia única.
- El **constructor** de la clase Singleton debe ser **privado**.

Pseudocódigo de Ejemplo: Conexión a Base de Datos Singleton

Este ejemplo muestra una clase `Database` implementada como un Singleton, asegurando que solo haya una conexión a la base de datos en toda la aplicación.

```
// Clase Database (Singleton)
class Database is
    // Campo estático para almacenar la única instancia del Singleton
    private static field instance: Database

    // Constructor privado para evitar la creación directa con 'new'
    private constructor Database() is
        // Código de inicialización de la conexión a la base de datos
        // (ej., conectar al servidor, establecer credenciales, etc.)
        // ...

    // Método estático y público para obtener la instancia del Singleton
    public static method getInstance() is
        // Lógica de inicialización diferida (Lazy Initialization)
        // La instancia se crea solo la primera vez que se solicita
        if (Database.instance == null) then
            // Para entornos multi-hilo, se necesita un bloqueo para evitar
            // que múltiples hilos creen la instancia simultáneamente.
            acquireThreadLock() and then
                // Doble verificación (Double-Checked Locking) para asegurar
                // que la instancia no se haya creado mientras esperábamos el bloqueo.
                if (Database.instance == null) then
                    Database.instance = new Database()
        return Database.instance

    // Lógica de negocio que se ejecuta en la instancia única
    public method query(sql) is
        // Por ejemplo, todas las consultas a la base de datos pasan por aquí.
        // Se podría añadir aquí lógica de throttling o caché.
        // ...

// Código de la Aplicación
class Application is
    method main() is
        // Obtener la primera instancia de la base de datos
        Database foo = Database.getInstance()
        foo.query("SELECT * FROM users")

        // ...

        // Obtener la "segunda" instancia de la base de datos
        Database bar = Database.getInstance()
        bar.query("INSERT INTO products VALUES (...)")

        // La variable `bar` contendrá *exactamente el mismo objeto* que la variable `foo`.
        // Esto garantiza que solo hay una conexión de base de datos activa.
```

Aplicabilidad

Utiliza el patrón Singleton cuando:

- **Una clase solo debe tener una instancia:** Esto es común para controladores, gestores de recursos (como conexiones de base de datos, gestores de archivos), configuraciones de aplicación o componentes de logging. El Singleton deshabilita otras formas de crear objetos de la clase, excepto su método de creación especial.

- **Necesites un control más estricto que las variables globales:** A diferencia de una variable global (que puede ser sobrescrita), el Singleton garantiza la unicidad y protege la instancia en caché de ser reemplazada por código externo (excepto por la propia clase Singleton). Puedes ajustar el método `getInstance()` si, excepcionalmente, necesitas un número limitado de instancias en lugar de solo una.

Cómo Implementar

1. **Campo estático privado:** Añade un campo estático y privado a la clase para almacenar la única instancia.
2. **Método de creación estático público:** Declara un método estático y público (ej., `getInstance()`) para obtener la instancia.
3. **Inicialización diferida:** Implementa la lógica dentro del método estático: si la instancia no ha sido creada, créala y guárdala; de lo contrario, devuelve la instancia existente.
4. **Constructor privado:** Declara el constructor de la clase como privado.
5. **Reemplazar llamadas al constructor:** Revisa el código cliente y sustituye todas las llamadas directas al constructor de la clase por llamadas al método de creación estático (`getInstance()`).

Pros y Contras

Ventajas (Pros):

- **Garantía de instancia única:** Te asegura que solo existe una instancia de la clase.
- **Punto de acceso global:** Proporciona una forma controlada de acceder a esa instancia desde cualquier parte del programa.
- **Inicialización diferida (Lazy Initialization):** El objeto se crea solo cuando se necesita por primera vez, ahorrando recursos si la instancia no se utiliza.

Desventajas (Contras):

- **Violación del SRP:** Resuelve dos problemas (control de instancia y acceso global) a la vez, lo que puede considerarse una violación del Principio de Responsabilidad Única.
- **Encubrimiento de mal diseño:** Puede enmascarar un diseño deficiente donde los componentes del programa están demasiado acoplados entre sí.
- **Manejo de concurrencia:** Requiere un tratamiento especial (bloqueos) en entornos multi-hilo para evitar que se creen múltiples instancias simultáneamente.
- **Dificultad en pruebas unitarias:** Puede complicar las pruebas unitarias del código cliente, ya que muchos frameworks de prueba dependen de la herencia o la inyección de dependencias para crear objetos simulados (`mock objects`), lo cual es difícil con constructores privados y métodos estáticos. Esto a menudo lleva a soluciones complejas o a evitar las pruebas para el Singleton.

Relaciones con Otros Patrones

- **Facade:** Una clase Fachada a menudo puede ser un Singleton, ya que una única instancia de la fachada suele ser suficiente para coordinar subsistemas complejos.
- **Flyweight:** Podría parecer similar a Singleton si se reduce todo el estado compartido a un único objeto Flyweight. Sin embargo, se diferencian en que:
    - Solo hay **una instancia Singleton**, mientras que una clase Flyweight puede tener **varias instancias** con diferentes estados intrínsecos.
    - Un objeto Singleton puede ser **mutable**, mientras que los objetos Flyweight son generalmente **inmutables**.

El patrón Singleton es potente para el control de recursos, pero sus desventajas, especialmente en la prueba y el diseño modular, han llevado a muchos desarrolladores a preferir la **Inyección de Dependencias** como una alternativa para gestionar la vida útil de

los objetos y el acceso a los recursos compartidos, ya que promueve un acoplamiento más bajo.

Patrón de Diseño: Adapter

El **Adapter** (también conocido como Adaptador, Envoltorio o Wrapper) es un **patrón de diseño estructural** que permite la **colaboración entre objetos con interfaces incompatibles**. Actúa como un puente entre dos interfaces diferentes, permitiendo que clases que de otro modo no podrían trabajar juntas, lo hagan sin problemas.

Propósito

Permite que clases con interfaces incompatibles colaboren entre sí.

Problema: Interfaces Incompatibles

Imagina que tienes una aplicación de monitoreo del mercado de valores que procesa datos en formato XML. Decides integrar una nueva biblioteca de análisis de terceros, pero esta biblioteca solo funciona con datos en formato JSON.

- **Incompatibilidad directa:** No puedes usar la biblioteca "tal cual" porque espera un formato de datos diferente al que tu aplicación produce.
- **Restricciones de modificación:**
    - Cambiar la biblioteca para que acepte XML podría romper su funcionalidad existente o sus dependencias internas.
    - A menudo, no se tiene acceso al código fuente de las bibliotecas de terceros, haciendo imposible su modificación.

Solución: Introducir un Adaptador

La solución es crear un **adaptador**. Este es un objeto especial que:

1. **Convierte la interfaz** de un objeto para que otro objeto pueda entenderla.
2. **Envuelve uno de los objetos** (el "servicio") para esconder la complejidad de la conversión que ocurre "tras bambalinas". El objeto envuelto no es consciente de la existencia del adaptador.

¿Cómo funciona?

El adaptador se sitúa entre el **cliente** (tu código existente) y el **servicio** (la biblioteca incompatible).

- El adaptador implementa una **interfaz compatible con el cliente**.
- El cliente invoca los métodos del adaptador a través de esta interfaz.
- Al recibir la llamada, el adaptador **traduce la solicitud** (tanto los datos como la estructura de la llamada) al formato y orden que el objeto de servicio espera, y luego delega la llamada a este último.

Volviendo al ejemplo del mercado de valores: Crearías adaptadores de XML a JSON. Tu código se comunicaría con la biblioteca de análisis solo a través de estos adaptadores. Cuando un adaptador recibe datos XML, los traduce a JSON y los pasa a los métodos correspondientes del objeto de análisis envuelto.

Analogía en el Mundo Real

Un **adaptador de corriente universal** es una excelente analogía. Cuando viajas de un país a otro (ej., de Europa a EE. UU.), los tipos de enchufes pueden ser incompatibles. Un adaptador de corriente te permite conectar tu dispositivo (europeo) a la toma de corriente (americana), traduciendo la conexión física sin que tu dispositivo necesite cambiar su enchufe.

Estructura del Patrón Adapter

Existen dos implementaciones principales del patrón Adapter:

1. Adaptador de Objetos (Object Adapter)

Esta es la implementación más común y utiliza el principio de **composición de objetos**: el adaptador implementa la interfaz que el cliente espera y envuelve una instancia del objeto de servicio incompatible.

- **Cliente (Client):** Contiene la lógica de negocio existente del programa y espera interactuar con objetos a través de una `Interfaz con el Cliente`.
- **Interfaz con el Cliente (Client Interface / Target):** Define el protocolo que el cliente espera y con el que puede interactuar.
- **Servicio (Service / Adaptee):** Es la clase útil existente (a menudo de terceros o heredada) cuya interfaz es incompatible con el cliente. No puedes o no quieres modificarla.
- **Adaptador (Adapter):**
  - Implementa la `Interfaz con el Cliente`.
  - Contiene una referencia (envuelve) a un objeto de `Servicio`.
  - Recibe llamadas del `Cliente` a través de la `Interfaz con el Cliente` y las traduce en llamadas al `Servicio`, adaptando el formato o la estructura de los datos/métodos.
  - El cliente no está acoplado al `Adaptador` concreto, siempre que trabaje con él a través de la `Interfaz con el Cliente`. Esto permite introducir nuevos tipos de adaptadores sin afectar al cliente.

2. Adaptador de Clase (Class Adapter)

Esta implementación utiliza la **herencia múltiple**, por lo que solo es posible en lenguajes que la soportan (como C++).

- La **Clase Adaptadora** hereda *simultáneamente* de la `Interfaz con el Cliente` y de la clase `Servicio`.
- No necesita envolver un objeto de servicio porque hereda sus comportamientos directamente.
- La adaptación se realiza dentro de los métodos sobrescritos, donde los métodos de la `Interfaz con el Cliente` llaman directamente a los métodos heredados del `Servicio`.

Pseudocódigo de Ejemplo: Piezas Cuadradas en Hoyos Redondos

Este ejemplo clásico ilustra el `Adapter` para encajar una `SquarePeg` (pieza cuadrada) en un `RoundHole` (hoyo redondo).

```
// Clase Cliente y su Interfaz compatible (RoundHole y RoundPeg)
class RoundHole is
    constructor RoundHole(radius) { ... }
    method getRadius() is // Devuelve el radio del agujero
    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }
    method getRadius() is // Devuelve el radio de la pieza redonda

// Clase de Servicio con interfaz incompatible
class SquarePeg is
    constructor SquarePeg(width) { ... }
    method getWidth() is // Devuelve la anchura de la pieza cuadrada

// Clase Adaptadora (implementa RoundPeg y envuelve SquarePeg)
class SquarePegAdapter extends RoundPeg is // Actúa como RoundPeg (interfaz con el
cliente)
    private field peg: SquarePeg // Envuelve la SquarePeg (el servicio)
```

```
    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // El adaptador simula ser una pieza redonda.
        // Calcula el radio del círculo más pequeño que puede contener el cuadrado.
        // radio = (ancho * sqrt(2)) / 2
        return peg.getWidth() * Math.sqrt(2) / 2

// En el código cliente
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // verdadero (encaja)

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)

// hole.fits(small_sqpeg) // ¡ERROR! Tipos incompatibles, no compila.

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)

hole.fits(small_sqpeg_adapter) // verdadero (el adaptador traduce y encaja)
hole.fits(large_sqpeg_adapter) // falso (el cuadrado grande no cabe incluso con el adaptador)
```

Aplicabilidad

Utiliza el patrón Adapter cuando:

- **Necesites usar una clase existente cuya interfaz no sea compatible** con el resto de tu código. Es ideal para integrar bibliotecas de terceros, código heredado o clases con interfaces "extrañas" sin modificarlas.
- **Quieras reutilizar varias subclases existentes que carecen de una funcionalidad común** que no puede añadirse a su superclase. En lugar de duplicar código en cada subclase, puedes crear un adaptador que envuelva estos objetos y les proporcione la funcionalidad faltante dinámicamente. Esto es similar al patrón Decorator, pero el Adapter se centra en la compatibilidad de interfaces.

Cómo Implementar

1. **Identifica clases incompatibles:** Asegúrate de tener al menos una clase `Servicio` útil (que no puedes cambiar) y una o varias clases `Cliente` que se beneficiarían de usarla.
2. **Declara la Interfaz con el Cliente:** Define una interfaz que describa cómo el `Cliente` se comunica con el `Servicio`.
3. **Crea la Clase Adaptadora:** Implementa la `Interfaz con el Cliente` en la nueva clase `Adaptadora`.
4. **Añade una Referencia al Servicio:** Incluye un campo privado en la clase `Adaptadora` para almacenar una referencia al objeto `Servicio`. Lo más común es inicializarlo a través del constructor.
5. **Implementa los Métodos del Adaptador:** Para cada método de la `Interfaz con el Cliente`, implementa el cuerpo del método en el `Adaptador` delegando el trabajo real al objeto `Servicio`. La clave es la traducción de interfaces y/o formatos de datos.

6. **Actualiza el Código Cliente:** Haz que las clases `Cliente` utilicen la clase `Adaptadora` a través de la `Interfaz con el Cliente`. Esto te permite cambiar o extender los adaptadores sin afectar el código cliente.

Pros y Contras

Ventajas (Pros):

- **Principio de Responsabilidad Única (SRP):** Separa el código de conversión de interfaz/datos de la lógica de negocio principal del programa.
- **Principio Abierto/Cerrado (OCP):** Puedes introducir nuevos tipos de adaptadores sin modificar el código cliente existente, siempre que trabajen a través de la `Interfaz con el Cliente`. Esto es ideal para integrar nuevas bibliotecas o versiones de API.

Desventajas (Contras):

- **Aumento de complejidad:** Introduce nuevas interfaces y clases, lo que puede aumentar la complejidad general del código, especialmente si el problema podría resolverse con un cambio directo simple en la clase de servicio.

Relaciones con Otros Patrones

- **Bridge vs. Adapter:**
  - **Bridge** se diseña **por adelantado** para permitir el desarrollo independiente de partes de una aplicación.
  - **Adapter** se usa comúnmente con una aplicación **existente** para hacer que clases incompatibles trabajen juntas.
- **Decorator vs. Adapter:**
  - **Adapter** proporciona una **interfaz completamente diferente** para acceder a un objeto existente.
  - **Decorator** mantiene la **misma interfaz o la amplía** (agrega responsabilidades) y permite la composición recursiva.
- **Proxy vs. Adapter:**
  - Con **Adapter**, accedes a un objeto existente a través de una **interfaz diferente**.
  - Con **Proxy**, la interfaz **sigue siendo la misma**, pero se controla el acceso al objeto.
  - Con **Decorator**, accedes al objeto a través de una **interfaz mejorada**.
- **Facade vs. Adapter:**
  - **Facade** define una **nueva interfaz simplificada** para un subsistema complejo de objetos existente.
  - **Adapter** intenta hacer una **interfaz existente utilizable** por un cliente incompatible. Un `Adapter` usualmente envuelve un solo objeto, mientras que un `Facade` trabaja con un subsistema completo.

El patrón Adapter es una herramienta valiosa para lidiar con la interoperabilidad en sistemas donde no se pueden modificar las interfaces existentes, promoviendo la reutilización de código y la flexibilidad.

Patrón de Diseño: Bridge

El patrón **Bridge** (también conocido como Puente) es un **patrón de diseño estructural** que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en **dos jerarquías separadas: abstracción e implementación**. Estas dos jerarquías pueden entonces **desarrollarse independientemente** la una de la otra.

Propósito

Desacopla una abstracción de su implementación, de modo que ambas puedan variar independientemente.

Problema: Explosión de la Jerarquía de Clases por Herencia Múltiple
El problema que resuelve el patrón Bridge surge a menudo cuando intentamos extender una clase en **múltiples dimensiones ortogonales (independientes)** usando la herencia. Consideremos un ejemplo sencillo con formas y colores:

- Tienes una jerarquía de formas: `Forma`, con subclases `Círculo` y `Cuadrado`.
- Ahora quieres añadir colores a estas formas, por ejemplo, `Rojo` y `Azul`.

Si usas la herencia directa para esto, terminarás con una explosión de clases: `CírculoRojo`, `CírculoAzul`, `CuadradoRojo`, `CuadradoAzul`.
El problema se agrava exponencialmente:

- Añadir un nuevo tipo de forma (`Triángulo`) requeriría crear `TriánguloRojo` y `TriánguloAzul` (2 nuevas clases).
- Añadir un nuevo color (`Verde`) requeriría crear `CírculoVerde`, `CuadradoVerde`, `TriánguloVerde` (3 nuevas clases).

La herencia simple no es adecuada para manejar este tipo de crecimiento multidimensional porque fuerza a todas las combinaciones posibles en una única jerarquía, lo que lleva a un código difícil de mantener, entender y extender.

Solución: Composición sobre Herencia
El patrón Bridge resuelve este problema cambiando de la herencia a la **composición de objetos**. La idea es extraer una de las dimensiones a una jerarquía de clases separada. La clase original (la Abstracción) entonces referencia un objeto de la nueva jerarquía (la Implementación), delegando el trabajo relacionado a ese objeto. Esta referencia actúa como un "puente".
Aplicado al ejemplo de formas y colores:

1. **Extraer la dimensión "color":** El código relacionado con el color se mueve a su propia jerarquía: `Color` como clase base, con subclases `Rojo` y `Azul`.
2. **Referencia en la "forma":** La clase `Forma` obtiene un campo de referencia que apunta a un objeto de `Color`.
3. **Delegación:** Cualquier trabajo relacionado con el color ahora es delegado por `Forma` al objeto de `Color` vinculado.

Ahora, añadir nuevos colores no requiere cambiar la jerarquía de formas, y viceversa. La jerarquía de formas (`Círculo`, `Cuadrado`) y la jerarquía de colores (`Rojo`, `Azul`) pueden desarrollarse y extenderse de forma totalmente independiente.

Abstracción e Implementación (Los Términos de la GoF)
Los términos **Abstracción** e **Implementación** son fundamentales en el patrón Bridge, aunque a veces puedan sonar complejos:

- **Abstracción:** Es la **capa de control de alto nivel** de una entidad. No realiza el trabajo real por sí misma; en cambio, **delega** ese trabajo a la capa de Implementación. (En nuestro ejemplo, la `Forma` es la Abstracción).
- **Implementación:** Define la **interfaz para las operaciones de bajo nivel** o "plataforma" que la Abstracción necesita. Contiene el código específico de la plataforma o la variación. (En nuestro ejemplo, el `Color` es la Implementación).

Importante: Cuando hablamos de "abstracción" e "implementación" en el contexto de Bridge, no nos referimos a las `interfaces` o `clases abstractas` de un lenguaje de programación per se, sino a conceptos lógicos de capas de funcionalidad.
En aplicaciones reales, un ejemplo común es una aplicación multiplataforma:

- **Abstracción:** La Interfaz Gráfica de Usuario (GUI), que maneja la lógica de control de alto nivel (ej., un botón es clickeable).
- **Implementación:** Las APIs del sistema operativo subyacente (ej., cómo se dibuja realmente un botón en Windows, Linux o macOS).

Sin Bridge, tendrías una "explosión" de clases combinando cada tipo de GUI con cada API de SO. Con Bridge, la GUI (Abstracción) tiene una referencia a una API específica (Implementación) y le delega las operaciones de bajo nivel. Puedes cambiar la GUI sin afectar las APIs, y añadir soporte para un nuevo SO solo requiere una nueva subclase en la jerarquía de Implementación.

Estructura del Patrón Bridge

1. **Abstracción (Abstraction):**
   - Proporciona la lógica de control de alto nivel.
   - Contiene una referencia (un "puente") a un objeto de la jerarquía de **Implementación**.
   - Delega la mayor parte del trabajo a este objeto de implementación.
2. **Implementación (Implementation):**
   - Declara la **interfaz común** que todas las implementaciones concretas deben seguir.
   - La Abstracción solo se comunica con un objeto de implementación a través de los métodos declarados aquí.
   - Normalmente, esta interfaz proporciona **operaciones primitivas**, mientras que la Abstracción define operaciones de más alto nivel que se construyen sobre ellas.
3. **Abstracciones Refinadas (Refined Abstraction):**
   - Proporcionan **variantes de la lógica de control** de alto nivel.
   - Extienden la `Abstracción` base y trabajan con diferentes implementaciones a través de la interfaz general de `Implementación`.
4. **Implementaciones Concretas (Concrete Implementations):**
   - Contienen el **código específico de la plataforma** o las variaciones concretas.
   - Implementan la interfaz de `Implementación`.
5. **Cliente (Client):**
   - Normalmente solo interactúa con la `Abstracción`.
   - Es responsable de vincular un objeto de `Abstracción` con un objeto de `Implementación` (generalmente a través del constructor de la `Abstracción`).

Pseudocódigo de Ejemplo: Dispositivos y Controles Remotos

Este ejemplo muestra cómo Bridge divide el código de una aplicación que gestiona dispositivos electrónicos (TV, Radio) y sus controles remotos.

```
// 1. Interfaz de Implementación (Device)
interface Device is
    method isEnabled()
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)

// 2. Implementaciones Concretas (Concrete Implementations)
class Tv implements Device is
    // ... Implementación específica para TV ...
class Radio implements Device is
    // ... Implementación específica para Radio ...

// 3. Abstracción (RemoteControl)
```

```
class RemoteControl is
    protected field device: Device // Referencia al objeto de implementación

    constructor RemoteControl(device: Device) is
        this.device = device

    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)

// 4. Abstracciones Refinadas (Refined Abstraction)
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)

// 5. Código Cliente (Client)
// El cliente vincula la abstracción con una implementación concreta
tv = new Tv()
remote = new RemoteControl(tv) // Un control remoto básico para TV
remote.togglePower()

radio = new Radio()
advancedRemote = new AdvancedRemoteControl(radio) // Un control remoto avanzado
para Radio
advancedRemote.togglePower()
advancedRemote.mute()
```

Aquí, la jerarquía de `RemoteControl` (Abstracción) se desarrolla independientemente de la jerarquía de `Device` (Implementación). El mismo `RemoteControl` puede operar una `Tv` o una `Radio` porque ambos implementan la interfaz `Device`. Añadir un nuevo tipo de remoto o un nuevo tipo de dispositivo no afecta a la otra jerarquía.

Aplicabilidad
Utiliza el patrón Bridge cuando:

- **Necesites dividir y organizar una clase monolítica** que tiene muchas variaciones de una sola funcionalidad (ej., una clase que interactúa con múltiples tipos de bases de datos o sistemas de archivos). Bridge te permite descomponerla en jerarquías separadas, facilitando el mantenimiento y minimizando el riesgo de errores.
- **Necesites extender una clase en varias dimensiones ortogonales (independientes)**. El patrón Bridge te permite extraer una jerarquía de clase separada para cada dimensión, y la clase original delega el trabajo a los objetos de esas jerarquías.
- **Necesites la capacidad de cambiar implementaciones durante el tiempo de ejecución.** Bridge permite reemplazar el objeto de implementación dentro de la

abstracción simplemente asignando una nueva referencia. (Aunque esto es una característica, no es la razón principal del patrón, y puede confundirse con el patrón Strategy, que se centra en el cambio de algoritmos).

Cómo Implementar

1. **Identifica las dimensiones ortogonales:** Encuentra los conceptos independientes en tus clases (ej., abstracción/plataforma, dominio/infraestructura, front-end/back-end, interfaz/implementación).
2. **Define la Interfaz de Abstracción:** Crea la clase base de la Abstracción y define las operaciones de alto nivel que el cliente necesita.
3. **Define la Interfaz de Implementación:** Crea una interfaz general para la Implementación, declarando las operaciones primitivas que la Abstracción necesitará.
4. **Crea Implementaciones Concretas:** Para cada plataforma o variación, crea una clase que implemente la interfaz de Implementación.
5. **Establece la Referencia de Implementación:** En la clase de Abstracción, añade un campo de referencia para el tipo de Implementación. La Abstracción delegará la mayor parte de su trabajo a este objeto referenciado.
6. **Crea Abstracciones Refinadas (Opcional):** Si tienes diferentes variantes de la lógica de alto nivel, crea subclases que extiendan la Abstracción base.
7. **Conexión por el Cliente:** El código cliente debe pasar un objeto de `Implementación` al constructor de la `Abstracción` para asociarlos. Después, el cliente puede interactuar solo con la `Abstracción`.

Pros y Contras

Ventajas (Pros):

- **Independencia de clases y aplicaciones:** Permite crear clases y aplicaciones independientes de plataformas o variaciones específicas.
- **Desacoplamiento:** El código cliente trabaja con abstracciones de alto nivel y no se expone a los detalles de la implementación de bajo nivel.
- **Principio Abierto/Cerrado (OCP):** Puedes introducir nuevas abstracciones e implementaciones de forma independiente, sin modificar el código existente.
- **Principio de Responsabilidad Única (SRP):** Permite enfocar la lógica de alto nivel en la abstracción y los detalles de la plataforma en la implementación.

Desventajas (Contras):

- **Mayor complejidad inicial:** Puede complicar el código si se aplica a una clase que ya está muy cohesionada o si el problema no presenta una clara separación de dimensiones. Introduce una capa adicional de abstracción que quizás no siempre sea necesaria.

Relaciones con Otros Patrones

- **Bridge vs. Adapter:**
  - **Bridge** se diseña **por adelantado** para permitir el desarrollo independiente de componentes.
  - **Adapter** se usa con una aplicación **existente** para hacer que clases incompatibles trabajen juntas.
- **Bridge, State, Strategy y Adapter:** Estos patrones tienen estructuras similares (basadas en composición y delegación) pero resuelven problemas diferentes. Un patrón es más que su estructura; comunica la intención y el problema que aborda.
- **Abstract Factory y Bridge:** Puedes combinar estos patrones. `Abstract Factory` puede encapsular las relaciones complejas entre abstracciones de `Bridge` e implementaciones específicas, ocultando esa complejidad al cliente.
- **Builder y Bridge:** La `directora` de Builder puede actuar como la `abstracción`, y las diferentes `constructoras` como las `implementaciones`.

El patrón Bridge es una herramienta poderosa para manejar la complejidad inherente a sistemas con múltiples dimensiones de variación, promoviendo un diseño modular y mantenible.

Patrón de Diseño: Facade

El patrón **Facade** (también llamado Fachada) es un **patrón de diseño estructural** que proporciona una **interfaz simplificada** a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Propósito

Ofrece una interfaz unificada y de alto nivel a un subsistema que de otro modo sería complejo.

Problema: Complejidad y Acoplamiento con Subsistemas Grandes

Imagina que tu código necesita interactuar con una biblioteca o framework muy sofisticado. Para usar sus funcionalidades, a menudo debes:

- Inicializar múltiples objetos.
- Gestionar sus dependencias.
- Ejecutar métodos en un orden específico.
- Proporcionar datos en formatos específicos.

El resultado es que la lógica de negocio de tus propias clases se vuelve **fuertemente acoplada a los detalles de implementación** de esas clases de terceros. Esto hace que tu código sea:

- **Difícil de comprender:** Debes conocer los detalles internos del subsistema.
- **Difícil de mantener:** Cualquier cambio en la biblioteca subyacente (ej., una actualización de versión) podría requerir modificaciones significativas en muchas partes de tu código.

Solución: La Clase Fachada

Una **fachada** es una clase que actúa como una **interfaz simplificada** para un subsistema complejo. Es como un "envoltorio" que expone solo la funcionalidad que es realmente importante para los clientes, ocultando la complejidad subyacente.

- **Funcionalidad limitada, pero suficiente:** La fachada puede no exponer todas las funcionalidades del subsistema, sino solo las que son relevantes para las necesidades del cliente.
- **Oculta la complejidad:** La fachada sabe a dónde dirigir las solicitudes del cliente dentro del subsistema y cómo coordinar sus diversas partes.

Ejemplo: Si estás creando una aplicación que sube videos de gatos a redes sociales y usas una biblioteca de conversión de video profesional, es probable que solo necesites una función simple como `codificar(nombreArchivo, formato)`. Una clase `VideoConverter` que encapsule la complejidad de la biblioteca (códecs, lectores de bitrate, mezcladores de audio, etc.) y exponga solo ese método simple sería tu fachada. Esto te protege de los detalles internos de la biblioteca.

Analogía en el Mundo Real

Piensa en **pedir comida por teléfono** a un restaurante grande. Cuando llamas, un **operador** (la Fachada) es tu único punto de contacto. No necesitas hablar con el cocinero, el cajero o el repartidor directamente. El operador te proporciona una interfaz de voz simple para realizar tu pedido, y él se encarga de coordinar internamente con la cocina, el sistema de pago y el servicio de entrega.

Estructura del Patrón Facade

- **Fachada (Facade):**

    - Proporciona una interfaz simplificada para una parte específica de la funcionalidad del subsistema.
    - Conoce las responsabilidades de las clases del subsistema y cómo orquestarlas.
    - Redirige las peticiones del cliente a los objetos apropiados del subsistema.
- **Subsistema Complejo (Complex Subsystem):**
    - Consiste en un grupo de objetos diversos y complejos que realizan el trabajo real.
    - Las clases del subsistema no tienen conocimiento de la Fachada; operan y se comunican entre sí directamente dentro del sistema.
    - Para que hagan algo significativo, el cliente debería entender sus detalles de implementación (inicialización, orden de llamadas, formatos de datos).
- **Cliente (Client):**
    - Utiliza la `Fachada` en lugar de interactuar directamente con los objetos individuales del `Subsistema Complejo`.
- **Fachada Adicional (Additional Facade - Opcional):**
    - Se puede crear para evitar que una única fachada se vuelva demasiado grande y compleja con funcionalidades no relacionadas.
    - Pueden ser usadas tanto por clientes como por otras fachadas.

Pseudocódigo de Ejemplo: Conversor de Video

Este ejemplo muestra cómo el patrón Facade simplifica la interacción con un framework complejo de conversión de video.

```
// Estas son algunas de las clases del framework de conversión de video de terceros.
// No controlamos este código, por lo que no podemos simplificarlo directamente.
class VideoFile { /* ... */ }
class OggCompressionCodec { /* ... */ }
class MPEG4CompressionCodec { /* ... */ }
class CodecFactory { /* ... */ }
class BitrateReader { /* ... */ }
class AudioMixer { /* ... */ }

// La clase Fachada para esconder la complejidad del framework
class VideoConverter is
    method convert(filename, format): File is
        // Lógica compleja para la conversión de video
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory()).extract(file)

        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()

        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// Las clases de la Aplicación usan la Fachada, no el framework directamente.
class Application is
    method main() is
        converter = new VideoConverter()
        mp4Video = converter.convert("funny-cats-video.ogg", "mp4")
        mp4Video.save()
```

```
// ... Ahora el código cliente es mucho más simple y desacoplado.
```

En este ejemplo, la clase `VideoConverter` es la Fachada. La aplicación cliente solo necesita interactuar con `VideoConverter.convert()`, sin preocuparse por la docena de clases internas del framework de video. Si el framework cambia, solo la fachada necesita ser actualizada.

Aplicabilidad
Utiliza el patrón Facade cuando:

- **Necesitas una interfaz limitada pero sencilla a un subsistema complejo.** Los subsistemas pueden volverse muy complejos con el tiempo, y Facade proporciona un atajo a las funciones más usadas que mejor se adaptan a las necesidades del cliente.
- **Quieres estructurar un subsistema en capas.** Puedes crear fachadas para definir puntos de entrada a cada nivel, reduciendo el acoplamiento entre subsistemas al obligarlos a comunicarse solo a través de sus fachadas. (Esto es similar al patrón Mediator, pero con un enfoque en la simplificación de interfaces).

Cómo Implementar

1. **Identifica la simplificación:** Determina si una interfaz más simple a un subsistema existente es posible y beneficiosa, es decir, si desacopla el código cliente de las clases internas del subsistema.
2. **Crea la clase Fachada:** Declara e implementa esta nueva interfaz en una clase fachada. La fachada redirigirá las llamadas del cliente a los objetos apropiados del subsistema.
3. **Gestiona el subsistema:** La fachada debe ser responsable de inicializar el subsistema y gestionar su ciclo de vida, a menos que el cliente ya se encargue de esto.
4. **Aislar el cliente:** Asegúrate de que todo el código cliente se comunique con el subsistema **únicamente a través de la fachada**. Esto protege el código cliente de cambios en el subsistema.
5. **Refinar la Fachada (Opcional):** Si una fachada se vuelve demasiado grande o abarca demasiadas responsabilidades, considera extraer parte de su comportamiento en nuevas fachadas más pequeñas y especializadas.

Pros y Contras
Ventajas (Pros):

- **Aislamiento de la complejidad:** Puedes aislar tu código de la complejidad de un subsistema grande y complicado.
- **Desacoplamiento:** Reduce el acoplamiento entre el código cliente y el subsistema.

Desventajas (Contras):

- **"Dios-objeto" (God Object):** Una fachada puede volverse un "objeto todopoderoso" si se acopla a demasiadas clases de una aplicación, lo que va en contra del principio de responsabilidad única.

Relaciones con Otros Patrones

- **Adapter vs. Facade:**
  - **Adapter** proporciona una interfaz **diferente** para acceder a un objeto existente con una interfaz incompatible. Usualmente envuelve un solo objeto.
  - **Facade** define una **nueva interfaz simplificada** para un subsistema completo de objetos.
- **Abstract Factory vs. Facade:**

    - `Abstract Factory` es una alternativa a `Facade` si solo quieres ocultar cómo se crean los objetos del subsistema al código cliente.
- **Flyweight vs. Facade:**
    - `Flyweight` se enfoca en crear muchos objetos pequeños para ahorrar memoria.
    - `Facade` se enfoca en crear un único objeto que represente un subsistema completo.
- **Mediator vs. Facade:**
    - Ambos organizan la colaboración entre muchas clases.
    - `Facade` define una interfaz simplificada a un subsistema, pero el subsistema no conoce la fachada y sus objetos pueden comunicarse directamente.
    - `Mediator` centraliza la comunicación entre componentes, y los componentes solo conocen al mediador, no se comunican directamente entre sí.
- **Singleton vs. Facade:**
    - Una clase `Fachada` a menudo puede ser un `Singleton`, ya que una única instancia suele ser suficiente para una fachada.
- **Proxy vs. Facade:**
    - `Facade` es similar a `Proxy` en que ambos pueden almacenar temporalmente una entidad compleja e inicializarla. Sin embargo, a diferencia de `Facade`, `Proxy` tiene la **misma interfaz** que su objeto de servicio, haciéndolos intercambiables.

El patrón Facade es extremadamente útil para mejorar la claridad del código, reducir la complejidad y aumentar la mantenibilidad, especialmente cuando se trabaja con APIs externas o subsistemas grandes.

Sistemas Distribuidos: Conceptos Fundamentales y Desafíos de Diseño

Prácticamente todos los grandes sistemas informáticos modernos son sistemas **distribuidos**. A diferencia de los sistemas centralizados que ejecutan todos sus componentes en una única computadora, un sistema distribuido **involucra numerosas computadoras independientes** que, para el usuario, **aparecen como un único sistema coherente** (Tanenbaum y Van Steen, 2007).
Si bien la ingeniería de sistemas distribuidos comparte muchos principios con la ingeniería de software en general, presentan desafíos específicos debido a que sus componentes se ejecutan en máquinas con administración independiente y se comunican a través de una red.

Ventajas de los Sistemas Distribuidos

Coulouris y colaboradores (2005) identifican varias ventajas clave de adoptar un enfoque distribuido para el desarrollo de sistemas:

1. **Compartición de Recursos:** Permiten compartir recursos de hardware y software (discos, impresoras, archivos, compiladores) asociados a computadoras en red, optimizando su uso.
2. **Apertura:** Suelen ser sistemas abiertos, diseñados en torno a **protocolos estándar** que posibilitan la combinación de equipos y software de diferentes proveedores, promoviendo la interoperabilidad.
3. **Concurrencia:** Procesos grandes pueden ejecutarse simultáneamente en computadoras conectadas en red. Estos procesos pueden comunicarse entre sí, pero no es estrictamente necesario.

4. **Escalabilidad:** Teóricamente, son escalables, lo que significa que sus capacidades pueden aumentarse agregando nuevos recursos para satisfacer demandas crecientes. En la práctica, la red puede ser un factor limitante.
5. **Tolerancia a Fallas:** La disponibilidad de múltiples computadoras y la posibilidad de replicar información permiten que estos sistemas toleren fallas de hardware y software. Aunque puede haber un servicio degradado, la pérdida completa de servicio solo ocurre con una falla de red generalizada.

Para sistemas organizacionales a gran escala, estas ventajas han llevado a que los sistemas distribuidos reemplacen a muchos sistemas mainframe heredados. Sin embargo, muchas aplicaciones de computadora personal (como la edición de fotografía) y la mayoría de los sistemas embebidos no son distribuidos.

Desafíos de Diseño en Sistemas Distribuidos

Los sistemas distribuidos son inherentemente más complejos que los sistemas centralizados, lo que los hace más difíciles de diseñar, implementar y probar. La complejidad de las interacciones entre componentes y la infraestructura del sistema dificulta la comprensión de sus propiedades emergentes. Por ejemplo, el rendimiento depende de factores como el ancho de banda y la carga de la red, y la velocidad de todas las computadoras involucradas, lo que genera una **respuesta impredecible** que puede variar drásticamente.

Los conflictos de diseño más importantes a considerar en la ingeniería de sistemas distribuidos incluyen:

1. **Transparencia:** ¿En qué medida el sistema distribuido debe parecer un solo sistema para el usuario? ¿Cuándo es útil que los usuarios entiendan que es distribuido? Lograr una transparencia total es casi imposible debido a la falta de control centralizado y los inevitables retrasos de red. A menudo, es mejor exponer la distribución para que los usuarios puedan anticipar y prepararse para las consecuencias (retrasos, fallas). El **middleware** (software intermedio) es clave para crear abstracciones de recursos y gestionar interacciones.
2. **Apertura:** ¿Debe diseñarse el sistema usando protocolos estándar para soportar la interoperabilidad, o protocolos más especializados que restrinjan la libertad del diseñador? Los sistemas abiertos se construyen con estándares generalmente aceptados (como los protocolos de Internet), permitiendo la integración de componentes de cualquier proveedor. Estándares como CORBA (aunque no logró una adopción masiva) y, más recientemente, los servicios web (como RESTful) buscan esta apertura.
3. **Escalabilidad:** ¿Cómo puede construirse el sistema para que su capacidad pueda aumentar en respuesta a demandas crecientes? Neuman (1994) identifica tres dimensiones:
    - **Tamaño:** Posibilidad de agregar más recursos (usuarios). Distingue entre **expandir** (sustituir recursos por más poderosos) y **ampliar** (agregar recursos adicionales), siendo esta última a menudo más costo-efectiva pero requiriendo procesamiento concurrente.
    - **Distribución:** Posibilidad de dispersar geográficamente los componentes sin reducir el rendimiento.
    - **Manejabilidad:** Posibilidad de administrar el sistema a medida que crece, incluso si las partes se ubican en organizaciones independientes.
4. **Seguridad:** ¿Cómo se definen e implementan políticas de seguridad útiles que se apliquen a través de un conjunto de sistemas administrados de manera independiente? Un sistema distribuido tiene muchas más vulnerabilidades que uno centralizado (intercepción, interrupción -incluidos ataques de denegación de servicio-, modificación, fabricación). La dificultad radica en aplicar una política de seguridad coherente cuando diferentes organizaciones poseen partes del sistema con políticas incompatibles.

5. **Calidad de Servicio (QoS):** ¿Cómo se especifica e implementa el sistema para entregar una calidad de servicio aceptable (confiabilidad, tiempo de respuesta, rendimiento total)? A menudo es costoso diseñar para cargas pico, y los parámetros de QoS pueden ser contradictorios (ej., mayor fiabilidad puede reducir el rendimiento). La computación en la nube ofrece una solución parcial a esto. Para datos críticos en el tiempo (audio/video), si la QoS cae por debajo de un umbral, el servicio puede degradarse a niveles inaceptables, requiriendo mecanismos de negociación y gestión de QoS.
6. **Gestión de Fallas:** ¿Cómo se detectan, contienen (para minimizar efectos) y reparan las fallas del sistema? Las fallas son inevitables en sistemas distribuidos. Se deben aplicar técnicas de tolerancia a fallas para que el sistema siga entregando la mayor cantidad posible de servicios a pesar de una falla y, si es posible, se recupere automáticamente.

Modelos de Interacción

Existen dos tipos fundamentales de interacción entre computadoras en un sistema distribuido:

1. **Interacción Procedimental:** Implica que una computadora solicita un servicio conocido ofrecido por otra y generalmente espera una respuesta. Es una serie de interacciones sincrónicas. Esto se implementa comúnmente con **Llamadas a Procedimientos Remotos (RPC)** o, en Java, **Invocaciones de Métodos Remotos (RMI)**. En RPC, un componente solicita otro como si fuera un procedimiento local; el middleware intercepta la solicitud, la transmite al componente remoto, este realiza la computación y devuelve el resultado. Requiere que ambos (solicitante y solicitado) estén disponibles y se conozcan mutuamente en el momento de la comunicación.
    - **Ejemplo:** Un comensal ordenando paso a paso a un camarero (pregunta-respuesta sincrónica).
2. **Interacción Basada en Mensajes:** Implica que una computadora "emisora" define la información requerida en un mensaje y lo envía a otra computadora. Los mensajes suelen transmitir más información en una sola interacción que una solicitud de procedimiento. A diferencia de RPC, puede tolerar la indisponibilidad (el mensaje permanece en una cola hasta que el receptor esté disponible) y los componentes no necesitan conocerse directamente, solo interactuar con el middleware.
    - **Ejemplo:** El camarero enviando un único mensaje (pedido completo) a la cocina.

Middleware en Sistemas Distribuidos

Los componentes en un sistema distribuido pueden estar implementados en diferentes lenguajes de programación y ejecutarse en distintos tipos de procesadores, con modelos de datos y protocolos de comunicación diversos. El **middleware** es el software que se sitúa en el centro, entre los componentes distribuidos y el sistema operativo, para gestionar estas diversidades y asegurar la comunicación e intercambio de datos.

El middleware, generalmente un conjunto de librerías y un sistema de tiempo de ejecución, proporciona dos tipos de soporte:

1. **Soporte de Interacción:** Coordina las interacciones entre componentes, ofreciendo **transparencia de ubicación** (los componentes no necesitan saber la ubicación física de otros), conversión de parámetros (para diferentes lenguajes), detección de eventos, y comunicación (RPC/RMI, intercambio de mensajes).
2. **Provisión de Servicios Comunes:** Ofrece implementaciones reutilizables de servicios que varios componentes pueden requerir, independientemente de su funcionalidad específica. Estos pueden incluir servicios de seguridad (autenticación, autorización), notificación, nomenclatura y gestión de transacciones. A menudo,

estos servicios son proporcionados por un "contenedor middleware" donde se implementan los componentes de la aplicación.

Computación Cliente-Servidor

Los sistemas distribuidos accedidos por Internet se organizan típicamente como sistemas **cliente-servidor**. En este modelo:

- El **cliente** es un programa que se ejecuta en la computadora local del usuario (ej., navegador web, aplicación móvil) y el usuario interactúa con él.
- El **servidor** es un programa que se ejecuta en una computadora remota y proporciona servicios (ej., acceso a páginas web, datos).

Aunque comúnmente se asocia con sistemas distribuidos, el modelo cliente-servidor es un **modelo arquitectónico general** que también puede aplicarse a interacciones lógicas dentro de una sola máquina.

En una arquitectura cliente-servidor, una aplicación se modela como un conjunto de servicios proporcionados por servidores, a los que los clientes acceden. Los clientes conocen los servidores disponibles, pero no a otros clientes. Clientes y servidores son procesos separados. Es normal que muchos procesos cliente se ejecuten en un solo procesador (ej., un PC con cliente de correo, navegador web, cliente de impresión). Los procesos servidor pueden ejecutarse en el mismo procesador o en sistemas multiprocesador con balanceo de carga para distribuir las peticiones.

Los sistemas cliente-servidor requieren una **clara separación entre la presentación de información y los cálculos**. Por ello, se diseñan en **capas lógicas** con interfaces claras, lo que permite que cada capa se distribuya en diferentes computadoras:

- **Capa de Presentación:** Muestra información al usuario y gestiona interacciones.
- **Capa de Gestión de Datos:** Gestiona datos que fluyen hacia y desde el cliente (validaciones, generación de páginas web).
- **Capa de Procesamiento de Aplicación:** Implementa la lógica de negocio y funcionalidad principal.
- **Capa de Base de Datos:** Almacena datos y ofrece servicios de gestión de transacciones.

Este modelo también subyace al concepto de **Software como Servicio (SaaS)**.

Patrones Arquitectónicos para Sistemas Distribuidos

Al diseñar una aplicación distribuida, se debe elegir un estilo arquitectónico que soporte los **requisitos no funcionales críticos** (rendimiento, confiabilidad, seguridad, manejabilidad).

No hay un modelo universalmente adecuado. Algunos estilos comunes son:

1. **Arquitectura Maestro-Esclavo:** Usada en sistemas de tiempo real donde se requieren tiempos de respuesta garantizados. Hay un proceso **maestro** que controla y coordina procesos **esclavos** que realizan las computaciones.
2. **Arquitectura Cliente-Servidor de Dos Niveles:** Sencilla, donde un cliente se comunica directamente con un servidor. Útil para sistemas simples o donde la centralización por seguridad es clave (la comunicación suele estar encriptada).
3. **Arquitectura Cliente-Servidor Multinivel (N-Tier):** Extiende el modelo de dos niveles con capas adicionales (presentación, lógica de negocio, datos, etc.). Se usa cuando hay un gran volumen de transacciones a procesar por el servidor, distribuyendo la carga de trabajo entre las capas.
4. **Arquitectura de Componentes Distribuidos:** Permite combinar recursos de diferentes sistemas y bases de datos. Puede ser un modelo de implementación para sistemas cliente-servidor multinivel.
5. **Arquitectura Peer-to-Peer (P2P):** Los clientes intercambian información almacenada localmente, y el servidor puede limitarse a presentar a los clientes entre sí. También es útil para realizar un gran número de cálculos independientes (ej., computación distribuida).

En resumen, los sistemas distribuidos ofrecen enormes ventajas en términos de recursos, apertura, concurrencia, escalabilidad y tolerancia a fallas. Sin embargo, su naturaleza distribuida introduce una complejidad inherente y desafíos significativos en el diseño, implementación y prueba, que los ingenieros de software deben abordar diligentemente.

Protocolo HTTP: Generalidades

El **HTTP (Hypertext Transfer Protocol)** es el protocolo fundamental que permite el intercambio de datos y recursos, como documentos HTML, en la **Web**. Funciona bajo una **arquitectura cliente-servidor**, donde un **cliente** (usualmente un navegador web) inicia la petición de datos, y un **servidor** los proporciona. Una página web completa se ensambla a partir de múltiples "sub-documentos" (CSS, imágenes, scripts, videos, texto, etc.) recibidos a través de este protocolo.

Comunicación y Mensajes

La comunicación en HTTP se basa en el **intercambio de mensajes individuales**. Los mensajes enviados por el **cliente** se denominan **peticiones**, y los mensajes enviados por el **servidor** se conocen como **respuestas**. Este diseño, concebido a principios de los años 90, es **ampliable** y ha evolucionado con el tiempo.
HTTP opera en la **capa de aplicación** y se transmite principalmente sobre **TCP (Transmission Control Protocol)** o su versión encriptada, **TLS (Transport Layer Security)**, aunque teóricamente podría usar cualquier protocolo de transporte fiable. Su extensibilidad le permite no solo transmitir documentos de hipertexto (HTML), sino también imágenes, videos, y enviar datos a servidores (como en formularios), e incluso actualizar partes de documentos en tiempo real.

Arquitectura de Sistemas Basados en HTTP

La arquitectura de HTTP se fundamenta en el modelo **cliente-servidor**.

- **Cliente (Agente del Usuario):** Es la entidad que inicia la petición. En la mayoría de los casos, es un **navegador web**, pero podría ser un robot de exploración (crawler) o una herramienta de desarrollo. El navegador siempre inicia la comunicación, solicitando un documento HTML, luego procesando este documento y enviando más peticiones para obtener los recursos adicionales (scripts, CSS, imágenes, etc.) necesarios para componer y presentar la página web final al usuario. También ejecuta scripts que pueden generar peticiones adicionales.
- **Servidor Web:** Al otro lado del canal de comunicación, el servidor "sirve" los datos solicitados por el cliente. Un servidor es conceptualmente una única entidad, pero puede estar compuesto por múltiples elementos físicos o lógicos (balanceadores de carga, servidores de caché, bases de datos, etc.) que colaboran para generar la respuesta. Varios servidores lógicos pueden operar en un único equipo físico, e incluso compartir la misma dirección IP (gracias al estándar HTTP/1.1 y la cabecera `Host`).
- **Proxies:** Son intermediarios entre el cliente y el servidor que gestionan los mensajes HTTP. Muchos dispositivos intermedios (routers, módems) operan en capas de protocolo inferiores (transporte, red, física) y son transparentes para HTTP. Los **proxies**, sin embargo, operan y procesan la capa de aplicación HTTP. Pueden ser transparentes o no (modificando las peticiones) y realizan funciones como:
    - **Caching:** Almacenan copias de recursos para acelerar futuras peticiones (la caché puede ser pública o privada, como la del navegador).
    - **Filtrado:** Implementan políticas de seguridad (antivirus, control parental).
    - **Balanceo de Carga:** Distribuyen peticiones entre múltiples servidores para manejar una mayor carga.
    - **Autenticación:** Controlan el acceso a recursos protegidos.
    - **Registro de Eventos:** Mantienen un historial de las interacciones.

Características Clave del Protocolo HTTP

- **Sencillo:** A pesar de la complejidad añadida en HTTP/2 (que encapsula mensajes), HTTP está diseñado para ser **legible y fácilmente interpretable por humanos** (especialmente en HTTP/1.1 y versiones anteriores). Esto facilita la depuración y reduce la curva de aprendizaje.
- **Extensible:** Introducidas en HTTP/1.0, las **cabeceras HTTP** han permitido que el protocolo sea altamente extensible. Nuevas funcionalidades pueden desarrollarse si el cliente y el servidor entienden la semántica de cabeceras específicas.
- **Sin Estado con Sesiones:** HTTP es un **protocolo sin estado**, lo que significa que no guarda información sobre peticiones anteriores en la misma sesión. Esto presenta desafíos para interacciones coherentes (como un carrito de compras). Sin embargo, el uso de **HTTP cookies**, a través de la extensibilidad del protocolo, permite mantener un contexto común para cada sesión de comunicación, simulando un estado.
- **HTTP y Conexiones:** La gestión de conexiones ocurre a nivel de la capa de transporte (normalmente **TCP**, que es fiable). HTTP no necesita una conexión continua, solo un protocolo subyacente fiable que detecte mensajes perdidos.
    - En HTTP/1.0, cada petición/respuesta implicaba abrir y cerrar una conexión TCP, lo que era lento.
    - HTTP/1.1 introdujo el **'pipelining'** (enviar varias peticiones sin esperar la primera respuesta) y **conexiones persistentes** (reutilizar la misma conexión TCP).
    - HTTP/2 avanzó con la **multiplexación de mensajes** sobre una única conexión, mejorando significativamente la eficiencia.
    - Actualmente, se investigan nuevos protocolos de transporte, como **QUIC** (de Google), que se basa en UDP para mayor fiabilidad y eficiencia.

Funcionalidades Controlables con HTTP

La extensibilidad de HTTP ha permitido la implementación de diversas funciones de control y seguridad en la Web:

- **Caché:** HTTP permite especificar cómo los documentos deben almacenarse en caché, indicando al servidor y proxies qué almacenar y por cuánto tiempo. Los clientes también pueden instruir a los proxies para ignorar la caché.
- **Flexibilidad del Requisito de Origen (CORS):** Para prevenir invasiones de privacidad, los navegadores originalmente solo permitían que páginas del mismo origen compartieran datos. HTTP, a través de cabeceras específicas, permite relajar esta restricción, facilitando la comunicación entre orígenes diferentes.
- **Autenticación:** HTTP provee servicios básicos de autenticación (ej., mediante cabeceras `WWW-Authenticate` o sesiones basadas en cookies) para proteger el acceso a recursos.
- **Proxies y Tunneling:** Las peticiones HTTP utilizan proxies para acceder a servidores o clientes en intranets o con IPs ocultas. Algunos proxies (como SOCKS) operan a niveles inferiores, permitiendo servir otros protocolos (ej., FTP) a través de ellos.
- **Sesiones:** A pesar de ser un protocolo sin estado, el uso de **HTTP cookies** permite relacionar peticiones con el estado del servidor, definiendo así las sesiones. Esto es crucial para aplicaciones de comercio electrónico y sitios con configuraciones de usuario personalizables.

Flujo de Comunicación HTTP

Cuando un cliente desea comunicarse con un servidor (directamente o a través de un proxy), sigue estos pasos:

1. **Abre una conexión TCP:** Esta conexión se utilizará para una o varias peticiones y para recibir las respuestas. El cliente puede abrir una nueva conexión, reutilizar una existente o abrir varias simultáneamente.

2. **Hace una petición HTTP:**
    - En HTTP/1.1 y anteriores, los mensajes son de texto plano y legibles.
    - En HTTP/2, los mensajes se encapsulan en tramas binarias, no directamente interpretables, pero el principio operativo es el mismo.
    - Una petición consta de:
        - Un **método HTTP** (verbo como `GET`, `POST`, o nombre como `OPTIONS`, `HEAD`) que define la operación.
        - La **dirección del recurso** (URL relativa).
        - La **versión del protocolo HTTP**.
        - **Cabeceras HTTP** opcionales (información adicional para el servidor).
        - Un **cuerpo del mensaje** opcional (para métodos como `POST`).
3. **Lee la respuesta del servidor:**
    - Una respuesta consta de:
        - La **versión del protocolo HTTP**.
        - Un **código de estado** (indica éxito o tipo de error).
        - Un **mensaje de estado** (breve descripción del código).
        - **Cabeceras HTTP** (similares a las de petición).
        - Opcionalmente, el **recurso solicitado**.
4. **Cierre o reuso de la conexión:** La conexión se cierra o se mantiene para futuras peticiones.

El **HTTP pipelining** (enviar varias peticiones sin esperar respuestas) ha sido difícil de implementar de forma robusta y ha sido reemplazado por la **multiplexación** en HTTP/2, que permite enviar múltiples peticiones y respuestas concurrentemente sobre una única conexión.

Mensajes HTTP: Evolución

- **HTTP/1.1 y anteriores:** Los mensajes son de **texto plano** y directamente legibles por humanos.
- **HTTP/2:** Los mensajes se estructuran en un **formato binario** usando "tramas", lo que permite la compresión de cabeceras y la multiplexación. Aunque el formato de transmisión cambia, la **semántica** de los mensajes sigue siendo la misma que en HTTP/1.1, lo que permite interpretarlos de forma equivalente.

Conclusión

HTTP es un protocolo **ampliable y fácil de usar**. Su diseño **cliente-servidor** y la flexibilidad de las **cabeceras** le han permitido evolucionar para satisfacer las demandas de las aplicaciones modernas de Internet. Aunque HTTP/2 introduce complejidad con su formato binario, mejora el rendimiento, manteniendo la estructura y semántica de los mensajes. El flujo de comunicaciones sigue siendo sencillo y puede ser fácilmente monitoreado para estudio y depuración.

Mensajes HTTP: El Corazón de la Comunicación Web

Los **mensajes HTTP** son el medio fundamental para el **intercambio de datos entre clientes y servidores** en la web. Existen dos tipos principales:

1. **Peticiones:** Enviadas por el **cliente** (generalmente un navegador web) al servidor para iniciar una acción.
2. **Respuestas:** Enviadas por el **servidor** al cliente en contestación a una petición.

Tradicionalmente, en **HTTP/1.1** y versiones anteriores, estos mensajes eran **texto plano codificado en ASCII** y se transmitían de forma abierta. Sin embargo, **HTTP/2.0** introdujo un cambio significativo: los mensajes se conforman mediante **tramas binarias codificadas**. Este cambio, diseñado para optimizar el rendimiento de la transmisión, es **transparente para los desarrolladores** y no requiere modificaciones en las APIs o archivos de configuración existentes.

Los desarrolladores rara vez codifican estos mensajes HTTP directamente; en su lugar, los especifican a través de archivos de configuración (para proxies y servidores) o APIs (para navegadores).

Estructura Común de los Mensajes HTTP

Tanto las peticiones como las respuestas HTTP comparten una estructura similar, compuesta por cuatro partes principales:

1. **Línea de Inicio (`start-line`):** Una única línea que describe la petición a realizar o el estado de la respuesta (éxito o fracaso).
2. **Cabeceras HTTP (`HTTP headers`):** Un grupo opcional de líneas que proporcionan metainformación sobre la petición o describen el cuerpo del mensaje. Son pares `Nombre-Cabecera: Valor`.
3. **Línea Vacía (`empty-line`):** Una línea en blanco que indica el final de las cabeceras y el inicio del cuerpo (si lo hay).
4. **Cuerpo del Mensaje (`body`):** Un campo opcional que contiene los datos asociados con la petición (como el contenido de un formulario HTML) o los datos asociados a una respuesta (como una página HTML, un archivo de audio o video). La presencia y el tamaño del cuerpo se indican en la línea de inicio y/o las cabeceras.

La **línea de inicio** y las **cabeceras HTTP** se conocen colectivamente como la **cabeza** del mensaje, mientras que el contenido de datos es el **cuerpo** del mensaje.

Peticiones HTTP

Las peticiones HTTP son los mensajes que un cliente envía a un servidor para solicitar una acción.

Línea de Inicio de una Petición

La línea de inicio de una petición HTTP consta de tres elementos:

1. **Método HTTP:** Un verbo (ej., `GET`, `PUT`, `POST`) o un nombre (ej., `HEAD`, `OPTIONS`) que describe la acción que se solicita.
    - **`GET`**: Solicita que un archivo sea enviado al cliente.
    - **`POST`**: Indica que se enviarán datos al servidor (para crear o modificar un recurso, o generar un documento temporal).
2. **Objetivo de la Petición:** Generalmente una **URL**, aunque su formato varía según el método HTTP:
    - **Formato de Origen (`origin form`):** La dirección absoluta seguida de `?` y una cadena de consulta. Es el formato más común para `GET`, `POST`, `HEAD` y `OPTIONS`.
        - `GET /background.png HTTP/1.0`
        - `POST / HTTP/1.1`
    - **Formato Absoluto (`absolute form`):** Una URL completa. Mayormente usado con `GET` cuando se conecta a un proxy.
        - `GET http://developer.mozilla.org/es/docs/Web/HTTP/Messages HTTP/1.1`
    - **Formato de Autoridad (`authority form`):** El componente de autoridad de una URL (dominio y opcionalmente puerto). Usado únicamente con `CONNECT` para establecer un túnel HTTP.
        - `CONNECT developer.mozilla.org:80 HTTP/1.1`
    - **Formato de Asterisco (`asterisk form`):** Un asterisco (`*`) usado con `OPTIONS`, representando al servidor completo.
        - `OPTIONS * HTTP/1.1`

3. **Versión de HTTP:** Define la estructura de los mensajes y actúa como un indicador de la versión esperada para la respuesta.

Cabeceras de Petición

Las cabeceras HTTP de una petición siguen el formato `Nombre-Cabecera: Valor`, sin distinción entre mayúsculas y minúsculas en el nombre. Cada cabecera debe estar en una única línea. Se clasifican en varios grupos:

- **Cabeceras Generales:** Afectan al mensaje como una unidad completa (ej., `Via`).
- **Cabeceras de Petición:** Modifican la petición, añadiendo detalles (ej., `Accept-Language`), contexto (ej., `Referer`), o restricciones condicionales (ej., `If-None-Match`).
- **Cabeceras de Entidad:** Se aplican al **cuerpo de la petición**. Si no hay cuerpo, no se transmiten (ej., `Content-Length`, `Content-Type`).

Cuerpo de la Petición

El cuerpo es la parte final de la petición y no todas las peticiones lo llevan. Métodos como `GET`, `HEAD`, `DELETE` u `OPTIONS` no suelen necesitarlo. Sin embargo, métodos como `POST` lo utilizan para enviar datos al servidor (por ejemplo, datos de un formulario HTML).

Los cuerpos de petición pueden ser:

- **Cuerpos de un único dato:** Un solo archivo definido por las cabeceras `Content-Type` y `Content-Length`.
- **Cuerpos con múltiples datos:** Compuestos por distintos contenidos, comúnmente asociados con formularios HTML que incluyen subidas de archivos.

Respuestas HTTP

Las respuestas HTTP son los mensajes que un servidor envía al cliente en respuesta a una petición.

Línea de Estado de una Respuesta

La línea de inicio de una respuesta HTTP se denomina **línea de estado** y contiene:

1. **Versión del Protocolo:** Normalmente `HTTP/1.1`.
2. **Código de Estado:** Un número de tres dígitos que indica el éxito o fracaso de la petición (ej., `200` para éxito, `404` para "No Encontrado", `302` para "Redirección").
3. **Texto de Estado:** Una breve descripción informativa del código de estado, legible para personas.

Ejemplo de línea de estado: `HTTP/1.1 404 Not Found`.

Cabeceras de Respuesta

Las cabeceras HTTP para respuestas tienen la misma estructura que las de petición: `Nombre-Cabecera: Valor`.

Se clasifican en:

- **Cabeceras Generales:** Afectan al mensaje completo (ej., `Via`).
- **Cabeceras de Respuesta:** Proporcionan información adicional sobre el servidor que no cabe en la línea de estado (ej., `Vary`, `Accept-Ranges`).
- **Cabeceras de Entidad:** Se aplican al **cuerpo de la respuesta**. Si no hay cuerpo, no se transmiten (ej., `Content-Length`, `Content-Type`).

Cuerpo de la Respuesta

La última parte del mensaje de respuesta es el cuerpo. Al igual que en las peticiones, no todas las respuestas lo tienen (ej., respuestas con código de estado `201` - Created o `204` - No Content).

Los cuerpos de respuesta pueden ser:

- **Cuerpos con un único dato (longitud conocida):** Un archivo simple cuya longitud y tipo están definidos por `Content-Type` y `Content-Length`.
- **Cuerpos con un único dato (longitud desconocida):** Un archivo simple codificado en partes (`chunked`), indicado por la cabecera `Transfer-Encoding: chunked`.

- **Cuerpos con múltiples datos:** Consisten en varias secciones de información, cada una con su propio contenido. Este caso es relativamente raro.

Conclusión

Los mensajes HTTP son esenciales para la comunicación en la web. Su estructura clara y su naturaleza extensible, impulsada por las cabeceras, han permitido que el protocolo HTTP evolucione y se adapte a las crecientes demandas de Internet. Aunque HTTP/2 introduce la complejidad de un formato binario y un mecanismo de tramas para mejorar el rendimiento, lo hace construyéndose sobre los fundamentos probados de HTTP/1.x, sin alterar radicalmente la semántica de los mensajes. Este flujo de comunicación sencillo y estandarizado facilita el estudio y la depuración de las interacciones web.

Transferencia de Estado Representacional (REST): Un Estilo de Arquitectura para la Web

La **Transferencia de Estado Representacional (REST)** es un **estilo de arquitectura de software** diseñado para **sistemas hipermedia distribuidos**, como la World Wide Web. El término fue acuñado en el año 2000 por **Roy Fielding**, uno de los principales arquitectos del protocolo HTTP, en su tesis doctoral sobre la web. Desde entonces, ha sido ampliamente adoptado por la comunidad de desarrollo.

Historia de REST

A mediados de los años 90, con la creciente adopción de la Web, surgió la necesidad de una descripción formal de su arquitectura. Se estaban añadiendo extensiones experimentales a HTTP, y era crucial contar con un marco arquitectónico para evaluar su impacto.

Roy Fielding, involucrado en la creación de los estándares iniciales de la Web (URI, HTTP 1.0 y 1.1), dedicó seis años a desarrollar el estilo arquitectónico REST. Probó sus principios con los estándares existentes, utilizándolos para definir mejoras y detectar inconsistencias. Su disertación doctoral de 2000, "Estilos arquitectónicos y el diseño de arquitecturas de software basadas en redes", formalizó la definición de REST.

Descripción de REST

Originalmente, **REST** se refería a un conjunto de principios arquitectónicos. Sin embargo, en la actualidad, el término se usa más ampliamente para describir cualquier interfaz entre sistemas que **utiliza directamente HTTP para obtener o manipular datos**, en cualquier formato (como **XML, JSON**), sin las abstracciones adicionales de protocolos basados en patrones de intercambio de mensajes, como **SOAP**. Es fundamental entender que una interfaz XML/HTTP que utilice un estilo de **Llamada a Procedimiento Remoto (RPC)**, aunque no use SOAP, no es un ejemplo de REST. Esta distinción es clave para evitar confusiones técnicas.

REST sostiene que la escalabilidad de la Web se debe a varios diseños fundamentales:

- **Protocolo Cliente/Servidor sin Estado:** Cada mensaje HTTP contiene toda la información necesaria para que el servidor entienda la petición, sin que ni el cliente ni el servidor necesiten recordar estados de comunicaciones anteriores. Aunque en la práctica se usan **cookies** y otros mecanismos para mantener el estado de sesión, REST estricto no permite ciertas prácticas como la reescritura de URLs para este fin.
- **Operaciones Bien Definidas:** HTTP en sí define un pequeño conjunto de operaciones aplicables a todos los recursos, siendo las más importantes `POST`, `GET`, `PUT` y `DELETE`. Estas operaciones a menudo se equiparan a las operaciones **CRUD** (Crear, Leer, Actualizar, Borrar) en bases de datos.
- **Sintaxis Universal para Identificar Recursos:** En un sistema REST, cada recurso es única y directamente direccionable a través de su **URI (Uniform Resource Identifier)**.

- **Uso de Hipermedios:** Tanto para la información de la aplicación como para las transiciones de estado. La representación de este estado suele ser **HTML** o **XML**. Esto permite navegar de un recurso REST a otros simplemente siguiendo enlaces, sin necesidad de registros o infraestructura adicional.

Recursos en REST

Un concepto central en REST es la existencia de **recursos** (elementos de información), accesibles mediante un identificador global (URI). Los componentes de la red (clientes y servidores) se comunican a través de una interfaz estándar (HTTP) e intercambian **representaciones** de estos recursos (los archivos que se descargan y envían).
Las peticiones pueden ser transmitidas a través de varios conectores (clientes, servidores, cachés, túneles, etc.), pero cada uno lo hace sin "ver más allá" de su propia petición. Esta característica de ser **"sin estado" (`stateless`)** es una restricción clave de REST. Así, una aplicación puede interactuar con un recurso conociendo su identificador y la acción requerida, sin necesidad de saber sobre cachés, proxies, cortafuegos o túneles intermedios. Sin embargo, la aplicación debe comprender el formato de la información devuelta (la representación), que usualmente es HTML o XML, pero puede ser cualquier contenido.

REST vs. RPC

La principal diferencia entre REST y **RPC (Llamada a Procedimiento Remoto)** radica en su enfoque:

- **RPC:** El énfasis está en la **diversidad de operaciones o "verbos"** del protocolo.
  - Ejemplos de operaciones RPC: `getUser()`, `addUser()`, `updateLocation()`, `listUsers()`.
- **REST:** El énfasis está en los **recursos o "sustantivos"** y los nombres asignados a cada tipo de recurso.
  - Ejemplos de recursos REST: `Usuario {}`, `Localización {}`.
  - Cada recurso tiene su propio identificador (ej., `http://www.example.org/locations/us/ny/new_york_city`). Los clientes interactúan con estos recursos usando las **operaciones estándar de HTTP** (`GET`, `POST`, `PUT`, `DELETE`).
  - `GET` se usa para descargar una copia de un recurso.
  - `PUT` se usa para actualizar un recurso (por ejemplo, descargar un XML, modificarlo y subirlo de nuevo).
  - `POST` se utiliza generalmente para acciones con efectos secundarios, como enviar una orden de compra o añadir datos a una colección.

Una distinción clave es que los verbos HTTP no proporcionan mecanismos estándar para "descubrir" recursos (no hay un `LIST` o `FIND` directo como en RPC). En REST, las colecciones de resultados de búsqueda se tratan como **otro tipo de recurso**, lo que significa que los diseñadores deben definir URLs adicionales para buscar o mostrar recursos. Por ejemplo:

- `GET http://www.example.org/locations/us/ny/` podría devolver una lista de enlaces a todas las localizaciones posibles en Nueva York.
- `GET http://www.example.org/users?surname=Michaels` podría devolver una lista de enlaces a todos los usuarios con el apellido "Michaels".

REST sugiere el uso de lenguajes de formularios (como HTML) para especificar consultas parametrizadas como parte de su restricción de "hipermedia como motor del estado de la aplicación".

Principios de REST

En su disertación, Roy T. Fielding define REST por un conjunto de principios de ingeniería de software que guían su diseño:

1. **Arquitectura Cliente-Servidor:** Enfatiza la **separación de responsabilidades** y la **portabilidad**. El servidor y el cliente son independientes, lo que permite un mayor desacoplamiento y facilita los cambios en los componentes. REST está diseñado para sistemas distribuidos centralizados, no P2P.
2. **Ausencia de Estado (`Stateless`):** El estado de la sesión se mantiene completamente en el cliente, no en el servidor. Cada petición del cliente debe contener toda la información necesaria para ser procesada. Esto mejora la escalabilidad del servidor. Sin embargo, el servidor puede facilitar transiciones de estado a través de redirecciones transparentes para el cliente.
3. **Habilitación y Uso de la Caché:** Todas las respuestas deben declarar si son cacheables (ej., usando la cabecera `Cache-Control` de HTTP). Esto permite que las respuestas se sirvan desde cachés intermedias, reduciendo la carga del servidor y mejorando el rendimiento.
4. **Sistema por Capas:** Un cliente solo necesita conocer la capa con la que interactúa directamente. No necesita preocuparse por detalles de implementación subyacentes como bases de datos, cachés, proxies o balanceadores de carga. La seguridad, por ejemplo, se puede añadir en una capa superior a los servicios web.
5. **Interfaz Uniforme:** Este es un principio clave y el más restrictivo, que impulsa la simplicidad y la visibilidad de las interacciones. Se compone de cuatro sub-principios:
    - **Identificación de Recursos en las Peticiones:** Las peticiones identifican recursos individuales mediante URIs. Los recursos son conceptualmente distintos de sus representaciones (HTML, XML, JSON). El formato se puede negociar mediante cabeceras HTTP.
    - **Manipulación de Recursos a Través de Representaciones:** Una vez que un cliente tiene una representación de un recurso (con metadatos), tiene suficiente información para modificar o eliminar su estado. Las APIs RESTful suelen ser "autodocumentadas" en este sentido.
    - **Mensajes Autodescriptivos:** Cada mensaje contiene toda la información necesaria para que el cliente lo entienda, sin requerir documentación externa o mensajes adicionales.
    - **Hipermedia como Motor del Estado de la Aplicación (HATEOAS):** Después de acceder a la URI inicial de la aplicación, el cliente REST debe ser capaz de descubrir todos los recursos disponibles y las transiciones de estado simplemente siguiendo los enlaces provistos dinámicamente en las respuestas del servidor. Esto elimina la necesidad de "hardcodear" la estructura o referencias de la aplicación en el cliente.

Modelo de Madurez de Richardson

Debido a la amplitud de las directrices REST, el **Modelo de Madurez de Richardson** describe los niveles por los que pasa una especificación de API REST, desde su creación hasta que incorpora controles de hipermedia:

- **Nivel 0:** Un único URI para todas las operaciones, con recursos poco definidos. No se considera una API RESTful.
- **Nivel 1:** Introduce recursos con URIs individuales para acciones separadas, en lugar de un punto de acceso universal. Los recursos son más concretos. Aún no es RESTful, pero avanza en esa dirección.
- **Nivel 2:** El sistema empieza a usar los **verbos HTTP** (`GET`, `POST`, `PUT`, `DELETE`) para las operaciones. Esto aumenta la especialización y la granularidad de los recursos.
- **Nivel 3:** El nivel más alto, introduce la **representación hipermedia (HATEOAS)**. Los mensajes de respuesta incluyen enlaces que permiten al cliente descubrir recursos relacionados y posibles transiciones de estado. Por ejemplo, una respuesta

de un sistema de reservas de hotel podría incluir el número de habitaciones disponibles y enlaces para reservar habitaciones específicas.

Diferencias con SOAP

Característica	REST	SOAP
Enfoque	Centrado en **datos** (recursos).	Orientado a **servicios** que operan con datos.
Naturaleza	Estilo arquitectónico (menos restrictivo).	Protocolo (más estandarizado y restrictivo).
Formato Mensajes	Generalmente **HTML, XML, JSON**. JSON es muy extendido.	Mayormente **XML**, fuertemente tipado, más verboso, mayor ancho de banda.
Pruebas	Se puede probar directamente con un **navegador** (si es RESTful).	Requiere **herramientas ad-hoc** (ej., SoapUI).
Uso de Librerías	Menos dependiente de librerías para lo básico.	Uso más extendido de librerías específicas.
Operación Frecuente	Principalmente `GET` (lectura).	Principalmente `POST`.
Complejidad Cliente	Generalmente más sencillo.	Puede conllevar el desarrollo de clientes más complejos.

Ventajas e Inconvenientes de REST

Ventajas:

- **Consenso y Adopción:** Gran parte de las APIs modernas (especialmente las públicas) se diseñan para ser RESTful, creando un consenso en la comunidad sobre su funcionamiento.
- **Sencillez de Implementación:** A menudo más sencillo de implementar en cliente y servidor, con muchos *frameworks* ofreciéndolo "out-of-the-box" (ej., Django).
- **Visibilidad:** Las URIs públicas facilitan el descubrimiento y la interacción.
- **Cacheabilidad:** Facilita el uso de caché, mejorando el rendimiento.

Inconvenientes:

- **Estandarización Abierta:** Aunque es un estilo, la falta de un estándar de implementación tan estricto como SOAP deja más decisiones abiertas al diseñador.
- **Seguridad y Visibilidad de URIs:** Las URIs públicas pueden suponer un problema de seguridad si no se gestionan bien.
- **Limitaciones de Longitud de Parámetros:** Las limitaciones en la longitud de los parámetros en `GET` pueden ser un problema para grandes cantidades de información (solucionado con `POST`).

- **Descubrimiento de Operaciones:** Requiere que el diseñador de la aplicación defina cómo se descubren recursos o se realizan búsquedas, ya que HTTP no tiene operaciones `LIST` o `FIND` directas.

Implementaciones Públicas Notables

Dado que la definición de REST es amplia, se puede decir que existe un enorme número de aplicaciones REST en la web. De forma más restrictiva, algunas implementaciones notables, aunque no siempre "totalmente RESTful" (es decir, no siempre respetan todas las restricciones, especialmente HATEOAS, siendo a menudo "Accidentalmente RESTful"), incluyen:

- La **blogosfera** (RSS/Atom).
- APIs de **Amazon.com**, **Bloglines**, **Yahoo!**, **Facebook**, **Twitter**, **MEGA**, **MercadoLibre**.
- Proyectos como "Seniors Canada On-line" del Gobierno de Canadá.
- Mecanismos de enrutamiento en *frameworks* como **Ruby on Rails** y **Catalyst** (basados en MVC).
- Implementaciones como **ADO.NET Data Services Framework** (Microsoft) y **RestLet** (Java).
- Sistemas de orquestación de la nube como **OpenNebula** (usando JSON y REST).

Estas implementaciones se inspiran en REST y respetan sus aspectos más significativos, en particular la **restricción de "interfaz uniforme"**.

Clean Architecture: Agregando lo Mejor de Otras Arquitecturas

La **Arquitectura Limpia (Clean Architecture)** es un esfuerzo por combinar las características más destacadas de otras arquitecturas de software en un modelo unificado. Se inspira en conceptos clave de:

- **Arquitectura Hexagonal (Puertos y Adaptadores)** de Alistair Cockburn.
- **Arquitectura Cebolla** de Jeffrey Palermo.
- **Datos, Contexto e Interacción (DCI)** de James Coplien y Trygve Reenskaug.
- **Entidad de Control de Límites (BCE)** de Ivar Jacobson.

Principios Guía

La esencia de la Arquitectura Limpia radica en la **separación de responsabilidades**. Esto se logra mediante capas dedicadas a casos de uso, reglas de negocio y otros conceptos fundamentales. Todas las capas, a excepción de las reglas de negocio, tienen una dependencia hacia una capa más interna.

Los cuatro principios rectores de la Arquitectura Limpia son:

1. **Separación de Responsabilidades (`Separation of Concerns`):** Cada parte del sistema tiene una única y clara responsabilidad.
2. **Principio de Responsabilidad Única (`Single Responsibility Principle`):** Cada módulo o clase debe tener una sola razón para cambiar.
3. **Principio de Inversión de Dependencias (`Dependency Inversion Principle`):** Los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles; los detalles deben depender de las abstracciones.
4. **Principio de Dependencias Explícitas (`Explicit Dependencies Principle`):** Las dependencias de un componente deben ser claramente declaradas, usualmente a través de constructores o propiedades.

Cómo Funciona

La Arquitectura Limpia se visualiza a menudo como **círculos concéntricos**, donde las capas más internas son las más abstractas y estables, y las más externas son las más concretas y variables. La clave es la **dirección de las dependencias**, que siempre apuntan hacia adentro:

- **Entidades (`Entities`):** En el **núcleo** del sistema, no dependen de nada. Contienen las **reglas de negocio corporativas y críticas** de la aplicación, las cuales son las menos propensas a cambiar.
- **Casos de Uso (`Use Cases`):** Dependen de las **Entidades**. Aquí reside la **lógica de negocio específica de la aplicación**. Orquestan el flujo de datos hacia y desde las Entidades.
- **Adaptadores de Interfaz (`Interface Adapters`):** Dependen de los **Casos de Uso**. Convierten los datos del formato más conveniente para los Casos de Uso y Entidades al formato más conveniente para los **Frameworks y Drivers externos**, y viceversa. Incluyen Presentadores, Vistas, Controladores, Gateways, etc.
- **Frameworks y Drivers (`Frameworks & Drivers`):** La capa más externa. Depende de los **Adaptadores de Interfaz**. Contiene los detalles de implementación específicos, como la **Interfaz de Usuario (UI), Web, APIs, Bases de Datos y Servicios Externos**.

Esta estructura asegura que las **capas internas no dependen de las capas externas**. Esta es una de las razones fundamentales para usar Clean Architecture, ya que **separa la lógica de negocio central de las decisiones técnicas y de infraestructura**.

Beneficios

La adopción de la Arquitectura Limpia conlleva varias ventajas significativas:

- **Separación de Responsabilidades:** Cada capa tiene un objetivo bien definido, lo que facilita la gestión y comprensión del código.
- **Testabilidad:** La lógica de negocio fundamental está aislada de las dependencias externas, permitiendo un **testing unitario más sencillo, rápido y fiable**.
- **Mantenibilidad:** Las fronteras claras entre capas y las dependencias unidireccionales reducen el riesgo de que un cambio en una parte del sistema afecte a otras, lo que simplifica el mantenimiento a largo plazo.
- **Escalabilidad:** La arquitectura soporta la incorporación de nuevas funcionalidades o tecnologías sin requerir cambios importantes en la lógica de negocio principal.
- **Flexibilidad:** Componentes externos como bases de datos, UI o *frameworks* pueden ser intercambiados o actualizados con un **impacto mínimo** en la lógica central de la aplicación.

Inconvenientes

A pesar de sus beneficios, la Arquitectura Limpia también presenta desafíos:

- **Complejidad:** La estructura puede introducir capas y abstracciones adicionales que podrían ser excesivas y **superfluas para proyectos pequeños o simples**.
- **Costo Inicial:** Requiere un **esfuerzo y una disciplina iniciales mayores** para configurar y mantener la arquitectura correctamente.
- **Curva de Aprendizaje Pronunciada:** Los desarrolladores no familiarizados con sus conceptos pueden encontrar difícil comprenderla e implementarla correctamente.
- **Posible Sobre-Ingeniería:** Una aplicación excesiva de la Arquitectura Limpia puede llevar a una **complejidad innecesaria**, especialmente en proyectos que no requieren una estructura tan robusta.
- **Desarrollo Más Lento al Inicio:** La estricta separación de responsabilidades y la adherencia a los principios pueden **ralentizar el desarrollo inicial**, particularmente en las etapas tempranas de un proyecto.

Clean Architecture es una propuesta robusta que busca ofrecer una solución duradera y flexible para el desarrollo de software complejo, priorizando la testabilidad y la mantenibilidad al desacoplar la lógica de negocio de los detalles técnicos.

ASP.NET Core: Un Marco Moderno para el Desarrollo Web

ASP.NET Core es un **marco de trabajo multiplataforma y de alto rendimiento** para crear **aplicaciones web modernas**. Este marco de **código abierto** permite a los desarrolladores construir aplicaciones web, servicios y APIs que se ejecutan en **Windows, macOS y Linux**. Está diseñado para el desarrollo a gran escala y puede manejar cualquier volumen de trabajo, lo que lo convierte en una opción robusta para aplicaciones de nivel empresarial.
Con ASP.NET Core, puedes:

- Crear aplicaciones web y servicios, aplicaciones de **Azure IoT (Internet de las cosas)** y **backends móviles**.
- Utilizar tus **herramientas de desarrollo favoritas** en Windows, macOS y Linux.
- Realizar **despliegues tanto locales como en la nube**.
- Ejecutar aplicaciones en la plataforma **.NET**.

¿Por Qué Elegir ASP.NET Core?

ASP.NET Core es un **rediseño de ASP.NET 4.x**, con cambios arquitectónicos que resultan en un marco más **sencillo y modular**. Ofrece numerosas ventajas:

- **Caso unificado** para crear **APIs web y una interfaz de usuario web**.
- Diseñado para la **capacidad de prueba**.
- **Blazor** permite usar **C# en el navegador** (junto con JavaScript), posibilitando compartir lógica de aplicación tanto del lado cliente como del lado servidor, todo escrito en .NET.
- **Razor Pages** facilita y acelera el desarrollo de escenarios centrados en páginas.
- Capacidad para desarrollar y ejecutar en **Windows, macOS y Linux**.
- Es **código abierto** y está **centrado en la comunidad**.
- Integración con **marcos modernos del lado cliente** y flujos de trabajo de desarrollo.
- Soporte para el alojamiento de servicios de **llamada a procedimiento remoto (RPC) mediante gRPC**.
- Un **sistema de configuración basado en entorno y listo para la nube**.
- **Inyección de dependencias** integrada.
- Una canalización de solicitud HTTP **ligera, de alto rendimiento y modular**.
- Capacidad para **hospedar en diversas plataformas**: Kestrel (servidor web propio), IIS, HTTP.sys, Nginx, y Docker.
- **Control de versiones en paralelo** (permite ejecutar diferentes versiones de .NET en la misma máquina).
- Herramientas que simplifican el desarrollo web moderno.

Creación de API Web e Interfaces de Usuario Web con ASP.NET Core MVC

ASP.NET Core MVC proporciona características robustas para construir APIs web y aplicaciones web:

- El **patrón Model-View-Controller (MVC)** facilita la capacidad de prueba de APIs web y aplicaciones web.
- **Blazor:** Un marco de UI web basado en componentes que utiliza C#, compatible con renderizado del lado del servidor y del lado del cliente a través de WebAssembly.
- **Razor Pages:** Un modelo de programación basado en páginas que simplifica y aumenta la productividad en la creación de interfaces de usuario web.
- **Razor Markup:** Proporciona una sintaxis productiva para Razor Pages y vistas MVC.

- **Asistentes de Etiquetas (`Tag Helpers`):** Permiten que el código del lado del servidor participe en la creación y renderizado de elementos HTML en archivos Razor.
- **Compatibilidad integrada con múltiples formatos de datos y negociación de contenido:** Permite a las APIs web comunicarse con una amplia gama de clientes (navegadores, dispositivos móviles).
- **Vinculación de modelos (`Model Binding`):** Asigna automáticamente datos de solicitudes HTTP a parámetros de métodos de acción.
- **Validación del modelo (`Model Validation`):** Realiza automáticamente la validación tanto del lado del cliente como del lado del servidor.

Desarrollo del Lado del Cliente

ASP.NET Core no solo incluye **Blazor** para crear interfaces de usuario web interactivas, sino que también se integra con otros **marcos populares de JavaScript front-end** como **Angular, React, Vue y Bootstrap**. Esto ofrece flexibilidad para el desarrollo del lado del cliente.

Plataformas de Destino de ASP.NET Core

A partir de **ASP.NET Core 3.x**, las aplicaciones **solo pueden tener como destino .NET** (anteriormente conocido como .NET Core). Esto trae consigo varias ventajas significativas en comparación con el antiguo .NET Framework:

- **Multiplataforma:** Soporte completo para Windows, macOS y Linux.
- **Rendimiento Mejorado.**
- **Control de Versiones en Paralelo.**
- **Nuevas APIs.**
- **Código Abierto.**

Ruta de Aprendizaje Recomendada

Para una introducción a ASP.NET Core, se sugiere la siguiente secuencia de tutoriales:

1. **Tutorial según el tipo de aplicación:**
    - **Aplicación web (UI Cliente):** Empieza con Blazor.
    - **Aplicación web (UI Servidor, nuevo desarrollo):** Introducción a Razor Pages.
    - **Aplicación web (Mantener MVC):** Introducción a MVC.
    - **API Web (Servicios HTTP RESTful):** Creación de una API web (hay un tutorial interactivo que no requiere instalación local).
    - **Aplicación de Llamada a Procedimiento Remoto (Servicios con búferes de protocolo):** Introducción a un servicio gRPC.
    - **Aplicación en Tiempo Real (Comunicación bidireccional):** Empieza con SignalR.
2. **Tutorial de Acceso a Datos Básico:**
    - **Nuevo desarrollo:** Blazor con Entity Framework Core o Razor Pages con Entity Framework Core.
    - **Mantener MVC:** MVC con Entity Framework Core.
3. **Introducción a los aspectos básicos de ASP.NET Core** que se aplican a todos los tipos de aplicaciones.
4. Explorar el resto de la documentación para temas de interés.

Migración y Ejemplos

- **Migración desde .NET Framework:** Para actualizar aplicaciones de ASP.NET 4.x a ASP.NET Core, consulta la guía "Actualización de ASP.NET a ASP.NET Core".
- **Descarga de Ejemplos:** Muchos artículos y tutoriales incluyen enlaces a código de ejemplo. Estos ejemplos se pueden descargar como un archivo ZIP del repositorio de ASP.NET.

- **Directivas de Preprocesador en Código de Ejemplo:** Algunos ejemplos utilizan directivas de preprocesador de C# (`#define`, `#if-#else/#elif-#endif`) para compilar y ejecutar selectivamente diferentes secciones de código. Para cambiar el escenario de un ejemplo, se debe modificar la directiva `#define` en la parte superior de los archivos C# relevantes.

ASP.NET Core es una plataforma robusta y flexible que ofrece a los desarrolladores herramientas de última generación para construir aplicaciones web de alto rendimiento y preparadas para el futuro. Su naturaleza multiplataforma y su enfoque en la modularidad lo convierten en una opción poderosa para cualquier tipo de proyecto, desde pequeñas aplicaciones hasta sistemas empresariales complejos.

Middleware en ASP.NET Core: La Canalización de Solicitudes y Respuestas

El **middleware** en ASP.NET Core es un componente de software fundamental que se integra en una **canalización de aplicaciones** para gestionar las solicitudes HTTP entrantes y las respuestas salientes. Cada componente de middleware tiene la capacidad de:

- Decidir si pasa la solicitud al siguiente componente en la canalización.
- Realizar operaciones **antes y/o después** de que el siguiente componente de la canalización haya procesado la solicitud.

Los **delegados de solicitudes** son los bloques de construcción para esta canalización, encargándose de manejar cada solicitud HTTP. Se configuran mediante métodos de extensión como **`Run`**, **`Map`**, y **`Use`**. Un delegado de solicitud puede definirse en línea como un método anónimo (conocido como middleware en línea) o en una clase reutilizable. Cada componente de middleware en la canalización es responsable de **invocar al siguiente componente** o de **"cortocircuitar" la canalización** si es necesario. Un middleware que cortocircuita se denomina **middleware terminal** porque impide que los componentes subsiguientes procesen la solicitud.

Rol del Middleware por Tipo de Aplicación

- Las **Blazor Web Apps, Razor Pages y las aplicaciones MVC** que procesan solicitudes del navegador en el servidor utilizan una canalización de middleware.
- Las **aplicaciones Blazor WebAssembly independientes** se ejecutan completamente en el cliente y **no procesan solicitudes con una canalización de middleware**, por lo que este artículo no se aplica a ellas.

Creación de una Canalización de Middleware con `WebApplication`

La canalización de solicitudes de ASP.NET Core es una secuencia de delegados de solicitud que se invocan uno tras otro.

Concepto Básico:

Cada delegado puede realizar operaciones antes y después de llamar al siguiente. Los delegados que manejan excepciones deben colocarse al principio de la canalización para que puedan capturar cualquier excepción que ocurra en etapas posteriores.

El `Run` Delegado:

La aplicación ASP.NET Core más sencilla puede configurar un único delegado de solicitud usando `app.Run()`. Este delegado es **siempre terminal** y finaliza la canalización, lo que significa que no se llamará a ningún middleware posterior:

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

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

```
});

app.Run(); // Este Run es el que ejecuta la aplicación, no un middleware adicional
```

Encadenamiento con `Use`:
Para encadenar múltiples delegados de solicitud, se utiliza el método `app.Use()`. El parámetro `next` representa el siguiente delegado en la canalización. Al no llamar a `next.Invoke()`, se puede **cortocircuitar la canalización**. Normalmente, se realizan acciones antes y después de llamar a `next`, como se muestra a continuación:

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

app.Use(async (context, next) =>
{
    // Realiza trabajo antes del siguiente componente (puede escribir en la respuesta).
    await next.Invoke();
    // Realiza trabajo después del siguiente componente (ej. logging, sin escribir en la
respuesta).
});

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from 2nd delegate.");
});

app.Run();
```

Cortocircuitar la Canalización de Solicitudes:
Cuando un delegado **no pasa la solicitud al siguiente delegado**, se dice que está **cortocircuitando la canalización**. Esto es útil para evitar trabajo innecesario. Por ejemplo, el **middleware de archivos estáticos** puede actuar como middleware terminal al procesar una solicitud de un archivo estático y detener el procesamiento.

> **Advertencia:** No intentes llamar a `next.Invoke()` ni modificar la respuesta una vez que los encabezados ya han sido enviados al cliente. Esto puede causar excepciones o dañar el formato de la respuesta. La propiedad `HttpContext.Response.HasStarted` es una pista útil para saber si la respuesta ya ha comenzado a enviarse.

Orden del Middleware: La Clave de la Funcionalidad y Seguridad
El orden en que se agregan los componentes de middleware en el método `Program.cs` **define el orden en que se invocarán** para las solicitudes y el orden inverso para las respuestas. Por motivos de **seguridad, rendimiento y funcionalidad**, el orden es **crítico**. Un diagrama típico de la canalización de procesamiento de solicitudes para aplicaciones ASP.NET Core MVC y Razor Pages muestra la siguiente secuencia general (el middleware de `Punto de conexión` ejecuta la canalización de filtro para MVC o Razor Pages):

Solicitud HTTP -> Excepción/Manejo de Errores -> HSTS -> Redirección HTTPS -> Archivos Estáticos -> Cookie Policy -> Enrutamiento -> Autenticación -> Autorización -> Sesión -> Punto de Conexión (MVC/Razor Pages) -> ... -> Respuesta HTTP

Ejemplo de orden recomendado para middleware de seguridad en `Program.cs`:

```
// ... (Configuración de servicios) ...

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

```
if (app.Environment.IsDevelopment())
{
   app.UseMigrationsEndPoint(); // Para entornos de desarrollo
}
else
{
   app.UseExceptionHandler("/Error"); // Para producción, captura excepciones
   app.UseHsts(); // Para producción, añade encabezado Strict-Transport-Security
}

app.UseHttpsRedirection(); // Redirige HTTP a HTTPS
app.UseStaticFiles();     // Sirve archivos estáticos (sin autorizar)
// app.UseCookiePolicy();   // Gestiona las políticas de cookies (GDPR, etc.)

app.UseRouting();         // Determina la ruta para la solicitud
// app.UseRateLimiter();    // Limita la tasa de solicitudes (si se usa después de UseRouting para endpoints específicos)
// app.UseRequestLocalization(); // Soporte de localización
// app.UseCors();           // Habilita CORS (debe ir antes de UseResponseCaching)

app.UseAuthentication();  // Intenta autenticar al usuario
app.UseAuthorization();   // Autoriza el acceso a recursos
// app.UseSession();        // Gestiona el estado de sesión (después de CookiePolicy, antes de MVC)
// app.UseResponseCompression(); // Comprime respuestas
// app.UseResponseCaching(); // Cachea respuestas

app.MapRazorPages();       // Mapea las Razor Pages a puntos de conexión
app.MapDefaultControllerRoute(); // Mapea las rutas por defecto para controladores MVC

app.Run();
```

Consideraciones importantes sobre el orden:

- **`UseCors`**, **`UseAuthentication`** y **`UseAuthorization`** deben aparecer en el orden mostrado.
- **`UseCors`** debe ir antes de **`UseResponseCaching`**.
- **`UseRequestLocalization`** debe aparecer antes de cualquier middleware que dependa de la cultura de la solicitud (ej., `app.UseStaticFiles()`).
- **`UseRateLimiter`** debe ir después de `UseRouting` si se usan APIs específicas de limitación de tasa por punto de conexión.
- El **middleware de archivos estáticos** se llama al principio para que pueda **cortocircuitar solicitudes** de archivos estáticos sin pasar por los componentes restantes. **No realiza comprobaciones de autorización**, por lo que los archivos servidos son públicos.
- **`UseExceptionHandler`** es el primer middleware de error, lo que le permite capturar excepciones de todos los middleware subsiguientes.
- El orden para el almacenamiento en caché y la compresión (`UseResponseCaching`, `UseResponseCompression`) puede variar según el escenario para optimizar la CPU o el almacenamiento de representaciones comprimidas.
- El **Middleware de Encabezados Reenviados** (`Forwarded Headers Middleware`) debe ejecutarse **antes** de otro middleware que dependa de la información de encabezados reenviados (esquema, host, IP de cliente, etc.).

Creación de Ramas en la Canalización de Middleware

ASP.NET Core permite crear ramas en la canalización de middleware para manejar rutas o condiciones específicas:

`Map`: Crea una rama de la canalización basada en una coincidencia de ruta. Si la ruta de la solicitud comienza con la ruta proporcionada, se ejecuta la rama. Los segmentos de ruta coincidentes se eliminan de `HttpRequest.Path` y se anexan a `HttpRequest.PathBase` dentro de la rama.

```
app.Map("/map1", HandleMapTest1);
app.Map("/map2", HandleMapTest2);
app.Run(async context => { await context.Response.WriteAsync("Hello from non-Map
delegate."); });

static void HandleMapTest1(IApplicationBuilder app) {
    app.Run(async context => { await context.Response.WriteAsync("Map Test 1"); });
}
// Si la solicitud es /map1, se ejecuta HandleMapTest1 y cortocircuita.
// Si la solicitud es /map3, se ejecuta el delegado final "Hello from non-Map delegate.".
```

- `Map` soporta anidación y puede coincidir con múltiples segmentos.

`MapWhen`: Crea una rama de la canalización basada en el resultado de un predicado `Func<HttpContext, bool>`. Se usa para detectar la presencia de variables de cadena de consulta, encabezados, etc. Si el predicado es `true`, se ejecuta la rama.

C#

```
app.MapWhen(context => context.Request.Query.ContainsKey("branch"), HandleBranch);
app.Run(async context => { await context.Response.WriteAsync("Hello from non-Map
delegate."); });

static void HandleBranch(IApplicationBuilder app) {
    app.Run(async context => {
        var branchVer = context.Request.Query["branch"];
        await context.Response.WriteAsync($"Branch used = {branchVer}");
    });
}
//Si la solicitud es /?branch=main, se ejecuta HandleBranch y cortocircuita.
```

`UseWhen`: Similar a `MapWhen`, pero la rama se **vuelve a unir a la canalización principal** si no contiene un middleware terminal. Esto permite que el middleware de la rama realice acciones condicionales sin detener el procesamiento de la solicitud por el resto de la canalización.

```
app.UseWhen(context => context.Request.Query.ContainsKey("branch"),
    appBuilder => HandleBranchAndRejoin(appBuilder));
app.Run(async context => { await context.Response.WriteAsync("Hello from non-Map
delegate."); });

void HandleBranchAndRejoin(IApplicationBuilder app) {
    // ... (middleware que se ejecuta en la rama) ...
    app.Use(async (context, next) => {
        // ... (log o trabajo que no escribe en la respuesta) ...
        await next(); // Se llama al siguiente delegado en la rama, que puede ser parte de la
canalización principal
        // ... (más trabajo después) ...
    });
}
```

```
// Si la solicitud es /?branch=main, el middleware en HandleBranchAndRejoin se ejecuta,
// pero la canalización principal continúa y también se ejecuta "Hello from non-Map delegate.".
```

Middleware Integrado

ASP.NET Core incluye una serie de componentes de middleware preconstruidos para tareas comunes, con un orden y comportamiento específicos:

Middleware	Descripción	Orden / Comentarios
Antiforgería	Protección contra falsificación de solicitudes.	Después de autenticación y autorización, antes de los puntos de conexión.
Autenticación	Soporte para autenticación de usuarios.	Antes de que `HttpContext.User` sea necesario. Terminal para callbacks de OAuth.
Autorización	Soporte para autorización de usuarios.	Inmediatamente después del middleware de autenticación.
Cookie Policy	Gestión del consentimiento de cookies y aplicación de estándares.	Antes del middleware que emite cookies (autenticación, sesión, MVC TempData).
CORS	Configura el intercambio de recursos entre orígenes.	Antes de los componentes que usan CORS. Actualmente, debe ir antes de `UseResponseCaching`.
DeveloperExceptionPage	Genera una página de errores detallada para desarrollo.	Antes de los componentes que pueden generar errores. Se registra automáticamente como el primer middleware en entornos de desarrollo.
Diagnóstico	Varios middleware para depuración y control de errores.	Antes de los componentes que generan errores. Terminal para excepciones o para servir la página web predeterminada.
Encabezados Reenviados	Reenvía encabezados de proxy a la solicitud actual.	**Debe ejecutarse antes** de los componentes que consumen los campos de encabezado actualizados (esquema, host, IP de cliente, método).

Comprobación de Estado	Verifica el estado de la aplicación y sus dependencias.	Terminal si una solicitud coincide con un punto de conexión de comprobación de estado.
Propagación de Encabezados	Propaga encabezados HTTP de la solicitud entrante a solicitudes salientes.	-
Registro HTTP	Registra solicitudes y respuestas HTTP.	Al principio de la canalización.
Invalidación del Método HTTP	Permite que una solicitud POST anule el método.	Antes de los componentes que consumen el método actualizado.
Redireccionamiento HTTPS	Redirige todas las solicitudes HTTP a HTTPS.	Antes de los componentes que consumen la URL.
Seguridad de Transporte Estricta (HSTS)	Añade un encabezado de seguridad especial en la respuesta.	Antes de que se envíen las respuestas y después de los componentes que modifican las solicitudes (ej., encabezados reenviados, reescritura de URL).
MVC	Procesa solicitudes con MVC/Razor Pages.	Si la solicitud coincide con una ruta, es terminal.
OWIN	Interoperabilidad con aplicaciones, servidores y middleware basados en OWIN.	Si el middleware OWIN procesa completamente la solicitud, es terminal.
Almacenamiento en Caché de Resultados	Soporte para almacenamiento en caché de respuestas.	Antes de los componentes que requieren el almacenamiento en caché. `UseRouting` debe ser anterior a `UseOutputCaching`. `UseCORS` debe ser anterior a `UseOutputCaching`.

Cacheo de Respuestas	Soporte para el almacenamiento en caché de respuestas. Requiere participación del cliente.	Antes de los componentes que requieren el almacenamiento en caché. `UseCORS` debe ser anterior a `UseResponseCaching`. No recomendado en UI (ej., Razor Pages) por los encabezados de navegador. El almacenamiento en caché de resultados es mejor para UIs.
Descompresión de Solicitudes	Soporte para descomprimir solicitudes.	Antes de los componentes que leen el cuerpo de la solicitud.
Compresión de Respuesta	Soporte para compresión de respuestas.	Antes de los componentes que requieren compresión.
Localización de Solicitudes	Soporte de localización.	Antes de los componentes que dependen de la ubicación. Después del middleware de enrutamiento cuando se usa `RouteDataRequestCultureProvider`.
Tiempos de Espera de Solicitudes	Soporte para configurar tiempos de espera.	`UseRequestTimeouts` debe ir después de `UseExceptionHandler`, `UseDeveloperExceptionPage` y `UseRouting`.
Enrutamiento de Punto de Conexión	Define y restringe las rutas de la solicitud.	Terminal para el emparejamiento de rutas.
SPA	Gestiona solicitudes de Aplicaciones de Página Única.	En un punto posterior de la cadena, para que otros middleware (archivos estáticos, MVC) tengan prioridad.
Sesión	Soporte para gestión de sesiones de usuario.	Antes de los componentes que requieren `Session`.
Archivos Estáticos	Soporte para servir archivos estáticos y explorar directorios.	Si la solicitud coincide con un archivo, es terminal.

Reescritura de URL	Soporte para reescritura de URL y redirección.	Antes de los componentes que consumen la URL.
W3C Logging	Genera registros de acceso al servidor en formato W3C.	Al principio de la canalización.
WebSockets	Habilita el protocolo WebSockets.	Antes de los componentes necesarios para aceptar solicitudes de WebSocket.

El sistema de middleware en ASP.NET Core es increíblemente potente y flexible, permitiendo a los desarrolladores construir canalizaciones de procesamiento de solicitudes altamente personalizables para sus aplicaciones. Entender su funcionamiento y el orden de los componentes es esencial para desarrollar aplicaciones robustas y eficientes.

API basadas en controlador vs. API mínimas en ASP.NET Core

ASP.NET Core ofrece dos enfoques principales para construir APIs: las **API basadas en controlador** y las **API mínimas**. Aunque ambas permiten crear servicios web, difieren en su estructura y la forma en que se define la lógica.

Descripción General

Las **API basadas en controlador** utilizan **clases dedicadas** (controladores) que se derivan de `ControllerBase`. Estos controladores agrupan la lógica relacionada con un recurso o conjunto de recursos, y suelen seguir patrones de **programación orientada a objetos**, facilitando la **inyección de dependencias** a través de sus constructores o propiedades. Por otro lado, las **API mínimas** definen los puntos de conexión directamente en el archivo `Program.cs` (o archivos auxiliares) utilizando **expresiones lambda o métodos estáticos**. Este enfoque busca ser más conciso y reduce la sobrecarga de clases, ocultando la clase `Host` por defecto. La configuración y extensibilidad se manejan a través de métodos de extensión, y la inyección de dependencias se realiza accediendo directamente al proveedor de servicios.

Ejemplos de Código

Para ilustrar las diferencias, consideremos una API que devuelve una previsión meteorológica:

API Basada en Controlador

```
// Program.cs
namespace APIWithControllers;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers(); // Agrega el soporte para controladores
        var app = builder.Build();
```

```
        app.UseHttpsRedirection();
        app.MapControllers(); // Mapea las rutas definidas en los controladores
        app.Run();
    }
}

// Controllers/WeatherForecastController.cs
using Microsoft.AspNetCore.Mvc;

namespace APIWithControllers.Controllers;

[ApiController]
[Route("[controller]")] // Define la ruta base para este controlador (ej. /WeatherForecast)
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;

    // Inyección de dependencias a través del constructor
    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")] // Define un punto de conexión GET
    public IEnumerable<WeatherForecast> Get()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

API Mínima

```
// Program.cs
namespace MinimalAPI;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);
        var app = builder.Build();

        app.UseHttpsRedirection();
```

```
        var summaries = new[] // Datos definidos directamente en Program.cs
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering",
"Scorching"
        };

        // Define el punto de conexión directamente con una expresión lambda
        app.MapGet("/weatherforecast", (HttpContext httpContext) =>
        {
            var forecast = Enumerable.Range(1, 5).Select(index =>
                new WeatherForecast
                {
                    Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                    TemperatureC = Random.Shared.Next(-20, 55),
                    Summary = summaries[Random.Shared.Next(summaries.Length)]
                })
                .ToArray();
            return forecast;
        });
        app.Run();
    }
}
```

Ambos enfoques usan la siguiente clase para el modelo de datos:

```
C#
namespace APIWithControllers; // Puede estar en cualquier namespace, se comparte
public class WeatherForecast
{
    public DateOnly Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    public string? Summary { get; set; }
}
```

Comparativa de Funcionalidades

Las API mínimas han evolucionado para ofrecer gran parte de la funcionalidad de las API basadas en controlador, incluyendo la capacidad de escalar, manejar rutas complejas, aplicar reglas de autorización y controlar el contenido de las respuestas.

Sin embargo, hay algunas funcionalidades que actualmente **no están soportadas de forma nativa** o no son tan directas en las API mínimas como lo son en las basadas en controlador:

- **Sin soporte integrado para el enlace de modelos (`IModelBinderProvider`, `IModelBinder`):** Aunque se puede añadir soporte con una implementación personalizada, no viene de fábrica.
- **Sin soporte integrado para la validación (`IModelValidator`):** La validación del modelo no es tan fluida como con los controladores.
- **No se admiten elementos de aplicación (`Application Parts`) ni el modelo de aplicación (`Application Model`):** Esto significa que no hay una forma incorporada de aplicar o crear convenciones personalizadas a nivel de la aplicación.
- **No hay soporte integrado para la renderización de vistas:** Para este propósito, se recomienda usar **Razor Pages**.
- **No se admite `JsonPatch`.**
- **No se admite OData.**

Elección entre los enfoques

La elección entre API basadas en controlador y API mínimas depende de las necesidades específicas del proyecto:

- **API mínimas** son ideales para:
  - **Microservicios pequeños y simples:** Donde la sobrecarga de un controlador completo no es necesaria.
  - **APIs ligeras:** Para tareas muy específicas o de un solo propósito.
  - **Prototipado rápido:** Permiten una puesta en marcha muy veloz.
  - **Desarrolladores que prefieren un estilo más funcional:** Ya que la lógica se define a menudo en lambdas.
- **API basadas en controlador** son más adecuadas para:
  - **APIs grandes y complejas:** Que se benefician de la organización en clases y la separación de responsabilidades.
  - **Proyectos que requieren características avanzadas de MVC:** Como el modelo de aplicación, validación de modelos integrada, o la renderización de vistas (aunque para esto último, Razor Pages es una alternativa).
  - **Equipos grandes:** Donde una estructura más formal puede mejorar la mantenibilidad y la colaboración.
  - **Proyectos que necesitan una fuerte adhesión a los principios de POO y DI:** La inyección de constructores en controladores es un patrón bien establecido.

En resumen, las API mínimas ofrecen una forma más ligera y concisa de construir APIs, ideal para escenarios donde la simplicidad es clave. Las API basadas en controlador, por su parte, proporcionan una estructura más robusta y un conjunto más completo de características para aplicaciones más grandes y complejas.

Comprobaciones de Estado en ASP.NET Core: Monitorizando la Salud de tu Aplicación

ASP.NET Core proporciona un **middleware de comprobaciones de estado** y bibliotecas asociadas para informar sobre la salud y el estado de los componentes de infraestructura de tu aplicación. Estas comprobaciones se exponen como **puntos de conexión HTTP** configurables, ideales para la monitorización en tiempo real.

¿Por qué usar Comprobaciones de Estado?

Los puntos de conexión de comprobaciones de estado son cruciales para:

- **Orquestadores de contenedores y equilibradores de carga:** Utilizan estos sondeos para verificar el estado de una aplicación. Si una comprobación de estado falla, un orquestador podría detener un despliegue, reiniciar un contenedor, o un equilibrador de carga podría redirigir el tráfico lejos de una instancia poco saludable.
- **Monitorización de recursos:** Permiten supervisar el uso de memoria, disco y otros recursos del servidor físico para determinar su estado operativo.
- **Prueba de dependencias:** Las comprobaciones de estado pueden verificar la disponibilidad y el funcionamiento normal de las dependencias de la aplicación, como bases de datos y servicios externos.

Es fundamental decidir qué **sistema de monitorización externa o orquestador de contenedores** vas a usar antes de implementar las comprobaciones de estado, ya que esto determinará el tipo de comprobaciones y cómo configurar sus puntos de conexión.

Sondeo de Estado Básico

Para muchas aplicaciones, una configuración básica que informa sobre la disponibilidad general de la aplicación para procesar solicitudes es suficiente.

Configuración:

1. Registra los servicios de comprobación de estado con `builder.Services.AddHealthChecks()` en `Program.cs`.

2. Crea un punto de conexión de comprobación de estado llamando a **`app.MapHealthChecks()`** con la URL deseada.

Por defecto, no se registran comprobaciones de estado específicas para dependencias. La aplicación se considera "saludable" si puede responder a la URL del punto de conexión. El escritor de respuestas predeterminado devuelve el estado (**`HealthStatus.Healthy`**, **`HealthStatus.Degraded`**, **o** **`HealthStatus.Unhealthy`**) como texto plano.

Ejemplo:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddHealthChecks(); // Registra los servicios de comprobación de estado
var app = builder.Build();

app.MapHealthChecks("/healthz"); // Crea un punto de conexión en /healthz

app.Run();
```

Docker HEALTHCHECK: Puedes integrar esto con la directiva `HEALTHCHECK` de Docker para monitorizar tus contenedores:

Dockerfile

```
HEALTHCHECK CMD curl --fail http://localhost:5000/healthz || exit 1
```

Creación de Comprobaciones de Estado Personalizadas

Para crear comprobaciones de estado más sofisticadas, implementa la interfaz **`IHealthCheck`**. El método **`CheckHealthAsync`** debe devolver un **`HealthCheckResult`** indicando el estado (`Healthy`, `Degraded`, o `Unhealthy`). Este resultado se escribe como texto plano con un código de estado configurable y puede incluir pares clave-valor opcionales.

Esquema Básico:

```
public class SampleHealthCheck : IHealthCheck
{
   public Task<HealthCheckResult> CheckHealthAsync(
      HealthCheckContext context, CancellationToken cancellationToken = default)
   {
      var isHealthy = true; // Aquí iría tu lógica real de comprobación

      if (isHealthy)
      {
         return Task.FromResult(HealthCheckResult.Healthy("A healthy result."));
      }

      // Si falla, puedes usar el FailureStatus configurado o Unhealthy por defecto
      return Task.FromResult(
         new HealthCheckResult(
            context.Registration.FailureStatus, "An unhealthy result."));
   }
}
```

Si `CheckHealthAsync` lanza una excepción, se reportará como `FailureStatus` con la excepción interna.

Registro de los Servicios de Comprobación de Estado

Para registrar una comprobación de estado personalizada, usa **`AddCheck`**:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>("Sample"); // Registra tu comprobación por su tipo
```

Puedes especificar un **`failureStatus`** personalizado y **`tags`** para filtrar las comprobaciones:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>(
        "Sample",
        failureStatus: HealthStatus.Degraded, // Estado si falla la comprobación
        tags: new[] { "sample" });           // Etiquetas para filtrar
```

También puedes registrar comprobaciones de estado directamente con una **función lambda** o usar **`AddTypeActivatedCheck`** para pasar argumentos al constructor de tu comprobación.

Uso del Enrutamiento de Comprobaciones de Estado

El método **`app.MapHealthChecks("/healthz")`** registra un punto de conexión de comprobación de estado que coincide y se ejecuta junto con otros puntos de conexión de la aplicación.

Opciones Avanzadas de Enrutamiento:

`RequireHost()`: Restringe el punto de conexión para que responda solo a hosts específicos (puedes incluir el puerto, útil en contenedores).

```
app.MapHealthChecks("/healthz")
    .RequireHost("www.contoso.com:5001");
```

- **Advertencia:** Las API que dependen del encabezado `Host` son vulnerables a suplantación. Usa `HttpContext.Connection.LocalPort` o filtrado de hosts para mitigar esto, y considera `RequireAuthorization()`.

`RequireAuthorization()`: Aplica el middleware de autorización al punto de conexión de comprobaciones de estado. Puedes especificar políticas de autorización.

C#

```
app.MapHealthChecks("/healthz")
    .RequireAuthorization();
```

-
- **`RequireCors()`:** Habilita el middleware CORS para el punto de conexión (aunque no es común para comprobaciones de estado directas).

Opciones de Comprobación de Estado

La clase **`HealthCheckOptions`** permite personalizar el comportamiento del middleware de comprobaciones de estado:

`Predicate`: Filtra las comprobaciones de estado que se ejecutarán.

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("sample") // Solo ejecuta las con la etiqueta "sample"
});
```

`ResultStatusCodes`: Personaliza la asignación de los estados de salud a códigos de estado HTTP.

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
```

```
{
    ResultStatusCodes =
    {
        [HealthStatus.Healthy] = StatusCodes.Status200OK,
        [HealthStatus.Degraded] = StatusCodes.Status200OK,
        [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable // Por defecto
    }
});
```

- **`AllowCachingResponses`:** Controla si el middleware agrega encabezados HTTP para evitar el almacenamiento en caché de la respuesta (`false` por defecto).

`ResponseWriter`: Permite personalizar completamente la salida del informe de comprobaciones de estado, por ejemplo, para devolver JSON.

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    ResponseWriter = WriteResponse // Tu delegado personalizado
});

// Ejemplo de delegado WriteResponse (genera JSON)
private static Task WriteResponse(HttpContext context, HealthReport healthReport)
{
    context.Response.ContentType = "application/json; charset=utf-8";
    // ... (lógica para serializar HealthReport a JSON) ...
    return
context.Response.WriteAsync(Encoding.UTF8.GetString(memoryStream.ToArray()));
}
```

- **Nota:** La API de comprobaciones de estado no tiene soporte JSON complejo integrado; debes personalizar el `ResponseWriter` según tu sistema de monitorización.

Sondeo de Bases de Datos

Puedes integrar comprobaciones de estado para tus bases de datos:

SQL Server (`AspNetCore.HealthChecks.SqlServer`): Ejecuta una consulta simple (ej., `SELECT 1`) para confirmar la conexión.

```
builder.Services.AddHealthChecks()
    .AddSqlServer(conStr);
```

- **Advertencia:** Elige una consulta que se devuelva rápidamente para evitar sobrecargar la base de datos. A menudo, una conexión exitosa es suficiente.

Entity Framework Core (`Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore`): `AddDbContextCheck` verifica que la aplicación puede comunicarse con la base de datos configurada para un `DbContext` de EF Core, llamando a `CanConnectAsync` por defecto.

```
builder.Services.AddHealthChecks()
    .AddDbContextCheck<SampleDbContext>();
```

Sondeos de Preparación (`Readiness`) y Ejecución (`Liveness`)

En algunos escenarios de alojamiento (como Kubernetes), se utilizan dos tipos de comprobaciones de estado distintas:

- **Preparación (`Readiness`):** Indica si la aplicación se está ejecutando pero **aún no está lista para recibir solicitudes** (ej., está cargando una configuración inicial grande).

- **Ejecución (`Liveness`):** Indica si la aplicación **se ha bloqueado y necesita ser reiniciada**.

Puedes implementar esto con una tarea en segundo plano y una comprobación de estado que exponga una propiedad que indique cuando la tarea de inicio ha finalizado:

```
// Servicio en segundo plano que simula una tarea de inicio larga
public class StartupBackgroundService : BackgroundService
{
    private readonly StartupHealthCheck _healthCheck;
    public StartupBackgroundService(StartupHealthCheck healthCheck) => _healthCheck =
healthCheck;
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);
        _healthCheck.StartupCompleted = true; // Indica que la tarea ha finalizado
    }
}

// Comprobación de estado que reporta el progreso de la tarea de inicio
public class StartupHealthCheck : IHealthCheck
{
    private volatile bool _isReady;
    public bool StartupCompleted { get => _isReady; set => _isReady = value; }
    public Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,
CancellationToken cancellationToken = default)
    {
        if (StartupCompleted) return Task.FromResult(HealthCheckResult.Healthy("Startup task
completed."));
        return Task.FromResult(HealthCheckResult.Unhealthy("Startup task still running."));
    }
}
```

Registro y mapeo de puntos de conexión separados:

```
builder.Services.AddHostedService<StartupBackgroundService>();
builder.Services.AddSingleton<StartupHealthCheck>(); // Registra como Singleton para que
el servicio pueda actualizarla

builder.Services.AddHealthChecks()
    .AddCheck<StartupHealthCheck>("Startup", tags: new[] { "ready" }); // Etiqueta para el
sondeo de preparación

app.MapHealthChecks("/healthz/ready", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("ready") // Solo las etiquetadas
con "ready"
});

app.MapHealthChecks("/healthz/live", new HealthCheckOptions
{
    Predicate = _ => false // Excluye todas las comprobaciones, solo verifica si el proceso está
activo
});
```

Este enfoque es particularmente útil en entornos como **Kubernetes**, que distinguen entre sondeos de preparación y ejecución para gestionar el ciclo de vida de los pods.

Publicadores de Comprobación de Estado (`IHealthCheckPublisher`)

Cuando agregas una implementación de `IHealthCheckPublisher` al contenedor de servicios, el sistema de comprobación de estado **ejecutará periódicamente las comprobaciones de estado y llamará a `PublishAsync` con el resultado**. Esto es útil para sistemas de monitorización basados en "push" (que esperan que la aplicación les envíe su estado).

Puedes configurar el `Delay` (retraso inicial), `Period` (frecuencia de ejecución), `Predicate` (filtro de comprobaciones) y `Timeout` a través de `HealthCheckPublisherOptions`.

```
public class SampleHealthCheckPublisher : IHealthCheckPublisher
{
    public Task PublishAsync(HealthReport report, CancellationToken cancellationToken)
    {
        if (report.Status == HealthStatus.Healthy) { /* ... */ }
        else { /* ... */ }
        return Task.CompletedTask;
    }
}

builder.Services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.FromSeconds(2);
    options.Predicate = healthCheck => healthCheck.Tags.Contains("sample");
});
builder.Services.AddSingleton<IHealthCheckPublisher, SampleHealthCheckPublisher>();
```

Inyección de Dependencias y Comprobaciones de Estado

Las comprobaciones de estado pueden consumir servicios a través de la **inyección de dependencias (DI)**, útil para inyectar opciones o configuración global.

C#

```
public class SampleHealthCheckWithDI : IHealthCheck
{
    private readonly SampleHealthCheckWithDiConfig _config;
    public SampleHealthCheckWithDI(SampleHealthCheckWithDiConfig config) => _config =
config;

    public Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,
CancellationToken cancellationToken = default)
    {
        // Usa _config en tu lógica de comprobación
        return Task.FromResult(HealthCheckResult.Healthy("A healthy result."));
    }
}

builder.Services.AddSingleton<SampleHealthCheckWithDiConfig>(new
SampleHealthCheckWithDiConfig { BaseUriToCheck = new
Uri("https://sample.contoso.com/api/") });
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheckWithDI>("With Dependency Injection", tags: new[] {
"inject" });
```

`UseHealthChecks` vs. `MapHealthChecks`

Existen dos formas de exponer las comprobaciones de estado:

- **`UseHealthChecks`:** Registra el middleware para manejar las solicitudes de comprobaciones de estado. Este método **finaliza la canalización** si una solicitud coincide, lo cual es deseable para evitar trabajo innecesario (logging, otros middleware). Se usa principalmente para configurar el middleware de comprobación de estado directamente en la canalización.
- **`MapHealthChecks`:** Registra un punto de conexión de comprobaciones de estado como parte del sistema de enrutamiento de puntos de conexión. La ventaja es que permite usar **middleware compatible con puntos de conexión** (como la autorización) y un mayor control específico sobre la política de coincidencia.

En resumen, las comprobaciones de estado son una herramienta vital en ASP.NET Core para asegurar que tus aplicaciones no solo están funcionando, sino que también están **saludables y listas para servir tráfico**. Su flexibilidad permite adaptarlas a una amplia gama de escenarios de monitorización.

Creación de API Web con ASP.NET Core: Uso de Controladores

ASP.NET Core ofrece la flexibilidad de crear API web tanto con **controladores** como con **API mínimas**. Este artículo se centra en la creación de API web utilizando **controladores**, que son clases que derivan de `ControllerBase` y se activan y eliminan por cada solicitud.

Clase `ControllerBase`

Una API web basada en controlador se compone de una o más clases de controlador que heredan de **`ControllerBase`**. Esta clase proporciona un conjunto robusto de propiedades y métodos útiles para manejar solicitudes HTTP. Un ejemplo común, como el proporcionado por la plantilla de proyecto de API web, es:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Es importante destacar que los controladores de API web generalmente deben derivar de `ControllerBase` en lugar de `Controller`. **`Controller`** deriva de `ControllerBase` pero añade soporte para vistas, lo cual es apropiado para manejar páginas web, no solicitudes puramente de API. Si un controlador necesita soportar tanto vistas como API web, entonces debe derivar de `Controller`.

Ejemplos de métodos útiles en `ControllerBase`:

Método	Notas
`BadRequest()`	Devuelve el código de estado **400 Bad Request**.
`NotFound()`	Devuelve el código de estado **404 Not Found**.
`CreatedAtAction()`	Devuelve un código de estado **201 Created**, a menudo con una URL a un recurso recién creado. Por ejemplo: `return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);`
`PhysicalFile()`	Devuelve un archivo.

`TryUpdateModelAsync()`	Invoca el **enlace de modelo**.
`TryValidateModel()`	Invoca la **validación de modelos**.

Para una lista completa de propiedades y métodos, consulta la documentación de `ControllerBase`.

Atributos

El espacio de nombres `Microsoft.AspNetCore.Mvc` ofrece una variedad de atributos para configurar el comportamiento de los controladores y métodos de acción de la API web. Estos atributos permiten definir cómo las acciones responden a las solicitudes HTTP.

Ejemplo:

```
[HttpPost] // Indica que esta acción responde a solicitudes HTTP POST
[ProducesResponseType(StatusCodes.Status201Created)] // Documenta que puede devolver un 201
[ProducesResponseType(StatusCodes.Status400BadRequest)] // Documenta que puede devolver un 400
public ActionResult<Pet> Create(Pet pet)
{
    // ... lógica para crear una mascota ...
    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

Algunos atributos comunes:

Atributo	Notas
`[Route]`	Especifica el patrón de URL para un controlador o una acción.
`[Bind]`	Especifica el prefijo y las propiedades a incluir en el enlace de modelo.
`[HttpGet]`	Identifica una acción que soporta el verbo HTTP GET.
`[Consumes]`	Especifica los **tipos de datos (media types)** que una acción acepta en el cuerpo de la solicitud (ej., `application/json`).
`[Produces]`	Especifica los **tipos de datos (media types)** que una acción devuelve en la respuesta.

Para una lista exhaustiva, consulta el espacio de nombres `Microsoft.AspNetCore.Mvc`.

Atributo `[ApiController]`

El atributo **`[ApiController]`** se aplica a una clase de controlador (o a un ensamblado completo) para habilitar una serie de comportamientos específicos de la API que simplifican el desarrollo y garantizan la conformidad con las mejores prácticas REST:

Requisito de enrutamiento mediante atributos: Fuerza el uso de enrutamiento por atributos (`[Route]`, `[HttpGet]`, etc.), deshabilitando las rutas convencionales.

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

1. **Respuestas HTTP 400 automáticas:** Si la validación del modelo falla (`ModelState.IsValid` es `false`), `[ApiController]` genera automáticamente una respuesta HTTP 400 Bad Request. Esto elimina la necesidad de comprobaciones manuales como:

```
 if (!ModelState.IsValid)
{
   return BadRequest(ModelState);
}
```

2. La respuesta predeterminada para un error de validación es de tipo **`ValidationProblemDetails`**, que cumple con la [especificación RFC 7807] para errores legibles por máquina en API web. Puedes personalizar el comportamiento de estas respuestas configurando `InvalidModelStateResponseFactory` en `ApiBehaviorOptions`. También puedes deshabilitar este comportamiento automático estableciendo `SuppressModelStateInvalidFilter = true`.
3. **Inferencia de parámetro de origen de enlace:** Simplifica la configuración al inferir automáticamente de dónde deben provenir los datos para los parámetros de acción. No es necesario aplicar explícitamente atributos como `[FromQuery]`, `[FromBody]`, etc., en muchos casos.
   - **`[FromServices]`** se infiere para tipos complejos registrados en el contenedor de DI.
   - **`[FromBody]`** se infiere para tipos complejos *no* registrados en DI (a menos que sean tipos especiales como `IFormCollection`).
   - **`[FromForm]`** se infiere para `IFormFile` y `IFormFileCollection`.
   - **`[FromRoute]`** se infiere para nombres de parámetros que coinciden con segmentos de ruta.
   - **`[FromQuery]`** se infiere para cualquier otro parámetro.
4. **Nota sobre `[FromBody]`:** Solo se puede tener un parámetro inferido como `[FromBody]` por acción. Intentar inferir o aplicar `[FromBody]` a varios parámetros en una misma acción resultará en una excepción.
   Nota sobre `[FromServices]`: En casos aislados, la inferencia automática de `[FromServices]` podría causar conflictos si tienes un tipo en DI que también se usa como argumento de acción no destinado a DI. Puedes deshabilitar esta inferencia para un parámetro específico con otro atributo de origen de enlace (`[FromBody]`, etc.) o globalmente configurando `DisableImplicitFromServicesParameters = true` en `ApiBehaviorOptions`.
5. **Inferencia de solicitud de varios elementos o datos de formulario:** Para parámetros de tipo `IFormFile` y `IFormFileCollection`, el atributo `[ApiController]` infiere que el tipo de contenido de la solicitud es `multipart/form-data`. Puedes deshabilitar esto con `SuppressConsumesConstraintForFormFileParameters = true`.

Detalles de problemas de los códigos de estado de error: MVC transforma automáticamente los resultados de error (códigos de estado 400 o superiores, como los

devueltos por `NotFound()`, `BadRequest()`, etc.) en un objeto `ProblemDetails` (`application/problem+json`). Esto proporciona un formato estandarizado para los mensajes de error de la API.

```
if (pet == null)
{
    return NotFound(); // Genera un 404 con un cuerpo ProblemDetails
}
```

6. Puedes deshabilitar la creación automática de `ProblemDetails` estableciendo `SuppressMapClientErrors = true` en `ApiBehaviorOptions`.

Definición de Tipos de Contenido con el Atributo `[Consumes]`

Por defecto, una acción de API puede aceptar cualquier tipo de contenido compatible con los formatadores de entrada configurados (ej., JSON, XML). El atributo **`[Consumes]`** te permite **limitar los tipos de contenido** que una acción o controlador específico aceptará.

```
[HttpPost]
[Consumes("application/xml")] // Esta acción solo acepta XML
public IActionResult CreateProduct(Product product)
{
    // ...
}
```

Si una solicitud no especifica un encabezado `Content-Type` que coincida con lo definido en `[Consumes]`, se generará una respuesta **415 Unsupported Media Type**.

`[Consumes]` también es crucial para **resolver ambigüedades** cuando varias acciones pueden coincidir con la misma ruta y verbo HTTP, pero difieren en el tipo de contenido que consumen.

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController : ControllerBase
{
    [HttpPost]
    [Consumes("application/json")]
    public IActionResult PostJson(IEnumerable<int> values) =>
        Ok(new { Consumes = "application/json", Values = values });

    [HttpPost]
    [Consumes("application/x-www-form-urlencoded")]
    public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
        Ok(new { Consumes = "application/x-www-form-urlencoded", Values = values });
}
```

En este ejemplo, `PostJson` manejará solicitudes POST con `Content-Type: application/json`, mientras que `PostForm` manejará solicitudes con `Content-Type: application/x-www-form-urlencoded`, incluso si comparten la misma ruta base.

El uso de controladores y los atributos proporcionados por ASP.NET Core facilita enormemente la creación de API web robustas y conformes a los estándares

Enrutamiento en ASP.NET Core: La Navegación de tu Aplicación

El enrutamiento en ASP.NET Core es el mecanismo fundamental que se encarga de **hacer coincidir las solicitudes HTTP entrantes y dirigirlas a los puntos de conexión ejecutables** dentro de tu aplicación. Los **puntos de conexión** son las unidades de código de control de solicitudes, como controladores, Razor Pages, servicios SignalR o gRPC, middleware específico (como comprobaciones de estado), o incluso delegados y expresiones lambda.
El sistema de enrutamiento no solo dirige el tráfico, sino que también puede **extraer valores de la URL** para su procesamiento posterior y **generar URL** que se mapean a puntos de conexión específicos.

Fundamentos del Enrutamiento
En su núcleo, el enrutamiento de ASP.NET Core se basa en un par de middleware:

1. **`UseRouting`**: Este middleware se encarga de la **coincidencia de rutas**. Examina el conjunto de puntos de conexión definidos en la aplicación y selecciona la "mejor coincidencia" basándose en la URL y el método HTTP de la solicitud entrante.
2. **`UseEndpoints`**: Este middleware es el responsable de la **ejecución del punto de conexión**. Una vez que `UseRouting` ha seleccionado un punto de conexión, `UseEndpoints` ejecuta el delegado de solicitud asociado a dicho punto.

Aunque `WebApplicationBuilder` suele configurar automáticamente una canalización que incluye `UseRouting` y `UseEndpoints`, puedes llamarlos explícitamente para controlar el orden en que se ejecutan en tu canalización de middleware, permitiendo que otro middleware se ejecute antes o después de la lógica de enrutamiento.
Ejemplo Básico:

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

app.MapGet("/", () => "Hello World!"); // Define un punto de conexión para GET /

app.Run();
```

Este código define un único punto de conexión que responde a solicitudes HTTP GET en la URL raíz (`/`) con "Hello World!".

Puntos de Conexión (`Endpoints`)
Un punto de conexión en ASP.NET Core es una unidad de funcionalidad que se puede:

- **Seleccionar**: Coincide con una URL y un método HTTP.
- **Ejecutar**: Ejecuta un delegado de solicitud (`RequestDelegate`).
- **Extender**: Tiene una colección de `Metadata` (metadatos) arbitrarios.
- **Enumerar**: La colección de puntos de conexión se puede enumerar a través de `EndpointDataSource` desde la Inyección de Dependencias (DI).

Los puntos de conexión se configuran en `UseEndpoints` o directamente con métodos como `MapGet`, `MapPost`, y también a través de métodos que integran características del framework:

- `MapRazorPages` para Razor Pages
- `MapControllers` para controladores
- `MapHub<THub>` para SignalR
- `MapGrpcService<TService>` para gRPC
- `MapHealthChecks` para comprobaciones de estado

Metadatos de Puntos de Conexión: Los puntos de conexión pueden tener **metadatos** adicionales adjuntos, que son datos que pueden ser procesados por middleware compatible con el enrutamiento. Por ejemplo, una directiva de autorización

(`RequireAuthorization()`) es un tipo de metadato que el middleware de autorización puede inspeccionar para aplicar reglas de seguridad específicas al punto de conexión.

```
app.UseAuthentication();
app.UseAuthorization();

app.MapHealthChecks("/healthz").RequireAuthorization(); // Aplica autorización al punto de conexión de salud
app.MapGet("/", () => "Hello World!");
```

Este ejemplo muestra cómo el middleware de autenticación y autorización puede ejecutarse entre `UseRouting` y `UseEndpoints` para aplicar políticas basadas en los metadatos del punto de conexión seleccionado.

Conceptos de Enrutamiento

El `Endpoint` seleccionado para la solicitud actual se puede recuperar desde `HttpContext.GetEndpoint()`. Puedes inspeccionar sus propiedades, incluyendo sus metadatos. Los objetos `Endpoint` son inmutables una vez creados.

El middleware puede influir en el enrutamiento antes de que se ejecute `UseRouting` (modificando la solicitud, por ejemplo) o reaccionar a él después de `UseRouting` pero antes de `UseEndpoints` (inspeccionando metadatos para tomar decisiones, como la auditoría o la autorización).

Middleware Terminal vs. Enrutamiento

- **Middleware Terminal**: Es un middleware que, al encontrar una condición de coincidencia (ej. `context.Request.Path == "/"`), ejecuta una funcionalidad y **devuelve un valor en lugar de invocar `next()`** en la canalización, terminando así el procesamiento de la solicitud.
- **Enrutamiento**: Los puntos de conexión definidos a través del sistema de enrutamiento son inherentemente **terminales**.

Diferencias Clave:

- **Control del Orden**: El middleware terminal puede ubicarse en cualquier punto de la canalización, mientras que los puntos de conexión se ejecutan en la posición de `UseEndpoints`.
- **Lógica de Coincidencia**: El middleware terminal requiere que escribas tu propia lógica de coincidencia de URL. El enrutamiento proporciona soluciones predefinidas y robustas para patrones de URL comunes.
- **Integración con Características**: Los puntos de conexión se integran de forma natural con middleware como `UseAuthorization` y `UseCors`. Para lograr esto con middleware terminal, se necesita una interacción manual con los sistemas subyacentes.

Aunque el middleware terminal es potente, a menudo requiere más código y pruebas, y una integración manual con otros sistemas. Se recomienda considerar la integración con el enrutamiento antes de implementar un middleware terminal personalizado.

Coincidencia de Dirección URL

Es el proceso por el cual el enrutamiento empareja una solicitud entrante con un punto de conexión. Se basa en datos de ruta y encabezados, y se puede extender.

El proceso de coincidencia de URL ocurre en fases:

1. Se procesa la ruta de la URL con todas las plantillas de ruta y se recopilan **todas las coincidencias**.
2. Se eliminan las coincidencias que fallan en las **restricciones de ruta** aplicadas.
3. Se eliminan las coincidencias que fallan en las políticas de coincidencia (`MatcherPolicy`).

4. `EndpointSelector` toma la decisión final, eligiendo el punto de conexión de **mayor prioridad** entre las coincidencias. Si hay múltiples coincidencias con la misma prioridad, se lanza una excepción de ambigüedad.

La prioridad se basa en:

- `RouteEndpoint.Order`: Un valor de orden explícito definido en el punto de conexión.
- **Precedencia de la plantilla de ruta**: Un sistema automático que asigna mayor prioridad a las plantillas de ruta "más específicas" (más segmentos literales, segmentos de parámetros con restricciones, etc.). Las plantillas con comodines (`*` o `**`) son las menos específicas.

Generación de Direcciones URL

La generación de direcciones URL es el proceso por el cual el enrutamiento puede **crear una ruta de URL** basándose en un conjunto de valores de ruta. Esto permite una separación lógica entre los puntos de conexión y las URL que los acceden.

La API `LinkGenerator`, disponible como servicio `singleton` desde la DI, es la base de la generación de enlaces en ASP.NET Core. `Mvc.IUrlHelper` (y los asistentes de etiquetas o HTML) utilizan internamente `LinkGenerator`.

Puedes usar métodos de extensión como `GetPathByAction`, `GetUriByAction`, `GetPathByPage`, y `GetUriByPage` para generar enlaces a controladores o Razor Pages.

- Los métodos `GetPath*` generan una ruta de acceso absoluta (ej. `/Products/List`).
- Los métodos `GetUri*` generan un URI absoluto (incluyendo esquema y host).

Ejemplo de Uso en Middleware:

```
public class ProductsMiddleware
{
   private readonly LinkGenerator _linkGenerator;

   public ProductsMiddleware(RequestDelegate next, LinkGenerator linkGenerator) =>
      _linkGenerator = linkGenerator;

   public async Task InvokeAsync(HttpContext httpContext)
   {
      httpContext.Response.ContentType = MediaTypeNames.Text.Plain;
      var productsPath = _linkGenerator.GetPathByAction("Products", "Store"); // Genera un
enlace a la acción Products del controlador Store
      await httpContext.Response.WriteAsync(
         $"Go to {productsPath} to see our products.");
   }
}
```

Plantillas de Ruta

Las plantillas de ruta utilizan tokens entre llaves `{}` para definir **parámetros de ruta** que se enlazan con los valores de la URL.

Características Clave:

- **Parámetros de Ruta**: `/{id}`. Deben tener un nombre y, opcionalmente, pueden incluir atributos adicionales. Múltiples parámetros en un segmento deben estar separados por un valor literal (ej., `/{param1}-{param2}`).
- **Texto Literal**: Cualquier texto fuera de `{}` debe coincidir exactamente con la URL (sin distinción de mayúsculas y minúsculas). Para que las llaves literales coincidan, duplícalas (ej., `{{` o `}}`).
- **Parámetros Comodín (`*` o `**`)**:

    - Pueden usarse como prefijo en un parámetro de ruta (ej., `blog/{**slug}`).
    - Enlazan el **resto del URI** al parámetro.
    - Pueden coincidir con una cadena vacía.
    - Los parámetros comodín escapan los caracteres de barra (`/`) al generar URL (ej., `foo/{*path}` con `{ path = "my/path" }` genera `foo/my%2Fpath`).

El enrutamiento es una parte integral y altamente configurable de ASP.NET Core, esencial para definir cómo interactúan las solicitudes entrantes con la lógica de tu aplicación.

Códigos de Estado de Respuesta HTTP: Una Guía Esencial

Los **códigos de estado de respuesta HTTP** son números de 3 dígitos que el servidor envía en la respuesta a una solicitud HTTP, indicando el resultado de esa solicitud. Se agrupan en cinco categorías, cada una con un rango numérico específico.

1. Respuestas Informativas (100–199)

Estos códigos indican que la solicitud ha sido recibida y el proceso continúa. Son respuestas provisionales.

- **100 Continue**: El cliente debe continuar con su solicitud.
- **101 Switching Protocol**: El servidor acepta cambiar al protocolo solicitado.
- **102 Processing (WebDAV)**: El servidor ha recibido y está procesando la solicitud, sin respuesta aún disponible.
- **103 Early Hints**: Permite que el cliente comience a precargar recursos mientras el servidor prepara la respuesta principal.

2. Respuestas Satisfactorias (200–299)

Indican que la solicitud ha sido recibida, comprendida y aceptada con éxito.

- **200 OK**: La solicitud se ha completado con éxito. Es el código más común para una respuesta exitosa.
- **201 Created**: La solicitud ha tenido éxito y se ha creado un nuevo recurso. Típicamente usado después de peticiones `POST` o `PUT` que crean algo.
- **202 Accepted**: La solicitud ha sido aceptada para procesamiento, pero no ha sido completada aún. No hay garantía de que la acción sea realizada.
- **204 No Content**: La solicitud ha tenido éxito, pero no hay contenido que devolver en el cuerpo de la respuesta. Útil para actualizaciones en las que el cliente no necesita recargar la página.
- **205 Reset Content**: La solicitud se completó sin contenido, y el cliente debe reiniciar la vista del documento que envió la solicitud (útil para borrar formularios).
- **206 Partial Content**: Se está sirviendo solo una parte del recurso debido a una solicitud de rango.

3. Redirecciones (300–399)

Estos códigos informan al cliente que necesita realizar una acción adicional para completar la solicitud, generalmente dirigiéndolo a una nueva URL.

- **300 Multiple Choice**: La solicitud tiene múltiples respuestas posibles.
- **301 Moved Permanently**: El recurso ha sido movido permanentemente a una nueva URL. Las futuras solicitudes deben usar la nueva URL.
- **302 Found**: El recurso ha sido movido temporalmente a una nueva URL. El cliente debe seguir usando la URL original para futuras solicitudes.
- **303 See Other**: El servidor indica que el cliente debe obtener el recurso solicitado desde una URL diferente usando un método `GET`.
- **304 Not Modified**: Indica al cliente que la versión en caché del recurso sigue siendo válida y no necesita ser transferida nuevamente.
- **307 Temporary Redirect**: Similar a 302 Found, pero especifica que el **método HTTP no debe cambiar** en la redirección.
- **308 Permanent Redirect**: Similar a 301 Moved Permanently, pero especifica que el **método HTTP no debe cambiar** en la redirección.

4. Errores de los Clientes (400–499)

Estos códigos indican que ha habido un error en la solicitud del cliente que impide que el servidor la procese.

- **400 Bad Request**: El servidor no pudo entender la solicitud debido a una sintaxis inválida.
- **401 Unauthorized**: La autenticación es necesaria para obtener la respuesta. El cliente no ha proporcionado credenciales válidas.
- **403 Forbidden**: El cliente no tiene permisos para acceder al contenido, incluso si se autentica. El servidor se niega a otorgar el acceso.
- **404 Not Found**: El servidor no pudo encontrar el recurso solicitado. Es uno de los errores más comunes en la web.
- **405 Method Not Allowed**: El método HTTP utilizado en la solicitud no está permitido para el recurso.
- **406 Not Acceptable**: El servidor no puede producir una respuesta que coincida con los criterios de `Accept` del cliente.
- **408 Request Timeout**: El servidor no recibió una solicitud completa del cliente dentro del tiempo especificado.
- **409 Conflict**: La solicitud entra en conflicto con el estado actual del servidor (ej., un intento de actualizar un recurso que ha sido modificado por otro usuario).
- **410 Gone**: El recurso ha sido eliminado permanentemente del servidor y no volverá.
- **413 Payload Too Large**: La entidad de la solicitud es más grande de lo que el servidor está dispuesto o puede procesar.
- **414 URI Too Long**: La URL proporcionada es demasiado larga para que el servidor la procese.
- **415 Unsupported Media Type**: El formato de los datos de la solicitud no es compatible con el servidor para ese recurso o método.
- **429 Too Many Requests**: El usuario ha enviado demasiadas solicitudes en un período de tiempo dado (limitación de tasa).
- **451 Unavailable For Legal Reasons**: El recurso no está disponible debido a razones legales (ej., censura).

5. Errores de los Servidores (500–599)

Estos códigos indican que el servidor encontró una condición inesperada que le impidió cumplir con la solicitud. El error no es del cliente.

- **500 Internal Server Error**: El servidor ha encontrado una situación que no sabe cómo manejar. Es un error genérico del lado del servidor.
- **501 Not Implemented**: El servidor no soporta el método HTTP solicitado o carece de la capacidad para cumplir la solicitud.
- **502 Bad Gateway**: El servidor, actuando como puerta de enlace o proxy, recibió una respuesta inválida del servidor *upstream* (el servidor al que intentaba acceder).
- **503 Service Unavailable**: El servidor no está listo para manejar la solicitud, a menudo debido a mantenimiento o sobrecarga. Generalmente, se espera que sea temporal.
- **504 Gateway Timeout**: El servidor, actuando como puerta de enlace o proxy, no recibió una respuesta a tiempo del servidor *upstream*.
- **505 HTTP Version Not Supported**: El servidor no soporta la versión del protocolo HTTP utilizada en la solicitud.

Comprender estos códigos es fundamental para el desarrollo web, ya que permiten diagnosticar problemas, depurar aplicaciones y construir APIs y sitios web que se comuniquen de manera efectiva y significativa con los clientes.

Tipos de Retorno de la Acción del Controlador en la API Web de ASP.NET Core

En el desarrollo de API web con ASP.NET Core, elegir el **tipo de retorno adecuado para las acciones de su controlador** es crucial para la claridad, la eficiencia y la correcta

documentación. ASP.NET Core ofrece cuatro opciones principales, cada una con sus propias ventajas y casos de uso:

1. **Tipo Específico (`T`)**
2. **`IActionResult`**
3. **`ActionResult<T>`**
4. **`HttpResults` (interfaz `IResult` y `Results<T>`)**

Exploremos cuándo usar cada uno.

1. Tipo Específico (`T`)

Esta es la opción más sencilla y directa. Una acción de controlador devuelve directamente un tipo de dato primitivo (como `string`, `int`) o un objeto complejo personalizado (como un objeto `Product`).

Cuándo usarlo: Es ideal cuando la acción **siempre devuelve un único tipo de datos** y no hay condiciones conocidas que puedan llevar a diferentes códigos de estado HTTP (ej., errores de validación, recursos no encontrados).

Ejemplo:

```
[HttpGet]
public Task<List<Product>> Get() =>
    _productContext.Products.OrderBy(p => p.Name).ToListAsync();
```

En este caso, la acción simplemente devuelve una lista de productos. Si no se encuentran productos, devolverá una lista vacía, no un error HTTP.

Consideraciones sobre `IEnumerable<T>` o `IAsyncEnumerable<T>`:

Cuando se devuelven colecciones, ASP.NET Core tiende a **almacenar en búfer** el resultado antes de escribirlo en la respuesta. Para mejorar el rendimiento y garantizar la **iteración asincrónica** (especialmente si los datos se obtienen de forma asíncrona), considere devolver **`IAsyncEnumerable<T>`**.

- Con `System.Text.Json`, `IAsyncEnumerable<T>` permite el *streaming* del resultado directamente a la respuesta.
- Con `Newtonsoft.Json` o formateadores basados en XML, el resultado se seguirá almacenando en el búfer.

Ejemplo con `IAsyncEnumerable<Product>`:

```
[HttpGet("asyncsale")]
public async IAsyncEnumerable<Product> GetOnSaleProductsAsync()
{
    var products = _productContext.Products.OrderBy(p => p.Name).AsAsyncEnumerable();

    await foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product; // Permite el streaming de productos en oferta
        }
    }
}
```

2. Tipo `IActionResult`

`IActionResult` es un tipo de retorno más flexible, adecuado cuando una acción puede devolver **múltiples tipos de resultados `ActionResult`** y, por lo tanto, **múltiples códigos de estado HTTP** diferentes. Los tipos `ActionResult` representan resultados de acciones específicas que corresponden a códigos de estado HTTP (ej., `OkResult` para 200, `NotFoundResult` para 404, `BadRequestResult` para 400).

Los métodos de conveniencia en la clase `ControllerBase` (como `Ok()`, `NotFound()`, `BadRequest()`, `CreatedAtAction()`) son atajos para crear y devolver estas instancias de `ActionResult`.

Cuándo usarlo: Cuando una acción tiene **múltiples rutas de ejecución** que pueden resultar en diferentes códigos de estado HTTP, o cuando la acción no devuelve un tipo de datos fijo pero indica el resultado de la operación (ej., éxito, no encontrado, error).

Atributo `[ProducesResponseType]`: Es fundamental usar el atributo `[ProducesResponseType]` con `IActionResult`. Este atributo documenta explícitamente los tipos de respuesta y códigos de estado HTTP que la acción puede devolver, lo cual es invaluable para herramientas de ayuda de API como Swagger/OpenAPI.

Ejemplos:

```
[HttpGet("{id}")]
[ProducesResponseType<Product>(StatusCodes.Status200OK)] // Documenta que puede devolver un Product con 200 OK
[ProducesResponseType(StatusCodes.Status404NotFound)]   // Documenta que puede devolver un 404 Not Found
public IActionResult GetById_IActionResult(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? NotFound() : Ok(product);
}

[HttpPost()]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)] // Documenta que puede devolver un 201 Created
[ProducesResponseType(StatusCodes.Status400BadRequest)] // Documenta que puede devolver un 400 Bad Request
public async Task<IActionResult> CreateAsync_IActionResult(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest(); // Devuelve 400 Bad Request
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();
    return CreatedAtAction(nameof(GetById_IActionResult), new { id = product.Id }, product); // Devuelve 201 Created
}
```

Nota: Si el atributo `[ApiController]` está aplicado, los errores de validación de modelos generarán automáticamente un 400 Bad Request, eliminando la necesidad de una comprobación `ModelState.IsValid` explícita.

3. Tipo `ActionResult<T>`

`ActionResult<T>` es un tipo de retorno genérico que combina la flexibilidad de `IActionResult` con la tipificación fuerte de un tipo específico. Permite devolver **un tipo específico (`T`) o cualquier tipo derivado de `ActionResult`**.

Cuándo usarlo: Es la opción preferida para la mayoría de las acciones de API web que pueden devolver tanto un resultado exitoso (con datos específicos) como diferentes errores HTTP. Ofrece un buen equilibrio entre flexibilidad y claridad.

Ventajas:

- **Inferencia de `[ProducesResponseType]`**: El tipo esperado de la acción se infiere automáticamente de `T`, lo que a menudo elimina la necesidad de especificar `Type = typeof(T)` en `[ProducesResponseType]`.
- **Operadores de Conversión Implícitos**: Permite convertir directamente tanto `T` como `ActionResult` a `ActionResult<T>`, simplificando el código. Por ejemplo, `return product;` se convierte implícitamente en `return new OkObjectResult(product);` y `return NotFound();` funciona directamente.

Ejemplos:

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)] // Infiere tipo Product
[ProducesResponseType(StatusCodes.Status404NotFound)]
public ActionResult<Product> GetById_ActionResultOfT(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? NotFound() : product; // Retorna NotFoundResult o el objeto
Product (implícitamente un OkObjectResult)
}

[HttpPost()]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<Product>> CreateAsync_ActionResultOfT(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest(); // Retorna 400 Bad Request
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();
    return CreatedAtAction(nameof(GetById_ActionResultOfT), new { id = product.Id },
product); // Retorna 201 Created
}
```

4. Tipo `HttpResults` (`IResult` y `Results<T>`)

Los tipos `HttpResults` son una adición más reciente a ASP.NET Core, diseñados para ser utilizados tanto en **API mínimas** como en **API Web con controladores**. Implementan la interfaz `IResult` y representan el resultado de un punto de conexión HTTP. La clase estática `Results` se utiliza para crear distintas instancias de `IResult`.

Cuándo usarlo: Cuando se busca la máxima **consistencia de código entre API mínimas y API web basadas en controladores**, o cuando se desea un control más explícito sobre la respuesta HTTP sin depender de los formateadores de MVC (y, por lo tanto, sin negociación de contenido).

Características:

- Son una implementación de resultados que se procesa llamando a `IResult.ExecuteAsync`.
- **NO aprovechan los formateadores configurados de MVC**, lo que significa que la negociación de contenido no está disponible y la implementación de `HttpResults` decide el `Content-Type` producido.

`IResult` (Interfaz y clase `Results`)

```
[HttpGet("{id}")]
```

```
[ProducesResponseType<Product>(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IResult GetById(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? Results.NotFound() : Results.Ok(product); // Usa
Results.NotFound() y Results.Ok()
}

[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType<Product>(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IResult> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return Results.BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    var location = Url.Action(nameof(CreateAsync), new { id = product.Id }) ?? $"/{product.Id}";
    return Results.Created(location, product); // Usa Results.Created()
}
```

`Results<TResult1, TResultN>` (Uniones Tipadas con `TypedResults`)

Esta es una mejora sobre `IResult` cuando se tienen múltiples tipos de retorno posibles. `Results<TResult1, TResultN>` es un tipo de unión genérico que **mantiene automáticamente los metadatos del punto de conexión**, eliminando la necesidad de `[ProducesResponseType]` y proporcionando **comprobación en tiempo de compilación** de que la acción solo devuelve los tipos declarados. Se utiliza la clase estática `TypedResults` para crear las instancias.

Ventajas:

- **Autodocumentación**: Los metadatos del punto de conexión se generan automáticamente, lo que puede eliminar la necesidad de `[ProducesResponseType]`.
- **Seguridad de Tipo en Tiempo de Compilación**: El compilador verifica que solo se devuelvan los tipos declarados en la unión.

Ejemplos:

```
[HttpGet("{id}")]
// No se necesita ProducesResponseType aquí gracias a la inferencia de Results<NotFound,
Ok<Product>>
public Results<NotFound, Ok<Product>> GetById(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? TypedResults.NotFound() : TypedResults.Ok(product);
}
[HttpPost]
public async Task<Results<BadRequest, Created<Product>>> CreateAsync(Product
product)
{
    if (product.Description.Contains("XYZ Widget"))
```

```
    {
        return TypedResults.BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    var location = Url.Action(nameof(CreateAsync), new { id = product.Id }) ?? $"/{product.Id}";
    return TypedResults.Created(location, product);
}
```

Elegir el tipo de retorno adecuado dependerá de la complejidad de su acción de API y de cuántos tipos de respuesta HTTP diferentes pueda generar. Para la mayoría de los casos de uso, **`ActionResult<T>`** es una excelente opción que equilibra la flexibilidad con la tipificación fuerte. Si busca la máxima reutilización de código entre controladores y API mínimas, **`HttpResults`** puede ser la clave.

Aplicación de Formato a Datos de Respuesta en ASP.NET Core Web API

ASP.NET Core Web API ofrece potentes capacidades para formatear los datos de respuesta, permitiéndote controlar cómo se presentan los datos a los clientes. Esto se puede lograr mediante formatos específicos, negociación de contenido basada en las preferencias del cliente, o configuraciones globales.

1. Resultados de Acción Específicos del Formato

Puedes hacer que una acción siempre devuelva datos en un formato particular, ignorando las preferencias del cliente.

`JsonResult`: Devuelve datos siempre en formato JSON.

```
[HttpGet]
public IActionResult Get() =>
    Ok(_todoItemStore.GetList()); // Por defecto, Ok() devuelve JSON
```

- Si usas `Ok()` (un método auxiliar de `ControllerBase`), la respuesta tendrá por defecto el encabezado `Content-Type: application/json; charset=utf-8`.

`ContentResult`: Devuelve datos de cadena con formato de texto sin formato.

```
[HttpGet("Version")]
public ContentResult GetVersion()
    => Content("v1.0.0"); // Devuelve text/plain
```

- La respuesta tendrá el encabezado `Content-Type: text/plain`.

Devolviendo Objetos POCO (Plain Old CLR Objects): Si una acción devuelve directamente un objeto que no es un `IActionResult`, ASP.NET Core lo serializará automáticamente usando el `IOutputFormatter` apropiado.

```
[HttpGet("{id:long}")]
public TodoItem? GetById(long id)
    => _todoItemStore.GetById(id);
```

- En este caso, si `TodoItem` se encuentra, se devolverá un `200 OK` con el objeto serializado (por defecto JSON). Si `TodoItem` es `null`, se devolverá un `204 No Content` (a menos que se cambie la configuración del formateador).

2. Negociación de Contenido

La negociación de contenido es el proceso mediante el cual el servidor decide el formato de la respuesta basándose en el **encabezado `Accept`** enviado por el cliente en la solicitud HTTP.

- **Implementación**: La negociación de contenido es implementada por `ObjectResult` y se integra en los métodos auxiliares de resultado de acción (como `Ok()`, `NotFound()`, etc.) que internamente usan `ObjectResult`.
- **Tipos de Medios por Defecto**: Por defecto, ASP.NET Core soporta:
  - `application/json`
  - `text/json`
  - `text/plain`

Cómo funciona la negociación:

1. Cuando la solicitud tiene un encabezado `Accept`, ASP.NET Core enumera los tipos de medios en orden de preferencia del cliente.
2. Intenta encontrar un formateador que pueda generar una respuesta en uno de esos formatos.
3. **Si no hay un formateador que satisfaga la solicitud:**
   - Devuelve `406 Not Acceptable` si `MvcOptions.ReturnHttpNotAcceptable` se establece en `true`.
   - De lo contrario, intenta encontrar el primer formateador que pueda formatear la respuesta.
4. **Si no hay encabezado `Accept` o si contiene `*/*` (y `RespectBrowserAcceptHeader` es `false`):**
   - El primer formateador que puede manejar el objeto se usa para serializar la respuesta. No hay negociación, el servidor decide.

Exploradores y Negociación de Contenido: Por defecto, ASP.NET Core **ignora los encabezados `Accept` de los navegadores web** (que a menudo especifican muchos formatos y comodines) y devuelve el contenido en JSON. Esto busca una experiencia más consistente. Para que la aplicación **respete los encabezados `Accept` del navegador**, debe establecer `options.RespectBrowserAcceptHeader = true` en `AddControllers()`:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers(options =>
{
    options.RespectBrowserAcceptHeader = true;
});
```

3. Configuración de Formateadores

Puedes agregar soporte para formatos adicionales o configurar el comportamiento de los formateadores existentes.

Adición de Soporte XML:

Para usar formateadores XML implementados con `XmlSerializer`:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers()
    .AddXmlSerializerFormatters(); // Agrega soporte para XML
```

Con esto, si el cliente envía `Accept: application/xml`, obtendrá una respuesta XML.

Configuración de Formateadores Basados en `System.Text.Json`:

Para configurar las opciones de serialización predeterminadas (ej., cambiar de `camelCase` a `PascalCase` para nombres de propiedades):

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers()
    .AddJsonOptions(options =>
```

```
    {
        options.JsonSerializerOptions.PropertyNamingPolicy = null; // Deshabilita camelCase
    });
```

Para acciones específicas, puedes pasar `JsonSerializerOptions` directamente a `JsonResult`:

```
[HttpGet]
public IActionResult Get()
    => new JsonResult(
        _todoItemStore.GetList(),
        new JsonSerializerOptions { PropertyNamingPolicy = null });
```

Adición de Soporte JSON Basado en `Newtonsoft.Json`:

Si necesitas características específicas de `Newtonsoft.Json` (como atributos `[JsonProperty]`, configuraciones de serialización complejas o `JsonPatch`), instala el paquete `Microsoft.AspNetCore.Mvc.NewtonsoftJson` y configúralo:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers()
    .AddNewtonsoftJson(options =>
    {
        options.SerializerSettings.ContractResolver = new DefaultContractResolver(); // Ejemplo de configuración
    });
```

De manera similar, puedes configurar opciones para acciones específicas usando `JsonResult` con `JsonSerializerSettings`.

Formato de Respuestas `ProblemDetails` y `ValidationProblemDetails`:

- **`ProblemDetails` (RFC 7807)**: Las respuestas de `ProblemDetails` (usadas por `ControllerBase.Problem()` y para errores de servidor) siempre son `camelCase` por especificación, independientemente de la configuración global de nomenclatura.

- **`ValidationProblemDetails`**: Cuando se aplica el atributo `[ApiController]` y la validación de modelos falla, se genera automáticamente una respuesta `ValidationProblemDetails`. Por defecto, las claves de error en el diccionario `errors` usan los nombres de propiedad del modelo sin cambios.

Para formato con `System.Text.Json`: Agrega `SystemTextJsonValidationMetadataProvider` a `MvcOptions.ModelMetadataDetailsProviders` para formatear las claves de error (por defecto `camelCase`):

```
builder.Services.AddControllers();
builder.Services.Configure<MvcOptions>(options =>
{
    options.ModelMetadataDetailsProviders.Add(
        new SystemTextJsonValidationMetadataProvider());
});
```

    - Puedes usar `[JsonPropertyName("customName")]` en tu modelo para sobrescribir el nombre de la clave de error.

Para formato con `Newtonsoft.Json`: Si usas `AddNewtonsoftJson()`, agrega `NewtonsoftJsonValidationMetadataProvider` para formatear las claves de error (por

defecto `camelCase`):

```
builder.Services.AddControllers()
    .AddNewtonsoftJson();
builder.Services.Configure<MvcOptions>(options =>
{
    options.ModelMetadataDetailsProviders.Add(
        new NewtonsoftJsonValidationMetadataProvider());
});
```

- Puedes usar `[JsonProperty("customName")]` en tu modelo para sobrescribir el nombre de la clave de error.

4. Especificación de un Formato (Filtro `[Produces]`)

Puedes forzar un formato de respuesta específico para una acción, un controlador completo o globalmente usando el filtro `[Produces]`.

```
[ApiController]
[Route("api/[controller]")]
[Produces("application/json")] // Fuerza JSON para todo el controlador
public class TodoItemsController : ControllerBase
{
    // ...
}
```

Este filtro asegura que las respuestas siempre serán JSON, incluso si el cliente solicita otro formato o si hay otros formateadores configurados.

5. Formateadores de Casos Especiales

- **`string` como `text/plain`**: Por defecto, los valores de retorno `string` se formatean como `text/plain` (o `text/html` si el navegador lo solicita). Puedes eliminar `StringOutputFormatter` para que el formateador JSON (o XML si JSON no está presente) maneje las cadenas.
- **`null` a `204 No Content`**: Por defecto, las acciones que devuelven un objeto de modelo (POCO) y cuyo valor es `null` resultan en una respuesta `204 No Content`. Puedes eliminar `HttpNoContentOutputFormatter` para que los objetos `null` sean formateados por el formateador configurado (ej., JSON devuelve `null` en el cuerpo).

Para eliminar estos formateadores:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers(options =>
{
    options.OutputFormatters.RemoveType<StringOutputFormatter>();
    options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

6. Asignaciones de URL de Formato de Respuesta (`[FormatFilter]`)

Los clientes pueden solicitar un formato particular como parte de la URL (ej., usando una extensión de archivo como `.json` o `.xml`). El atributo `[FormatFilter]` permite que la API reconozca y use esta extensión para seleccionar el formateador apropiado.

```
[ApiController]
[Route("api/[controller]")]
[FormatFilter] // Habilita el filtro de formato
public class TodoItemsController : ControllerBase
{
    private readonly TodoItemStore _todoItemStore;
```

```
    public TodoItemsController(TodoItemStore todoItemStore)
        => _todoItemStore = todoItemStore;

    [HttpGet("{id:long}.{format?}")] // Permite una extensión opcional en la URL
    public TodoItem? GetById(long id)
        => _todoItemStore.GetById(id);
}
```

- `/api/todoitems/5` : Usa el formateador de salida predeterminado (negociación de contenido o JSON si no hay `Accept` específico).
- `/api/todoitems/5.json` : Fuerza el uso del formateador JSON.
- `/api/todoitems/5.xml` : Fuerza el uso del formateador XML (si está configurado).

7. Deserialización Polimórfica

Mientras que las características integradas de ASP.NET Core proporcionan una serialización polimórfica limitada, **no admiten la deserialización polimórfica por defecto**. Para esto, generalmente se requiere la implementación de un convertidor personalizado.

Control de Errores en las API Web Basadas en el Controlador de ASP.NET Core

El manejo adecuado de errores es fundamental para construir API web robustas y amigables para el cliente. ASP.NET Core ofrece varias estrategias para capturar, personalizar y responder a errores en sus controladores de API web.

1. Página de Excepciones para el Desarrollador (`DeveloperExceptionPage`)

La **Página de Excepciones para el Desarrollador** es una herramienta invaluable durante la fase de desarrollo. Muestra información detallada sobre excepciones no controladas, incluyendo el seguimiento de la pila, parámetros de consulta, cookies, encabezados y metadatos del punto de conexión.

- **Activación**: Por defecto, se activa automáticamente en el **entorno de desarrollo** para aplicaciones creadas con `WebApplication.CreateBuilder`. Las aplicaciones más antiguas pueden activarla con `app.UseDeveloperExceptionPage()`.
- **Precaución**: **Nunca la habilite en entornos de producción** para evitar exponer información sensible a los usuarios.
- **Comportamiento**: Si la solicitud tiene un encabezado `Accept: text/plain`, la página devuelve texto sin formato en lugar de HTML.

Ejemplo de uso: Para verla en acción, puede agregar una acción de controlador que lance una excepción y luego ejecutar la aplicación en el entorno de desarrollo y acceder a ese punto de conexión.

```
[HttpGet("Throw")]
public IActionResult Throw() =>
    throw new Exception("Sample exception.");
```

2. Controlador de Excepciones (`UseExceptionHandler`)

Para entornos que no son de desarrollo (producción, staging, etc.), es crucial utilizar un mecanismo de control de errores que no revele detalles internos del servidor. El **middleware de control de excepciones** (`UseExceptionHandler`) es la solución recomendada para producir una carga de error consistente y segura.

Configuración en `Program.cs`:

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

```
app.UseHttpsRedirection();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error"); // Redirige a una ruta de error
}

app.UseAuthorization();
app.MapControllers();
app.Run();
```

Acción de Controlador para el Error: Debe definir una acción de controlador que responda a la ruta especificada (`/error` en el ejemplo). La forma más sencilla es usar `Problem()` para devolver una carga compatible con **RFC 7807** (Problem Details).

```
[Route("/error")]
[ApiExplorerSettings(IgnoreApi = true)] // Excluye de la documentación OpenAPI/Swagger
public IActionResult HandleError() =>
    Problem(); // Devuelve una respuesta Problem Details estándar
```

Importante: No marque la acción del controlador de errores con atributos de método HTTP (como `[HttpGet]`) para asegurar que pueda manejar cualquier verbo HTTP. También, permita acceso anónimo si los usuarios no autenticados deben recibir el error.

Controladores de Excepciones por Entorno: Puede tener controladores de excepciones diferentes para desarrollo y producción:

```
if (app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error-development");
}
else
{
    app.UseExceptionHandler("/error");
}
```

Y luego, acciones de controlador correspondientes que, en desarrollo, pueden revelar más detalles de la excepción:

```
C#
[Route("/error-development")]
public IActionResult HandleErrorDevelopment(
    [FromServices] IHostEnvironment hostEnvironment)
{
    if (!hostEnvironment.IsDevelopment())
    {
        return NotFound(); // Asegura que no se muestre en producción
    }

    var exceptionHandlerFeature =
        HttpContext.Features.Get<IExceptionHandlerFeature>()!;

    return Problem(
        detail: exceptionHandlerFeature.Error.StackTrace, // Detalles del stack trace
        title: exceptionHandlerFeature.Error.Message);   // Mensaje de la excepción
}

[Route("/error")]
public IActionResult HandleError() =>
    Problem();
```

3. Uso de Excepciones para Modificar la Respuesta

Puedes modificar el contenido de la respuesta de error de forma más flexible utilizando **excepciones personalizadas y filtros de acción**.

Cree una Excepción Personalizada:

```
public class HttpResponseException : Exception
{
    public HttpResponseException(int statusCode, object? value = null) =>
        (StatusCode, Value) = (statusCode, value);

    public int StatusCode { get; }
    public object? Value { get; }
}
```

1. **Cree un Filtro de Acción**: Este filtro interceptará la excepción personalizada y generará una respuesta HTTP específica.

   Fragmento de código

```
public class HttpResponseExceptionFilter : IActionFilter, IOrderedFilter
{
    public int Order => int.MaxValue - 10; // Se ejecuta casi al final

    public void OnActionExecuting(ActionExecutingContext context) { }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        if (context.Exception is HttpResponseException httpResponseException)
        {
            context.Result = new ObjectResult(httpResponseException.Value)
            {
                StatusCode = httpResponseException.StatusCode
            };
            context.ExceptionHandled = true; // Marca la excepción como manejada
        }
    }
}
```

2. **Registre el Filtro**: En `Program.cs`, añada el filtro a la colección de filtros de MVC.

```
builder.Services.AddControllers(options =>
{
    options.Filters.Add<HttpResponseExceptionFilter>();
});
```

3. Ahora, cualquier controlador puede lanzar `HttpResponseException` y el filtro la convertirá en una respuesta HTTP.

4. Respuesta de Error Ante Errores de Validación

Cuando la **validación del modelo falla** en los controladores de API web (especialmente con `[ApiController]`), MVC responde por defecto con un tipo de respuesta `ValidationProblemDetails`.

Personalización de `InvalidModelStateResponseFactory`: Puedes reemplazar la fábrica predeterminada para personalizar cómo se construyen estas respuestas. Por ejemplo, para soportar formato XML además de JSON:

```
builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
            new BadRequestObjectResult(context.ModelState)
            {
                ContentTypes =
```

```
            {
                System.Net.Mime.MediaTypeNames.Application.Json,
                System.Net.Mime.MediaTypeNames.Application.Xml
            }
        };
    })
    .AddXmlSerializerFormatters(); // Necesario si quieres XML
```

Formato de claves de error en `ValidationProblemDetails`: Por defecto, las claves en el diccionario `errors` usan los nombres de propiedad del modelo sin cambios. Para formatearlas (ej., a `camelCase` por defecto con `System.Text.Json`):

C#

```
builder.Services.Configure<MvcOptions>(options =>
{
    options.ModelMetadataDetailsProviders.Add(
        new SystemTextJsonValidationMetadataProvider());
});
```

Si usa `Newtonsoft.Json`, use `NewtonsoftJsonValidationMetadataProvider`. También puede usar `[JsonPropertyName("customName")]` (para System.Text.Json) o `[JsonProperty("customName")]` (para Newtonsoft.Json) para personalizar nombres de propiedades individuales.

5. Respuesta de Error del Cliente (`ProblemDetails` y `IProblemDetailsService`)

Un "resultado de error del cliente" se define como cualquier resultado con un código de estado HTTP de **400 o superior**. ASP.NET Core transforma automáticamente los resultados de error en `ProblemDetails` (compatibles con RFC 7807) para los controladores de API web.

Servicio de Detalles del Problema (`AddProblemDetails`)

`IProblemDetailsService` permite la creación de respuestas `ProblemDetails` para API HTTP. Al agregar `AddProblemDetails()` y `UseExceptionHandler()`/`UseStatusCodePages()`, su aplicación generará automáticamente respuestas `ProblemDetails` para errores de cliente y servidor que aún no tienen contenido en el cuerpo.

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddProblemDetails(); // Habilita el servicio de ProblemDetails

var app = builder.Build();
app.UseExceptionHandler(); // Captura excepciones y usa ProblemDetails
app.UseStatusCodePages();  // Maneja códigos de estado de error (400+) sin contenido
// ...
app.MapControllers();
app.Run();
```

Con esta configuración, tanto la entrada inválida de controladores como las excepciones no controladas o URIs no coincidentes generarán una respuesta `ProblemDetails`.

Desactivar `ProblemDetails` automático (`SuppressMapClientErrors`): Puedes deshabilitar la creación automática de `ProblemDetails` para errores de cliente estableciendo `SuppressMapClientErrors` en `true`:

```
builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
```

```
    options.SuppressMapClientErrors = true;
});
```

- Esto hará que, por ejemplo, un `BadRequest()` devuelva un `HTTP 400` sin cuerpo de respuesta.

Personalización de `ProblemDetails` (`CustomizeProblemDetails`): Puedes personalizar el contenido de `ProblemDetails` utilizando `CustomizeProblemDetails` en `ProblemDetailsOptions`. Esto te permite añadir detalles específicos o cambiar el tipo y título basándose en el contexto del error.

```
builder.Services.AddProblemDetails(options =>
    options.CustomizeProblemDetails = (context) =>
    {
        // Lógica para personalizar ProblemDetails
        // Accede a context.ProblemDetails para modificar sus propiedades
        // Puedes usar HttpContext.Features para pasar información adicional
    }
);
```

- **Implementar `ProblemDetailsFactory`**: Para un control más profundo, puedes registrar una implementación personalizada de `Microsoft.AspNetCore.Mvc.Infrastructure.ProblemDetailsFactory`. Esta fábrica es utilizada por MVC para generar todas las instancias de `ProblemDetails` y `ValidationProblemDetails`.
  C#
  ```
  builder.Services.AddTransient<ProblemDetailsFactory,
  SampleProblemDetailsFactory>();
  ```
- **Usar `ApiBehaviorOptions.ClientErrorMapping`**: Puedes configurar propiedades específicas de la respuesta `ProblemDetails` para códigos de estado de error individuales. Por ejemplo, para añadir un enlace específico para errores `404 Not Found`:
  C#
  ```
  builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
  ```
- Un control de errores efectivo es vital para la usabilidad y la estabilidad de su API. Al comprender y aplicar estas técnicas, puede ofrecer respuestas claras y significativas a sus clientes, independientemente de si la solicitud fue exitosa o no.

Entity Framework Core: Una Visión General

Entity Framework (EF) Core es una versión ligera, extensible, de código abierto y multiplataforma de la popular tecnología de acceso a datos Entity Framework. Actúa como un **Mapeador Objeto-Relacional (O/RM)**, lo que permite a los desarrolladores de .NET interactuar con una base de datos utilizando objetos .NET, eliminando la necesidad de escribir gran parte del código de acceso a datos tradicional. EF Core es compatible con una amplia variedad de motores de bases de datos.

El Modelo

El acceso a datos en EF Core se realiza a través de un **modelo**. Este modelo se compone de **clases de entidad** (que representan las tablas o colecciones de la base de datos) y un **objeto de contexto** (que actúa como una sesión con la base de datos, permitiendo consultar y guardar datos).

Enfoques de Desarrollo del Modelo:

1. **Generar un modelo a partir de una base de datos existente (Database First)**: Ideal si ya tienes una base de datos y quieres generar tu código .NET a partir de ella.
2. **Codificar un modelo a mano para que coincida con la base de datos (Code First)**: Permite definir tus entidades en código y luego generar la base de datos a partir de ese código.
3. **EF Migrations**: Una vez que tienes un modelo, puedes usar las Migraciones de EF para crear y evolucionar la base de datos a medida que tu modelo cambia.

Ejemplo de Modelo (BloggingContext, Blog, Post):

```
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace Intro;

public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; } // Representa la tabla Blogs
    public DbSet<Post> Posts { get; set; } // Representa la tabla Posts

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer( // Configura para usar SQL Server

@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;ConnectRet
ryCount=0");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
    public List<Post> Posts { get; set; } // Propiedad de navegación para Posts relacionados
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; } // Clave foránea a Blog
    public Blog Blog { get; set; } // Propiedad de navegación a Blog
}
```

Consultando Datos

Las instancias de tus clases de entidad se recuperan de la base de datos utilizando **Language Integrated Query (LINQ)**, lo que te permite escribir consultas en C# que se traducen a SQL.

Ejemplo de Consulta LINQ:

```
using (var db = new BloggingContext())
{
    var blogs = await db.Blogs
```

```
        .Where(b => b.Rating > 3) // Filtra blogs con rating mayor a 3
        .OrderBy(b => b.Url)     // Ordena por URL
        .ToListAsync();          // Ejecuta la consulta y obtiene la lista de forma asíncrona
}
```

Guardando Datos

La creación, eliminación y modificación de datos en la base de datos se realiza manipulando instancias de tus clases de entidad y luego llamando a `SaveChanges()` en el contexto.

Ejemplo de Guardado de Datos:

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog); // Agrega un nuevo blog al contexto
    await db.SaveChangesAsync(); // Guarda los cambios en la base de datos
}
```

Consideraciones sobre EF O/RM

Aunque EF Core simplifica muchos detalles de programación, es crucial tener en cuenta algunas mejores prácticas aplicables a cualquier O/RM para evitar problemas comunes en aplicaciones de producción:

- **Conocimiento de la Base de Datos Subyacente**: Un conocimiento intermedio o avanzado del servidor de base de datos es esencial para diseñar, depurar, perfilar y migrar datos en aplicaciones de alto rendimiento. Esto incluye entender claves primarias y foráneas, índices, normalización, sentencias DML/DDL, tipos de datos, etc.
- **Pruebas Funcionales y de Integración**: Es vital replicar el entorno de producción lo más fielmente posible para:
    - Encontrar problemas que solo aparecen con versiones o ediciones específicas del servidor de base de datos.
    - Detectar cambios importantes al actualizar EF Core y otras dependencias (ej., ASP.NET Core, OData, AutoMapper), que pueden afectar a EF Core de formas inesperadas.
- **Pruebas de Rendimiento y Estrés**: Realiza pruebas con cargas representativas. Algunas características usadas ingenuamente pueden no escalar bien (ej., múltiples `Includes` en colecciones, uso intensivo de carga perezosa, consultas condicionales en columnas no indexadas, actualizaciones e inserciones masivas con valores generados por la base de datos, falta de manejo de concurrencia, modelos grandes, políticas de caché inadecuadas).
- **Revisión de Seguridad**: Asegura un manejo adecuado de cadenas de conexión y otros secretos, permisos de base de datos para operaciones no de implementación, validación de entrada para SQL puro y cifrado de datos sensibles.
- **Registro y Diagnóstico**: Asegúrate de que el registro y los diagnósticos sean suficientes y utilizables (ej., configuración de registro adecuada, etiquetas de consulta, Application Insights).
- **Recuperación de Errores**: Prepara planes de contingencia para escenarios de fallo comunes, como reversiones de versiones, servidores de respaldo, escalado horizontal y equilibrio de carga, mitigación de ataques DoS y copias de seguridad de datos.
- **Implementación y Migración de Aplicaciones**: Planifica cómo se aplicarán las migraciones durante la implementación. Realizarlas al inicio de la aplicación puede causar problemas de concurrencia y requiere permisos más elevados de los necesarios para el funcionamiento normal. Usa entornos de staging para facilitar la recuperación de errores fatales durante la migración.

- **Examen Detallado y Pruebas de las Migraciones Generadas**: Las migraciones deben probarse exhaustivamente antes de aplicarse a datos de producción. La forma del esquema y los tipos de columna no pueden cambiarse fácilmente una vez que las tablas contienen datos de producción. Por ejemplo, `nvarchar(max)` y `decimal(18, 2)` son los tipos predeterminados de EF para `string` y `decimal` respectivamente, pero rara vez son los mejores tipos para tu escenario específico.

Vida útil, Configuración e Inicialización de DbContext en Entity Framework Core

Este artículo explora los patrones fundamentales para inicializar y configurar instancias de `DbContext` en Entity Framework Core (EF Core), cubriendo su ciclo de vida, integración con la inyección de dependencias de ASP.NET Core y otras formas de creación.

La Vida Útil del `DbContext`

La vida útil de una instancia de `DbContext` comienza con su creación y termina con su **disposición (`Dispose`)**. Una instancia de `DbContext` está diseñada para ser utilizada para una **única unidad de trabajo (Unit-of-Work)**, lo que significa que su vida útil suele ser muy corta.

Una Unidad de Trabajo Típica en EF Core Implica:

1. **Creación de la instancia de `DbContext`**.
2. **Seguimiento de instancias de entidad por el contexto**: Las entidades comienzan a ser rastreadas al ser devueltas de una consulta o al ser añadidas o adjuntadas al contexto.
3. **Realización de cambios** en las entidades rastreadas para implementar reglas de negocio.
4. Llamada a **`SaveChanges()`** o **`SaveChangesAsync()`** para que EF Core detecte los cambios y los persista en la base de datos.
5. **Disposición de la instancia de `DbContext`**.

Importante:

- Es crucial **disponer el `DbContext` después de su uso** para liberar recursos no administrados y anular el registro de eventos, evitando fugas de memoria.
- **`DbContext` no es seguro para subprocesos (thread-safe)**. No comparta contextos entre subprocesos. Asegúrese de `await` todas las llamadas asincrónicas antes de continuar usando la instancia del contexto.
- Una `InvalidOperationException` lanzada por código de EF Core indica un error de programa y generalmente deja el contexto en un estado irrecuperable.

`DbContext` en Inyección de Dependencias para ASP.NET Core

En aplicaciones web, cada solicitud HTTP a menudo corresponde a una unidad de trabajo única, lo que hace que vincular la vida útil del contexto a la de la solicitud sea un buen valor predeterminado.

Las aplicaciones ASP.NET Core están configuradas usando **inyección de dependencias**. EF Core se puede integrar registrándolo con `AddDbContext` en `Program.cs`:

```
var connectionString =
    builder.Configuration.GetConnectionString("DefaultConnection")
        ?? throw new InvalidOperationException("Connection string"
        + "'DefaultConnection' not found.");

builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Este código registra `ApplicationDbContext` (una subclase de `DbContext`) como un **servicio con alcance (`scoped service`)**. Esto significa que una nueva instancia de `ApplicationDbContext` se crea para cada solicitud HTTP y se inyecta en los

controladores o servicios que la requieren. La configuración del contexto (por ejemplo, el proveedor de base de datos y la cadena de conexión) se pasa a través del parámetro `DbContextOptions<ApplicationDbContext>`.

El Constructor del Contexto: La clase `ApplicationDbContext` debe exponer un constructor público que acepte un parámetro `DbContextOptions<ApplicationDbContext>`:

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Uso en un Controlador (Inyección de Constructor):

```
public class MyController
{
    private readonly ApplicationDbContext _context;

    public MyController(ApplicationDbContext context) // Inyección del DbContext
    {
        _context = context;
    }
}
```

El resultado final es que se crea una instancia de `ApplicationDbContext` para cada solicitud, se pasa al controlador para realizar una unidad de trabajo y se desecha cuando finaliza la solicitud.

Inicialización Básica de `DbContext` con `new`

También es posible construir instancias de `DbContext` directamente con la palabra clave `new`. La configuración se puede realizar sobrescribiendo el método `OnConfiguring` o pasando opciones al constructor.

Sobrescribiendo `OnConfiguring`:

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0");
    }
}
```

Pasando la Cadena de Conexión al Constructor:

```
public class ApplicationDbContext : DbContext
{
    private readonly string _connectionString;

    public ApplicationDbContext(string connectionString)
    {
        _connectionString = connectionString;
    }
```

```
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connectionString);
    }
}
```

Creando `DbContextOptions` Explícitamente: Para un `DbContext` configurado para inyección de dependencias, también se puede construir explícitamente creando un objeto `DbContextOptions` y pasándolo al constructor:

```
var contextOptions = new DbContextOptionsBuilder<ApplicationDbContext>()
    .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0")
    .Options;
using var context = new ApplicationDbContext(contextOptions); // Construcción explícita
```

Uso de una Fábrica de `DbContext` (`AddDbContextFactory`)

Algunos tipos de aplicaciones (como ASP.NET Core Blazor) o escenarios donde se necesitan **múltiples unidades de trabajo dentro del mismo alcance** (ej., varias operaciones en una sola solicitud HTTP) requieren un enfoque diferente. En estos casos, se puede usar `AddDbContextFactory` para registrar una fábrica para la creación de instancias de `DbContext`.

Configuración de la Fábrica en `Program.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContextFactory<ApplicationDbContext>(
        options => options.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0"));
}
```

La clase `ApplicationDbContext` aún debe tener el constructor que acepta `DbContextOptions<ApplicationDbContext>`.

Uso de la Fábrica en un Servicio:

```
private readonly IDbContextFactory<ApplicationDbContext> _contextFactory;
public MyController(IDbContextFactory<ApplicationDbContext> contextFactory)
{
    _contextFactory = contextFactory;
}
public async Task DoSomething()
{
    using (var context = _contextFactory.CreateDbContext()) // Crea una nueva instancia del contexto
    {
        // ...
    }
}
```

Las instancias de `DbContext` creadas de esta manera **no son administradas por el proveedor de servicios de la aplicación** y, por lo tanto, **deben ser dispuestas explícitamente** por la aplicación (usando un bloque `using` o llamando a `Dispose()`).

Opciones de `DbContext` (`DbContextOptionsBuilder`)

El punto de partida para toda la configuración de `DbContext` es `DbContextOptionsBuilder`. Se puede obtener en `AddDbContext`, en `OnConfiguring` o construyéndolo explícitamente con `new`.

Configuración del Proveedor de Base de Datos

Cada instancia de `DbContext` debe configurarse para usar **un único proveedor de base de datos**. Esto se hace con una llamada específica a `Use*` (ej., `UseSqlServer`, `UseSqlite`, `UseInMemoryDatabase`). Estos métodos son métodos de extensión proporcionados por el paquete NuGet del proveedor de base de datos.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
   optionsBuilder.UseSqlServer(
      @"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0");
}
```

Consejo: Si el compilador no encuentra un método `Use*`, asegúrate de que el paquete NuGet del proveedor esté instalado y que tengas `using Microsoft.EntityFrameworkCore;`.

También se pueden configurar opciones específicas del proveedor. Por ejemplo, la resiliencia de la conexión para Azure SQL:

```
optionsBuilder.UseSqlServer(
      @"Server=(localdb)\mssqllocaldb;Database=Test",
      providerOptions => { providerOptions.EnableRetryOnFailure(); });
```

Otras Configuraciones de `DbContext`

Otras configuraciones del `DbContext` se pueden encadenar con la llamada `Use*`. Ejemplos comunes incluyen:

- **`EnableSensitiveDataLogging()`**: Incluye datos de la aplicación en excepciones y registros (¡solo para desarrollo!).
- **`UseQueryTrackingBehavior()`**: Establece el comportamiento de seguimiento predeterminado para las consultas.
- **`LogTo()`**: Una forma sencilla de obtener registros de EF Core.
- **`UseLazyLoadingProxies()`**: Habilita la carga perezosa a través de proxies dinámicos (requiere `Microsoft.EntityFrameworkCore.Proxies`).

`DbContextOptions` vs. `DbContextOptions<TContext>`

- La mayoría de las subclases de `DbContext` que aceptan opciones deben usar la variación **genérica `DbContextOptions<TContext>`**. Esto asegura que las opciones correctas para el subtipo específico de `DbContext` se resuelvan desde la inyección de dependencias, incluso cuando se registran múltiples subtipos.
- Si la subclase de `DbContext` está destinada a ser **heredada**, debe exponer un constructor `protected` que tome un `DbContextOptions` no genérico.

Configuración del `DbContext` en Tiempo de Diseño

Las herramientas de diseño de EF Core (como las de migraciones) necesitan poder descubrir y crear una instancia de un tipo `DbContext` para recopilar detalles sobre los tipos de entidad y su mapeo al esquema de la base de datos. Esto funciona automáticamente si la herramienta puede crear el `DbContext` de una manera similar a como se configuraría en tiempo de ejecución. Los patrones específicos para esto se detallan en la documentación de "Design-Time Context Creation".

Evitando Problemas de Hilos en `DbContext`

EF Core no soporta múltiples operaciones paralelas en la misma instancia de `DbContext`. Esto incluye la ejecución paralela de consultas asincrónicas y cualquier uso concurrente explícito desde múltiples hilos.

- **Siempre `await` las llamadas asincrónicas de EF Core inmediatamente.** No hacerlo puede corromper el estado del `DbContext`.

- **Compartición implícita de instancias de `DbContext` a través de inyección de dependencias**: Por defecto, `AddDbContext` registra los tipos `DbContext` con una **vida útil con alcance (`scoped lifetime`)**. Esto es seguro en la mayoría de las aplicaciones ASP.NET Core porque cada solicitud del cliente se ejecuta en un solo hilo y obtiene una instancia de `DbContext` separada. Sin embargo, en modelos como Blazor Server, una solicitud lógica mantiene el circuito del usuario, por lo que solo una instancia `scoped DbContext` está disponible por circuito de usuario.

Cualquier código que ejecute explícitamente múltiples hilos en paralelo debe asegurarse de que las instancias de `DbContext` nunca se accedan simultáneamente. Esto se puede lograr registrando el contexto como `scoped` y creando **ámbitos (`scopes`)** separados para cada hilo (usando `IServiceScopeFactory`), o registrando el `DbContext` como **transient** (usando la sobrecarga de `AddDbContext` que toma un parámetro `ServiceLifetime`).

Creación y Configuración de un Modelo en Entity Framework Core

Entity Framework (EF) Core utiliza un **modelo de metadatos** para describir cómo los tipos de entidad de una aplicación se asignan a la base de datos subyacente. Este modelo se construye utilizando **convenciones** (heurísticas que buscan patrones comunes), y luego se puede personalizar con **atributos de mapeo (Data Annotations)** o llamadas a los métodos de **`ModelBuilder` (Fluent API)** en `OnModelCreating`.

La configuración de Fluent API tiene la mayor precedencia, seguida por las Data Annotations, y finalmente las convenciones integradas.

1. Uso de Fluent API para Configurar un Modelo

La **Fluent API** es el método más potente para configurar tu modelo en EF Core. Permite especificar la configuración sin modificar tus clases de entidad, lo que la hace ideal para mantener tus modelos limpios y separados de las preocupaciones de mapeo. La configuración de Fluent API tiene la precedencia más alta.

Para usarla, sobrescribe el método `OnModelCreating` en tu contexto derivado (`DbContext`):

```
using Microsoft.EntityFrameworkCore;
namespace EFModeling.EntityProperties.FluentAPI.Required;
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>() // Selecciona la entidad Blog
            .Property(b => b.Url)   // Selecciona la propiedad Url
            .IsRequired();          // Configura la propiedad como requerida
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; } // Esta propiedad será requerida en la DB
}
```

Agrupando Configuración

Para organizar tu código y reducir el tamaño del método `OnModelCreating`, puedes extraer la configuración de cada tipo de entidad a una clase separada que implemente `IEntityTypeConfiguration<TEntity>`:

```
public class BlogEntityTypeConfiguration : IEntityTypeConfiguration<Blog>
{
    public void Configure(EntityTypeBuilder<Blog> builder)
    {
        builder
            .Property(b => b.Url)
            .IsRequired();
    }
}
```

Luego, invoca el método `Configure` desde `OnModelCreating`:

```
new BlogEntityTypeConfiguration().Configure(modelBuilder.Entity<Blog>());
```

Aplicando Todas las Configuraciones en un Ensamblado

Puedes aplicar todas las configuraciones especificadas en tipos que implementen `IEntityTypeConfiguration` en un ensamblado dado. Esto es útil para aplicar configuraciones a granel.

```
modelBuilder.ApplyConfigurationsFromAssembly(typeof(BlogEntityTypeConfiguration).Assembly);
```

Nota: El orden en que se aplicarán estas configuraciones es indefinido, por lo que este método solo debe usarse cuando el orden no importa.

Usando `EntityTypeConfigurationAttribute` en Tipos de Entidad

Alternativamente, puedes colocar un `EntityTypeConfigurationAttribute` en el tipo de entidad para que EF Core encuentre y use la configuración apropiada:

```
[EntityTypeConfiguration(typeof(BookConfiguration))]
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Isbn { get; set; }
}

public class BookConfiguration : IEntityTypeConfiguration<Book>
{
    public void Configure(EntityTypeBuilder<Book> builder)
    {
        builder.Property(b => b.Isbn).HasMaxLength(13);
    }
}
```

Para que EF Core descubra este atributo, el tipo de entidad `Book` debe incluirse en el modelo (por ejemplo, a través de un `DbSet<Book>` en tu `DbContext` o registrándolo en `OnModelCreating`).

2. Uso de Data Annotations para Configurar un Modelo

Puedes aplicar ciertos atributos (conocidos como **Data Annotations**) a tus clases y propiedades. Las Data Annotations anulan las convenciones, pero son anuladas por la configuración de Fluent API.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace EFModeling.EntityProperties.DataAnnotations.Annotations;

internal class MyContext : DbContext
```

```
{
    public DbSet<Blog> Blogs { get; set; }
}

[Table("Blogs")] // Mapea la clase a una tabla llamada "Blogs"
public class Blog
{
    public int BlogId { get; set; }

    [Required] // Hace que la propiedad Url sea requerida
    public string Url { get; set; }
}
```

3. Convenciones Integradas

EF Core incluye muchas convenciones de construcción de modelos que están habilitadas por defecto. Estas convenciones aplican configuraciones comunes basadas en patrones (ej., una propiedad `Id` se convierte en clave primaria).

Eliminando una Convención Existente

A veces, una convención predeterminada puede no ser apropiada para tu aplicación. Puedes eliminarla sobrescribiendo el método `ConfigureConventions` en tu `DbContext`:

```
protected override void ConfigureConventions(ModelConfigurationBuilder
configurationBuilder)
{
    // Elimina la convención que crea índices automáticamente para columnas de clave
foránea
    configurationBuilder.Conventions.Remove(typeof(ForeignKeyIndexConvention));
}
```

Ejemplo: La `ForeignKeyIndexConvention` crea índices automáticamente para columnas de clave foránea. Si no deseas este comportamiento para todas tus FKs (debido a la sobrecarga del índice o porque las indexarás manualmente), puedes eliminar esta convención. Después de eliminarla, los índices para FKs solo se crearán si se configuran explícitamente (usando `[Index]` o Fluent API).

4. Vista de Depuración (`Debug View`)

La vista de depuración del constructor de modelos es una herramienta invaluable para entender cómo EF Core está construyendo tu modelo. Muestra la estructura de las entidades, propiedades, claves, relaciones, índices y otras configuraciones aplicadas. Puedes acceder a ella desde el depurador de tu IDE (ej., Visual Studio) o directamente desde el código:

```
Console.WriteLine(context.Model.ToDebugString()); // Forma corta
```

También existe una **forma larga** que incluye todas las anotaciones (metadatos relacionales o específicos del proveedor), que puede ser útil para una inspección más profunda:

```
Console.WriteLine(context.Model.ToDebugString(MetadataDebugStringOptions.LongDefault)
); // Forma larga
```

Esta vista es extremadamente útil para diagnosticar problemas de mapeo o para confirmar que tus configuraciones (convenciones, data annotations, fluent API) se están aplicando como esperas.

Tipos de Entidad en Entity Framework Core

En Entity Framework Core (EF Core), un **tipo de entidad** es una clase de .NET que EF Core puede leer y escribir desde/hacia la base de datos. Si usas una base de datos relacional, EF Core puede crear tablas para tus entidades a través de migraciones.

1. Incluyendo Tipos en el Modelo

Por convención, EF Core incluye tipos en el modelo si:

- Están expuestos en propiedades `DbSet` en tu contexto.
- Se especifican explícitamente en el método `OnModelCreating` (usando `modelBuilder.Entity<TipoEntidad>()`).
- Se descubren recursivamente a través de las propiedades de navegación de otros tipos de entidad ya descubiertos.

Ejemplo:

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; } // Blog incluido por DbSet

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>(); // AuditEntry incluido explícitamente
    }
}
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public List<Post> Posts { get; set; } // Post incluido a través de la navegación desde Blog
}
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public Blog Blog { get; set; }
}
public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

2. Excluyendo Tipos del Modelo

Si no quieres que un tipo se incluya en el modelo (por ejemplo, si es una clase auxiliar que no necesita persistencia en la base de datos), puedes excluirlo usando:

Data Annotations: El atributo `[NotMapped]` en la clase.

```
[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

- **Fluent API**: El método `modelBuilder.Ignore<Tipo>()` en `OnModelCreating`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<BlogMetadata>();
}
```

- **Excluyendo de Migraciones**A veces, puedes querer que un tipo de entidad sea parte del modelo, pero que **su tabla no sea creada o gestionada por las migraciones** (ej., si la tabla ya existe y es gestionada externamente).

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<IdentityUser>()
        .ToTable("AspNetUsers", t => t.ExcludeFromMigrations()); // Excluye de migraciones
}
```

Con esta configuración, la tabla `AspNetUsers` no será creada por las migraciones, pero `IdentityUser` seguirá siendo parte del modelo y se podrá usar normalmente para consultas y actualizaciones. Si necesitas volver a gestionarla con migraciones, deberás crear una nueva migración sin esta exclusión.

3. Nombre de la Tabla

Por convención, cada tipo de entidad se mapea a una tabla de base de datos con el mismo nombre que la propiedad `DbSet` que expone la entidad. Si no existe un `DbSet` para la entidad, se utiliza el nombre de la clase.

Puedes configurar manualmente el nombre de la tabla:

Data Annotations: Atributo `[Table("NombreTabla")]`.

```
[Table("blogs")] // Mapea la clase Blog a la tabla "blogs"
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Fluent API: Método `modelBuilder.Entity<Tipo>().ToTable("NombreTabla")`.

C#

```
modelBuilder.Entity<Blog>().ToTable("blogs");
```

4. Esquema de la Tabla

En bases de datos relacionales, las tablas se crean por convención en el esquema predeterminado de la base de datos (ej., `dbo` en SQL Server). Puedes configurar un esquema específico:

Data Annotations: Atributo `[Table("NombreTabla", Schema = "NombreEsquema")]`.

```
[Table("blogs", Schema = "blogging")] // Mapea la tabla "blogs" al esquema "blogging"
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Fluent API: Método `modelBuilder.Entity<Tipo>().ToTable("NombreTabla", schema: "NombreEsquema")`.

```
modelBuilder.Entity<Blog>().ToTable("blogs", schema: "blogging");
```

- También puedes definir un **esquema predeterminado para todo el modelo** con la Fluent API, lo que afectará a todos los objetos de base de datos (tablas, secuencias) a menos que se sobrescriba para una entidad específica:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("blogging");
}
```

5. Mapeo a Vistas (`ToView`)

Los tipos de entidad pueden mapearse a vistas de base de datos usando la Fluent API. EF Core asume que la vista ya existe y **no la creará automáticamente en una migración**.

```
modelBuilder.Entity<Blog>()
```

```
    .ToView("blogsView", schema: "blogging");
```

Mapear a una vista elimina el mapeo predeterminado de la tabla. Sin embargo, el tipo de entidad también puede mapearse a una tabla explícitamente. En este caso, el mapeo de la vista se usará para consultas y el mapeo de la tabla para actualizaciones.

6. Mapeo a Funciones con Valor de Tabla (TVF - `ToFunction`)

Es posible mapear un tipo de entidad a una función con valor de tabla (TVF) en lugar de una tabla. Esto es útil para entidades que representan resultados de consultas complejas. La función debe ser **sin parámetros**.

Ejemplo:

Entidad Keyless (sin clave primaria explícita):

```
public class BlogWithMultiplePosts
{
    public string Url { get; set; }
    public int PostCount { get; set; }
}
```

- **Mapeo en `OnModelCreating`:**
  `modelBuilder.Entity<BlogWithMultiplePosts>().HasNoKey().ToFunction("BlogsWithMultiplePosts");`
- Cuando se consulta esta entidad, EF Core llamará a la TVF en la base de datos y mapeará las columnas resultantes a las propiedades de la entidad. Las propiedades se mapean convencionalmente a columnas con nombres coincidentes, pero se pueden configurar explícitamente con `HasColumnName`.

7. Comentarios de la Tabla (`[Comment]`)

Puedes establecer un comentario de texto arbitrario que se almacenará en la tabla de la base de datos, lo que te permite documentar tu esquema directamente en la base de datos.

Data Annotations: Atributo `[Comment("Tu comentario aquí")]`.

```
[Comment("Blogs gestionados en el sitio web")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Fluent API: Método `modelBuilder.Entity<Tipo>().HasComment("Tu comentario aquí")`.

C#

```
modelBuilder.Entity<Blog>().HasComment("Blogs gestionados en el sitio web");
```

8. Tipos de Entidad de Tipo Compartido (Shared-type entity types)

Los tipos de entidad que usan el mismo tipo CLR (ej., `Dictionary<string, object>`) se conocen como tipos de entidad de tipo compartido. Necesitan configurarse con un **nombre único** que debe proporcionarse cada vez que se usa el tipo de entidad de tipo compartido (ej., en `DbSet` o en la configuración de `ModelBuilder`).

```
internal class MyContext : DbContext
{
    // Usa Set<TEntity>(string name) para especificar el nombre único
    public DbSet<Dictionary<string, object>> Blogs => Set<Dictionary<string, object>>("Blog");
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Configura el tipo de entidad de tipo compartido con su nombre único
```

```
        modelBuilder.SharedTypeEntity<Dictionary<string, object>>(
            "Blog", bb =>
            {
                bb.Property<int>("BlogId");
                bb.Property<string>("Url");
                bb.Property<DateTime>("LastUpdated");
            });
    }
}
```

Ciclo de Vida, Configuración e Inicialización de `DbContext` en Entity Framework Core

Este artículo detalla los patrones fundamentales para la inicialización y configuración de instancias de `DbContext` en Entity Framework Core (EF Core), abordando su ciclo de vida, su integración con la inyección de dependencias en ASP.NET Core, y otras estrategias de creación.

1. El Ciclo de Vida de `DbContext`

La vida útil de un `DbContext` comienza con su creación y termina con su **disposición (`Dispose`)**. Una instancia de `DbContext` está diseñada para ser utilizada para una **única unidad de trabajo (Unit-of-Work)**, lo que implica que su ciclo de vida es típicamente muy corto.

Una unidad de trabajo típica con EF Core incluye:

1. **Creación** de una instancia de `DbContext`.
2. **Seguimiento** de instancias de entidad por el contexto (ya sea porque son devueltas por una consulta o porque se añaden/adjuntan al contexto).
3. **Realización de cambios** en las entidades rastreadas según las reglas de negocio.
4. Llamada a `SaveChanges()` o `SaveChangesAsync()` para que EF Core detecte los cambios y los persista en la base de datos.
5. **Disposición** de la instancia de `DbContext`.

Consideraciones importantes:

- Es crucial **disponer el `DbContext` después de su uso** para liberar recursos no administrados y anular el registro de eventos, previniendo fugas de memoria.
- **`DbContext` no es seguro para subprocesos (`Not thread-safe`)**. No se deben compartir contextos entre hilos. Siempre se deben `await` todas las llamadas asincrónicas antes de continuar usando la instancia del contexto.
- Una `InvalidOperationException` lanzada por el código de EF Core generalmente indica un error de programación y puede dejar el contexto en un estado irrecuperable.

2. `DbContext` en la Inyección de Dependencias para ASP.NET Core

En aplicaciones web, cada solicitud HTTP a menudo corresponde a una única unidad de trabajo, lo que hace que vincular el ciclo de vida del contexto al de la solicitud sea una práctica estándar en ASP.NET Core.

Las aplicaciones ASP.NET Core utilizan la inyección de dependencias. EF Core se integra registrando el `DbContext` mediante `AddDbContext` en `Program.cs`:

```
var connectionString =
    builder.Configuration.GetConnectionString("DefaultConnection")
        ?? throw new InvalidOperationException("Connection string"
        + "'DefaultConnection' not found.");

builder.Services.AddDbContext<ApplicationDbContext>(options =>
```

```
    options.UseSqlServer(connectionString));
```

Este código registra `ApplicationDbContext` (una subclase de `DbContext`) como un **servicio con ámbito (`scoped service`)**. Esto significa que se crea una nueva instancia de `ApplicationDbContext` para cada solicitud HTTP y se inyecta donde sea necesario. La configuración (proveedor de base de datos, cadena de conexión) se pasa a través del parámetro `DbContextOptions<ApplicationDbContext>`.

Requisito del Constructor: La clase `ApplicationDbContext` debe tener un constructor público que acepte un parámetro `DbContextOptions<ApplicationDbContext>`:

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Uso en un Controlador (Inyección de Constructor):

```
public class MyController
{
    private readonly ApplicationDbContext _context;

    public MyController(ApplicationDbContext context) // El contexto se inyecta
automáticamente por DI
    {
        _context = context;
    }
}
```

El resultado es una instancia de `ApplicationDbContext` creada por solicitud, utilizada para la unidad de trabajo y dispuesta al finalizar la solicitud.

3. Inicialización Básica de `DbContext` con `new`

Las instancias de `DbContext` se pueden construir directamente usando `new` en C#. La configuración se puede realizar sobrescribiendo el método `OnConfiguring` o pasando opciones al constructor.

Sobrescribiendo `OnConfiguring`:

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0");
    }
}
```

Pasando la Cadena de Conexión al Constructor:

```
public class ApplicationDbContext : DbContext
{
    private readonly string _connectionString;

    public ApplicationDbContext(string connectionString)
    {
        _connectionString = connectionString;
    }
```

```
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connectionString);
    }
}
```

Creando `DbContextOptions` explícitamente: Se puede crear un objeto `DbContextOptions` con `DbContextOptionsBuilder` y pasarlo al constructor del `DbContext`, permitiendo construir explícitamente un contexto configurado para la inyección de dependencias:

```
var contextOptions = new DbContextOptionsBuilder<ApplicationDbContext>()
    .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0")
    .Options;
using var context = new ApplicationDbContext(contextOptions); // Se construye
explícitamente el contexto
```

4. Uso de una Fábrica de `DbContext` (`AddDbContextFactory`)

Para tipos de aplicaciones como ASP.NET Core Blazor o escenarios donde se requieren **múltiples unidades de trabajo dentro del mismo ámbito** (ej., varias operaciones en una única solicitud HTTP), se puede usar `AddDbContextFactory` para registrar una fábrica de instancias de `DbContext`.

Configuración de la Fábrica:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContextFactory<ApplicationDbContext>(
        options => options.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0"));
}
```

La clase `ApplicationDbContext` debe tener el mismo constructor que acepta `DbContextOptions<ApplicationDbContext>`.

Uso de la Fábrica Inyectada:

```
private readonly IDbContextFactory<ApplicationDbContext> _contextFactory;
public MyController(IDbContextFactory<ApplicationDbContext> contextFactory)
{
    _contextFactory = contextFactory;
}

public async Task DoSomething()
{
    using (var context = _contextFactory.CreateDbContext()) // Se crea una nueva instancia
del contexto
    {
        // ...
    }
}
```

Es importante notar que las instancias de `DbContext` creadas de esta manera **no son gestionadas por el proveedor de servicios** de la aplicación y, por lo tanto, **deben ser dispuestas explícitamente** por la aplicación (a menudo con un bloque `using`).

5. Opciones de `DbContext` (`DbContextOptionsBuilder`)

El `DbContextOptionsBuilder` es el punto de partida para toda la configuración del `DbContext`. Se puede obtener de tres maneras: en `AddDbContext`, en `OnConfiguring` o construyéndolo explícitamente con `new`.

Configuración del Proveedor de Base de Datos

Cada instancia de `DbContext` debe configurarse para usar **un único proveedor de base de datos**. Esto se hace mediante una llamada `Use*` específica (ej., `UseSqlServer`, `UseSqlite`). Es necesario instalar el paquete NuGet del proveedor de base de datos correspondiente.

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer( // Configura para usar SQL Server
            @"Server=(localdb)\mssqllocaldb;Database=Test;ConnectRetryCount=0");
    }
}
```

Las configuraciones opcionales específicas del proveedor se realizan en un `providerOptions` adicional. Por ejemplo, `EnableRetryOnFailure` para la resiliencia de la conexión en Azure SQL:

```
optionsBuilder
    .UseSqlServer(
        @"Server=(localdb)\mssqllocaldb;Database=Test",
        providerOptions => { providerOptions.EnableRetryOnFailure(); });
```

Otras Configuraciones de `DbContext`

Otras configuraciones del `DbContext` se pueden encadenar antes o después de la llamada `Use*`. Ejemplos comunes incluyen:

- `EnableSensitiveDataLogging()`: Incluye datos de la aplicación en excepciones y logs (¡solo para desarrollo!).
- `UseQueryTrackingBehavior()`: Establece el comportamiento de seguimiento predeterminado para las consultas.
- `LogTo()`: Una forma sencilla de obtener logs de EF Core.
- `UseLazyLoadingProxies()`: Habilita la carga perezosa (requiere el paquete `Microsoft.EntityFrameworkCore.Proxies`).

`DbContextOptions` vs. `DbContextOptions<TContext>`

- La mayoría de las subclases de `DbContext` deben usar la variación **genérica `DbContextOptions<TContext>`** en su constructor. Esto asegura que las opciones correctas para el subtipo específico se resuelvan desde la inyección de dependencias.
- Si una subclase de `DbContext` está diseñada para ser **heredada**, debe exponer un constructor `protected` que tome un `DbContextOptions` no genérico, permitiendo que múltiples subclases concretas lo usen.

6. Configuración de `DbContext` en Tiempo de Diseño

Las herramientas de tiempo de diseño de EF Core (como las de migraciones) necesitan poder descubrir y crear una instancia funcional de un tipo `DbContext` para obtener detalles sobre los tipos de entidad y su mapeo. Esto funciona automáticamente si el `DbContext` puede ser creado y configurado de manera similar a como lo sería en tiempo de ejecución.

7. Evitando Problemas de Hilos en `DbContext`

Entity Framework Core no soporta múltiples operaciones paralelas en la misma instancia de `DbContext`. Esto incluye la ejecución paralela de consultas asincrónicas y cualquier uso concurrente explícito desde múltiples hilos.

- **Siempre `await` las llamadas asincrónicas de EF Core inmediatamente.** No hacerlo puede corromper el estado del `DbContext`.
- **Compartir implícitamente instancias de `DbContext` a través de la inyección de dependencias**: Por defecto, `AddDbContext` registra los tipos `DbContext` con una **vida útil con ámbito (`scoped lifetime`)**. Esto es seguro en la mayoría de las aplicaciones ASP.NET Core (un hilo por solicitud, instancia de `DbContext` separada por solicitud). Sin embargo, en modelos como Blazor Server, donde un circuito de usuario mantiene una solicitud lógica, solo una instancia `scoped DbContext` está disponible por circuito.

Cualquier código que ejecute explícitamente múltiples hilos en paralelo debe asegurar que las instancias de `DbContext` nunca se accedan concurrentemente. Esto se puede lograr registrando el contexto como `scoped` y creando **ámbitos (`scopes`)** separados para cada hilo (usando `IServiceScopeFactory`), o registrando el `DbContext` como **transient** (usando la sobrecarga de `AddDbContext` que toma un parámetro `ServiceLifetime`).
Given the current date is June 18, 2025, and the provided article about "Creating and Configuring a Model" is from March 27, 2023, the information is still highly relevant and accurate for current (and foreseeable) versions of Entity Framework Core. The core concepts of conventions, Data Annotations, and Fluent API for model configuration have remained stable.
Here's the summary of the article, maintaining its structure and content.

Creación y Configuración de un Modelo en Entity Framework Core

Entity Framework (EF) Core construye un **modelo de metadatos** para describir cómo los tipos de entidad de una aplicación se mapean a la base de datos subyacente. Este modelo se crea utilizando un conjunto de **convenciones** (heurísticas que buscan patrones comunes). Posteriormente, el modelo se puede personalizar mediante **atributos de mapeo (también conocidos como Data Annotations)** y/o llamadas a los métodos de **`ModelBuilder` (también conocidos como Fluent API)** en el método `OnModelCreating`. Ambos mecanismos anulan la configuración realizada por las convenciones.
La mayoría de las configuraciones pueden aplicarse a un modelo dirigido a cualquier almacén de datos. Los proveedores también pueden habilitar configuraciones específicas para un almacén de datos particular o ignorar configuraciones no soportadas o no aplicables.

1. Uso de Fluent API para Configurar un Modelo

La **Fluent API** es el método más potente para configurar tu modelo en EF Core. Permite especificar la configuración sin modificar tus clases de entidad, manteniendo así tus modelos limpios de preocupaciones de mapeo. La configuración de Fluent API tiene la **mayor precedencia** y anulará tanto las convenciones como las Data Annotations. La configuración se aplica en el orden en que se llaman los métodos, y las llamadas posteriores anulan las configuraciones especificadas previamente en caso de conflictos.
Para usarla, sobrescribe el método `OnModelCreating` en tu contexto derivado (`DbContext`):

```
using Microsoft.EntityFrameworkCore;
namespace EFModeling.EntityProperties.FluentAPI.Required;
```

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    #region Required
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>() // Selecciona la entidad Blog
            .Property(b => b.Url)   // Selecciona la propiedad Url
            .IsRequired();          // Configura la propiedad como requerida
    }
    #endregion
}
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; } // Esta propiedad se mapeará como requerida en la DB
}
```

Agrupando Configuración

Para mejorar la organización del código y reducir el tamaño del método `OnModelCreating`, la configuración de un tipo de entidad específico puede extraerse a una clase separada que implemente `IEntityTypeConfiguration<TEntity>`.

```
public class BlogEntityTypeConfiguration : IEntityTypeConfiguration<Blog>
{
    public void Configure(EntityTypeBuilder<Blog> builder)
    {
        builder
            .Property(b => b.Url)
            .IsRequired();
    }
}
```

Luego, se invoca el método `Configure` desde `OnModelCreating`:

```
new BlogEntityTypeConfiguration().Configure(modelBuilder.Entity<Blog>());
```

Aplicando Todas las Configuraciones en un Ensamblado

Es posible aplicar todas las configuraciones especificadas en tipos que implementen `IEntityTypeConfiguration` dentro de un ensamblado dado.

```
modelBuilder.ApplyConfigurationsFromAssembly(typeof(BlogEntityTypeConfiguration).Assembly);
```

Nota: El orden en que se aplicarán estas configuraciones es indefinido. Este método solo debe usarse cuando el orden de aplicación no es relevante.

Usando `EntityTypeConfigurationAttribute` en Tipos de Entidad

En lugar de llamar explícitamente a `Configure`, se puede colocar un `EntityTypeConfigurationAttribute` en el tipo de entidad para que EF Core encuentre y utilice la configuración apropiada.

```
[EntityTypeConfiguration(typeof(BookConfiguration))]
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Isbn { get; set; }
}
```

Este atributo indica que EF Core usará la implementación `IEntityTypeConfiguration` especificada siempre que el tipo de entidad `Book` se incluya en un modelo (por ejemplo, mediante una propiedad `DbSet<Book>` o registrándolo en `OnModelCreating`).
Nota: Los tipos con `EntityTypeConfigurationAttribute` no se descubren automáticamente en un ensamblado. Los tipos de entidad deben añadirse al modelo antes de que el atributo sea descubierto en ese tipo de entidad.

2. Uso de Data Annotations para Configurar un Modelo
También se pueden aplicar ciertos atributos (conocidos como **Data Annotations**) a tus clases y propiedades. Las Data Annotations anulan la configuración de las convenciones, pero son a su vez anuladas por la configuración de Fluent API.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;
namespace EFModeling.EntityProperties.DataAnnotations.Annotations;
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
}
[Table("Blogs")] // Mapea la clase a una tabla llamada "Blogs"
public class Blog
{
    public int BlogId { get; set; }

    [Required] // Hace que la propiedad Url sea requerida en la base de datos
    public string Url { get; set; }
}
```

3. Convenciones Integradas
EF Core incluye numerosas convenciones de construcción de modelos que están habilitadas por defecto. Estas convenciones aplican configuraciones comunes automáticamente (ej., una propiedad llamada `Id` se configura como clave primaria). Se pueden encontrar en la lista de clases que implementan la interfaz `IConvention`.
Las aplicaciones pueden eliminar o reemplazar cualquiera de estas convenciones, así como añadir nuevas **convenciones personalizadas** para aplicar configuración a patrones no reconocidos por EF de forma predeterminada.

Eliminando una Convención Existente
A veces, una convención incorporada puede no ser apropiada para tu aplicación y puede ser eliminada.

Ejemplo: No crear índices para columnas de clave foránea
Normalmente, EF Core crea índices para columnas de clave foránea (FK) mediante la convención `ForeignKeyIndexConvention`. Sin embargo, los índices tienen una sobrecarga y quizás no siempre sea apropiado crearlos para todas las columnas FK. Para deshabilitar esto, la `ForeignKeyIndexConvention` puede ser eliminada al construir el modelo:

```
protected override void ConfigureConventions(ModelConfigurationBuilder
configurationBuilder)
{
    // Elimina la convención que crea índices automáticamente para columnas de clave
foránea
    configurationBuilder.Conventions.Remove(typeof(ForeignKeyIndexConvention));
}
```

Después de eliminar esta convención, los índices en las FKs no se crearán automáticamente. Sin embargo, los índices pueden crearse explícitamente para las columnas de clave foránea cuando se deseen, ya sea usando el `IndexAttribute` o con configuración en `OnModelCreating`.

4. Vista de Depuración (`Debug View`)

La vista de depuración del constructor de modelos es una herramienta esencial para comprender cómo EF Core está construyendo tu modelo. Se puede acceder a ella en el depurador de tu IDE (ej., Visual Studio).
También se puede acceder directamente desde el código, por ejemplo, para enviar la vista de depuración a la consola:

```
Console.WriteLine(context.Model.ToDebugString()); // Forma corta
```

La vista de depuración tiene una forma corta y una forma larga. La forma larga incluye todas las anotaciones, lo que puede ser útil para ver metadatos relacionales o específicos del proveedor:

```
Console.WriteLine(context.Model.ToDebugString(MetadataDebugStringOptions.LongDefault)
); // Forma larga
```

Esta vista es muy útil para depurar problemas de mapeo o para verificar que las configuraciones (convenciones, data annotations, fluent API) se están aplicando como se espera.

Claves en Entity Framework Core

Una clave sirve como identificador único para cada instancia de entidad en Entity Framework Core (EF Core). La mayoría de las entidades tienen una única clave, que se mapea al concepto de **clave primaria** en bases de datos relacionales. Las entidades pueden tener claves adicionales además de la clave primaria, conocidas como **claves alternativas**.

1. Configuración de una Clave Primaria

Por convención, una propiedad nombrada `Id` o `<nombreTipo>Id` será configurada como la clave primaria de una entidad.

Ejemplo por Convención:

```
internal class Car
{
    public string Id { get; set; } // Se configura como PK por convención
    public string Make { get; set; }
    public string Model { get; set; }
}

internal class Truck
{
    public string TruckId { get; set; } // Se configura como PK por convención
    public string Make { get; set; }
    public string Model { get; set; }
}
```

Nota: Los tipos de entidad poseídos (`Owned entity types`) usan reglas diferentes para definir claves.

Puedes configurar explícitamente una sola propiedad para que sea la clave primaria:

Data Annotations: Usando el atributo `[Key]`.

```
internal class Car
{
    [Key] // La propiedad LicensePlate se configura como PK
```

```
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Fluent API: Usando el método `HasKey()`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => c.LicensePlate); // Configura LicensePlate como PK
}
```

También puedes configurar múltiples propiedades para ser la clave de una entidad, lo que se conoce como una **clave compuesta**. Las convenciones solo establecerán una clave compuesta en casos específicos, como para una colección de tipos poseídos.

Data Annotations: Usando el atributo `[PrimaryKey(nameof(Prop1), nameof(Prop2))]`.

```
[PrimaryKey(nameof(State), nameof(LicensePlate))] // Clave compuesta por State y LicensePlate
internal class Car
{
    public string State { get; set; }
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Fluent API: Pasando un objeto anónimo con las propiedades al método `HasKey()`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => new { c.State, c.LicensePlate }); // Configura una clave compuesta
}
```

2. Generación de Valores

Para claves primarias numéricas y GUID no compuestas, EF Core configura la generación de valores por convención. Por ejemplo, una clave primaria numérica en SQL Server se configura automáticamente como una columna `IDENTITY`.

Importante: Si una propiedad clave tiene su valor generado por la base de datos y se especifica un valor no predeterminado al añadir una entidad, EF Core asumirá que la entidad ya existe en la base de datos e intentará actualizarla en lugar de insertar una nueva. Para evitar esto, desactiva la generación de valores o especifica explícitamente valores para propiedades generadas.

3. Nombre de la Clave Primaria

Por convención, en bases de datos relacionales, las claves primarias se crean con el nombre `PK_<nombreTipo>`. Puedes configurar el nombre de la restricción de clave primaria con la Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasKey(b => b.BlogId)
```

```
        .HasName("PrimaryKey_BlogId"); // Configura el nombre de la restricción PK
}
```

4. Tipos y Valores de Clave

EF Core soporta el uso de propiedades de cualquier tipo primitivo como clave primaria, incluyendo `string`, `Guid`, `byte[]`, entre otros. Sin embargo, no todas las bases de datos soportan todos los tipos como claves. En algunos casos, los valores de clave pueden convertirse automáticamente a un tipo compatible; de lo contrario, la conversión debe especificarse manualmente.

Las propiedades clave deben tener siempre un valor no predeterminado al añadir una nueva entidad al contexto. Sin embargo, algunos valores de clave pueden ser **generados por la base de datos**. En esos casos, EF Core intentará generar un valor temporal cuando la entidad sea añadida para fines de seguimiento. Después de llamar a `SaveChanges`, el valor temporal será reemplazado por el valor real generado por la base de datos.

5. Claves Alternativas

Una **clave alternativa** sirve como un identificador único adicional para cada instancia de entidad, aparte de la clave primaria. Puede ser el objetivo de una relación. En bases de datos relacionales, esto se mapea al concepto de un índice/restricción único en la(s) columna(s) de la clave alternativa y una o más restricciones de clave foránea que referencian dichas columnas.

Consejo: Si solo necesitas imponer la unicidad en una columna, define un índice único en lugar de una clave alternativa. Las claves alternativas en EF son de solo lectura y proporcionan semántica adicional sobre los índices únicos porque pueden ser usadas como el objetivo de una clave foránea.

Las claves alternativas se introducen típicamente de forma automática cuando son necesarias, por ejemplo, cuando se identifica una propiedad que no es la clave primaria como el objetivo de una relación.

Ejemplo de Clave Alternativa Implícita (por relación):

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogUrl) // BlogUrl es la FK en Post
            .HasPrincipalKey(b => b.Url);    // Url es la clave alternativa en Blog
    }
}
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; } // Esta propiedad se convierte en clave alternativa
    public List<Post> Posts { get; set; }
}
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
```

```
    public string BlogUrl { get; set; } // Propiedad que referencia la clave alternativa de Blog
    public Blog Blog { get; set; }
}
```

También puedes configurar explícitamente una sola propiedad para ser una clave alternativa:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasAlternateKey(c => c.LicensePlate); // Configura LicensePlate como clave alternativa
}
```

O configurar múltiples propiedades para ser una clave alternativa (clave alternativa compuesta):

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasAlternateKey(c => new { c.State, c.LicensePlate }); // Configura una clave
alternativa compuesta
}
```

Por convención, el índice y la restricción para una clave alternativa se nombran `AK_<nombreTipo>_<nombrePropiedad>` (para claves alternativas compuestas, `<nombrePropiedad>` se convierte en una lista de nombres de propiedades separadas por guiones bajos). Puedes configurar el nombre del índice y la restricción única de la clave alternativa:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasAlternateKey(c => c.LicensePlate)
        .HasName("AlternateKey_LicensePlate"); // Configura el nombre de la restricción de la
clave alternativa
}
```

Introducción a las Relaciones en Entity Framework Core

Este documento ofrece una introducción a la representación de las relaciones en los modelos de objetos y las bases de datos relacionales, incluyendo cómo EF Core las mapea entre ambos paradigmas.

1. Relaciones en Modelos de Objetos

Una relación define cómo dos entidades se vinculan entre sí. En un lenguaje orientado a objetos como C#, esta conexión se representa añadiendo referencias (propiedades de navegación) entre las clases.

Ejemplo de Modelo de Objetos (Blog y Post):

Inicialmente, las clases `Blog` y `Post` pueden no tener una relación explícita:

```
public class Blog
{
    public string Name { get; set; }
    public virtual Uri SiteUri { get; set; }
}

public class Post
{
    public string Title { get; set; }
```

```
    public string Content { get; set; }
    public DateTime PublishedOn { get; set; }
    public bool Archived { get; set; }
}
```

Para indicar una relación, se añaden propiedades de navegación. Por ejemplo, un `Post` puede referenciar al `Blog` al que pertenece:

```
public class Post
{
    // ... otras propiedades ...
    public Blog Blog { get; set; } // Propiedad de navegación del Post al Blog
}
```

De manera inversa, un `Blog` puede contener una colección de `Posts`:

```
C#
public class Blog
{
    // ... otras propiedades ...
    public ICollection<Post> Posts { get; } // Propiedad de navegación del Blog a sus Posts
}
```

Esta conexión bidireccional (de `Blog` a `Post` y de `Post` a `Blog`) se conoce como una "relación" en EF Core. **Importante**: Una *única* relación típicamente se puede recorrer en ambas direcciones. Las propiedades `Blog.Posts` y `Post.Blog` se denominan "navegaciones".

2. Relaciones en Bases de Datos Relacionales

Las bases de datos relacionales representan las relaciones utilizando **claves foráneas (Foreign Keys - FKs)**. Una clave foránea en una tabla (la tabla dependiente) referencia a una **clave primaria (Primary Key - PK)** en otra tabla (la tabla principal).

Ejemplo de Modelo Relacional (SQL Server):

```
SQL
CREATE TABLE [Posts] (
    [Id] int NOT NULL IDENTITY,
    [Title] nvarchar(max) NULL,
    [Content] nvarchar(max) NULL,
    [PublishedOn] datetime2 NOT NULL,
    [Archived] bit NOT NULL,
    [BlogId] int NOT NULL, -- Clave foránea que referencia a Blogs
    CONSTRAINT [PK_Posts] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Posts_Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blogs]
([Id]) ON DELETE CASCADE
);
CREATE TABLE [Blogs] (
    [Id] int NOT NULL IDENTITY, -- Clave primaria de Blogs
    [Name] nvarchar(max) NULL,
    [SiteUri] nvarchar(max) NULL,
    CONSTRAINT [PK_Blogs] PRIMARY KEY ([Id])
);
```

En este modelo, la columna `BlogId` en la tabla `Posts` es una clave foránea que referencia la columna `Id` (clave primaria) de la tabla `Blogs`. Esta restricción asegura la integridad referencial: cualquier valor en `Posts.BlogId` debe coincidir con un `Blogs.Id` existente.

3. Mapeo de Relaciones en EF Core

El mapeo de relaciones en EF Core consiste en conectar la representación de clave primaria/clave foránea de la base de datos con las referencias entre objetos del modelo. Esto implica:

1. Añadir una propiedad de clave primaria a cada tipo de entidad.
2. Añadir una propiedad de clave foránea a un tipo de entidad (el dependiente).
3. Asociar las referencias (navegaciones) entre los tipos de entidad con las claves primarias y foráneas para formar una única configuración de relación.

Una vez hecho este mapeo, EF Core gestiona automáticamente la coherencia: cambia los valores de las claves foráneas cuando las referencias entre objetos cambian, y cambia las referencias entre objetos cuando los valores de las claves foráneas cambian.

Ejemplo de Entidades con Claves Primarias y Foráneas:

```
public class Blog
{
    public int Id { get; set; } // Propiedad de clave primaria
    public string Name { get; set; }
    public virtual Uri SiteUri { get; set; }

    public ICollection<Post> Posts { get; }
}
public class Post
{
    public int Id { get; set; } // Propiedad de clave primaria
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PublishedOn { get; set; }
    public bool Archived { get; set; }
    public int BlogId { get; set; } // Propiedad de clave foránea
    public Blog Blog { get; set; } // Propiedad de navegación
}
```

EF Core asocia automáticamente la clave primaria `Blog.Id` y la clave foránea `Post.BlogId` con las navegaciones `Blog.Posts` y `Post.Blog` en relaciones simples. Sin embargo, esto también puede especificarse explícitamente en `OnModelCreating` usando la Fluent API:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasMany(e => e.Posts)       // Un Blog tiene muchas Posts
        .WithOne(e => e.Blog)        // Una Post tiene un Blog
        .HasForeignKey(e => e.BlogId) // BlogId es la clave foránea en Post
        .HasPrincipalKey(e => e.Id);  // Id es la clave principal en Blog
}
```

Con esta configuración, todas estas propiedades trabajarán de manera coherente como una representación de una única relación entre `Blog` y `Post`.

4. Más Información sobre Relaciones

EF Core soporta varios tipos de relaciones y formas de configurarlas:

- **Relaciones Uno-a-Muchos**: Una entidad se asocia con cualquier número de otras entidades.
- **Relaciones Uno-a-Uno**: Una entidad se asocia con otra entidad única.
- **Relaciones Muchos-a-Muchos**: Cualquier número de entidades se asocia con cualquier número de otras entidades.

Para profundizar en las propiedades de las entidades involucradas en el mapeo de relaciones:

- **Claves foráneas y principales en relaciones**: Detalla cómo las claves foráneas se mapean a la base de datos.
- **Navegaciones de relación**: Describe cómo las navegaciones se superponen a una clave foránea para proporcionar una vista orientada a objetos de la relación.

Los modelos de EF Core se construyen mediante una combinación de:

- **Convenciones de relación**: Descubren tipos de entidad, sus propiedades y relaciones.
- **Atributos de mapeo de relación**: Alternativa a la API de `ModelBuilder` para algunos aspectos.
- **API de `ModelBuilder` (Fluent API)**: La fuente final de verdad para el modelo de EF, siempre tiene precedencia y ofrece control total.

Otros temas relacionados con las relaciones incluyen:

- **Eliminaciones en cascada**: Describe cómo las entidades relacionadas pueden eliminarse automáticamente.
- **Tipos de entidad poseídos (`Owned entity types`)**: Un tipo especial de relación que implica una conexión más fuerte y comportamientos específicos.

Se recomienda consultar el glosario de términos de relaciones para entender la terminología.

5. Uso de Relaciones

Las relaciones definidas en el modelo se pueden usar de varias maneras:

- **Consulta de datos relacionados**:
  - **Carga ansiosa (`Eagerly`)**: Con `Include` en consultas LINQ.
  - **Carga perezosa (`Lazily`)**: Usando proxies de carga perezosa o carga perezosa sin proxies.
  - **Carga explícita (`Explicitly`)**: Usando los métodos `Load` o `LoadAsync`.
- **Inicialización de datos (`data seeding`)**: Mediante la coincidencia de valores de PK con valores de FK.
- **Seguimiento de grafos de entidades**: El "change tracker" de EF Core utiliza las relaciones para:
  - Detectar cambios en las relaciones y realizar correcciones (`fixup`).
  - Enviar actualizaciones de claves foráneas a la base de datos con `SaveChanges` o `SaveChangesAsync`.

Guardar Datos en Entity Framework Core

Entity Framework (EF) Core ofrece dos enfoques fundamentales para guardar datos en la base de datos: el seguimiento de cambios con `SaveChanges()` y las operaciones de "actualización masiva" con `ExecuteUpdate()` y `ExecuteDelete()`.

1. Enfoque 1: Seguimiento de Cambios y `SaveChanges()`

Este es el método más común y versátil para guardar datos, ideal para escenarios donde se carga, modifica y luego se guarda un subconjunto específico de datos ("unidad de trabajo").

Funcionamiento Básico:

1. **Consulta y Seguimiento:** Se carga una entidad de la base de datos usando una consulta LINQ (las consultas de EF son de seguimiento por defecto). EF rastrea las entidades cargadas en su **rastreador de cambios interno**.
2. **Manipulación:** La instancia de entidad cargada se modifica directamente en el código .NET (ej., asignando un nuevo valor a una propiedad). EF no interviene en este paso.

3. **Persistencia:** Se llama a `DbContext.SaveChanges()` (o `SaveChangesAsync()`). En este punto, EF detecta automáticamente los cambios comparando el estado actual de las entidades con una "instantánea" tomada en el momento de la carga. Los cambios detectados se persisten en la base de datos (ej., mediante sentencias `UPDATE` para modificaciones, `INSERT` para adiciones con `DbSet<TEntity>.Add`, y `DELETE` para eliminaciones con `DbSet<TEntity>.Remove`).

Ejemplo de Actualización:

```
using (var context = new BloggingContext())
{
   var blog = await context.Blogs.SingleAsync(b => b.Url == "http://example.com");
   blog.Url = "http://example.com/blog"; // Modificación de la entidad rastreada
   await context.SaveChangesAsync(); // EF detecta el cambio y lo guarda
}
```

Ventajas de `SaveChanges()`:

- **Detección Automática de Cambios:** EF detecta automáticamente qué entidades y propiedades han cambiado, actualizando solo lo necesario, lo que mejora el rendimiento.
- **Gestión de Operaciones Complejas:** EF maneja complejidades como la obtención de claves generadas por la base de datos para entidades insertadas antes de insertar sus dependencias (ej., un Blog y sus Posts).
- **Control de Concurrencia:** EF puede detectar conflictos de concurrencia (cuando una fila ha sido modificada por otra parte) y alertar al desarrollador.
- **Transacciones Automáticas:** En bases de datos que lo soportan, `SaveChanges()` envuelve automáticamente múltiples cambios en una transacción, garantizando la consistencia de los datos.
- **Agrupamiento (Batching):** Agrupa múltiples cambios en una sola operación en muchos casos, reduciendo significativamente el número de viajes de ida y vuelta a la base de datos y mejorando el rendimiento.

2. Enfoque 2: `ExecuteUpdate()` y `ExecuteDelete()` ("Actualización Masiva")

Mientras que `SaveChanges()` es potente, puede ser ineficiente para operaciones masivas (grandes cantidades de entidades) porque requiere cargar y rastrear todas las entidades involucradas en la operación. Las operaciones masivas (`ExecuteUpdate` y `ExecuteDelete`) abordan esta limitación.

Funcionamiento Básico: Estas operaciones permiten expresar directamente sentencias SQL `UPDATE` o `DELETE` utilizando operadores LINQ, lo que se ejecuta eficientemente en la base de datos sin cargar datos ni involucrar el rastreador de cambios de EF.

Ejemplo de Eliminación Masiva:

```
// Elimina todos los blogs con una calificación menor a 3 directamente en la DB
context.Blogs.Where(b => b.Rating < 3).ExecuteDelete();
```

Esto genera una sentencia SQL `DELETE` con una cláusula `WHERE` para la base de datos, lo que es mucho más eficiente para grandes volúmenes de datos. De manera similar, `ExecuteUpdate()` permite expresar sentencias `UPDATE`.

Estas operaciones también pueden simplificar el código cuando se sabe exactamente qué propiedades de qué entidad se desean cambiar, evitando la complejidad del API de seguimiento de cambios (ej., `Attach`).

Ventajas de `ExecuteUpdate()` y `ExecuteDelete()`:

- **Eficiencia en Operaciones Masivas:** Permiten modificar o eliminar un gran número de entidades sin cargarlas en memoria, enviando una única sentencia SQL a la base de datos.

- **Mayor Rendimiento:** Evitan la sobrecarga del rastreador de cambios de EF.
- **Simplicidad para Cambios Específicos:** Facilitan la expresión de operaciones de actualización o eliminación muy específicas sin la necesidad de un seguimiento completo.

Limitaciones:

- **Ejecución Inmediata:** Estas operaciones se ejecutan inmediatamente y actualmente no se pueden agrupar (`batch`) con otras operaciones (a diferencia de `SaveChanges()`).
- **Responsabilidad del Desarrollador:** Como no hay seguimiento de cambios, el desarrollador es responsable de saber exactamente qué entidades y propiedades deben cambiarse.
- **Sin Control Automático de Concurrencia:** No aplican automáticamente el control de concurrencia; el desarrollador debe implementarlo explícitamente mediante una cláusula `Where`.
- **Solo Actualización y Eliminación:** Actualmente, solo soportan la actualización y eliminación; las inserciones deben hacerse a través de `DbSet<TEntity>.Add` y `SaveChanges()`.

Resumen: Cuándo Usar Cada Enfoque

Usar `SaveChanges()` cuando:

- No se sabe de antemano qué cambios se realizarán (EF los detectará automáticamente).
  - *Escenario:* "Quiero cargar un Blog de la base de datos y mostrar un formulario para que el usuario lo cambie".
- Se necesita manipular un grafo de objetos (múltiples objetos interconectados), ya que EF determinará el orden correcto de los cambios y cómo vincularlos.
  - *Escenario:* "Quiero actualizar un blog, cambiando algunos de sus posts y eliminando otros".

Usar `ExecuteUpdate()` y `ExecuteDelete()` cuando:

- Se desea cambiar un número potencialmente grande de entidades basándose en algún criterio.
  - *Escenario:* "Quiero dar un aumento a todos los empleados".
  - *Escenario:* "Quiero eliminar todos los blogs cuyo nombre comienza con X".
- Se sabe exactamente qué entidades se desean modificar y cómo.
  - *Escenario:* "Quiero eliminar el blog cuyo nombre es 'Foo'".
  - *Escenario:* "Quiero cambiar el nombre del blog con Id 5 a 'Bar'".

Guardar Cambios Básicos en Entity Framework Core (`DbContext.SaveChanges()`)

`DbContext.SaveChanges()` es una de las dos técnicas principales en Entity Framework Core (EF Core) para persistir cambios en la base de datos. Con este método, se realizan una o más **operaciones rastreadas** (añadir, actualizar, eliminar) y luego se aplican esos cambios invocando el método `SaveChanges()`. La alternativa es usar `ExecuteUpdate()` y `ExecuteDelete()` para operaciones masivas sin involucrar el rastreador de cambios.

1. Añadir Datos (`DbSet<TEntity>.Add`)

Para añadir nuevas instancias de tus clases de entidad, utiliza el método `DbSet<TEntity>.Add()`. Los datos se insertarán en la base de datos cuando se llame a `DbContext.SaveChanges()`.

Ejemplo:

```
using (var context = new BloggingContext())
{
   var blog = new Blog { Url = "http://example.com" };
   context.Blogs.Add(blog); // Se marca la entidad como 'Added'
   await context.SaveChangesAsync(); // La entidad se inserta en la base de datos
}
```

Nota: Los métodos `Add`, `Attach` y `Update` funcionan sobre el grafo completo de entidades pasadas. Alternativamente, se puede establecer el estado de una sola entidad usando `context.Entry(entity).State = EntityState.Added;`.

2. Actualizar Datos

EF Core detecta automáticamente los cambios realizados en una entidad existente que está siendo rastreada por el contexto. Esto incluye entidades que se cargan/consultan de la base de datos o que fueron añadidas y guardadas previamente. Simplemente se modifican los valores de las propiedades y luego se llama a `SaveChanges()`.

Ejemplo:

```
using (var context = new BloggingContext())
{
    // Se carga una entidad, que automáticamente es rastreada
    var blog = await context.Blogs.SingleAsync(b => b.Url == "http://example.com");
    blog.Url = "http://example.com/blog"; // Se modifica la propiedad de la entidad rastreada
    await context.SaveChangesAsync(); // EF detecta el cambio en 'Url' y genera un UPDATE
}
```

3. Eliminar Datos (`DbSet<TEntity>.Remove`)

Para eliminar instancias de tus clases de entidad, utiliza el método `DbSet<TEntity>.Remove()`.

Ejemplo:

```
using (var context = new BloggingContext())
{
    // Se carga una entidad, que automáticamente es rastreada
    var blog = await context.Blogs.SingleAsync(b => b.Url == "http://example.com/blog");
    context.Blogs.Remove(blog); // Se marca la entidad como 'Deleted'
    await context.SaveChangesAsync(); // La entidad se elimina de la base de datos
}
```

Si la entidad ya existe en la base de datos, se eliminará durante `SaveChanges()`. Si la entidad aún no ha sido guardada (es decir, está marcada como "añadida"), se eliminará del contexto y no se insertará.

4. Múltiples Operaciones en una Sola Llamada a `SaveChanges()`

Puedes combinar múltiples operaciones de añadir, actualizar y/o eliminar en una única llamada a `SaveChanges()`. EF Core gestionará la secuencia y la lógica necesaria para aplicar todos los cambios de manera coherente.

Ejemplo:

```
using (var context = new BloggingContext())
{
    // Preparación inicial de la base de datos
    context.Blogs.Add(new Blog { Url = "http://example.com/blog" });
    context.Blogs.Add(new Blog { Url = "http://example.com/another_blog" });
    await context.SaveChangesAsync(); // Guarda los blogs iniciales
}

using (var context = new BloggingContext())
{
    // Operaciones combinadas:
    // 1. Añadir
    context.Blogs.Add(new Blog { Url = "http://example.com/blog_one" });
    context.Blogs.Add(new Blog { Url = "http://example.com/blog_two" });
```

```
    // 2. Actualizar
    var firstBlog = await context.Blogs.FirstAsync();
    firstBlog.Url = ""; // Se modifica una entidad existente

    // 3. Eliminar
    var lastBlog = await context.Blogs.OrderBy(e => e.BlogId).LastAsync();
    context.Blogs.Remove(lastBlog); // Se marca una entidad para eliminación

    await context.SaveChangesAsync(); // Todas las operaciones anteriores se aplican en una transacción
}
```

Nota: Para la mayoría de los proveedores de bases de datos, `SaveChanges()` es **transaccional**. Esto significa que todas las operaciones se realizan con éxito o todas fallan, asegurando que los datos nunca queden parcialmente aplicados, manteniendo la consistencia de la base de datos.

Seguridad en ASP.NET Core: Autenticación, Autorización y Prevención de Vulnerabilidades

ASP.NET Core proporciona un robusto conjunto de herramientas y bibliotecas para configurar y gestionar la seguridad en aplicaciones web. Esto incluye características de autenticación, autorización, protección de datos, aplicación de HTTPS, almacenamiento seguro de secretos, y prevención de ataques como XSRF/CSRF y XSS.

1. Autenticación vs. Autorización

Estos son dos conceptos fundamentales en seguridad web:

- **Autenticación**: Es el proceso de verificar la identidad de un usuario. Un usuario proporciona credenciales (ej., nombre de usuario y contraseña) que se comparan con los datos almacenados. Si coinciden, el usuario se autentica exitosamente.
- **Autorización**: Es el proceso que determina qué acciones un usuario autenticado está permitido realizar. Una vez que un usuario es autenticado, la autorización decide a qué recursos o funcionalidades tiene acceso.

2. Características de Seguridad en ASP.NET Core

ASP.NET Core integra varias capacidades para asegurar las aplicaciones:

- **Proveedores de Identidad:** Soporte integrado para proveedores de identidad (ej., ASP.NET Core Identity) y servicios de terceros (ej., Facebook, Twitter, Google).
- **Almacenamiento de Secretos:** Diversos enfoques para almacenar secretos de aplicación de forma segura (ej., Secret Manager para desarrollo, Managed Identities para producción).
- **Prevención de Vulnerabilidades Comunes:**
    - **Ataques de Cross-Site Scripting (XSS):** Prevención de inyección de scripts maliciosos en páginas web.
    - **Ataques de Inyección SQL:** Mitigación de la inyección de código SQL malicioso.
    - **Ataques de Falsificación de Solicitud entre Sitios (XSRF/CSRF):** Prevención de solicitudes no autorizadas desde sitios de terceros.
    - **Ataques de Redirección Abierta:** Protección contra redirecciones a URLs maliciosas.

3. Conceptos de Autenticación en ASP.NET Core

La autenticación en ASP.NET Core es manejada por el servicio `IAuthenticationService` y el *middleware* de autenticación. Este servicio utiliza

manejadores de autenticación registrados para realizar acciones relacionadas con la autenticación (ej., autenticar un usuario, responder a intentos de acceso no autorizados).

Esquemas de Autenticación: Son nombres que corresponden a un manejador de autenticación y sus opciones de configuración. Se registran en `Program.cs` llamando a `AddAuthentication` y métodos de extensión específicos del esquema (ej., `AddJwtBearer`, `AddCookie`). El parámetro a `AddAuthentication` puede especificar el esquema por defecto.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("JwtSettings", options))
    .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("CookieSettings", options));
```

- **Middleware de Autenticación:** Se añade en `Program.cs` llamando a `UseAuthentication`. Debe invocarse *antes* de cualquier *middleware* que dependa de usuarios autenticados.
- **`ClaimsPrincipal`:** La autenticación es responsable de proporcionar un `ClaimsPrincipal` para que la autorización tome decisiones de permiso.
- **Manejador de Autenticación:** Es un tipo que implementa el comportamiento de un esquema (derivado de `IAuthenticationHandler` o `AuthenticationHandler<TOptions>`). Su responsabilidad principal es autenticar usuarios, construyendo objetos `AuthenticationTicket` que representan la identidad del usuario.
    - **`RemoteAuthenticationHandler<TOptions>`:** Para autenticación que requiere un paso remoto (ej., OAuth 2.0, OIDC con proveedores como Facebook, Google).
- **Acciones del Manejador de Autenticación:**
    - **`Authenticate`:** Construye la identidad del usuario basándose en el contexto de la solicitud (ej., desde cookies o un token JWT).
    - **`Challenge`:** Invocado por la autorización cuando un usuario no autenticado solicita un recurso restringido. Indica al usuario qué mecanismo de autenticación usar (ej., redirigir a una página de inicio de sesión, retornar un 401 con `www-authenticate`).
    - **`Forbid`:** Llamado por la autorización cuando un usuario *autenticado* intenta acceder a un recurso al que no tiene permiso. Indica al usuario que está autenticado pero no autorizado (ej., redirigir a una página de "acceso denegado", retornar un 403).

4. Flujos de Autenticación Seguros y Almacenamiento de Secretos

- **Identidades Administradas (Managed Identities):** Para servicios de Azure, las identidades administradas son la opción de autenticación más segura. Permiten autenticarse a servicios sin almacenar credenciales en código, variables de entorno o archivos de configuración (ej., Azure SQL, Azure Storage).
- **Evitar Resource Owner Password Credentials Grant (ROPC):** Se desaconseja el uso de ROPC Grant, ya que expone la contraseña del usuario al cliente y representa un riesgo de seguridad significativo.
- **Pautas para Datos de Configuración:**
    - Nunca almacenar contraseñas o datos sensibles en el código o en archivos de configuración de texto plano.
    - Usar la herramienta **Secret Manager** para almacenar secretos durante el desarrollo.
    - No usar secretos de producción en entornos de desarrollo o prueba.

    - Especificar secretos fuera del proyecto para evitar que se suban accidentalmente a repositorios de código fuente.

5. Proveedores de Autenticación por Inquilino (Multi-tenant)

ASP.NET Core no tiene una solución integrada para la autenticación multi-inquilino. Para este tipo de escenarios, se recomienda considerar frameworks de terceros como:

- **Orchard Core:** Un framework de aplicación modular y multi-inquilino de código abierto (también un CMS).
- **ABP Framework:** Soporta patrones arquitectónicos como microservicios y multi-tenancy.
- **Finbuckle.MultiTenant:** Una biblioteca ligera que proporciona resolución de inquilinos y aislamiento de datos.

ASP.NET Core Identity

Es una API completa diseñada para gestionar la funcionalidad de inicio de sesión de la interfaz de usuario (UI) en aplicaciones ASP.NET Core. Permite a los desarrolladores administrar usuarios, contraseñas, datos de perfil, roles, reclamaciones (claims), tokens, confirmación por correo electrónico y más. Los usuarios pueden crear cuentas almacenadas directamente en Identity o utilizar proveedores de inicio de sesión externos como Facebook, Google, Microsoft Account y Twitter.

1. ¿Qué es ASP.NET Core Identity?

- **Funcionalidad Principal:** Proporciona un sistema robusto para la gestión de usuarios, incluyendo registro, inicio y cierre de sesión, restablecimiento de contraseñas, y verificación de correo electrónico.
- **Almacenamiento de Datos:** Típicamente se configura para usar una base de datos SQL Server para almacenar nombres de usuario, contraseñas y datos de perfil, aunque se puede adaptar a otros almacenes persistentes.
- **Distinción Clave:** ASP.NET Core Identity *no está relacionado* con la **Plataforma de Identidad de Microsoft** (Microsoft Identity Platform, evolución de Azure AD), que es una solución de identidad alternativa para autenticación y autorización en aplicaciones ASP.NET Core.

2. Seguridad para Web APIs y SPAs

Mientras que ASP.NET Core Identity añade funcionalidad de inicio de sesión UI a aplicaciones web tradicionales, para asegurar Web APIs y Single Page Applications (SPAs) se recomienda usar soluciones como:

- **Microsoft Entra ID**
- **Azure Active Directory B2C (Azure AD B2C)**
- **Duende Identity Server:** Un framework OpenID Connect y OAuth 2.0 para ASP.NET Core que habilita características como Autenticación como Servicio (AaaS), Single Sign-On/Off (SSO), control de acceso para APIs y Federation Gateway. (Nota importante: Duende Identity Server puede requerir una tarifa de licencia para uso en producción).

3. Creación de Proyectos con Autenticación (Cuentas Individuales)

Las plantillas de proyecto de ASP.NET Core permiten crear aplicaciones con autenticación integrada utilizando "Cuentas Individuales". Los componentes de UI de Identity se incluyen directamente en el proyecto o como una Razor Class Library (RCL).

- **Blazor Web App:** Al crear un proyecto Blazor Web App, se selecciona "Individual Accounts" como tipo de autenticación. Los componentes Razor de Identity (ej., `/Components/Account/Pages/Register`, `/Login`) se generan directamente en la carpeta `Components/Account`.

- **Razor Pages App:** Para Razor Pages, al seleccionar "Individual Accounts", ASP.NET Core Identity se proporciona como una Razor Class Library (RCL). Las páginas de Identity se exponen a través del área `Identity` (ej., `/Areas/Identity/Pages/Account/Register`).
- **MVC App:** Similar a Razor Pages, las aplicaciones MVC también obtienen Identity como una RCL basada en Razor Pages, con las mismas rutas de área `Identity`.

Una vez creado el proyecto, es necesario **aplicar las migraciones** (ej., `Update-Database` en PMC) para inicializar la base de datos de Identity.

4. Configuración de Servicios de Identity (`Program.cs`)

La configuración de Identity se realiza en el archivo `Program.cs`. El patrón típico implica:

1. **Configurar `DbContext`**: Se añade el `DbContext` de la aplicación, que Identity utilizará para almacenar los datos del usuario.
2. **Añadir Identity**: Se utiliza `builder.Services.AddDefaultIdentity<IdentityUser>(...)` para añadir los servicios de Identity con la configuración por defecto. Esto es un atajo para `AddIdentity`, `AddDefaultUI` y `AddDefaultTokenProviders`.

Configurar Opciones de Identity (`IdentityOptions`): Se pueden personalizar las políticas de contraseña, configuración de bloqueo de cuenta, y reglas para nombres de usuario y correos electrónicos.

```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Configuración de contraseña, bloqueo de cuenta, usuario
    options.Password.RequiredLength = 6;
    options.Lockout.MaxFailedAccessAttempts = 5;
});
// Configuración de la cookie de autenticación (ej., tiempo de expiración)
builder.Services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Identity/Account/Login";
});
```

3. **Habilitar Middleware**: En el pipeline de solicitudes (`app.Use...`), `app.UseAuthentication()` debe llamarse antes de `app.UseAuthorization()` y después de `app.UseRouting()`.

5. Flujo de Autenticación Detallado (Registrar, Iniciar Sesión, Cerrar Sesión)

Las plantillas de Identity proporcionan la lógica de UI y backend para estas operaciones:

- **Registro (`Register`)**:
  - Cuando un usuario se registra, `_userManager.CreateAsync(user, password)` crea la cuenta.
  - Se genera un token de confirmación de correo electrónico (`GenerateEmailConfirmationTokenAsync`).
  - Si `options.SignIn.RequireConfirmedAccount` es `true`, el usuario es redirigido a la página de confirmación de registro y no se inicia sesión automáticamente.
  - En producción, `DisplayConfirmAccountLink` en `RegisterConfirmation.cshtml.cs` debe ser `false` para deshabilitar la confirmación de cuenta automática (solo para pruebas).
- **Inicio de Sesión (`Login`)**:
  - Al enviar el formulario de inicio de sesión, `_signInManager.PasswordSignInAsync()` intenta autenticar al usuario.

    - Maneja resultados como éxito, requerimiento de doble factor de autenticación (`RequiresTwoFactor`), o cuenta bloqueada (`IsLockedOut`).
- **Cierre de Sesión (`Logout`)**:
    - La acción `OnPost` en `LogoutModel` (invocada típicamente por un formulario POST) llama a `_signInManager.SignOutAsync()`.
    - `SignOutAsync()` limpia las reclamaciones del usuario almacenadas en la cookie de autenticación, cerrando la sesión. Se requiere una redirección para que el navegador realice una nueva solicitud y actualice el estado del usuario.

6. Pruebas y Exploración de Identity

- Para probar Identity, se puede añadir el atributo `[Authorize]` a una página o controlador. Al intentar acceder a un recurso protegido sin estar autenticado, se redirigirá a la página de inicio de sesión.
- Se puede explorar el código fuente completo de la UI de Identity, generar el código fuente completo de la UI de Identity en el proyecto para examinarlo, o usar el depurador para entender el flujo.
- Los paquetes NuGet de Identity forman parte del *shared framework* de ASP.NET Core, siendo `Microsoft.AspNetCore.Identity` el paquete principal con las interfaces, y `Microsoft.AspNetCore.Identity.EntityFrameworkCore` para la integración con EF Core.

ASP.NET Core Identity utiliza valores por defecto para configuraciones como la política de contraseñas, el bloqueo de cuentas y la configuración de cookies. Estos ajustes pueden ser personalizados y sobrescritos durante el inicio de la aplicación.

1. Opciones de Identity (`IdentityOptions`)

La clase `IdentityOptions` permite configurar el sistema de Identity y debe establecerse *después* de llamar a `AddIdentity` o `AddDefaultIdentity`.

- **`ClaimsIdentityOptions`**: Define los tipos de *claim* utilizados para roles, `security stamp`, identificador de usuario y nombre de usuario.
    - `DefaultRoleClaimType`: Por defecto, `ClaimTypes.Role`.
    - `SecurityStampClaimType`: Por defecto, `AspNet.Identity.SecurityStamp`.
    - `UserIdClaimType`: Por defecto, `ClaimTypes.NameIdentifier`.
    - `UserNameClaimType`: Por defecto, `ClaimTypes.Name`.
- **Opciones de Bloqueo (`LockoutOptions`)**: Controlan el comportamiento de bloqueo de cuentas tras intentos fallidos de inicio de sesión.
    - `DefaultLockoutTimeSpan`: Tiempo que un usuario permanece bloqueado (por defecto, 5 minutos).
    - `MaxFailedAccessAttempts`: Número de intentos fallidos antes del bloqueo (por defecto, 5).
    - `AllowedForNewUsers`: Si un nuevo usuario puede ser bloqueado (por defecto, `true`).
    - **Nota**: Un inicio de sesión exitoso reinicia el contador de intentos fallidos y el temporizador de bloqueo. El `PasswordSignInAsync` debe tener `lockoutOnFailure: true` para activar el bloqueo.

```
builder.Services.Configure<IdentityOptions>(options =>
{
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
```

});

- **Opciones de Contraseña (`PasswordOptions`)**: Definen los requisitos para las contraseñas.
    - `RequireDigit`: Requiere un dígito (por defecto, `true`).
    - `RequiredLength`: Longitud mínima (por defecto, 6).
    - `RequireLowercase`: Requiere minúsculas (por defecto, `true`).
    - `RequireNonAlphanumeric`: Requiere caracteres no alfanuméricos (por defecto, `true`).
    - `RequiredUniqueChars`: Número de caracteres distintos requeridos (solo ASP.NET Core 2.0+, por defecto, 1).
    - `RequireUppercase`: Requiere mayúsculas (por defecto, `true`).

```
builder.Services.Configure<IdentityOptions>(options =>
{
   options.Password.RequireDigit = true;
   options.Password.RequireLowercase = true;
   options.Password.RequireNonAlphanumeric = true;
   options.Password.RequireUppercase = true;
   options.Password.RequiredLength = 6;
   options.Password.RequiredUniqueChars = 1;
});
```

- **Opciones de Inicio de Sesión (`SignInOptions`)**: Controlan los requisitos para iniciar sesión.
    - `RequireConfirmedEmail`: Requiere un email confirmado para iniciar sesión (por defecto, `false`).
    - `RequireConfirmedPhoneNumber`: Requiere un número de teléfono confirmado para iniciar sesión (por defecto, `false`).
- **Opciones de Tokens (`TokenOptions`)**: Gestionan los proveedores de tokens para diversas acciones (ej., autenticador, cambio de email/teléfono, confirmación de email, reseteo de contraseña).
- **Opciones de Usuario (`UserOptions`)**: Definen las restricciones para nombres de usuario y correos electrónicos.
    - `AllowedUserNameCharacters`: Caracteres permitidos en el nombre de usuario (por defecto, alfanuméricos y `-_@+.`).
    - `RequireUniqueEmail`: Requiere que cada usuario tenga un correo electrónico único (por defecto, `false`).

2. Opciones de Hash de Contraseña (`PasswordHasherOptions`)

Permiten configurar cómo se hashean las contraseñas.

- `CompatibilityMode`: Define el modo de compatibilidad del algoritmo de hashing (por defecto, `IdentityV3`).
- `IterationCount`: Número de iteraciones PBKDF2 utilizadas para hashear nuevas contraseñas cuando `CompatibilityMode` es `IdentityV3` (por defecto, 100000). Un mayor número de iteraciones aumenta la seguridad, pero también el tiempo de cálculo.

```
builder.Services.Configure<PasswordHasherOptions>(option =>
{
   option.IterationCount = 12000; // Ejemplo de configuración
});
```

3. Configuración de Cookies de Autenticación (`ConfigureApplicationCookie`)

Se configura la cookie de autenticación de la aplicación en `Program.cs`. Debe llamarse *después* de `AddIdentity` o `AddDefaultIdentity`.

```
builder.Services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Identity/Account/AccessDenied"; // Ruta de acceso denegado
    options.Cookie.Name = "YourAppCookieName"; // Nombre de la cookie
    options.Cookie.HttpOnly = true; // Solo accesible por HTTP, no JavaScript
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60); // Tiempo de expiración de la cookie
    options.LoginPath = "/Identity/Account/Login"; // Ruta de inicio de sesión
    options.SlidingExpiration = true; // Renueva la cookie si el usuario está activo
});
```

4. Requerir Autenticación Globalmente

Para que todos los usuarios deban estar autenticados por defecto, se puede configurar una política de autorización globalmente. (El artículo remite a otra sección para más detalles).

5. `ISecurityStampValidator` y Cierre de Sesión en Todas Partes

Identity utiliza la interfaz `ISecurityStampValidator` para regenerar el `ClaimsPrincipal` de un usuario cuando ocurren eventos sensibles a la seguridad (ej., cambio de contraseña, adición a un rol). El validador verifica periódicamente que el *security stamp claim* del usuario (almacenado en la cookie) no haya cambiado con respecto al almacenado en la base de datos. Si el `security stamp` cambia, la cookie se invalida, forzando al usuario a volver a autenticarse.

Esto permite implementar una funcionalidad de "cerrar sesión en todas partes" llamando a `userManager.UpdateSecurityStampAsync(user)`, lo que invalida todas las sesiones activas del usuario. El intervalo de validación se puede configurar:

```
builder.Services.Configure<SecurityStampValidatorOptions>(o =>
        o.ValidationInterval = TimeSpan.FromMinutes(1)); // Valida cada minuto
```

Autenticación JWT Bearer en APIs

La autenticación JWT Bearer es comúnmente utilizada para APIs y funciona de manera similar a la autenticación por cookies, pero con tokens que pueden enviarse a cualquier servidor para autenticación, no solo al dominio emisor. Un JWT es un token autocontenido que encapsula información sobre un recurso API o un cliente. El cliente que solicita el JWT puede acceder a los datos de una API utilizando el encabezado `Authorization` con un *bearer token*.

JWT Bearer Authentication proporciona:

- **Autenticación**: El `JwtBearerHandler` valida el token y extrae la identidad del usuario a partir de sus *claims*.
- **Autorización**: Los *bearer tokens* habilitan la autorización al proporcionar una colección de *claims* que representan los permisos del usuario o de la aplicación.
- **Autorización Delegada**: Cuando se usa un token de acceso específico de usuario para autenticar entre APIs, en lugar de un token de acceso a nivel de aplicación.

Tipos de Tokens

Se desaconseja generar tokens de acceso o ID propios, excepto para pruebas, ya que pueden llevar a vulnerabilidades de seguridad si no se adhieren a los estándares. Se recomienda usar **OpenID Connect 1.0** o un estándar **OAuth** para crear tokens de acceso.

- **Tokens de Acceso (`Access Tokens`)**:
    - Son cadenas usadas por una aplicación cliente para hacer solicitudes a un servidor que implementa una API.

    - No deben ser leídos o interpretados por la aplicación cliente; son solo para hacer solicitudes a la API.
    - Se envían típicamente en el encabezado `Authorization` como un *bearer token*.
    - Pueden ser **tokens de acceso de aplicación** (almacenados una vez por la app hasta su expiración) o **tokens de acceso delegados** (almacenados por usuario, en una cookie o caché segura del servidor). Se recomienda usar tokens de acceso delegados cuando un usuario está involucrado.
    - Pueden ser **sender-constrained tokens** (ej. DPoP, MTLS) que requieren que el cliente pruebe posesión de una clave privada para usar el token.
- **Tokens de ID (`ID Tokens`)**:
    - Confirman la autenticación exitosa de un usuario.
    - Siempre están en formato JWT y contienen *claims* con información del usuario.
    - **Nunca deben usarse para acceder a APIs**.
- **Otros Tokens**: Incluyen *refresh tokens* (para renovar el token de acceso sin reautenticación) y *OAuth JAR tokens* (para enviar solicitudes de autorización de forma segura).

Uso de Tokens JWT para Asegurar una API

Cuando una API usa tokens de acceso JWT para la autorización, solo se valida el token de acceso; la API concede o deniega el acceso en función de este. Si la solicitud no está autorizada, se devuelve una respuesta **401 (Unauthorized)** o **403 (Forbidden)**. La API no debe redirigir al usuario al proveedor de identidad. La aplicación cliente es responsable de adquirir el token adecuado.

- **401 Unauthorized**: Indica que el token de acceso proporcionado no cumple los estándares requeridos (ej., firma inválida, expiración, *claims* incorrectos como `aud` o `iss`). Debe incluir un encabezado `WWW-Authenticate`.
- **403 Forbidden**: Indica que el usuario autenticado carece de los permisos necesarios para acceder al recurso, distinto de un problema de autenticación. Se implementa en ASP.NET Core mediante requisitos y políticas de autorización, o autorización basada en roles.

Rol de OIDC y/o OAuth con Bearer Tokens

Una API que usa tokens de acceso JWT solo valida el token, no cómo se obtuvo. **OpenID Connect (OIDC)** y **OAuth 2.0** son los *frameworks* estandarizados y seguros para la adquisición de tokens. Se recomienda encarecidamente usarlos debido a la complejidad de la adquisición segura de tokens:

- **Para apps que actúan en nombre de un usuario y una aplicación**: OIDC es la opción preferida (flujo de código confidencial con PKCE para web apps). Los tokens de acceso pueden almacenarse en una cookie si la app es ASP.NET Core con autenticación OIDC del lado del servidor (`SaveTokens`).
- **Para apps sin usuario (servicio a servicio)**: El flujo de credenciales de cliente de OAuth 2.0 es adecuado para obtener tokens de acceso de aplicación.

Implementación de la Autenticación de Token JWT Bearer

El paquete NuGet `Microsoft.AspNetCore.Authentication.JwtBearer` se usa para validar los tokens JWT Bearer. La validación debe incluir:

- **Firma**: Para asegurar confianza e integridad.
- **Claim `iss` (Issuer)**: Con el valor esperado.
- **Claim `aud` (Audience)**: Con el valor esperado.
- **Expiración del token**.

- Otros *claims* requeridos para tokens de acceso OAuth 2.0: `sub`, `client_id`, `iat`, `jti`.

Configuración básica (`AddJwtBearer`):

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(jwtOptions =>
{
 jwtOptions.Authority = "https://{--your-authority--}"; // URL del emisor (IdP)
 jwtOptions.Audience = "https://{--your-audience--}"; // Audiencia esperada del token
});
```

Se puede realizar una **validación explícita** usando `TokenValidationParameters` para definir validaciones más específicas (ej., múltiples *issuers* o *audiences*).
Múltiples Esquemas JWT: Las APIs a menudo necesitan aceptar tokens de diferentes emisores. Esto se puede lograr con APIs separadas o usando `AddPolicyScheme` para definir múltiples esquemas de autenticación y seleccionar el apropiado basándose en las propiedades del token.
Forzar Autenticación Bearer:

- `SetDefaultPolicy` en `AddAuthorizationBuilder` puede requerir autenticación para todas las solicitudes, incluso sin el atributo `[Authorize]`.
- El atributo `[Authorize]` en controladores o Minimal APIs puede forzar la autenticación. Si hay múltiples esquemas, el esquema Bearer debe ser el predeterminado o especificarse explícitamente (`[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]`).

Enfoques Recomendados para Crear un JWT

- **Usar Estándares**: Siempre se deben usar estándares como OpenID Connect o OAuth al crear tokens de acceso en producción. Evitar la creación de tokens personalizados.
- **Usar Claves Asimétricas**: Siempre. La clave pública se usa para validar la firma del token.
- **Nunca Crear un Token de Acceso desde una Solicitud de Usuario/Contraseña**: Las solicitudes de nombre de usuario/contraseña no están autenticadas y son vulnerables. Los tokens de acceso deben crearse solo a través de flujos estándar de OpenID Connect o OAuth.
- **Usar Cookies (para Web Apps Seguras)**: Los tokens de acceso deben almacenarse en un servidor de confianza, y solo una cookie `HttpOnly` segura se comparte en el navegador del cliente.
- **APIs *Downstream* (Delegated Authorization)**: Para que una API acceda a otras APIs en nombre del usuario, se recomienda una estrategia de confianza cero con un token de acceso de usuario delegado. Esto se puede implementar con:
  - **OAuth 2.0 Token Exchange**: Solicitando un nuevo token delegado.
  - **Microsoft Identity Web (On-Behalf-Of flow)**: Una librería sencilla y segura para Azure Entra ID.
  - **Usar el mismo token delegado**: Fácil de implementar, pero el token tiene acceso completo a todas las APIs *downstream*.
  - **OAuth client credentials flow (con token de aplicación)**: Fácil de implementar, pero no es un token delegado de usuario.

Manejo de Tokens de Acceso

- Los tokens de acceso deben rotarse y persistirse en un servidor seguro.
- En una aplicación web, las cookies se usan para asegurar la sesión y pueden almacenar tokens vía la opción `SaveTokens`.

- `SaveTokens` no refresca automáticamente los tokens (funcionalidad planeada para .NET 10), por ahora se puede hacer manualmente o usar librerías como `Duende.AccessTokenManagement.OpenIdConnect`.
- **Nunca abrir tokens de acceso en una aplicación UI**: Los tokens de acceso están destinados solo para las APIs.
- **No enviar tokens de ID a las APIs**.

YARP (Yet Another Reverse Proxy)

YARP es una tecnología útil para manejar solicitudes HTTP y reenviarlas a otras APIs. Puede implementar lógica de seguridad para adquirir nuevas credenciales de acceso y se usa frecuentemente con la arquitectura de seguridad **Backend for Frontend (BFF)**.

Pruebas de APIs

Se pueden usar pruebas de integración y contenedores con tokens de acceso para probar APIs seguras. La herramienta `dotnet user-jwts` puede crear tokens JWT para pruebas.
Advertencia: No se deben implementar flujos de seguridad inseguros para pruebas en producción. Las herramientas como Swagger UI, Curl y otras, cuando se usan para pruebas, pueden requerir debilitar la seguridad de la API, lo cual debe limitarse estrictamente a entornos de prueba dedicados y aislados.

Introducción a la Autorización en ASP.NET Core

La **autorización** en ASP.NET Core es el proceso que determina **qué puede hacer un usuario** dentro de una aplicación, una vez que su identidad ha sido verificada. Es un concepto distinto pero complementario a la **autenticación**, que es el proceso de verificar la identidad del usuario. La autorización se apoya siempre en un mecanismo de autenticación previo.
Por ejemplo, un usuario administrativo podría tener autorización para crear, añadir, editar y eliminar documentos en una biblioteca, mientras que un usuario no administrativo solo podría estar autorizado para leerlos.

Tipos de Autorización

ASP.NET Core ofrece un modelo de autorización flexible y potente, que incluye dos tipos principales:

1. **Autorización Basada en Roles (Declarativa Simple):**
    - Este modelo permite asignar roles a los usuarios (ej., "Administrador", "Editor") y luego restringir el acceso a recursos o funcionalidades basándose en la pertenencia a esos roles. Es un enfoque directo y declarativo.
2. **Autorización Basada en Políticas (Modelo Rico):**
    - Este es un modelo más avanzado y expresivo. La autorización se define a través de **requisitos** (lo que se necesita para acceder) y **manejadores** que evalúan las *claims* (afirmaciones sobre la identidad del usuario) de un usuario contra esos requisitos.
    - Permite comprobaciones imperativas que pueden evaluar no solo la identidad del usuario, sino también propiedades específicas del recurso al que el usuario intenta acceder. Esto permite una lógica de autorización muy granular y personalizada.

Espacios de Nombres

Los componentes clave de autorización, incluyendo los atributos `AuthorizeAttribute` (para restringir el acceso) y `AllowAnonymousAttribute` (para permitir el acceso no autenticado), se encuentran en el espacio de nombres `Microsoft.AspNetCore.Authorization`.

Autorización Simple en ASP.NET Core

La autorización en ASP.NET Core se controla principalmente con el atributo `[Authorize]` y sus diversos parámetros. En su forma más básica, aplicar este atributo a un controlador, una acción o una Razor Page **limita el acceso a ese componente solo a usuarios autenticados**.

1. Uso del Atributo `[Authorize]`

En un Controlador Completo: Al aplicar `[Authorize]` a una clase de controlador, todas las acciones dentro de ese controlador requerirán que el usuario esté autenticado para acceder.

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login() { }
    public ActionResult Logout() { }
}
```

- En este ejemplo, tanto `Login` como `Logout` requerirían autenticación.

En una Acción Específica: Si solo una acción particular necesita autorización, se puede aplicar el atributo directamente a esa acción.

```
public class AccountController : Controller
{
   public ActionResult Login() { }

   [Authorize]
   public ActionResult Logout() { }
}
```

- Aquí, solo `Logout` requiere que el usuario esté autenticado.

Excluir con `[AllowAnonymous]`: Se puede usar el atributo `[AllowAnonymous]` para permitir el acceso a usuarios no autenticados (o anónimos) a acciones individuales, incluso si el controlador está marcado con `[Authorize]`.

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login() { } // Accesible por todos

    public ActionResult Logout() { } // Solo para usuarios autenticados
}
```

- **¡Advertencia Importante!**: `[AllowAnonymous]` anula completamente cualquier requisito de autorización. Si se aplica `[AllowAnonymous]` a nivel de controlador, cualquier atributo `[Authorize]` en el mismo controlador o en sus métodos de acción será ignorado. El *middleware* de autenticación seguirá ejecutándose, pero no necesitará tener éxito.

En una Razor Page: Se aplica `[Authorize]` a la clase `PageModel` de una Razor Page para restringir el acceso a la página completa a usuarios autenticados.

```
[Authorize]
public class LogoutModel : PageModel
{
    public async Task OnGetAsync() { }
    public async Task<IActionResult> OnPostAsync() { }
}
```

2. Atributo `[Authorize]` y Razor Pages (Manejadores de Página)

Es crucial entender que el atributo `[Authorize]` **no puede aplicarse directamente a los manejadores de página de Razor Pages** (ej., `OnGet`, `OnPost`, `OnGetAsync`).

- **Enfoque Recomendado por Microsoft**: Si se tienen requisitos de autorización diferentes para distintos manejadores dentro de lo que conceptualmente sería una "página", la recomendación es **usar un controlador ASP.NET Core MVC** en su lugar. Esta es la solución menos compleja y la preferida.

- **Soluciones Alternativas (si no se usa MVC)**:

    1. **Páginas Separadas**: Crear páginas Razor separadas para los manejadores que requieren autorización diferente. El contenido compartido se puede mover a vistas parciales (`partial views`). Esta es la opción recomendada cuando es posible.
    2. **Filtro Personalizado**: Para contenido que debe compartir una página común, se puede escribir un filtro personalizado que implemente la autorización como parte de `IAsyncPageFilter.OnPageHandlerSelectionAsync`. Este enfoque se demuestra en el proyecto `PageHandlerAuth` de GitHub y permite aplicar un atributo de autorización a manejadores específicos (ej., `[AuthorizePageHandler]` en `OnPostAuthorized`).
        - **Limitaciones del Filtro Personalizado**: Este enfoque **no se compone** con atributos de autorización aplicados a la página, el `PageModel` o globalmente. Esto significa que la autenticación y autorización podrían ejecutarse varias veces, y la solución no funciona en conjunto con el resto del sistema de autenticación y autorización de ASP.NET Core de forma predeterminada, requiriendo una verificación cuidadosa de su correcto funcionamiento.

Importante: No hay planes para soportar el atributo `[Authorize]` directamente en los manejadores de página de Razor Pages.

Habilitar Solicitudes de Origen Cruzado (CORS) en ASP.NET Core

La seguridad del navegador impone una **política de mismo origen** que impide que una página web realice solicitudes a un dominio diferente del que la sirvió. Esta restricción previene la lectura de datos sensibles por parte de sitios maliciosos. **Cross-Origin Resource Sharing (CORS)** es un estándar W3C que permite a un servidor relajar esta política, permitiendo explícitamente algunas solicitudes de origen cruzado mientras rechaza otras.

Importante: CORS *no* es una característica de seguridad; más bien, relaja la seguridad. Una API no es más segura por permitir CORS. Es el cliente (navegador) quien impone la política CORS; el servidor simplemente responde a la solicitud.

1. Mismo Origen

Dos URLs tienen el mismo origen si tienen esquemas, *hosts* y puertos idénticos (según RFC 6454). Cambiar cualquiera de estos elementos (ej., dominio, subdominio, esquema HTTP/HTTPS, puerto) resulta en un origen diferente.

2. Formas de Habilitar CORS

Hay tres maneras principales de habilitar CORS en ASP.NET Core:

1. **En el *middleware***: Usando una política con nombre o una política por defecto.
2. **Con el enrutamiento de endpoints*: Aplicando CORS a *endpoints* específicos.
3. **Con el atributo `[EnableCors]`**: Proporciona el control más fino.

Advertencia: El *middleware* `UseCors` debe llamarse en el orden correcto en el *pipeline* de la aplicación, generalmente después de `UseRouting` pero antes de `UseAuthorization` y `UseResponseCaching`.

3. CORS con Política con Nombre y *Middleware*

Este es un enfoque común para aplicar una política CORS a todos los *endpoints* de la aplicación:

Registrar la política en `Program.cs`: Se usa `AddCors` para añadir los servicios CORS y definir una o más políticas nombradas.

```
var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      policy =>
                      {
                          policy.WithOrigins("http://example.com", "http://www.contoso.com")
                                .AllowAnyHeader()
                                .AllowAnyMethod(); // Ejemplo de encadenamiento
                      });
});
```

1. *WithOrigins*: Define los orígenes permitidos. La URL *no* debe contener una barra final (`/`). *AllowAnyHeader()*: Permite cualquier encabezado de solicitud. *AllowAnyMethod()*: Permite cualquier método HTTP.

Habilitar el *middleware* CORS: Llamar a `UseCors` en el *pipeline* de la aplicación, especificando el nombre de la política.

```
app.UseRouting();
app.UseCors(MyAllowSpecificOrigins); // Debe ir después de UseRouting
app.UseAuthorization();
app.MapControllers();
```

2. Esto aplica la política `_myAllowSpecificOrigins` a todos los *endpoints* de los controladores.

4. CORS con Política por Defecto y *Middleware*

Se puede definir una política por defecto llamando a `AddDefaultPolicy` en `AddCors`. Luego, se habilita el *middleware* CORS sin especificar un nombre de política:

```
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(policy =>
    {
        policy.WithOrigins("http://example.com", "http://www.contoso.com");
    });
});
// ...
app.UseRouting();
app.UseCors(); // Habilita la política por defecto
// ...
```

Esto aplicará la política por defecto a todos los *endpoints*.

5. Habilitar CORS con *Endpoint Routing*

Con el enrutamiento de *endpoints*, CORS se puede habilitar por *endpoint* usando los métodos de extensión `RequireCors`:

```
// ... (AddCors y UseCors sin nombre de política)
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/echo", context => context.Response.WriteAsync("echo"))
```

```
        .RequireCors(MyAllowSpecificOrigins); // Aplica política a este endpoint
    endpoints.MapControllers()
        .RequireCors(MyAllowSpecificOrigins); // Aplica política a todos los controladores
    endpoints.MapGet("/echo2", context => context.Response.WriteAsync("echo2")); // No
permite CORS
});
```

En este caso, `app.UseCors()` habilita el *middleware* pero no aplica CORS si no hay una política por defecto. La política se aplica explícitamente a los *endpoints* individuales. El atributo `[DisableCors]` **no desactiva CORS** habilitado por `RequireCors`.

6. Habilitar CORS con Atributos (`[EnableCors]`)

El atributo `[EnableCors]` permite un control más granular, habilitando CORS solo para *endpoints* seleccionados:

- `[EnableCors]` (sin parámetros) especifica la política por defecto.
- `[EnableCors("{Policy String}")]` especifica una política con nombre.

Puede aplicarse a:

- `PageModel` de Razor Page.
- Controladores.
- Métodos de acción del controlador.

Advertencia: No se recomienda combinar políticas (ej., `[EnableCors]` en un controlador y `UseCors` en el *middleware* global). Elija uno u otro. Para el control más fino, use `[EnableCors("MyPolicy")]` sin una política por defecto global ni `endpoint routing` con `RequireCors`.

7. Deshabilitar CORS (`[DisableCors]`)

El atributo `[DisableCors]` se puede usar para deshabilitar CORS en una acción específica, incluso si el controlador tiene `[EnableCors]`. Sin embargo, `[DisableCors]` **no deshabilita CORS habilitado por el *endpoint routing***.

```
[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // ...
    [DisableCors]
    [HttpGet("{action}")]
    public IActionResult GetValues2() { /* ... */ } // CORS deshabilitado para esta acción
}
```

8. Opciones de Política CORS

`AddPolicy` en `Program.cs` permite configurar varias opciones:

- **`WithOrigins()`**: Define los orígenes permitidos. **No usar `AllowAnyOrigin()` junto con `AllowCredentials()`**, ya que es una configuración insegura y puede llevar a falsificación de solicitudes de sitio cruzado. `SetIsOriginAllowedToAllowWildcardSubdomains()` permite subdominios comodín.
- **`WithMethods()` / `AllowAnyMethod()`**: Especifica los métodos HTTP permitidos.
- **`WithHeaders()` / `AllowAnyHeader()`**: Define los encabezados de solicitud permitidos. `AllowAnyHeader` permite cualquier encabezado de autor. Una coincidencia de política con `WithHeaders` es estricta.

- **`WithExposedHeaders()`**: Expone encabezados de respuesta adicionales al cliente, ya que por defecto los navegadores solo exponen los encabezados de respuesta "simples".
- **`AllowCredentials()`**: Permite que las solicitudes de origen cruzado incluyan credenciales (cookies, esquemas de autenticación HTTP). El cliente debe configurar `XMLHttpRequest.withCredentials = true` o similar. Habilitar esto es un riesgo de seguridad si `WithOrigins` no es específico.
- **`SetPreflightMaxAge()`**: Establece el encabezado `Access-Control-Max-Age` para indicar cuánto tiempo se puede almacenar en caché la respuesta a una solicitud *preflight*.

9. Solicitudes Preflight

Para algunas solicitudes CORS (ej., métodos PUT/DELETE, encabezados personalizados), el navegador envía una solicitud `OPTIONS` adicional, llamada **solicitud *preflight***, antes de la solicitud real. Si la solicitud *preflight* es denegada, el navegador no realiza la solicitud de origen cruzado real, incluso si el servidor responde con un `200 OK`. ASP.NET Core responde automáticamente a las solicitudes `OPTIONS` *preflight* cuando CORS está habilitado globalmente o por atributo.

10. Consideraciones Adicionales

- **Redirección HTTP a HTTPS**: Las redirecciones de HTTP a HTTPS (`UseHttpsRedirection`) pueden causar fallos `ERR_INVALID_REDIRECT` en las solicitudes *preflight* de CORS. Se recomienda que los proyectos de API rechacen las solicitudes HTTP en lugar de redirigir.
- **CORS en IIS**: Al desplegar en IIS, CORS debe ejecutarse antes de la autenticación de Windows si el servidor no permite acceso anónimo. Es posible que se necesite el módulo CORS de IIS.
- **Pruebas**: El artículo proporciona un código de ejemplo descargable para probar las configuraciones de CORS. Se advierte sobre el uso de `https://localhost:<port>` para pruebas y la configuración de IIS Express para `anonymousAuthentication: true`.