De Desarrollador a Arquitecto .NET: “Guía Práctica para Construir Software Profesional”

🧩 Parte I – Fundamentos del Ecosistema .NET

1. Historia y evolución de .NET
Describe cómo Microsoft transformó .NET desde su versión inicial hasta la unificación moderna en .NET 8.
➡️ *Objetivo:* Comprender la evolución del framework y sus implicaciones en proyectos actuales.

2. Estructura interna del CLR y del runtime
Explora el Common Language Runtime, compilación JIT, garbage collector y el modelo de ejecución.
➡️ *Objetivo:* Entender cómo se ejecuta el código y optimizar el rendimiento.

3. .NET Framework vs .NET Core vs .NET 8
Comparación técnica y estratégica para elegir la plataforma adecuada.
➡️ *Objetivo:* Saber cuándo migrar y cómo decidir la versión tecnológica.

4. Herramientas esenciales: Visual Studio, VS Code, CLI
Revisión práctica de entornos de desarrollo, extensiones y automatización con línea de comandos.
➡️ *Objetivo:* Dominar las herramientas clave del ecosistema.

5. Lenguajes: C#, VB.NET y F#
Diferencias y aplicaciones de cada lenguaje.
➡️ *Objetivo:* Conocer el lenguaje ideal para cada tipo de solución.

💡 Parte II – Bases del Desarrollo Profesional

6. Principios SOLID y buenas prácticas
Desglose de los cinco principios SOLID con ejemplos reales.
➡️ *Objetivo:* Crear código mantenible y escalable.

7. Patrones de diseño (Factory, Singleton, Strategy, etc.)
Aplicación de patrones clásicos en proyectos .NET.
➡️ *Objetivo:* Reutilizar soluciones probadas a problemas comunes.

8. Pruebas unitarias y TDD con xUnit/NUnit
Estrategias de testing y cultura de calidad.
➡️ *Objetivo:* Implementar TDD en el ciclo de desarrollo.

9. Control de versiones con Git y GitHub
Buenas prácticas de versionado, ramas y flujo GitFlow.
➡️ *Objetivo:* Mantener control de cambios en proyectos colaborativos.

10. Integración continua y pipelines básicos
Construcción automática, pruebas y despliegues continuos.
➡️ *Objetivo:* Entender los fundamentos de CI/CD.

🧱 Parte III – Arquitectura de Aplicaciones .NET

11. Monolitos bien estructurados
Diseño limpio y separación por capas dentro de una aplicación monolítica.
➡️ *Objetivo:* Aprender que un monolito bien diseñado puede ser escalable.

12. Clean Architecture
Principios de separación por responsabilidades y dependencias inversas.
➡️ *Objetivo:* Implementar Clean Architecture en proyectos reales.

13. Domain-Driven Design (DDD)
Modelado de dominios complejos con entidades, agregados y repositorios.
➡️ *Objetivo:* Alinear la arquitectura con el negocio.

14. CQRS y Mediator Pattern
Separación de lecturas y escrituras para mejorar el rendimiento y la mantenibilidad.
➡️ *Objetivo:* Implementar CQRS y patrones de mediación en APIs.

15. Inyección de dependencias y configuración avanzada
Uso de contenedores IoC en .NET.
➡️ *Objetivo:* Aplicar DI para desacoplar componentes.

🌐 Parte IV – APIs, Servicios y Comunicación

16. ASP.NET Core y Web API
Creación de APIs REST con controladores, endpoints y middlewares.
➡️ *Objetivo:* Construir servicios web robustos y seguros.

17. REST, GraphQL y gRPC
Comparativa y elección de protocolos modernos.
➡️ *Objetivo:* Seleccionar el enfoque adecuado según el escenario.

18. Versionado, logging y Swagger
Buenas prácticas de documentación y trazabilidad.
➡️ *Objetivo:* Exponer APIs mantenibles y auto documentadas.

19. Autenticación y autorización (JWT, OAuth2, Identity)
Diseño de seguridad en APIs modernas.
➡️ *Objetivo:* Proteger endpoints con estándares industriales.

20. Caching y rendimiento
Mejoras de velocidad con caching distribuido y en memoria.
➡️ *Objetivo:* Optimizar tiempos de respuesta.

🗄️ Parte V – Persistencia y Datos

21. Entity Framework Core y LINQ avanzado
ORM moderno y consultas complejas.
➡️ *Objetivo:* Dominar la persistencia de datos en .NET.

22. Repositorios y Unit of Work
Abstracción del acceso a datos.
➡️ *Objetivo:* Aplicar patrones estructurados para la capa de datos.

23. Integración con SQL Server, PostgreSQL, MongoDB
Diseño de persistencia híbrida (relacional/noSQL).
➡️ *Objetivo:* Elegir y combinar bases según necesidad.

24. Data Migrations y Seeders
Gestión del ciclo de vida de bases de datos.
➡️ *Objetivo:* Automatizar cambios en entornos productivos.

25. Optimización de consultas y caching distribuido
Técnicas de mejora de rendimiento y escalabilidad.
➡️ *Objetivo:* Evitar cuellos de botella de base de datos.

⚙️ Parte VI – Microservicios y Contenedores

26. Principios de microservicios
Diseño independiente y despliegue modular.
➡️ *Objetivo:* Comprender cuándo aplicar microservicios.

27. Comunicación entre servicios (HTTP, RabbitMQ, Kafka)
Mensajería asíncrona y eventos distribuidos.
➡️ *Objetivo:* Implementar integración entre servicios.

28. Docker y Kubernetes con .NET
Contenerización y orquestación de aplicaciones.
➡️ *Objetivo:* Crear, ejecutar y desplegar microservicios en contenedores.

29. Configuración centralizada y resiliencia (Polly)
Gestión de fallos y configuración distribuida.
➡️ *Objetivo:* Construir aplicaciones tolerantes a errores.

30. Observabilidad y logging distribuido
Monitorización y métricas con Application Insights.
➡️ *Objetivo:* Detectar problemas antes de que afecten al usuario.

☁️ Parte VII – Nube y DevOps

31. Introducción a Azure
Visión general de los servicios más usados.
➡️ *Objetivo:* Comprender el entorno cloud de Microsoft.

32. Azure App Service, Functions y Storage
Implementación de aplicaciones sin servidor.
➡️ *Objetivo:* Publicar y mantener soluciones en Azure.

33. Bases de datos y seguridad en la nube
Protección de datos y secretos.
➡️ *Objetivo:* Asegurar la información en entornos distribuidos.

34. Azure DevOps y GitHub Actions
Automatización del ciclo completo de desarrollo.
➡️ *Objetivo:* Integrar CI/CD en la nube.

35. CI/CD y despliegues automatizados
Pipeline completo desde commit hasta producción.
➡️ *Objetivo:* Minimizar errores humanos y acelerar releases.

🔒 Parte VIII – Seguridad, Escalabilidad y Rendimiento

36. Seguridad en APIs y datos
Protección ante ataques comunes (XSS, CSRF, SQLi).
➡️ *Objetivo:* Implementar medidas proactivas de seguridad.

37. Escalabilidad horizontal y vertical
Estrategias de crecimiento controlado.
➡️ *Objetivo:* Diseñar arquitecturas adaptables a la demanda.

38. Caching distribuido y balanceadores
Uso de Redis, Load Balancers y CDNs.
➡️ *Objetivo:* Mejorar resiliencia y latencia.

39. Pruebas de carga y performance tuning
Medición y ajuste fino del sistema.
➡️ *Objetivo:* Detectar y corregir cuellos de botella.

40. Auditoría y trazabilidad
Registro de operaciones críticas.
➡️ *Objetivo:* Mantener cumplimiento y transparencia.

🧭 Parte IX – Soft Skills del Arquitecto

41. Liderazgo técnico y mentoring
Cómo guiar equipos hacia la excelencia.
➡️ *Objetivo:* Desarrollar habilidades de influencia positiva.

42. Documentación y decisiones arquitectónicas (ADR)
Registrar decisiones técnicas relevantes.
➡️ *Objetivo:* Crear trazabilidad de arquitectura.

43. Comunicación con equipos y stakeholders
Traducir lo técnico en valor de negocio.
➡️ *Objetivo:* Mejorar la comunicación interdepartamental.

44. Gestión de deuda técnica y evolución del producto
Equilibrar mantenimiento y nuevas funcionalidades.
➡️ *Objetivo:* Mantener la salud del sistema a largo plazo.

🧠 Parte X – Casos Prácticos

45. Arquitectura de un ERP modular en .NET
Diseño integral de un sistema empresarial.
46. Plataforma SaaS en Azure con microservicios
Ejemplo completo con CI/CD y contenedores.
47. API Gateway y autenticación centralizada
Implementación segura para múltiples clientes.
48. Monitoreo y logging distribuido con Application Insights
Análisis real de métricas y fallos.

➡️ *Objetivo común:* Integrar todos los conocimientos en escenarios reales.

🌐 Introducción – Del desarrollador al arquitecto

"El arquitecto no deja de programar: programa sistemas enteros."

Convertirse en **arquitecto .NET** no es un ascenso jerárquico, sino una **evolución natural del pensamiento técnico**.
Es pasar de escribir funciones a **diseñar cómo esas funciones viven, se comunican y evolucionan en el tiempo**.
Es mirar un sistema no como líneas de código, sino como un organismo vivo que respira datos, escala con la demanda y se adapta a los cambios.

Este libro nace con un propósito claro: **guiarte paso a paso desde el desarrollador que domina .NET hasta el arquitecto que domina la visión del sistema**.
No necesitas ser un experto en todo, sino aprender a **conectar las piezas**: código, infraestructura, procesos y personas.

🧩 Qué aprenderás

A lo largo de estas páginas aprenderás a:

- Diseñar aplicaciones con **arquitecturas limpias y mantenibles**.
- Entender **capas, patrones y principios SOLID** con profundidad práctica.
- Construir **APIs robustas** y servicios desacoplados.
- Aplicar **seguridad, caching y observabilidad** desde el inicio.
- Migrar a **microservicios**, contenedores y la nube con Azure.
- Implementar **DevOps, CI/CD y automatización completa**.
- Desarrollar las **habilidades humanas** del arquitecto: comunicación, liderazgo y mentoring.

Y, sobre todo, a **pensar como un arquitecto**, es decir:

ver el sistema completo, anticipar el cambio y diseñar con propósito.

⚙️ Por qué .NET

Desde su reinvención con **.NET Core**, el ecosistema .NET se ha convertido en una plataforma abierta, multiplataforma y cloud-ready.
Permite crear desde microservicios hasta soluciones SaaS globales, con un stack unificado: **C#, ASP.NET Core, EF Core, Blazor, Azure y DevOps**.
Dominar este entorno te convierte en un profesional preparado para liderar proyectos en cualquier industria.

💡 Filosofía del libro

Cada parte del libro te lleva un paso más lejos:

1. **Fundamentos:** construyes el pensamiento arquitectónico.
2. **Diseño:** aplicas patrones y principios limpios.
3. **Servicios y APIs:** diseñas comunicación moderna.
4. **Persistencia:** aprendes a pensar en datos, no solo tablas.
5. **Microservicios y nube:** distribuyes tu sistema.
6. **Seguridad y escalabilidad:** garantizas resiliencia.
7. **Soft skills:** lideras equipos y decisiones.
8. **Casos prácticos:** aplicas todo en proyectos reales.

🧭 Para quién está escrito

Este libro está pensado para:

- Desarrolladores .NET que quieren **dar el salto a arquitecto**.
- Líderes técnicos que buscan **estructurar sus decisiones**.
- Estudiantes o profesionales que desean **entender cómo se construye software de nivel empresarial**.
- Cualquiera que ame el código, pero quiera aprender a **ver más allá de él**.

🚀 Cómo aprovecharlo

- Lee cada parte en orden, pero **experimenta**: ejecuta los ejemplos, diseña tus propios diagramas, compara enfoques.
- Usa el índice como guía de estudio progresivo o como referencia rápida.
- Al terminar, crea tu propio **proyecto arquitectónico** aplicando los principios del libro.

✨ Un viaje hacia la visión

Ser arquitecto .NET es unir técnica, visión y propósito.
No se trata de conocer todos los frameworks, sino de **saber cuándo usarlos y por qué**.
De escribir menos código, pero tomar **mejores decisiones**.
De construir sistemas que no solo funcionen, sino que **puedan vivir y evolucionar** durante años.

Este libro es una invitación:
a pensar diferente, a diseñar con claridad y a crear software que deje huella.

Bienvenido a tu viaje para convertirte en **Arquitecto .NET**.

✍️ Sobre el autor

Óscar de la Cuesta
@oscardelacuesta — palentino.es

Apasionado por la tecnología, la arquitectura de software y la enseñanza práctica, Óscar de la Cuesta combina más de una década de experiencia en desarrollo, consultoría y diseño de sistemas empresariales basados en .NET y Azure.

En *Arquitecto .NET: del código al diseño de sistemas inteligentes*, ofrece una **guía moderna, completa y accesible** que acompaña al lector desde los fundamentos del framework hasta las configuraciones más avanzadas de nube, microservicios y automatización, integrando en todo momento la **inteligencia artificial como herramienta de apoyo y aprendizaje**.

El libro refleja una filosofía clara: el conocimiento debe compartirse. Por eso, esta obra se publica bajo licencia abierta para inspirar a nuevos profesionales a crear, aprender y construir software de calidad.

🪪 Licencia

🧩 Parte I – Fundamentos del Ecosistema .NET

"No se puede diseñar una buena arquitectura sin entender los cimientos sobre los que se construye."

El primer paso para convertirse en arquitecto .NET es dominar los fundamentos de su ecosistema.
Esta parte ofrece una visión completa del entorno técnico, su evolución, sus componentes internos y las herramientas que hacen posible construir software moderno, escalable y seguro.
Comprenderás no solo **cómo funciona .NET**, sino **por qué** fue diseñado así y **cómo aprovecharlo al máximo** en proyectos empresariales.

Capítulo 1 – Historia y evolución de .NET

🌎 Introducción

El framework .NET nació en 2002 como respuesta de Microsoft a la creciente necesidad de crear aplicaciones multiplataforma, seguras y orientadas a servicios.
Inicialmente diseñado para Windows, evolucionó hasta convertirse en una plataforma **abierta, modular y de alto rendimiento**, con soporte para Linux, macOS, contenedores y la nube.

🔁 De .NET Framework a .NET Core

- **.NET Framework (2002-2018):** centrado en Windows y aplicaciones de escritorio (WinForms, WPF, ASP.NET clásico).
- **.NET Core (2016):** reescritura total del framework, ahora *open source* y multiplataforma.
- **.NET 5, 6, 7 y 8:** unificación definitiva. Microsoft fusionó todas las ramas en una sola plataforma: **"One .NET"**, con soporte nativo para web, escritorio, móvil, IoT e IA.

🚀 Impacto en la arquitectura

El paso a .NET Core y posteriores versiones cambió la manera de pensar la arquitectura:

- Adopción natural de **microservicios y APIs REST**.
- **Despliegue modular** mediante contenedores Docker.
- Soporte directo para **DevOps y CI/CD**.
- Alto rendimiento con **Kestrel** y compilación AOT (Ahead-of-Time).

🧠 Lecciones para el arquitecto

Un arquitecto .NET debe entender:

1. La **evolución tecnológica** de su plataforma para tomar decisiones informadas.
2. La **compatibilidad y migración** entre versiones.
3. Los cambios en **rendimiento, seguridad y despliegue** que afectan al diseño global.

✅ Objetivos de aprendizaje

- Conocer la historia y los hitos clave de .NET.
- Entender la transición a la plataforma unificada moderna.
- Saber cómo esa evolución influye en las decisiones arquitectónicas actuales.

Capítulo 2 – Estructura interna del CLR y del runtime

"Un arquitecto que entiende cómo se ejecuta su código puede diseñar sistemas más eficientes y predecibles."

⚙️ Introducción

En el corazón de toda aplicación .NET se encuentra el **CLR (Common Language Runtime)**, el motor que ejecuta el código administrado.
Conocer su estructura es esencial para comprender **cómo se comportan las aplicaciones en tiempo de ejecución**, cómo se gestionan los recursos y qué decisiones afectan el rendimiento y la estabilidad del sistema.

🧩 ¿Qué es el CLR?

El **CLR** es el componente central de .NET que proporciona:

- **Gestión automática de memoria (GC)**
- **Seguridad de tipos y aislamiento del código**
- **Compilación Just-In-Time (JIT)**
- **Manejo de excepciones unificado**
- **Interoperabilidad con código nativo (P/Invoke)**

En resumen, el CLR convierte tu código C#, VB.NET o F# en **código máquina optimizado** que se ejecuta sobre el sistema operativo host.

💉 El proceso de ejecución

Cuando compilas un programa en .NET:

1. **El compilador (CSC o VBC)** transforma el código fuente en **IL (Intermediate Language)**.
2. El IL se almacena dentro de un **ensamblado (.dll o .exe)** junto con metadatos (tipos, referencias, atributos).
3. En tiempo de ejecución, el **JIT Compiler** convierte el IL en código nativo específico de la CPU.

4. El CLR ejecuta ese código bajo su entorno administrado, supervisando memoria, seguridad y ejecución.

Este ciclo puede representarse así:

Código fuente → IL (MSIL/CIL) → JIT → Código máquina → Ejecución en CLR

Garbage Collector (GC)

El **GC** libera automáticamente la memoria que ya no se utiliza.
Opera por generaciones (0, 1 y 2), clasificando los objetos según su tiempo de vida.
Esto evita fugas de memoria y mejora la eficiencia, aunque en arquitecturas de alto rendimiento el arquitecto debe:

- Evitar la creación excesiva de objetos.
- Liberar manualmente recursos no administrados.
- Usar patrones como using o Dispose.

JIT vs AOT

- **JIT (Just-In-Time):** compila en el momento de ejecución. Rápido en desarrollo, más flexible.
- **AOT (Ahead-of-Time):** compila todo el código antes de ejecutarse, ideal para **microservicios, IoT y entornos de baja latencia**.

Con .NET 8, el AOT es una herramienta clave para arquitectos que buscan **rendimiento y tiempos de arranque ultrarrápidos**.

Rol del arquitecto

El arquitecto debe conocer:

- Cómo el CLR maneja los hilos y el *thread pool*.
- Qué impacto tienen los *async/await* en la ejecución.
- Cómo configurar la recolección de basura en entornos de servidor (Server GC).
- Cómo afecta el tipo de compilación al rendimiento global.

✅ Objetivos de aprendizaje

- Comprender el funcionamiento interno del CLR y del JIT.
- Conocer el ciclo de vida de ejecución del código .NET.
- Aplicar decisiones arquitectónicas basadas en rendimiento y eficiencia de memoria.

Capítulo 3 – .NET Framework vs .NET Core vs .NET 8

"Conocer el pasado de .NET te ayuda a diseñar arquitecturas sostenibles para el futuro."

🌎 Introducción

Durante más de dos décadas, Microsoft ha transformado .NET de un framework cerrado y dependiente de Windows a una **plataforma unificada, libre y multiplataforma**.
El arquitecto moderno debe saber qué versión elegir para cada contexto, cuándo migrar y qué impacto tiene en infraestructura, seguridad y rendimiento.

.NET Framework: la era del monolito corporativo

Período: 2002 – 2018
Plataforma: Solo Windows

Aplicaciones típicas: WinForms, WPF, ASP.NET WebForms, servicios WCF, aplicaciones de escritorio y servidores IIS.

Ventajas:

- Estabilidad y madurez comprobadas.
- Amplia compatibilidad con librerías históricas.
- Soporte nativo para Active Directory y entorno Windows.

Limitaciones:

- No multiplataforma.
- No soporta contenedores ni Docker.
- Sin actualizaciones significativas desde 4.8.

➡️ **Conclusión:** Adecuado para sistemas legados, pero obsoleto para nuevas implementaciones.

⚙️ .NET Core: la revolución moderna

Período: 2016 – 2020
Plataforma: Windows, Linux, macOS
Aplicaciones: APIs REST, microservicios, contenedores, cloud-native apps.

Ventajas:

- *Open Source* y multiplataforma.
- Motor web Kestrel ultrarrápido.
- Despliegue independiente del sistema (self-contained).
- Integración nativa con Docker y CI/CD.

Limitaciones:

- Falta de soporte para WPF/WinForms (en las primeras versiones).
- Ecosistema en evolución constante.

➡️ **Conclusión:** Base sólida para arquitecturas modernas orientadas a servicios y nube.

🚀 .NET 5 → 8: la era de la unificación

Período: 2020 – presente
Plataforma: "One .NET"
Aplicaciones: Web, escritorio, móvil, nube, IA, IoT.

Novedades clave:

- Unificación completa (.NET Framework + Core + Mono).
- Soporte de **MAUI** para aplicaciones multiplataforma.
- Mejoras en **AOT (Ahead-of-Time)** y **NativeAOT**.
- Integración total con **Azure**, **Containers** y **Blazor**.
- Librerías unificadas (System.*), alto rendimiento y mantenimiento simplificado.

.NET 8 añade:

- **Long-Term Support (LTS)** hasta 2026.
- **MEF actualizado**, mejor rendimiento del GC.
- **Minimal APIs** y mejoras en **SignalR, gRPC y EF Core 8**.

➡️ **Conclusión:** Es la base recomendada para cualquier desarrollo nuevo. Flexible, rápido y preparado para IA y microservicios.

🧠 Decisiones arquitectónicas

Un arquitecto debe analizar:

Escenario	Versión recomendada
Sistemas legados de escritorio	.NET Framework 4.8
Microservicios y APIs modernas	.NET 8
Aplicaciones multiplataforma (Windows, Android, iOS)	.NET MAUI (.NET 8)
IoT, contenedores, nube	.NET 8 AOT
Aplicaciones internas en servidores antiguos	.NET Framework / migración progresiva

---

🔄 Estrategias de migración

1. **Evaluar dependencias** (WCF, WebForms, etc.).
2. **Modernizar librerías comunes** con netstandard2.0 o superior.
3. **Refactorizar a arquitectura limpia** antes de portar código.
4. **Contenerizar progresivamente** los módulos migrados.

✅ Objetivos de aprendizaje

- Comprender diferencias y compatibilidad entre versiones.
- Evaluar el impacto de migrar a .NET moderno.
- Elegir la plataforma adecuada según el tipo de solución.

Capítulo 4 – Herramientas esenciales: Visual Studio, VS Code y CLI

"Un buen arquitecto no solo diseña sistemas, también domina las herramientas que los hacen posibles."

⚙️ Introducción

Las herramientas no hacen al arquitecto, pero sí **potencian su visión**.
En el ecosistema .NET, **Visual Studio**, **VS Code** y la **CLI (Command Line Interface)** son los tres pilares sobre los que se construyen, prueban y despliegan soluciones modernas.
Este capítulo te enseñará a usarlas estratégicamente, integrándolas en un flujo de desarrollo ágil y automatizado.

🧩 Visual Studio: la suite integral

Visual Studio es el IDE más completo para desarrollo .NET empresarial.
Ofrece un entorno gráfico potente con depuración avanzada, integración con Azure y herramientas para arquitectos.

Características clave:

- **Diseño de soluciones complejas:** proyectos múltiples, dependencias y referencias visuales.
- **Perf Analyzer y Diagnostic Tools:** análisis de rendimiento y memoria.
- **Live Unit Testing:** ejecución continua de pruebas.
- **Integración con Azure DevOps y GitHub:** CI/CD, control de versiones y revisión de código.
- **IntelliTrace y Snapshot Debugging:** depuración histórica para entornos de producción.

Consejo del arquitecto:
Crea *Plantillas de soluciones (.zip)* con estructura limpia (API, Domain, Infrastructure, UI) para estandarizar proyectos en tu organización.

💻 Visual Studio Code: la herramienta ligera y versátil

VS Code es un editor multiplataforma, ligero y totalmente extensible.
Ideal para arquitecturas distribuidas, microservicios, proyectos con contenedores o entornos híbridos (C#, Node.js, Python).

Extensiones esenciales para .NET:

- **C# (OmniSharp):** soporte completo para IntelliSense y debugging.
- **NuGet Package Manager GUI:** instalación y gestión de librerías.
- **Docker y Dev Containers:** entornos reproducibles para cada servicio.
- **REST Client:** probar endpoints sin salir del editor.
- **GitLens y GitHub Copilot:** análisis de cambios y sugerencias asistidas por IA.

Ventajas para el arquitecto:

- Compatible con Linux y macOS.
- Ideal para proyectos distribuidos en contenedores.
- Integración con terminales y scripts DevOps.

🔧 CLI (Command Line Interface)

La **.NET CLI** permite compilar, probar, publicar y desplegar aplicaciones sin entorno gráfico.
Es indispensable para **automatización, pipelines de CI/CD y Dev Containers**.

Comandos más útiles:

```
dotnet new webapi        # Crea una API REST
dotnet build             # Compila el proyecto
dotnet run               # Ejecuta la aplicación
dotnet test              # Lanza las pruebas unitarias
dotnet publish -c Release -o out  # Publica binarios
```

Trucos arquitectónicos:

- Usa `dotnet new sln` y `dotnet sln add` para componer soluciones desde scripts.
- Genera plantillas internas con `dotnet new --install .\MiPlantilla`.
- Combina con PowerShell o Bash para automatizar builds y despliegues.

Integración con DevOps y contenedores

Las tres herramientas pueden convivir en un flujo unificado:

1. **Diseñas en Visual Studio.**
2. **Automatizas y pruebas en VS Code o CLI.**
3. **Contenerizas con Docker y despliegas con Azure DevOps.**

El arquitecto debe definir **scripts reproducibles** para cada entorno (dev, test, prod) evitando configuraciones manuales.

Objetivos de aprendizaje

- Conocer el potencial de Visual Studio y VS Code como entornos de arquitectura.
- Dominar la CLI para automatizar tareas.
- Integrar las herramientas en flujos CI/CD y contenedores.

Capítulo 5 – Lenguajes: C#, VB.NET y F#

"El arquitecto no se define por el lenguaje que usa, sino por cómo diseña soluciones que trascienden el código."

🧭 Introducción

El ecosistema .NET soporta varios lenguajes que comparten el mismo runtime (CLR) y librerías base.
Esto significa que, sin importar si programas en **C#**, **VB.NET** o **F#**, todos se compilan a **IL (Intermediate Language)** y se ejecutan dentro del mismo entorno controlado.
Sin embargo, **cada lenguaje tiene su personalidad, enfoque y ventajas**, y el arquitecto debe saber cuándo usar cada uno.

🦾 C#: el estándar de la arquitectura moderna

C# es el lenguaje más extendido y versátil del ecosistema .NET. Diseñado por Anders Hejlsberg, combina elegancia, potencia y un equilibrio perfecto entre rendimiento y productividad.

Ventajas:

- Sintaxis clara y moderna, cercana a Java y C++.
- Gran soporte para programación **orientada a objetos**, **asíncrona** y **funcional**.
- Base de casi todos los frameworks modernos: ASP.NET Core, Blazor, MAUI, Unity.
- Amplio soporte comunitario, herramientas, documentación y ejemplos.

Ejemplo breve:

```
public class Usuario
{
   public string Nombre { get; set; }
   public void Saludar() => Console.WriteLine($"Hola, {Nombre}");
}
```

Uso ideal:

- Aplicaciones web, microservicios, APIs REST, Azure Functions, videojuegos (Unity), IA (ML.NET).

VB.NET: el legado empresarial

VB.NET (Visual Basic .NET) proviene de la tradición Visual Basic clásica. Aunque su uso ha disminuido, sigue presente en muchas aplicaciones corporativas, sistemas ERP y herramientas de automatización interna.

Ventajas:

- Curva de aprendizaje suave.
- Sintaxis legible, ideal para usuarios no técnicos.
- Gran compatibilidad con proyectos antiguos de escritorio (WinForms).

Ejemplo breve:

```
Public Class Usuario
    Public Property Nombre As String
    Public Sub Saludar()
        Console.WriteLine($"Hola, {Nombre}")
    End Sub
End Class
```

Uso ideal:

- Mantenimiento de sistemas heredados.
- Aplicaciones de escritorio internas.
- Automatizaciones rápidas o prototipos.

Consejo del arquitecto:
Cuando trabajes con entornos VB.NET, **define una estrategia de modernización progresiva** hacia C# o .NET MAUI, manteniendo interoperabilidad durante la transición.

🧠 F#: el lenguaje para el pensamiento funcional

F# es un lenguaje funcional, conciso y poderoso, ideal para matemáticas, ciencia de datos y cálculos complejos.
Aunque menos popular, es una joya para arquitecturas que requieren **seguridad en tipos**, **inmutabilidad** y **código expresivo**.

Ventajas:

- Enfoque en la **programación funcional pura**.
- Menos errores y mayor previsibilidad.
- Ideal para análisis financiero, IA y simulaciones.

Ejemplo breve:

```
let saludar nombre =
    printfn "Hola, %s" nombre
```

Uso ideal:

- Modelos matemáticos, procesamiento de datos, algoritmos complejos.
- Microservicios de IA o módulos de cálculo intensivo dentro de arquitecturas híbridas.

⚖️ Comparativa arquitectónica

Característica	C#	VB.NET	F#
Enfoque principal	Generalista y moderno	Legado empresarial	Funcional
Popularidad	⭐⭐⭐⭐⭐	⭐⭐	⭐⭐
Rendimiento	Muy alto	Alto	Muy alto
Curva de aprendizaje	Media	Baja	Alta
Ideal para	APIs, Web, Nube, IA	WinForms, ERPs antiguos	Data science, IA, cálculos

---

🧩 Interoperabilidad total

Todos los lenguajes .NET pueden convivir en una misma solución.
Un arquitecto puede tener un **módulo en VB.NET**, una **API en C#** y una **librería analítica en F#**, todos ejecutándose sobre el mismo CLR.
Lo importante es mantener una **arquitectura limpia y desacoplada** entre los componentes.

✅ Objetivos de aprendizaje

- Comprender las diferencias entre C#, VB.NET y F#.
- Saber cuándo usar cada lenguaje en función del proyecto.
- Entender la interoperabilidad dentro del ecosistema .NET.
- Prepararse para elegir C# como estándar en arquitecturas modernas.

Parte II – Bases del Desarrollo Profesional

"Un arquitecto no escribe más código: escribe mejor código."

El arquitecto .NET debe dominar las bases del diseño de software limpio, mantenible y extensible.
Esta parte se centra en los **principios SOLID**, las **buenas prácticas de desarrollo**, los **patrones de diseño** y las **herramientas de calidad de código**.
Aquí se sientan los cimientos para construir sistemas que duren y evolucionen sin romperse.

Capítulo 6 – Principios SOLID y buenas prácticas

"SOLID no es una moda; es el lenguaje universal de la arquitectura limpia."

Introducción

Los principios **SOLID**, definidos por Robert C. Martin ("Uncle Bob"), son un conjunto de buenas prácticas que ayudan a crear sistemas **robustos, flexibles y fáciles de mantener**.
Cada letra representa un principio que guía la forma en que las clases y módulos deben relacionarse entre sí.

Aplicarlos correctamente separa a un **buen desarrollador** de un **arquitecto excelente**.

S — Single Responsibility Principle (SRP)

Cada clase debe tener una sola razón para cambiar.
Esto significa que una clase debe enfocarse en una única tarea o responsabilidad.

Ejemplo incorrecto:

```
public class ReporteService {
    public void GenerarPDF() { }
    public void EnviarPorCorreo() { }
}
```

Ejemplo correcto:

```
public class GeneradorPDF { public void Generar() { } }
public class EnviadorCorreo { public void Enviar() { } }
```

➡️ *Beneficio:* facilita el mantenimiento, testing y reuso.

🧩 O — Open/Closed Principle (OCP)

El código debe estar abierto a extensión, pero cerrado a modificación.
En lugar de alterar clases existentes, se deben extender mediante herencia o inyección de dependencias.

```
public interface INotificador { void Notificar(string mensaje); }

public class NotificadorEmail : INotificador { ... }
public class NotificadorSMS : INotificador { ... }
```

➡️ *Beneficio:* nuevos comportamientos sin romper lo existente.

🧩 L — Liskov Substitution Principle (LSP)

Las subclases deben poder reemplazar a sus clases base sin alterar el comportamiento esperado.

```
public class Ave { public virtual void Volar() {} }
public class Pinguino : Ave { /* No puede volar → violación de LSP */ }
```

➡️ *Beneficio:* asegura coherencia y confianza al usar herencia.

🧩 I — Interface Segregation Principle (ISP)

Es mejor tener interfaces específicas que una general gigante.

```
public interface IImpresora {
    void Imprimir();
    void Escanear(); // No todos los dispositivos lo hacen
}
```

➡️ Solución: separar en `IImpresora` y `IEscaner`.

➡️ *Beneficio:* reduce dependencias innecesarias y mejora la flexibilidad.

🧩 D — Dependency Inversion Principle (DIP)

Depende de abstracciones, no de implementaciones.
El código debe comunicarse a través de interfaces o clases abstractas.

```
public class PedidoService {
    private readonly IRepositorio _repo;
    public PedidoService(IRepositorio repo) => _repo = repo;
}
```

➡️ *Beneficio:* desacopla módulos y facilita pruebas unitarias.

🧠 Aplicar SOLID como arquitecto

- Define **interfaces claras** entre módulos.
- Aplica **inyección de dependencias (DI)** desde el inicio.
- Diseña **clases pequeñas y enfocadas**.
- Usa **patrones de diseño** (Factory, Strategy, Adapter) para extender sin romper.

⚙️ Buenas prácticas adicionales

- Nombra las clases y métodos con significado.
- Evita duplicación de código (DRY).
- Escribe pruebas unitarias desde el inicio.
- Documenta decisiones arquitectónicas (ADR).
- Revisa código mediante *Code Review* y herramientas como SonarLint o Roslyn Analyzers.

✅ Objetivos de aprendizaje

- Comprender los principios SOLID y su propósito.
- Aplicarlos para crear código modular y mantenible.
- Entender su relación con patrones de diseño y arquitectura limpia.

Capítulo 7 – Patrones de diseño (Factory, Singleton, Strategy, etc.)

"Los patrones son la gramática del lenguaje arquitectónico. No reinventes la rueda: entiéndela."

Introducción

Los **patrones de diseño** son soluciones probadas a problemas recurrentes en el desarrollo de software.
Un arquitecto .NET debe conocerlos y saber cuándo aplicarlos correctamente, equilibrando **simplicidad**, **mantenibilidad** y **rendimiento**.

Estos patrones no son recetas rígidas, sino **principios reutilizables** que mejoran la comunicación técnica entre desarrolladores y aseguran coherencia en los proyectos.

Clasificación general

Los patrones se agrupan en tres grandes familias:

Tipo	Propósito	Ejemplo
Creacionales	Cómo se crean los objetos	Singleton, Factory, Builder
Estructurales	Cómo se relacionan las clases	Adapter, Decorator, Facade
Comportamiento	Cómo colaboran los objetos	Strategy, Observer, Command

1. Singleton Pattern

Garantiza que solo exista una instancia de una clase y que sea accesible globalmente.

Ejemplo en C#:

```
public sealed class Logger
{
    private static readonly Logger _instancia = new Logger();
    private Logger() { }
    public static Logger Instancia => _instancia;
    public void Escribir(string mensaje) => Console.WriteLine(mensaje);
}
```

Uso típico: Logging, configuraciones globales o caches.

Consejo: Evita abusarlo. En arquitecturas grandes puede generar acoplamiento global.

🏭 2. Factory Method Pattern

Permite crear objetos sin exponer la lógica de instanciación.

```
public interface INotificador { void Enviar(string mensaje); }

public class EmailNotificador : INotificador { public void Enviar(string m)
=> Console.WriteLine($"Email: {m}"); }
public class SMSNotificador : INotificador { public void Enviar(string m) =>
Console.WriteLine($"SMS: {m}"); }

public static class NotificadorFactory
{
    public static INotificador Crear(string tipo) =>
        tipo switch
        {
            "Email" => new EmailNotificador(),
            "SMS" => new SMSNotificador(),
            _ => throw new ArgumentException("Tipo no soportado")
        };
}
```

Uso: creación flexible sin romper el principio de apertura/cierre (OCP).

🧠 3. Strategy Pattern

Permite cambiar el comportamiento de una clase en tiempo de ejecución.

```
public interface ICalculoImpuesto { decimal Calcular(decimal monto); }
public class IVA : ICalculoImpuesto { public decimal Calcular(decimal monto)
=> monto * 0.21m; }
public class SinImpuesto : ICalculoImpuesto { public decimal Calcular(decimal
monto) => monto; }

public class Factura
{
    private readonly ICalculoImpuesto _estrategia;
    public Factura(ICalculoImpuesto estrategia) => _estrategia = estrategia;
    public decimal Total(decimal monto) => _estrategia.Calcular(monto);
}
```

➡️ *Ventaja:* permite extender comportamientos sin modificar código existente.

🧩 4. Adapter Pattern

Permite que clases incompatibles trabajen juntas transformando interfaces.

```
public interface IImpresora { void Imprimir(); }

public class ImpresoraAntigua { public void Print() =>
Console.WriteLine("Imprimiendo..."); }

public class AdapterImpresora : IImpresora
{
    private readonly ImpresoraAntigua _impresora = new ImpresoraAntigua();
    public void Imprimir() => _impresora.Print();
}
```

➡️ *Uso típico:* integrar librerías antiguas o APIs externas en sistemas modernos.

🎨 5. Decorator Pattern

Agrega funcionalidades sin alterar la clase original.

```
public interface IReporte { void Generar(); }

public class ReporteBase : IReporte { public void Generar() =>
Console.WriteLine("Reporte base generado"); }

public class ReporteConFirma : IReporte
{
    private readonly IReporte _reporte;
    public ReporteConFirma(IReporte reporte) => _reporte = reporte;
    public void Generar()
    {
        _reporte.Generar();
        Console.WriteLine("Firma añadida");
    }
}
```

➡️ *Ejemplo real:* agregar logging, compresión o validaciones de manera no intrusiva.

6. Observer Pattern

Permite que múltiples objetos "escuchen" cambios de otro objeto.

```
public class Sujeto
{
    private readonly List<IObservador> _obs = new();
    public void Suscribir(IObservador o) => _obs.Add(o);
    public void Notificar(string msg) => _obs.ForEach(o =>
o.Actualizar(msg));
}
public interface IObservador { void Actualizar(string mensaje); }
```

➡️ *Uso:* eventos, interfaces gráficas, patrones de publicación/suscripción.

⚙️ Consejos del arquitecto .NET

- No apliques patrones por moda; **úsalos cuando resuelven un problema real.**
- Combina patrones: por ejemplo, **Factory + Strategy** o **Decorator + Observer**.
- Crea un **catálogo interno de patrones** para tu organización.
- Refuerza con **inyección de dependencias (DI)** para mantener bajo acoplamiento.

✅ Objetivos de aprendizaje

- Comprender las familias de patrones de diseño y su propósito.
- Implementar los patrones más comunes en .NET.
- Aplicarlos para aumentar la mantenibilidad y escalabilidad de un sistema.

Capítulo 8 – Pruebas unitarias y TDD con xUnit / NUnit

"El código sin pruebas es solo una hipótesis. El código probado es conocimiento comprobado."

🧭 Introducción

Un arquitecto .NET no solo diseña cómo funciona el sistema, sino cómo garantizar que **funcione correctamente, siempre**.
Las **pruebas unitarias** son la base de la calidad continua, y el enfoque **TDD (Test-Driven Development)** impulsa la creación de software más estable, modular y mantenible.

⚙️ ¿Qué son las pruebas unitarias?

Una **prueba unitaria** valida una pequeña porción del código (generalmente una función o método) para asegurar que se comporta como se espera.

Beneficios:

- Detectan errores antes de llegar a producción.
- Aumentan la confianza al refactorizar.
- Documentan el comportamiento esperado del sistema.
- Fomentan un diseño más desacoplado y limpio.

🧩 Frameworks más usados

- **xUnit.net:** moderno, rápido y adoptado por Microsoft.
- **NUnit:** veterano y compatible con versiones antiguas.
- **MSTest:** incluido en Visual Studio (útil para entornos empresariales clásicos).

Ejemplo práctico con xUnit

Clase a probar:

```
public class Calculadora
{
    public int Sumar(int a, int b) => a + b;
    public int Dividir(int a, int b)
    {
        if (b == 0) throw new DivideByZeroException();
        return a / b;
    }
}
```

Prueba unitaria:

```
using Xunit;

public class CalculadoraTests
{
    [Fact]
    public void Sumar_DeberiaRetornarSumaCorrecta()
    {
        var calc = new Calculadora();
        var resultado = calc.Sumar(2, 3);
        Assert.Equal(5, resultado);
    }

    [Fact]
    public void Dividir_EntreCero_DeberiaLanzarExcepcion()
    {
        var calc = new Calculadora();
```

```
        Assert.Throws<DivideByZeroException>(() => calc.Dividir(10, 0));
    }
}
```

Resultado:
Cada prueba validará el comportamiento esperado, mostrando verde si pasa o rojo si falla.

Ciclo TDD – *Red, Green, Refactor*

El **Test-Driven Development** se basa en tres pasos:

1. **Red:** escribe una prueba que falle (aún no hay código).
2. **Green:** implementa el mínimo código para pasar la prueba.
3. **Refactor:** mejora el código manteniendo la prueba verde.

Este ciclo promueve un diseño orientado a responsabilidad y simplicidad.

Buenas prácticas para el arquitecto

- Mantén las pruebas **aisladas** y **deterministas**.
- Usa **nombres descriptivos** (DebeHacer_X_Cuando_Y).
- Emplea **Mocks/Fakes** con frameworks como *Moq* para simular dependencias.
- Integra las pruebas en el **pipeline de CI/CD**.
- Define una **cobertura mínima del 80%** sin sacrificar calidad.

⚙️ Ejemplo con dependencias (uso de Moq)

```
var mockRepo = new Mock<IClienteRepositorio>();
mockRepo.Setup(r => r.Obtener(1)).Returns(new Cliente { Nombre = "Oscar" });

var servicio = new ClienteService(mockRepo.Object);
var cliente = servicio.ObtenerCliente(1);

Assert.Equal("Oscar", cliente.Nombre);
```

➡️ *Así se prueban componentes sin depender de bases de datos ni APIs externas.*

✅ Objetivos de aprendizaje

- Comprender el propósito y valor de las pruebas unitarias.
- Implementar pruebas con xUnit o NUnit en .NET.
- Adoptar el ciclo TDD para mejorar diseño y confiabilidad.
- Integrar pruebas en pipelines y metodologías DevOps.

Capítulo 9 – Control de versiones con Git y GitHub

"Un sistema sin control de versiones es un castillo sin cimientos: parece fuerte, hasta que algo cambia."

🧭 Introducción

El control de versiones es la columna vertebral de cualquier equipo de desarrollo profesional. Permite **rastrear cambios**, **colaborar en equipo**, y **mantener la trazabilidad del código fuente**.
El estándar actual en el mundo .NET —y en casi todo el desarrollo moderno— es **Git**, acompañado de plataformas como **GitHub**, **Azure DevOps** o **GitLab**.

Un arquitecto no solo debe usar Git, sino **diseñar estrategias de ramificación y flujo de trabajo** que aseguren estabilidad, productividad y control.

⚙️ ¿Qué es Git?

Git es un **sistema de control de versiones distribuido**, lo que significa que cada desarrollador tiene una copia completa del historial del proyecto.

Ventajas principales:

- Permite trabajar sin conexión.
- Facilita ramas independientes para nuevas funcionalidades.
- Permite revertir, comparar y fusionar cambios.
- Mantiene el historial completo de todo el código.

🔧 Comandos esenciales

```
git init                # Inicia un nuevo repositorio
git clone <url>         # Clona un repositorio remoto
git status              # Muestra el estado de cambios
git add .               # Agrega archivos al área de preparación
git commit -m "Mensaje" # Guarda los cambios localmente
git push origin main    # Envía los cambios al repositorio remoto
git pull origin main    # Actualiza desde el remoto
```

🌿 Estrategias de ramificación (branching)

Un arquitecto debe definir **una política clara de ramas**.
Los tres modelos más usados son:

Estrategia	Descripción	Ideal para
Main Only	Todo se hace en una rama principal	Equipos pequeños
Feature Branch	Cada nueva función se desarrolla aparte y luego se fusiona	Proyectos medianos
GitFlow	Ramas separadas para *main*, *develop*, *feature*, *release* y *hotfix*	Equipos grandes / CI/CD

Ejemplo visual de GitFlow:

main ← develop ← feature/login

↑

hotfix/seguridad

🔄 Pull Requests y Code Review

Un **Pull Request (PR)** es la forma estándar de integrar código revisado.
Antes de fusionar, otro desarrollador (o el arquitecto) revisa los cambios, comenta posibles mejoras y aprueba el merge.

Beneficios:

- Mejora la calidad del código.
- Evita errores en producción.
- Fomenta aprendizaje entre pares.

Consejo del arquitecto:
Define una **plantilla de PR** con checklist (pruebas, documentación, formato, naming).

🧩 Integración con GitHub y Azure DevOps

GitHub es la plataforma más usada para repositorios públicos y privados.
Permite:

- CI/CD con **GitHub Actions**.
- Seguimiento de issues y proyectos.
- Wikis y documentación técnica.
- Escaneo automático de vulnerabilidades.

Azure DevOps, en cambio, ofrece mayor integración con proyectos empresariales, pipelines, gestión ágil y tableros Kanban.

🧠 Buenas prácticas del arquitecto

- Establece una **convención de commits** (feat:, fix:, refactor:).
- Evita commits grandes sin contexto.
- Protege ramas principales con revisiones obligatorias.
- Usa etiquetas (tags) para marcar versiones estables.
- Automatiza el versionado con *semantic versioning (1.0.0 → 1.1.0)*.

✅ Objetivos de aprendizaje

- Comprender la importancia del control de versiones en proyectos .NET.
- Dominar Git como herramienta profesional.
- Aplicar flujos colaborativos (GitFlow, Pull Requests).
- Integrar GitHub y Azure DevOps en el ciclo DevOps.

Capítulo 10 – Integración continua y pipelines básicos

"Automatizar no es una opción: es el camino hacia la calidad constante."

🧭 Introducción

Un arquitecto .NET no solo diseña software, sino **procesos confiables de entrega**.
La **Integración Continua (CI)** y la **Entrega Continua (CD)** permiten que el código sea construido, probado y desplegado automáticamente cada vez que se realiza un cambio.
Esto garantiza calidad, velocidad y confianza en cada entrega.

En este capítulo aprenderás cómo implementar **pipelines básicos** con **GitHub Actions** y **Azure DevOps**, los dos entornos más usados en el ecosistema Microsoft.

⚙️ ¿Qué es CI/CD?

- **Integración Continua (CI):**
  Automatiza la compilación y pruebas del código cada vez que se realiza un commit o pull request.
  ➡️ *Objetivo:* detectar errores temprano.
- **Entrega Continua (CD):**
  Automatiza el empaquetado y despliegue en entornos (QA, Staging, Producción).
  ➡️ *Objetivo:* garantizar entregas rápidas y confiables.

Ventajas arquitectónicas:

- Reduce errores humanos.
- Acelera los ciclos de desarrollo.
- Establece trazabilidad completa de versiones.
- Mejora la colaboración entre desarrollo y operaciones (DevOps).

🧩 Ejemplo de pipeline básico con GitHub Actions

Archivo: .github/workflows/build.yml

```
name: Build and Test .NET
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: windows-latest
    steps:
    - uses: actions/checkout@v4
    - name: Instalar .NET
      uses: actions/setup-dotnet@v4
      with:
        dotnet-version: '8.0.x'
    - name: Restaurar dependencias
      run: dotnet restore
    - name: Compilar solución
      run: dotnet build --configuration Release --no-restore
    - name: Ejecutar pruebas unitarias
      run: dotnet test --no-build --verbosity normal
```

➡️ Este pipeline compila, prueba y valida cada commit automáticamente.

🧱 Ejemplo de pipeline en Azure DevOps

Archivo: azure-pipelines.yml

```
trigger:
  - main

pool:
  vmImage: 'windows-latest'

steps:
  - task: UseDotNet@2
    inputs:
      packageType: 'sdk'
      version: '8.0.x'

  - script: dotnet restore
    displayName: 'Restaurar dependencias'

  - script: dotnet build --configuration Release
    displayName: 'Compilar proyecto'

  - script: dotnet test --no-build
    displayName: 'Ejecutar pruebas unitarias'
```

➡️ En entornos corporativos, este pipeline puede extenderse con despliegue automático a **Azure App Service**, **Docker**, o **Kubernetes**.

🧠 Rol del arquitecto en CI/CD

El arquitecto define:

- La **estructura de los entornos** (Dev, QA, Stage, Prod).
- Las **reglas de despliegue** (manual o automático).
- Los **puntos de validación**: compilación, pruebas, escaneo de seguridad y performance.
- Los **monitores post-despliegue**, para detectar errores en tiempo real.

🚀 Buenas prácticas

- Mantén los pipelines en el mismo repositorio del código.
- Define **variables de entorno seguras** (Azure Key Vault, Secrets).
- Divide el pipeline por fases: *Build* → *Test* → *Deploy* → *Monitor*.
- Aplica el principio "**Shift Left**": probar lo antes posible.
- Documenta los pipelines como parte de la arquitectura del sistema.

✅ Objetivos de aprendizaje

- Entender los fundamentos de CI/CD.
- Crear pipelines básicos en GitHub Actions y Azure DevOps.
- Automatizar compilación, pruebas y despliegues.
- Adoptar una mentalidad DevOps orientada a calidad continua.

Parte III – Arquitectura de Aplicaciones .NET

"La arquitectura no se trata de complejidad, sino de orden."

En esta parte aprenderás a organizar proyectos .NET de forma profesional: cómo estructurar monolitos limpios, implementar **Clean Architecture**, aplicar **DDD (Domain-Driven Design)** y construir sistemas que crezcan sin romperse.
El arquitecto .NET no solo escribe código, sino que **define la forma en que todo el equipo colabora dentro del sistema**.

Capítulo 11 – Monolitos bien estructurados

"Un monolito bien diseñado puede ser más sólido y fácil de mantener que una mala red de microservicios."

🧭 Introducción

Antes de hablar de microservicios, es esencial dominar los **monolitos modulares**.
Lejos de ser obsoletos, los monolitos siguen siendo la base de la mayoría de aplicaciones empresariales.
Cuando están correctamente estructurados, pueden escalar, mantenerse y evolucionar durante años sin degradarse.

🧩 ¿Qué es un monolito?

Un **monolito** es una aplicación que contiene todos los módulos (UI, lógica de negocio y datos) en un único proceso.
Sin embargo, eso **no implica desorden** si se aplican principios de separación de responsabilidades y modularidad.

Problema común:
En muchos sistemas antiguos, la lógica está entremezclada con el acceso a datos o la interfaz, generando lo que se conoce como un "*Big Ball of Mud*".

Solución:
Diseñar **capas limpias y dependencias bien definidas**.

🧱 Estructura básica recomendada

Un monolito bien estructurado en .NET puede dividirse así:

/src

├── MyApp.API → Capa de presentación (Web API o MVC)

├── MyApp.Application → Casos de uso y lógica de negocio

├── MyApp.Domain → Entidades, reglas, interfaces

└── MyApp.Infrastructure → Persistencia, servicios externos, repositorios

Cada capa debe tener una **única responsabilidad** y depender solo de las capas inferiores.

🔗 Flujo de dependencias

1. **API (Presentación):** recibe solicitudes HTTP y las traduce a comandos o queries.
2. **Application:** ejecuta los casos de uso concretos (por ejemplo, "CrearPedido").
3. **Domain:** contiene las reglas de negocio puras y entidades.
4. **Infrastructure:** implementa repositorios, acceso a bases de datos, APIs externas.

➡️ *La dirección de las dependencias siempre va hacia el dominio.*

⚙️ Ejemplo simple en C#

Entidad de dominio:

```
public class Pedido
{
    public int Id { get; set; }
    public decimal Total { get; private set; }
    public void AgregarProducto(decimal precio) => Total += precio;
}
```

Caso de uso (Application):

```
public class CrearPedido
{
    private readonly IPedidoRepositorio _repo;
```

```
    public CrearPedido(IPedidoRepositorio repo) => _repo = repo;
    public void Ejecutar(Pedido pedido) => _repo.Guardar(pedido);
}
```

Controlador (API):

```
[HttpPost("pedidos")]
public IActionResult Crear([FromBody] Pedido pedido)
{
    _crearPedido.Ejecutar(pedido);
    return Ok();
}
```

🧠 **Ventajas de un monolito modular**

- **Simplicidad de despliegue:** un único artefacto.
- **Menor sobrecarga:** sin comunicación entre servicios.
- **Fácil refactorización:** control total del código.
- **Puerta de entrada hacia microservicios:** se puede dividir más adelante.

🚫 **Errores frecuentes**

- Lógica de negocio en controladores.
- Capas dependientes entre sí (por ejemplo, API usando EF directamente).
- Mezclar DTOs, entidades y modelos de base de datos.
- No definir interfaces claras para infraestructura.

Buenas prácticas

- Implementa **inyección de dependencias (DI)** desde el inicio.
- Usa **DTOs y mapeo** (AutoMapper) para aislar el dominio.
- Aplica **unit testing** en Application y Domain.
- Documenta las capas y flujos de datos.
- Divide el monolito en **módulos por contexto** (ventas, clientes, productos).

Objetivos de aprendizaje

- Comprender la importancia de los monolitos modulares.
- Aprender a estructurar capas limpias en .NET.
- Preparar el terreno para evolucionar hacia Clean Architecture.

Capítulo 12 – Clean Architecture y separaciones por capas

"La arquitectura limpia no se trata de frameworks, sino de independencia: del tiempo, del cambio y de la complejidad."
— Robert C. Martin ("Uncle Bob")

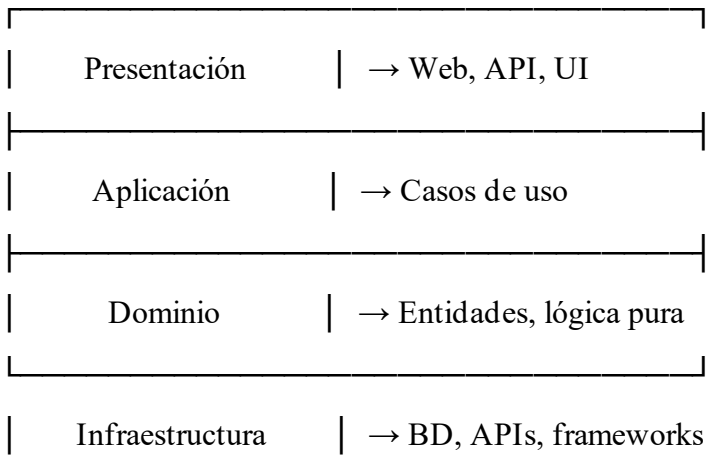
🧭 Introducción

Clean Architecture es un enfoque que busca que el software sea **fácil de mantener, probar y evolucionar**.
Su principio fundamental es que **las reglas de negocio no deben depender de detalles externos**, como bases de datos, frameworks o interfaces gráficas.

El arquitecto .NET debe garantizar que los **cambios tecnológicos nunca afecten al corazón del sistema**, protegiendo su dominio de negocio.

🧩 Estructura conceptual

Clean Architecture se representa como **anillos concéntricos**, donde el centro (el *dominio*) es el más protegido:



Regla clave:

Las dependencias siempre apuntan hacia el centro.
(Nada dentro del dominio debe conocer lo externo.)

⚙️ Capas principales en .NET

🧠 Dominio (Domain)

- Contiene **entidades**, **value objects** y **interfaces** que definen la lógica de negocio.
- No depende de ninguna tecnología.
- Ejemplo:

```
public class Producto
{
    public string Nombre { get; private set; }
    public decimal Precio { get; private set; }
    public Producto(string nombre, decimal precio)
    {
        if (precio <= 0) throw new ArgumentException("Precio inválido");
        Nombre = nombre; Precio = precio;
    }
}
```

⚙️ Aplicación (Application)

- Contiene los **casos de uso** y **servicios de aplicación**.
- Usa las interfaces del dominio, pero no implementa detalles.
- Ejemplo:

```
public class CrearPedidoHandler
{
    private readonly IPedidoRepositorio _repo;
    public CrearPedidoHandler(IPedidoRepositorio repo) => _repo = repo;
    public void Ejecutar(Pedido pedido) => _repo.Guardar(pedido);
}
```

💾 Infraestructura (Infrastructure)

- Implementa los **detalles técnicos**: bases de datos, archivos, APIs externas.
- Depende de las interfaces definidas en *Domain* o *Application*.
- Ejemplo:

```
public class PedidoRepositorio : IPedidoRepositorio
{
   private readonly DbContext _context;
   public void Guardar(Pedido pedido)
   {
      _context.Add(pedido);
      _context.SaveChanges();
   }
}
```

🌐 Presentación (Web/API/UI)

- Expone la aplicación al usuario o a otros sistemas.
- Usa controladores o vistas que invocan casos de uso.
- Ejemplo:

```
[HttpPost("pedido")]
public IActionResult Crear([FromBody] Pedido pedido)
{
   _crearPedidoHandler.Ejecutar(pedido);
   return Ok();
}
```

🧩 Inversión de dependencias

Para mantener el aislamiento, se utiliza **Dependency Injection (DI)**, generalmente con el contenedor nativo de .NET o librerías como Autofac o SimpleInjector.

Ejemplo:

```
builder.Services.AddScoped<IPedidoRepositorio, PedidoRepositorio>();

builder.Services.AddScoped<CrearPedidoHandler>();
```

🧠 Beneficios arquitectónicos

- **Alta mantenibilidad:** se cambia la base de datos o el framework sin afectar el dominio.
- **Facilidad para pruebas unitarias:** el núcleo puede probarse sin dependencias externas.
- **Escalabilidad progresiva:** permite migrar módulos a microservicios.
- **Longevidad:** el sistema resiste cambios tecnológicos (ORM, UI, nube, etc.).

🚫 Errores comunes

- Mezclar lógica de negocio en controladores.
- Crear dependencias inversas (por ejemplo, *Domain* usando *Infrastructure*).
- No definir interfaces claras para repositorios.
- Usar Clean Architecture como excusa para sobreingeniería.

💡 Consejo del arquitecto

"Empieza simple, pero empieza limpio."
Si tu monolito actual no cumple con Clean Architecture, aplica una **refactorización gradual por capas**, separando lo técnico de lo funcional.

✅ Objetivos de aprendizaje

- Comprender los principios de Clean Architecture.
- Separar responsabilidades en capas desacopladas.
- Implementar independencia tecnológica mediante DI.
- Preparar bases sólidas para evolucionar a DDD o microservicios.

Capítulo 13 – Domain-Driven Design (DDD) aplicado

"El verdadero poder del software no está en la tecnología, sino en el entendimiento compartido del dominio."
— Eric Evans, autor de "Domain-Driven Design"

🧭 Introducción

El **Domain-Driven Design (DDD)** es un enfoque de arquitectura que busca que el software refleje **el conocimiento profundo del negocio**.
No se trata de programar diferente, sino de **pensar como el negocio piensa**.
Un arquitecto .NET que domina DDD crea modelos ricos, expresivos y resistentes al cambio, donde el código se convierte en una **representación viva de las reglas del dominio**.

🧩 Conceptos fundamentales

🧠 Dominio

Es el área de conocimiento que el software aborda (por ejemplo, facturación, logística, educación).
El dominio define las **reglas, procesos y vocabulario** que gobiernan la aplicación.

🗣️ Lenguaje ubicuo (*Ubiquitous Language*)

Todo el equipo —analistas, programadores, testers y clientes— debe hablar **el mismo idioma**.
Los términos del negocio (Pedido, Cliente, Factura) se convierten en **clases, métodos y eventos** dentro del código.

■ Modelo de dominio

Representa las **entidades, valores y comportamientos** que expresan las reglas del negocio.
No se trata de datos, sino de **conceptos con significado**.

Bloques de construcción del DDD

1. Entidades (Entities)

Tienen identidad única y ciclo de vida.
Ejemplo:

```
public class Cliente
{
    public Guid Id { get; private set; } = Guid.NewGuid();
    public string Nombre { get; private set; }
    public void CambiarNombre(string nuevoNombre) => Nombre = nuevoNombre;
}
```

2. Objetos de valor (Value Objects)

No tienen identidad, solo valor.
Ejemplo:

```
public record Dinero(decimal Cantidad, string Moneda);
```

3. Agregados (Aggregates)

Agrupan entidades y objetos de valor bajo una **raíz de agregado**, que controla la consistencia.
Ejemplo:

```
public class Pedido
{
    private readonly List<ItemPedido> _items = new();
    public IReadOnlyCollection<ItemPedido> Items => _items;
    public void AgregarItem(string producto, decimal precio)
        => _items.Add(new ItemPedido(producto, precio));
}
```

4. Repositorios

Encapsulan la persistencia, simulando colecciones del dominio.
Ejemplo:

```
public interface IPedidoRepositorio
{
    Pedido? ObtenerPorId(Guid id);
    void Guardar(Pedido pedido);
}
```

5. Servicios de dominio

Contienen lógica que no pertenece a ninguna entidad específica.
Ejemplo:

```
public class CalculadorDescuento
{
    public decimal Aplicar(decimal total, decimal porcentaje) => total - (total * porcentaje / 100);
}
```

🧩 Contextos delimitados (Bounded Contexts)

Un sistema complejo se divide en **subdominios**, cada uno con su propio modelo y lenguaje.
Ejemplo:

- *Ventas*, *Inventario* y *Facturación* pueden ser contextos diferentes.

Cada contexto tiene autonomía y comunica con los demás mediante APIs, eventos o colas de mensajería.

[Ventas] → Evento "PedidoCreado" → [Facturación]

⚙️ DDD y Clean Architecture

Clean Architecture y DDD **se complementan perfectamente**:

- Clean define **cómo se organiza el código**.
- DDD define **qué representa el código**.

El dominio ocupa el centro en ambos enfoques, aislado del resto del sistema.

🧠 Rol del arquitecto en DDD

El arquitecto debe:

- Participar en el descubrimiento del dominio con expertos del negocio.
- Guiar la creación de modelos expresivos y cohesivos.
- Establecer límites claros entre contextos.
- Promover el lenguaje ubicuo en todo el equipo.
- Evitar el exceso de tecnicismos: DDD busca **claridad, no complejidad**.

✅ Objetivos de aprendizaje

- Comprender los principios del Domain-Driven Design.
- Modelar entidades, objetos de valor y agregados correctamente.
- Aplicar Bounded Contexts para dividir dominios complejos.
- Unir DDD con Clean Architecture para lograr sistemas robustos y expresivos.

Capítulo 14 – CQRS y Mediator Pattern

"Divide y vencerás: cuando separar responsabilidades mejora la claridad y el control."

🧭 Introducción

El patrón **CQRS (Command Query Responsibility Segregation)** surge de una idea simple pero poderosa:

"Las operaciones que modifican el estado no deberían ser las mismas que las que lo leen."

Aplicado correctamente, CQRS mejora la escalabilidad, facilita la auditoría y se integra perfectamente con **Clean Architecture** y **DDD**.
En .NET, suele implementarse junto al **patrón Mediator**, que organiza el flujo de comandos y consultas de manera limpia y centralizada.

🧩 ¿Qué es CQRS?

CQRS divide la aplicación en dos caminos distintos:

- **Comandos (Commands):** acciones que cambian el estado (crear, actualizar, eliminar).
- **Consultas (Queries):** operaciones de solo lectura (obtener, listar, filtrar).

Ejemplo conceptual:

CrearPedidoCommand → Handler → Guarda en BD

ObtenerPedidosQuery → Handler → Lee desde BD

⚙️ Estructura en .NET

Una aplicación CQRS típica tiene:

```
/Application
├── Commands
│   └── CrearPedido
│       ├── CrearPedidoCommand.cs
│       └── CrearPedidoHandler.cs
├── Queries
│   └── ObtenerPedidos
│       ├── ObtenerPedidosQuery.cs
│       └── ObtenerPedidosHandler.cs
```

Esto separa el flujo de datos y reduce el acoplamiento.

🧱 Implementación práctica

Command:

```
public record CrearPedidoCommand(string Cliente, decimal Total) : IRequest<Guid>;
```

Handler:

```
public class CrearPedidoHandler : IRequestHandler<CrearPedidoCommand, Guid>
{
    private readonly IPedidoRepositorio _repo;
    public CrearPedidoHandler(IPedidoRepositorio repo) => _repo = repo;

    public Task<Guid> Handle(CrearPedidoCommand request, CancellationToken ct)
    {
        var pedido = new Pedido(request.Cliente, request.Total);
        _repo.Guardar(pedido);
        return Task.FromResult(pedido.Id);
    }
```

```
}
```

Query:

```
public record ObtenerPedidosQuery : IRequest<IEnumerable<Pedido>>;
```

Handler de consulta:

```
public class ObtenerPedidosHandler : IRequestHandler<ObtenerPedidosQuery,
IEnumerable<Pedido>>
{
    private readonly IConsultaPedidos _consulta;
    public ObtenerPedidosHandler(IConsultaPedidos consulta) => _consulta = consulta;

    public Task<IEnumerable<Pedido>> Handle(ObtenerPedidosQuery request,
CancellationToken ct)
        => Task.FromResult(_consulta.ListarPedidos());
}
```

🧩 El patrón Mediator

El **Mediator Pattern** actúa como intermediario entre las solicitudes (commands/queries) y sus manejadores.
En .NET, se implementa fácilmente con la librería **MediatR**.

Configuración:

```
builder.Services.AddMediatR(typeof(CrearPedidoHandler));
```

Uso en el controlador:

```
[HttpPost("pedido")]
public async Task<IActionResult> Crear(CrearPedidoCommand cmd)
{
    var id = await _mediator.Send(cmd);
    return Ok(id);
}
```

➡️ *El controlador no conoce la lógica interna ni la capa de infraestructura.*

⚙️ Ventajas de CQRS + Mediator

- Separación clara entre lectura y escritura.
- Código más fácil de probar y mantener.
- Escalabilidad horizontal: lectura y escritura pueden tener bases distintas.
- Integración natural con **event sourcing**, **mensajería** o **microservicios**.

🚫 Desventajas o malas prácticas

- No aplicar CQRS en sistemas pequeños (añade complejidad).
- Mezclar comandos y consultas en un mismo servicio.
- Crear Handlers sin propósito o sobrecargar la capa de aplicación.

🧠 Consejo del arquitecto

"Usa CQRS donde haya flujo complejo o necesidad de escalar, no en cada botón."
Empieza simple, aplica CQRS por capas y añade MediatR cuando necesites controlar la comunicación entre componentes de forma limpia.

✅ Objetivos de aprendizaje

- Comprender el principio CQRS y cuándo aplicarlo.
- Implementar comandos y consultas desacoplados en .NET.
- Integrar MediatR como mediador central del flujo de datos.
- Diseñar sistemas más claros, probables y escalables.

Capítulo 15 – Inyección de dependencias y configuración avanzada

"La flexibilidad de un sistema no depende de cuántas clases tenga, sino de cuán débilmente estén acopladas entre sí."

🧭 Introducción

La **inyección de dependencias (DI)** es la base de las arquitecturas limpias y desacopladas en .NET.
Permite que los componentes no creen directamente sus dependencias, sino que las reciban desde un **contenedor IoC (Inversion of Control)**.
Esto simplifica pruebas, mantenimiento y configuración, y se integra de forma nativa en .NET desde la versión Core.

🧩 ¿Qué es la inyección de dependencias?

Sin DI:

```
public class PedidoService
{
    private readonly PedidoRepositorio _repo = new PedidoRepositorio(); // Acoplamiento fuerte
}
```

Con DI:

```
public class PedidoService
{
    private readonly IPedidoRepositorio _repo;
    public PedidoService(IPedidoRepositorio repo) => _repo = repo;
}
```

El contenedor IoC se encarga de crear e inyectar las instancias necesarias, **eliminando dependencias directas**.

Tipos de vida (lifetimes) en .NET

Al registrar los servicios, es importante definir su ciclo de vida:

Tipo	Descripción	Ejemplo
Transient	Nueva instancia cada vez que se solicita	AddTransient<IPedidoRepositorio, PedidoRepositorio>()
Scoped	Una instancia por solicitud HTTP	AddScoped<IClienteService, ClienteService>()
Singleton	Misma instancia durante toda la aplicación	AddSingleton<ILogger, Logger>()

Ejemplo completo

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

builder.Services.AddScoped<IPedidoRepositorio, PedidoRepositorio>();
builder.Services.AddTransient<INotificador, EmailNotificador>();
builder.Services.AddSingleton<ILogger, FileLogger>();

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

En ejecución, el framework inyectará las dependencias donde se necesiten:

```
public class CrearPedidoHandler
{
    private readonly IPedidoRepositorio _repo;
    private readonly INotificador _notificador;

    public CrearPedidoHandler(IPedidoRepositorio repo, INotificador notificador)
    {
        _repo = repo;
        _notificador = notificador;
    }
}
```

Inyección de configuración

.NET utiliza el patrón **Options** para manejar configuraciones limpias y tipadas.

Archivo appsettings.json:

```
{
  "Correo": {
    "Servidor": "smtp.miempresa.com",
    "Puerto": 587,
    "Remitente": "no-reply@miempresa.com"
  }
}
```

Clase de configuración:

```
public class CorreoOptions
{
    public string Servidor { get; set; }
    public int Puerto { get; set; }
    public string Remitente { get; set; }
}
```

Registro en Program.cs:

```
builder.Services.Configure<CorreoOptions>(
    builder.Configuration.GetSection("Correo"));
```

Uso en una clase:

```
public class EmailNotificador
{
    private readonly CorreoOptions _config;
    public EmailNotificador(IOptions<CorreoOptions> options)
    {
        _config = options.Value;
    }
}
```

➡️ *Ventaja:* cada entorno (desarrollo, producción, pruebas) puede tener su propio appsettings.*.json.

⚙️ Combinación de DI + MediatR + CQRS

En proyectos medianos/grandes, el arquitecto combina estos conceptos:

```
builder.Services.AddMediatR(typeof(CrearPedidoHandler));
builder.Services.AddScoped<IPedidoRepositorio, PedidoRepositorio>();
builder.Services.AddScoped<IEmailService, EmailService>();
```

De esta manera, **los handlers CQRS reciben automáticamente sus dependencias**, manteniendo el código limpio y desacoplado.

🚀 Buenas prácticas

- No abuses del patrón **Singleton**; puede generar estado compartido no deseado.
- Evita **resolver dependencias manualmente** (new ServiceProvider().GetService()).
- Divide el registro de servicios por módulos o capas.
- Documenta las configuraciones en un *README técnico*.
- Implementa validaciones en tus Options (por ejemplo, validar puertos o URLs).

✅ Objetivos de aprendizaje

- Comprender la inyección de dependencias como base del desacoplamiento.
- Registrar servicios con ciclos de vida apropiados.
- Gestionar configuraciones limpias y seguras.
- Integrar DI con CQRS, MediatR y Clean Architecture.

🌐 Parte IV – APIs, Servicios y Comunicación

"Una API bien diseñada es la voz del sistema: clara, coherente y segura."

Capítulo 16 – ASP.NET Core y Web API

"El API es la frontera entre la lógica del negocio y el resto del mundo: cuídala como un contrato."

🧭 Introducción

ASP.NET Core es el corazón de la creación de **servicios web y APIs RESTful** en .NET moderno.
Permite desarrollar aplicaciones ligeras, escalables y de alto rendimiento, integradas con **Clean Architecture**, **DDD** y **CQRS**.

El arquitecto .NET debe saber **diseñar APIs como contratos**, no solo como endpoints: predecibles, seguras, versionadas y bien documentadas.

⚙️ Estructura básica de una API en .NET

Una Web API típica tiene esta estructura:

/Controllers

  PedidoController.cs

/Application

  Commands/

  Queries/

/Domain

  Entidades/

Controlador ejemplo:

```
[ApiController]
[Route("api/[controller]")]
public class PedidoController : ControllerBase
{
    private readonly IMediator _mediator;
    public PedidoController(IMediator mediator) => _mediator = mediator;

    [HttpPost]
    public async Task<IActionResult> Crear([FromBody] CrearPedidoCommand cmd)
    {
        var id = await _mediator.Send(cmd);
        return CreatedAtAction(nameof(ObtenerPorId), new { id }, null);
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> ObtenerPorId(Guid id)
    {
        var pedido = await _mediator.Send(new ObtenerPedidoQuery(id));
        return pedido is null ? NotFound() : Ok(pedido);
    }
}
```

➡️ Aquí el controlador actúa solo como **puente** entre HTTP y la capa de aplicación, cumpliendo los principios de **Clean Architecture**.

🧩 Configuración de la API

En Program.cs:

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

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

📘 Buenas prácticas REST

1. **Usar nombres de recursos:** /api/pedidos, /api/clientes.
2. **Métodos HTTP adecuados:**
   - GET → obtener datos
   - POST → crear recursos
   - PUT → actualizar
   - DELETE → eliminar
3. **Códigos de estado claros:**
   - 200 OK, 201 Created, 400 BadRequest, 404 NotFound, 500 InternalError.
4. **Respuestas consistentes:** usa un formato estándar (JSON con campos data, error, timestamp).
5. **Versionado de API:** /api/v1/pedidos, /api/v2/pedidos.

🔒 Seguridad básica

- Autenticación con **JWT (JSON Web Tokens)**.
- Autorización mediante **[Authorize]** y roles.
- Validación de entrada con **Data Annotations** y **FluentValidation**.
- CORS configurado para permitir solo dominios de confianza:

```
builder.Services.AddCors(o => o.AddPolicy("Permitido",
    p => p.WithOrigins("https://app.miempresa.com").AllowAnyHeader().AllowAnyMethod()));
```

🧠 Consejo del arquitecto

"Un API no es solo para desarrolladores; es parte de la experiencia del usuario."
Diseña pensando en la claridad, consistencia y evolución.
Una API bien documentada y predecible **reduce soporte y aumenta adopción.**

🚀 Integración con herramientas

- **Swagger / OpenAPI:** genera documentación y pruebas automáticas.
- **Postman / REST Client:** para probar endpoints.
- **Serilog / Application Insights:** para registrar peticiones y métricas.
- **HealthChecks:** para monitorear estado del API.

✅ Objetivos de aprendizaje

- Diseñar y construir una API RESTful profesional con ASP.NET Core.
- Aplicar buenas prácticas de versionado, seguridad y validación.
- Integrar Clean Architecture y CQRS en la capa de exposición.
- Documentar y monitorear las APIs para entornos productivos.

Capítulo 17 – REST, GraphQL y gRPC

"No existe el protocolo perfecto: solo el adecuado para el propósito correcto."

🧭 Introducción

En arquitecturas distribuidas —ya sean **monolitos modulares, microservicios o sistemas híbridos**— la comunicación entre componentes es crítica.
.NET ofrece soporte nativo para **REST**, **GraphQL** y **gRPC**, tres formas distintas de exponer y consumir datos.
El arquitecto debe saber **cuándo y por qué** usar cada una.

🌎 1. REST (Representational State Transfer)

⚙️ Descripción

REST es el estándar más extendido. Usa HTTP y recursos identificados por URLs.
Cada acción se realiza mediante los métodos HTTP (GET, POST, PUT, DELETE) y devuelve datos en JSON.

Ejemplo de endpoint:

GET /api/pedidos/123

Respuesta:

```
{
  "id": 123,
  "cliente": "Beatriz",
  "total": 199.99
}
```

Ventajas:

- Simple, ampliamente compatible.
- Ideal para integraciones públicas o web.
- Totalmente soportado por ASP.NET Core.

Limitaciones:

- Sobrecarga de datos: devuelve más de lo necesario.
- Requiere múltiples llamadas para relaciones complejas.

Ideal para: APIs empresariales estándar, sistemas CRUD, integraciones externas.

🧩 2. GraphQL

⚙️ Descripción

GraphQL es un lenguaje de consultas desarrollado por Facebook.
Permite al cliente **pedir exactamente los datos que necesita**, combinando varias fuentes en una sola petición.

Ejemplo de consulta:

```
{
  pedido(id: 123) {
    id
    cliente { nombre email }
    productos { nombre precio }
  }
}
```

Respuesta:

```
{
  "pedido": {
    "id": 123,
    "cliente": { "nombre": "Beatriz", "email": "bea@dominio.com" },
    "productos": [
      { "nombre": "Teclado", "precio": 29.9 },
      { "nombre": "Ratón", "precio": 15.5 }
    ]
  }
}
```

Ventajas:

- Flexibilidad total en la respuesta.
- Menos llamadas a la red.
- Perfecto para frontends dinámicos (React, Vue, Angular).

Limitaciones:

- Más complejidad en el servidor.
- Riesgo de consultas pesadas si no se controlan.

Ideal para: portales, apps móviles, paneles ricos en datos.

En .NET: se implementa fácilmente con **Hot Chocolate (ChilliCream)** o **GraphQL.NET**.

---

3. gRPC (Google Remote Procedure Call)

Descripción

gRPC usa **HTTP/2** y **Protocol Buffers (protobuf)** para una comunicación binaria eficiente.
Se basa en llamadas a métodos remotos, no en recursos REST.

Definición de servicio (pedido.proto):

```
service PedidoService {
  rpc CrearPedido (PedidoRequest) returns (PedidoResponse);
}
```

Ventajas:

- Rendimiento superior (baja latencia, menos consumo).
- Soporte nativo para streaming bidireccional.
- Excelente para comunicación entre microservicios.

Limitaciones:

- No ideal para navegadores (requiere adaptadores).
- Menos legible que JSON.

Ideal para:
microservicios, IoT, sistemas en tiempo real o interconexión entre servidores .NET.

Configuración básica en .NET:

```
builder.Services.AddGrpc();
app.MapGrpcService<PedidoService>();
```

🧠 Elección arquitectónica

Característica	REST	GraphQL	gRPC
Formato	JSON	JSON	Binario (protobuf)
Velocidad	Media	Media	Alta
Complejidad	Baja	Media	Alta
Escenarios ideales	Web, APIs públicas	Frontend dinámico	Microservicios, IoT
Streaming	No	Parcial	Sí
Compatibilidad navegador	Total	Total	Limitada

💡 Consejo del arquitecto

"El protocolo no define la arquitectura, pero una mala elección puede limitarla."
Para APIs abiertas: **REST**.
Para clientes ricos: **GraphQL**.
Para microservicios o alto rendimiento: **gRPC**.
En entornos mixtos, **combina los tres según necesidad**.

✅ Objetivos de aprendizaje

- Entender las diferencias entre REST, GraphQL y gRPC.
- Evaluar qué protocolo usar según el caso.
- Conocer las implementaciones modernas en .NET.
- Diseñar arquitecturas híbridas y escalables de comunicación.

Capítulo 18 – Versionado, Logging y Swagger

"Una API sin documentación es como un mapa sin leyenda: funcional, pero inservible."

🧭 Introducción

A medida que las APIs .NET crecen y evolucionan, el arquitecto debe garantizar que cada versión sea **mantenible, trazable y entendible**.
El versionado permite **compatibilidad hacia atrás**, el logging proporciona **rastro y diagnóstico**, y **Swagger (OpenAPI)** convierte la API en un contrato vivo.
Estas tres piezas forman la base de una **API sostenible y auditable**.

⚙️ 1. Versionado de APIs

El versionado evita romper clientes existentes cuando cambian los endpoints.
.NET ofrece **Microsoft.AspNetCore.Mvc.Versioning**, un paquete oficial para gestionar múltiples versiones.

📦 Instalación

```
dotnet add package Microsoft.AspNetCore.Mvc.Versioning
```

⚙️ Configuración

```
builder.Services.AddApiVersioning(options =>
{
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.ReportApiVersions = true;
});
```

🧩 Uso en controladores

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
public class PedidosController : ControllerBase
{
    [HttpGet]
    public IActionResult GetV1() => Ok("Versión 1.0");
}

[ApiVersion("2.0")]
[Route("api/v{version:apiVersion}/[controller]")]
```

```
public class PedidosV2Controller : ControllerBase
{
    [HttpGet]
    public IActionResult GetV2() => Ok("Versión 2.0 mejorada");
}
```

➡️ Ahora los clientes pueden consumir `/api/v1/pedidos` o `/api/v2/pedidos`.

🧠 2. Logging profesional

El logging es el **sistema nervioso de la aplicación**.
Permite diagnosticar errores, analizar tráfico y detectar patrones de uso.
En .NET se implementa fácilmente con el sistema **ILogger** y se puede conectar a herramientas externas.

🔹 Ejemplo básico

```
public class PedidosController : ControllerBase
{
    private readonly ILogger<PedidosController> _logger;
    public PedidosController(ILogger<PedidosController> logger) => _logger =
logger;

    [HttpGet]
    public IActionResult Get()
    {
        _logger.LogInformation("Consultando pedidos...");
        return Ok("Pedidos consultados");
    }
}
```

🔹 Niveles de log

Nivel	Uso
Trace	Diagnóstico detallado
Debug	Desarrollo y depuración
Information	Eventos normales
Warning	Comportamientos inesperados
Error	Fallos recuperables
Critical	Fallos graves del sistema

🔗 Integración con proveedores externos

- **Serilog** → logs estructurados y exportación a archivos, SQL o Elastic.
- **Application Insights** → métricas y trazas distribuidas en Azure.
- **Seq** → panel de observabilidad local.

Ejemplo con Serilog:

```
Log.Logger = new LoggerConfiguration()
    .WriteTo.File("logs/api.log", rollingInterval: RollingInterval.Day)
    .WriteTo.Console()
    .CreateLogger();

builder.Host.UseSerilog();
```

📘 3. Swagger y OpenAPI

La documentación automática convierte la API en un **contrato vivo y navegable**.
.NET utiliza el paquete oficial `Swashbuckle.AspNetCore`.

📦 Instalación

```
dotnet add package Swashbuckle.AspNetCore
```

⚙️ Configuración

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "API de Pedidos",
        Version = "v1",
        Description = "Documentación oficial de la API de ejemplo"
    });
});
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
"Pedidos API v1"));
}
```

Acceso en navegador:
➡️ `https://localhost:5001/swagger`

Documentar endpoints con anotaciones

```
/// <summary>Obtiene la lista de pedidos activos.</summary>
/// <response code="200">Lista de pedidos.</response>
/// <response code="404">No se encontraron pedidos.</response>
[HttpGet]
public IActionResult GetPedidos() => Ok(servicio.Listar());
```

---

4. Buenas prácticas de documentación y trazabilidad

- Mantén el **versionado explícito** en URL o encabezados.
- Documenta **todas las respuestas posibles (200, 400, 500)**.
- Usa **Swagger + XML Comments** para generar doc automática.
- Incluye **correlación de logs por RequestId**.
- Define **niveles de log por entorno** (más detallado en dev, menos en prod).
- Guarda logs críticos de producción en almacenamiento seguro (SQL, Blob, Elastic).

Objetivo

Exponer APIs **mantenibles, observables y auto-documentadas**, donde cada versión pueda convivir sin romper compatibilidad, y cada evento quede registrado para análisis o auditoría.

Capítulo 19 – Autenticación y Autorización (JWT, OAuth2, Identity)

"En una arquitectura segura, cada petición cuenta una historia… y debe poder demostrarse quién la escribió."

🧭 Introducción

Toda API pública o privada necesita **proteger sus endpoints**.
No basta con tener usuarios; hay que controlar el **contexto, permisos y validez** de cada solicitud.
En este capítulo aprenderás a implementar seguridad moderna en .NET con **JWT (JSON Web Tokens)**, **OAuth2** y **ASP.NET Core Identity**, los pilares del acceso seguro y escalable.

⚙️ 1. Autenticación vs Autorización

Concepto	Propósito	Ejemplo
Autenticación	Verifica *quién eres*	Login con usuario y contraseña
Autorización	Define *qué puedes hacer*	Permiso "solo admin puede borrar"

Ambas deben convivir: primero se valida la identidad, luego se aplica la política de acceso.

🔐 2. Autenticación con JWT (JSON Web Tokens)

JWT es el estándar más usado para autenticar en APIs sin estado (*stateless*).
Se emite un **token firmado**, que el cliente envía en cada petición.

📦 Instalación

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

⚙️ Configuración en `Program.cs`

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
```

```
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "https://miserver.com",
        ValidAudience = "https://miserver.com",
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes("clave_super_secreta_123456789"))
    };
});
```

Activar middleware:

```
app.UseAuthentication();
app.UseAuthorization();
```

🧩 Generación de token

```
public string GenerarToken(string usuario)
{
    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, usuario),
        new Claim("role", "Admin")
    };

    var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("clave_super_secreta_123456789"))
;
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
        issuer: "https://miserver.com",
        audience: "https://miserver.com",
        claims: claims,
        expires: DateTime.Now.AddHours(1),
        signingCredentials: creds);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

El cliente lo envía en cada request:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...
```

🧠 3. Autorización por Roles y Políticas

🔹 Por roles

```
[Authorize(Roles = "Admin")]
[HttpDelete("{id}")]
public IActionResult Delete(int id) => Ok($"Pedido {id} eliminado");
```

🔹 Por políticas personalizadas

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("SoloMayoresDeEdad", policy =>
        policy.RequireClaim("Edad", "18"));
});
[Authorize(Policy = "SoloMayoresDeEdad")]
public IActionResult CrearPedido() => Ok("Pedido aceptado");
```

🔑 4. OAuth2 y OpenID Connect

OAuth2 permite delegar autenticación a un **proveedor de identidad externo** (Azure AD, Google, Auth0, etc.).
OpenID Connect extiende OAuth2 para incluir **información del usuario** (perfil, email, roles).

Escenarios comunes:

- Iniciar sesión con Google o Azure AD.
- APIs seguras compartidas por varias apps.
- Single Sign-On (SSO) corporativo.

Configuración típica (Azure AD):

```
builder.Services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme
)
.AddMicrosoftIdentityWebApp(builder.Configuration.GetSection("AzureAd"));
```

En `appsettings.json`:

```
"AzureAd": {
  "Instance": "https://login.microsoftonline.com/",
  "Domain": "empresa.com",
  "TenantId": "xxxxxx-xxxx-xxxx-xxxx",
  "ClientId": "xxxxxx-xxxx-xxxx-xxxx",
  "CallbackPath": "/signin-oidc"
}
```

🧩 5. ASP.NET Core Identity

Para proyectos con usuarios propios, Identity gestiona registro, login, recuperación de contraseña y roles.

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer("cadena_conexion"));
builder.Services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AppDbContext>()
    .AddDefaultTokenProviders();
```

Se integra automáticamente con JWT u OAuth2, ofreciendo seguridad unificada.

⚙️ 6. Buenas prácticas de seguridad en APIs

- ✅ **Usa HTTPS siempre.**
- ✅ **Expira tokens** y renueva con *refresh tokens*.
- ✅ **Evita roles estáticos** → usa *claims dinámicos*.
- ✅ **Registra intentos fallidos de login.**
- ✅ **Protege endpoints sensibles con MFA o scopes.**
- ✅ **Centraliza la emisión de tokens** (no la dupliques en cada microservicio).
- ✅ **Revisa dependencias de seguridad (OWASP).**

🧩 Ejemplo de flujo seguro

```
Cliente → /login (usuario, pass)
          ↓
API → Genera JWT firmado
          ↓
Cliente almacena token
          ↓
Cada request incluye Authorization: Bearer <token>
          ↓
API valida token, aplica roles/políticas y responde
```

✅ Objetivo

Diseñar APIs **seguras y escalables**, con autenticación moderna basada en JWT y OAuth2, integradas con ASP.NET Identity, permitiendo proteger cada endpoint sin sacrificar rendimiento ni mantenibilidad.

Capítulo 20 – Caching y Rendimiento

"La optimización no siempre consiste en hacer más rápido el código, sino en evitar hacerlo innecesariamente."

🧭 Introducción

El caching es una de las estrategias más efectivas para **mejorar la latencia y la respuesta de una API**.
En lugar de procesar o consultar los mismos datos repetidamente, los resultados se guardan temporalmente en memoria o en un sistema distribuido como **Redis**.
El arquitecto .NET debe saber **cuándo, qué y cómo cachear**, sin comprometer coherencia ni seguridad.

⚙️ 1. Conceptos clave de caching

Tipo	Ubicación	Duración	Uso típico
In-memory cache	RAM del servidor	Volátil	Datos ligeros y locales
Distributed cache	Redis, SQL, NCache	Compartido	Entornos escalados
Response cache	Nivel HTTP	Por cliente o proxy	APIs GET o públicas

---

🧩 2. Caching en memoria (IMemoryCache)

Ideal para aplicaciones monolíticas o de instancia única.

📦 Instalación

Ya incluida en ASP.NET Core:

builder.Services.AddMemoryCache();

⚙️ Uso en código

```
public class ProductoService
{
    private readonly IMemoryCache _cache;
    public ProductoService(IMemoryCache cache) => _cache = cache;
```

```
    public Producto ObtenerProducto(int id)
    {
        return _cache.GetOrCreate($"producto_{id}", entry =>
        {
            entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5);
            return ConsultarBaseDatos(id);
        });
    }
}
```

🔹 **Estrategias de expiración**

- **AbsoluteExpiration**: vence tras cierto tiempo.
- **SlidingExpiration**: se renueva mientras se use.
- **Priority**: prioridad de limpieza al liberar memoria.

🔗 3. Caché distribuido con Redis

Ideal para **microservicios o entornos con múltiples instancias**.

📦 **Instalación**

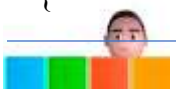
```
dotnet add package Microsoft.Extensions.Caching.StackExchangeRedis
```

⚙️ **Configuración**

```
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost:6379";
    options.InstanceName = "MiApp_";
});
```

🧩 **Ejemplo de uso**

```
public class CacheService
{
```



```
    private readonly IDistributedCache _cache;
    public CacheService(IDistributedCache cache) => _cache = cache;

    public async Task SetAsync(string key, string value)
    {
        await _cache.SetStringAsync(key, value,
            new DistributedCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)
            });
    }

    public async Task<string?> GetAsync(string key)
    {
        return await _cache.GetStringAsync(key);
    }
}
```

🚀 4. Response caching en APIs

Permite cachear respuestas HTTP completas.

📦 Habilitar middleware

```
builder.Services.AddResponseCaching();
app.UseResponseCaching();
```

📘 Ejemplo en controlador

```
[HttpGet]
[ResponseCache(Duration = 60, Location = ResponseCacheLocation.Any)]
public IActionResult GetProductos()
{
```

```
    return Ok(servicio.ObtenerTodos());
}
```

➡️ Las respuestas se servirán directamente desde cache durante 60 segundos.

🧠 5. Caching avanzado y patrones

Patrón	Descripción	Ejemplo
Cache-Aside	Consultar primero cache, luego BD	Redis o MemoryCache
Read-Through	El sistema gestiona automáticamente la carga	NCache, Redis Enterprise
Write-Through	Actualiza cache junto con la BD	Datos críticos
Invalidation	Elimina datos obsoletos al modificarse	Eventos o TTL

Ejemplo simple Cache-Aside:

```
var data = await _cache.GetAsync("usuarios");
if (data == null)
{
    data = ObtenerUsuariosDeBD();
    await _cache.SetAsync("usuarios", data);
}
```

🧩 6. Optimización de rendimiento complementaria

- Usa **async/await** para llamadas I/O.
- Configura **Pooling de conexiones** (SQL, HTTP).
- Activa **compresión de respuesta**:
- builder.Services.AddResponseCompression();
- app.UseResponseCompression();
- Usa **EF Core AsNoTracking()** para lecturas.
- Implementa **circuit breakers** con *Polly* para servicios lentos.
- Monitorea tiempos con **Application Insights**.

⚖️ 7. Buenas prácticas del arquitecto

- Cachea **solo datos estáticos o de lectura frecuente**.
- Nunca almacenes información sensible (tokens, contraseñas).
- Configura políticas de expiración y limpieza.
- Usa **Redis Cluster** o **Azure Cache for Redis** en producción.
- Instrumenta métricas para saber el "hit ratio" del cache.
- Diseña APIs con coherencia eventual: "cachear sí, pero con criterio".

✅ Objetivo

Optimizar tiempos de respuesta mediante caching en memoria y distribuido, reduciendo consultas costosas y mejorando la escalabilidad sin comprometer integridad o coherencia de los datos.

Parte V – Persistencia y Datos

Capítulo 21 – Entity Framework Core y LINQ avanzado

"Los datos son la memoria viva del sistema; gestionarlos bien es garantizar su historia y su futuro."

Introducción

Entity Framework Core (EF Core) es el ORM (Object-Relational Mapper) de Microsoft para .NET moderno.
Permite trabajar con bases de datos relacionales usando clases C# en lugar de SQL puro, facilitando la mantenibilidad y portabilidad.
Además, su integración con **LINQ** permite expresar consultas complejas de manera declarativa y segura.

---

1. Configuración básica de EF Core

Instalación

Ejemplo para SQL Server:

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

Configuración en Program.cs

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Clase de contexto

```
public class AppDbContext : DbContext
{
    public DbSet<Cliente> Clientes { get; set; }
    public DbSet<Pedido> Pedidos { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
}
```

📘 Entidades de dominio

```
public class Cliente
{
    public int Id { get; set; }
    public string Nombre { get; set; } = "";
    public List<Pedido> Pedidos { get; set; } = new();
}
public class Pedido
{
    public int Id { get; set; }
    public DateTime Fecha { get; set; }
    public decimal Total { get; set; }
    public int ClienteId { get; set; }
}
```

🧠 2. CRUD con EF Core

```
// CREATE
var cliente = new Cliente { Nombre = "Laura" };
context.Clientes.Add(cliente);
await context.SaveChangesAsync();

// READ
var clientes = await context.Clientes.ToListAsync();

// UPDATE
cliente.Nombre = "Laura Gómez";
context.Update(cliente);
await context.SaveChangesAsync();
```

```
// DELETE
context.Remove(cliente);
await context.SaveChangesAsync();
```

🧩 3. LINQ avanzado

LINQ (Language Integrated Query) permite escribir consultas SQL-like directamente en C#, con validación de tipos y soporte para IntelliSense.

- **Filtrado y ordenamiento**

```
var pedidos = await context.Pedidos
    .Where(p => p.Total > 100)
    .OrderByDescending(p => p.Fecha)
    .ToListAsync();
```

- **Proyecciones**

```
var resumen  = await context.Pedidos
    .Select(p => new { p.Id, p.Total, Cliente = p.Cliente.Nombre })
    .ToListAsync();
```

- **Agrupaciones y agregaciones**

```
var totales = await context.Pedidos
    .GroupBy(p => p.ClienteId)
    .Select(g => new { Cliente = g.Key, Total = g.Sum(x => x.Total) })
    .ToListAsync();
```

🧱 4. Relaciones y carga de datos

- **Carga explícita**

```
var cliente = await context.Clientes.FindAsync(1);
await context.Entry(cliente).Collection(c => c.Pedidos).LoadAsync();
```

- **Carga ansiosa (eager loading)**

```
var cliente = await context.Clientes
    .Include(c => c.Pedidos)
    .FirstAsync(c => c.Id == 1);
```

🔹 **Carga diferida (lazy loading)**

```
dotnet add package Microsoft.EntityFrameworkCore.Proxies
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseLazyLoadingProxies().UseSqlServer(connStr));
```

⚡ **5. Consultas sin seguimiento (lectura optimizada)**

Para mejorar rendimiento en lecturas:

```
var productos = await context.Productos.AsNoTracking().ToListAsync();
```

➡️ Evita que EF rastree entidades cuando no habrá cambios.

🧩 **6. Mapeo avanzado con Fluent API**

Permite personalizar reglas sin usar atributos.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Cliente>()
        .Property(c => c.Nombre)
        .HasMaxLength(100)
        .IsRequired();

    modelBuilder.Entity<Pedido>()
        .HasOne(p => p.Cliente)
        .WithMany(c => c.Pedidos)
        .HasForeignKey(p => p.ClienteId);
}
```

📊 7. Herramientas de desarrollo

Comando	Función
dotnet ef migrations add Inicial	Crea una migración
dotnet ef database update	Aplica migraciones
dotnet ef dbcontext list	Lista contextos del proyecto

🧠 8. Buenas prácticas del arquitecto

- Usa **AsNoTracking()** para lecturas masivas.
- Evita cargar datos innecesarios con Include().
- Define **índices** con Fluent API para campos buscados frecuentemente.
- Mantén **entidades livianas** (sin lógica extra).
- Centraliza el acceso mediante patrones (Repository / UoW).
- Controla las conexiones con using o inyección de dependencias.
- Monitorea consultas lentas con EnableSensitiveDataLogging() (solo en dev).

✅ Objetivo

Dominar el uso de **Entity Framework Core y LINQ avanzado** para manejar datos de manera eficiente, escalable y expresiva, garantizando control total sobre consultas, rendimiento y relaciones complejas.

Capítulo 22 – Repositorios y Unit of Work

"El buen arquitecto no consulta directamente la base de datos, conversa con ella a través de un contrato bien diseñado."

🧭 Introducción

Entity Framework Core es poderoso, pero mezclarlo directamente con la lógica de negocio puede generar **acoplamiento, duplicidad y dificultad para probar**.
Los patrones **Repository** y **Unit of Work (UoW)** actúan como una capa intermedia que **abstrae el acceso a los datos**, manteniendo la arquitectura limpia y testeable.

⚙️ 1. Patrón Repository

El patrón **Repository** centraliza el acceso a una entidad o agregado, permitiendo realizar operaciones CRUD sin exponer directamente EF Core.

📘 Interfaz genérica

```
public interface IRepository<T> where T : class
{
    Task<IEnumerable<T>> GetAllAsync();
    Task<T?> GetByIdAsync(int id);
    Task AddAsync(T entity);
    void Update(T entity);
    void Remove(T entity);
}
```

🧩 Implementación genérica

```
public class Repository<T> : IRepository<T> where T : class
{
    protected readonly AppDbContext _context;
    private readonly DbSet<T> _dbSet;

    public Repository(AppDbContext context)
```

```
    {
        _context = context;
        _dbSet = context.Set<T>();
    }

    public async Task<IEnumerable<T>> GetAllAsync() => await _dbSet.ToListAsync();
    public async Task<T?> GetByIdAsync(int id) => await _dbSet.FindAsync(id);
    public async Task AddAsync(T entity) => await _dbSet.AddAsync(entity);
    public void Update(T entity) => _dbSet.Update(entity);
    public void Remove(T entity) => _dbSet.Remove(entity);
}
```

🧠 2. Repositorios específicos

Para lógica particular de una entidad, se extiende el repositorio base:

```
public interface IClienteRepository : IRepository<Cliente>
{
    Task<IEnumerable<Cliente>> GetClientesVIPAsync();
}

public class ClienteRepository : Repository<Cliente>, IClienteRepository
{
    public ClienteRepository(AppDbContext context) : base(context) { }

    public async Task<IEnumerable<Cliente>> GetClientesVIPAsync()
    {
        return await _context.Clientes.Where(c => c.Pedidos.Count > 10).ToListAsync();
    }
}
```

🔄 3. Patrón Unit of Work (UoW)

Este patrón coordina varios repositorios bajo una misma transacción, asegurando consistencia.

📘 Interfaz UoW

```
public interface IUnitOfWork : IDisposable
{
    IClienteRepository Clientes { get; }
    IPedidoRepository Pedidos { get; }
    Task<int> SaveAsync();
}
```

🧩 Implementación UoW

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;
    public IClienteRepository Clientes { get; }
    public IPedidoRepository Pedidos { get; }

    public UnitOfWork(AppDbContext context,
                      IClienteRepository clienteRepo,
                      IPedidoRepository pedidoRepo)
    {
        _context = context;
        Clientes = clienteRepo;
        Pedidos = pedidoRepo;
    }

    public async Task<int> SaveAsync() => await _context.SaveChangesAsync();
    public void Dispose() => _context.Dispose();
```

```
}
```

🧩 4. Uso en la capa de servicios

```
public class PedidoService
{
    private readonly IUnitOfWork _uow;

    public PedidoService(IUnitOfWork uow) => _uow = uow;

    public async Task CrearPedidoAsync(Pedido pedido)
    {
        await _uow.Pedidos.AddAsync(pedido);
        await _uow.SaveAsync();
    }

    public async Task<IEnumerable<Cliente>> ListarClientesVIPAsync()
        => await _uow.Clientes.GetClientesVIPAsync();
}
```

🧱 5. Registro en el contenedor de dependencias

```
builder.Services.AddScoped<IClienteRepository, ClienteRepository>();
builder.Services.AddScoped<IPedidoRepository, PedidoRepository>();
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
```

6. Beneficios arquitectónicos

Beneficio	Descripción
Desacoplamiento	La lógica de negocio no depende de EF Core.
Testabilidad	Se pueden usar repositorios falsos o mocks.
Reutilización	Mismo repositorio para varios proyectos.
Consistencia	UnitOfWork garantiza transacciones atómicas.
Escalabilidad	Se puede cambiar la fuente de datos sin modificar la lógica.

Buenas prácticas

- Mantén **repositorios ligeros**; la lógica compleja debe estar en servicios.
- No dupliques métodos genéricos (usa el Repository<T> base).
- Inyecta **UnitOfWork** en controladores, no los repositorios sueltos.
- Centraliza transacciones con SaveAsync().
- Usa interfaces para facilitar pruebas unitarias.

Objetivo

Aplicar patrones estructurados para **abstraer el acceso a datos**, garantizando limpieza, mantenibilidad y coherencia transaccional entre múltiples repositorios.

Capítulo 23 – Integración con SQL Server, PostgreSQL y MongoDB

"No existe la mejor base de datos, sino la más adecuada para cada tipo de información."

🧭 Introducción

Las aplicaciones actuales suelen manejar distintos tipos de datos: **estructurados, semiestructurados y no estructurados**.
Por eso, una arquitectura moderna en .NET puede combinar **bases relacionales (SQL Server, PostgreSQL)** con **NoSQL (MongoDB)**.
Este enfoque híbrido ofrece **flexibilidad, rendimiento y escalabilidad**.

⚙️ 1. SQL Server con Entity Framework Core

SQL Server es el motor más usado en el ecosistema .NET y se integra de forma nativa con EF Core.

📦 Instalación

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

⚙️ Configuración

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("SqlServer")));
```

Ejemplo de conexión:

```
"ConnectionStrings": {
  "SqlServer": "Server=.;Database=MiEmpresaDB;Trusted_Connection=True;Encrypt=False;"
}
```

✅ Uso

Ideal para **datos estructurados, integridad referencial y transacciones** complejas.

🧩 2. PostgreSQL con EF Core

PostgreSQL es una alternativa potente, open source y compatible con SQL estándar.

📦 Instalación

dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL

⚙️ Configuración

```
builder.Services.AddDbContext<PostgresContext>(options =>
    options.UseNpgsql(builder.Configuration.GetConnectionString("Postgres")));
```

Ejemplo:

```
"ConnectionStrings": {
  "Postgres": "Host=localhost;Database=MiEmpresaPG;Username=postgres;Password=1234"
}
```

🧠 Diferencias clave con SQL Server

Característica	SQL Server	PostgreSQL
Licencia	Propietario	Open Source
JSON nativo	Parcial	Completo
Rendimiento analítico	Alto	Muy alto con CTE y particiones
Extensiones	Limitadas	Amplias (PostGIS, TimescaleDB)

➡️ PostgreSQL es ideal para **datos geográficos, analíticos o híbridos**.

🧱 3. MongoDB con .NET (NoSQL)

MongoDB almacena documentos JSON flexibles, ideales para datos dinámicos y esquemas variables.
Es muy usado en microservicios, catálogos, logs o configuraciones.

📦 Instalación

dotnet add package MongoDB.Driver

⚙️ Configuración

```
builder.Services.AddSingleton<IMongoClient>(sp =>
```

```
    new MongoClient(builder.Configuration.GetConnectionString("MongoDB")));
```

Ejemplo de conexión:

```
"ConnectionStrings": {
  "MongoDB": "mongodb://localhost:27017"
}
```

Ejemplo de uso

```
public class ProductoMongoRepository
{
    private readonly IMongoCollection<Producto> _productos;

    public ProductoMongoRepository(IMongoClient client)
    {
        var db = client.GetDatabase("MiEmpresaNoSQL");
        _productos = db.GetCollection<Producto>("Productos");
    }

    public async Task<List<Producto>> ObtenerTodosAsync() =>
        await _productos.Find(_ => true).ToListAsync();

    public async Task InsertarAsync(Producto p) =>
        await _productos.InsertOneAsync(p);
}
```

✅ **Ventajas**

- Esquema flexible.
- Excelente para datos JSON o semiestructurados.
- Escala horizontalmente.
- Muy rápido para lecturas masivas o consultas no relacionales.

🔗 **4. Arquitectura híbrida SQL + NoSQL**

Un sistema moderno puede combinar lo mejor de ambos mundos:

Pedidos y clientes → SQL Server

Logs y auditorías → MongoDB

Catálogo de productos → PostgreSQL

Ejemplo práctico:

```
public class DataHub
{
    public IUnitOfWork Sql { get; }
    public ProductoMongoRepository NoSql { get; }

    public DataHub(IUnitOfWork sql, ProductoMongoRepository mongo)
    {
        Sql = sql;
        NoSql = mongo;
    }

    public async Task SincronizarProductoAsync(int id)
    {
        var producto = await Sql.Productos.GetByIdAsync(id);
        await NoSql.InsertarAsync(producto);
    }
```

}

➡️ El arquitecto decide qué base usar según **tipo, frecuencia y estructura del dato**.

⚡ 5. Estrategias de sincronización

Escenario	Estrategia	Herramienta
Sincronización eventual	Mensajería o eventos	RabbitMQ, Kafka
Backup cruzado	Jobs periódicos	Hangfire, Azure Functions
Lecturas rápidas de datos	Cachear en Redis	Redis Cache
Auditoría en NoSQL	Replicación parcial	MongoSink, ETL personalizado

---

🧠 6. Buenas prácticas del arquitecto

- Centraliza las cadenas de conexión en appsettings.json.
- Aísla cada contexto (SQL, PostgreSQL, Mongo) en su propio repositorio.
- No mezcles transacciones entre motores distintos.
- Usa **mensajería eventual** para mantener sincronización.
- Mide latencia y throughput antes de decidir el motor.
- Documenta claramente la **responsabilidad de cada base**.

✅ Objetivo

Diseñar una persistencia híbrida que combine **bases relacionales y NoSQL**, eligiendo la herramienta correcta para cada tipo de dato y asegurando interoperabilidad sin perder consistencia ni rendimiento.

Capítulo 24 – Data Migrations y Seeders

"Una base de datos bien gestionada evoluciona junto al código, sin romper el pasado ni frenar el futuro."

🧭 Introducción

Cuando el modelo de datos cambia, la base debe evolucionar con él.
Entity Framework Core integra un sistema de **migraciones** que registra los cambios en el esquema y los aplica en distintos entornos.
Además, los **seeders** permiten poblar información inicial o de referencia automáticamente (roles, usuarios, parámetros, etc.).

El arquitecto .NET debe garantizar **consistencia, trazabilidad y automatización** en este proceso.

⚙️ 1. Migraciones con Entity Framework Core

Las migraciones son "versiones" de la base que reflejan los cambios del modelo a lo largo del tiempo.

📦 Crear la primera migración

dotnet ef migrations add Inicial

⚙️ Aplicar cambios en la base

dotnet ef database update

Esto genera una tabla interna llamada **__EFMigrationsHistory**, donde EF guarda el historial de migraciones aplicadas.

🧩 2. Ejemplo completo

Supongamos que agregamos una nueva entidad:

```
public class Categoria
{
    public int Id { get; set; }
    public string Nombre { get; set; } = "";
}
```

Agregamos el DbSet al contexto:

```
public DbSet<Categoria> Categorias { get; set; }
```

Creamos la nueva migración:

```
dotnet ef migrations add AgregarCategoria
dotnet ef database update
```

EF creará una tabla Categorias automáticamente.

⚙️ 3. Migraciones por entorno

En entornos grandes, puede ser necesario mantener migraciones separadas o condicionales.

Ejemplo en CI/CD:

```
dotnet ef database update --context AppDbContext --environment Production
```

O bien, aplicar migraciones desde código al iniciar la app:

```
using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    db.Database.Migrate();
}
```

➡️ Ideal para **despliegues automatizados** (sin ejecutar comandos manuales).

🧩 4. Revertir o eliminar migraciones

Si se creó una por error:

```
dotnet ef migrations remove
```

Para volver a un estado anterior:

```
dotnet ef database update NombreMigracionAnterior
```

⚡ 5. Seeders: Datos iniciales

Los **seeders** insertan información necesaria al iniciar la aplicación.
EF Core permite hacerlo directamente en el contexto.

Ejemplo en OnModelCreating

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Categoria>().HasData(
        new Categoria { Id = 1, Nombre = "Tecnología" },
        new Categoria { Id = 2, Nombre = "Hogar" }
    );
}
```

Cada HasData() se convierte en comandos INSERT al ejecutar database update.

---

6. Seeders personalizados

Para datos dinámicos o externos, se puede usar un servicio de inicialización.

```
public class DataSeeder
{
    private readonly AppDbContext _context;

    public DataSeeder(AppDbContext context) => _context = context;

    public async Task SeedAsync()
    {
        if (!_context.Roles.Any())
        {
            _context.Roles.AddRange(
                new Rol { Nombre = "Admin" },
                new Rol { Nombre = "Usuario" }
            );
            await _context.SaveChangesAsync();
        }
```

```
    }
}
```

Y ejecutarlo al inicio:

```
using (var scope = app.Services.CreateScope())
{
    var seeder = scope.ServiceProvider.GetRequiredService<DataSeeder>();
    await seeder.SeedAsync();
}
```

🧠 7. Buenas prácticas del arquitecto

- Cada cambio estructural debe ir acompañado de una **migración**.
- No edites archivos de migración manualmente.
- Versiona migraciones junto al código fuente (Git).
- Usa HasData() solo para datos fijos o catálogos.
- Evita sembrar grandes volúmenes con HasData (usa seeders personalizados).
- Aplica migraciones automáticamente en CI/CD, no manualmente.
- Mantén **un contexto por dominio**, no una base monolítica.

✅ Objetivo

Automatizar el ciclo de vida de la base de datos mediante **migraciones versionadas y seeders controlados**, asegurando coherencia entre código y datos en todos los entornos.

Capítulo 25 – Optimización de Consultas y Caching Distribuido

"El rendimiento no es magia, es el resultado de entender qué debe calcularse… y qué no."

🧭 Introducción

El rendimiento en la capa de datos define la **experiencia del usuario y la escalabilidad del sistema**.
Como arquitecto .NET, tu misión es detectar los puntos costosos (consultas lentas, exceso de lecturas, joins innecesarios) y mitigarlos con **estrategias de consulta inteligente y caching eficiente**.

⚙️ 1. Optimización de consultas en EF Core

🔹 Usa AsNoTracking() para lecturas

Evita que EF rastree entidades que no se modificarán:

```
var productos = await context.Productos.AsNoTracking().ToListAsync();
```

🔹 Proyecciones directas

Evita traer columnas innecesarias:

```
var resumen = await context.Productos
    .Select(p => new { p.Nombre, p.Precio })
    .ToListAsync();
```

🔹 Filtrado en base de datos, no en memoria

❌ Evita:

```
var datos = context.Productos.ToList().Where(p => p.Activo);
```

✅ Mejor:

```
var datos = await context.Productos.Where(p => p.Activo).ToListAsync();
```

🔹 Indexa columnas consultadas

Usa Fluent API para definir índices:

```
modelBuilder.Entity<Producto>()
    .HasIndex(p => p.Nombre)
    .HasDatabaseName("IX_Producto_Nombre");
```

🧩 2. Eager loading selectivo

Solo usa .Include() cuando sea necesario:

```
var cliente = await context.Clientes
    .Include(c => c.Pedidos.Where(p => p.Total > 1000))
    .FirstAsync();
```

➡️ Evita cargas masivas que traen datos innecesarios.

⚡ 3. Compilación de consultas

EF Core permite **cachear consultas compiladas** para reutilizarlas y mejorar la velocidad:

```
private static readonly Func<AppDbContext, decimal, IEnumerable<Pedido>> _consulta =
    EF.CompileQuery((AppDbContext ctx, decimal minTotal) =>
        ctx.Pedidos.Where(p => p.Total > minTotal));
```

Uso:

```
var pedidos = _consulta(context, 500);
```

🧱 4. Caching distribuido (Redis)

Cuando los datos cambian poco, puedes **cachear los resultados de consultas costosas** usando Redis.

📦 Instalación

```
dotnet add package Microsoft.Extensions.Caching.StackExchangeRedis
```

⚙️ Configuración

```
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost:6379";
    options.InstanceName = "AppCache_";
});
```

Ejemplo

```
public class ProductoCacheService
{
    private readonly IDistributedCache _cache;
    private readonly AppDbContext _context;

    public ProductoCacheService(IDistributedCache cache, AppDbContext context)
    {
        _cache = cache;
        _context = context;
    }

    public async Task<List<Producto>> GetProductosAsync()
    {
        var cacheKey = "productos_todos";
        var data = await _cache.GetStringAsync(cacheKey);
        if (data != null)
            return JsonSerializer.Deserialize<List<Producto>>(data)!;

        var productos = await _context.Productos.AsNoTracking().ToListAsync();
        var json = JsonSerializer.Serialize(productos);

        await _cache.SetStringAsync(cacheKey, json, new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)
        });

        return productos;
```

```
    }
}
```

➡️ La próxima llamada obtiene los datos directamente de Redis, sin tocar la base.

🔗 5. Estrategias avanzadas

Estrategia	Uso	Ejemplo
Cache-Aside	Consultar cache, luego BD si no existe	Redis o MemoryCache
Sliding Expiration	Mantener cache mientras se use	Configurar expiración dinámica
Cache por capa	API cachea datos ya filtrados	Resultados de consultas
Invalidación	Eliminar datos obsoletos al modificar	RemoveAsync() tras SaveChanges

---

🧠 6. Diagnóstico y métricas

- Usa **EnableSensitiveDataLogging()** (solo en desarrollo).
- Visualiza las consultas SQL generadas:
- context.Database.Log = Console.Write;
- Analiza tiempos con **Application Insights o MiniProfiler**.
- Mide **hit ratio** del cache: cuántas veces se evita ir a la BD.
- Implementa métricas Prometheus/Grafana para la capa de datos.

🔍 7. Escenarios de cuello de botella

Problema	Síntoma	Solución
Consultas sin índices	Lentitud creciente	Crear índices o particiones
N+1 queries	Exceso de lecturas por Include	Eager loading controlado
Sin cache distribuido	Carga excesiva del DB	Implementar Redis
Contexto largo	Bloqueos o inconsistencias	Usar Scoped o Transient
Consultas sin filtro	RAM alta	Paginar resultados

✅ Objetivo

Evitar cuellos de botella en la base de datos mediante **consultas optimizadas, uso eficiente de LINQ y caching distribuido**, logrando tiempos de respuesta mínimos y escalabilidad real en entornos productivos.

⚙️ Parte VI – Microservicios y Contenedores

Capítulo 26 – Principios de Microservicios

"No se trata de hacer sistemas pequeños, sino de hacer sistemas que crezcan sin romperse."

🧭 Introducción

Los **microservicios** son una evolución del modelo monolítico tradicional.
En lugar de una sola aplicación que hace todo, el sistema se divide en **servicios pequeños, autónomos y especializados**, que se comunican entre sí mediante APIs o mensajería.
Su objetivo: lograr **agilidad, escalabilidad y resiliencia**.

⚙️ 1. Qué son los microservicios

Un microservicio es una **unidad funcional independiente**, con su propia base de datos y ciclo de vida.
Cada uno se centra en un **bounded context** (contexto delimitado) dentro del dominio.

Ejemplo:

- Servicio de **Pedidos**
- Servicio de **Clientes**
- Servicio de **Inventario**
- Servicio de **Notificaciones**

Cada uno puede desarrollarse, desplegarse y escalarse **sin afectar a los demás**.

🧩 2. Diferencias entre Monolito y Microservicios

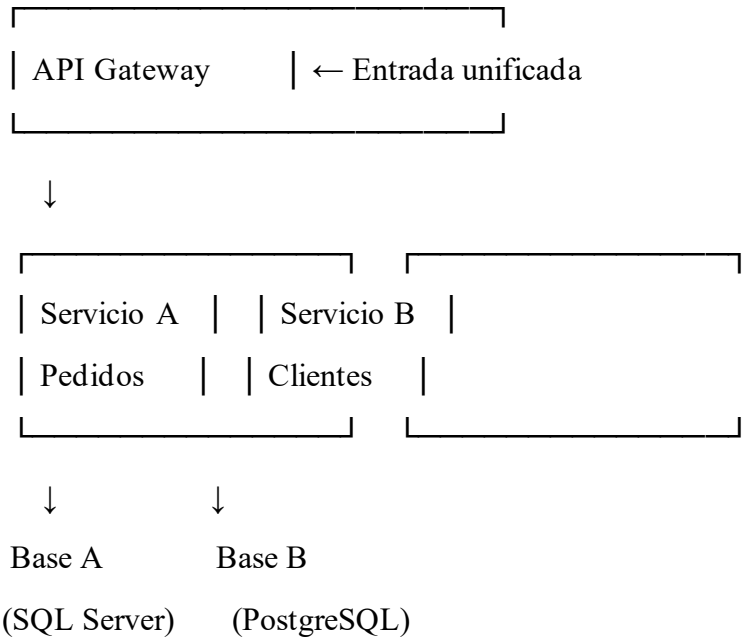
Característica	Monolito	Microservicios
Despliegue	Un único bloque	Independiente por servicio
Escalado	Completo	Por componente
Base de datos	Compartida	Por servicio
Fallos	Afecta a todo el sistema	Aislado por servicio
Tecnología	Uniforme	Libre por módulo

Característica	Monolito	Microservicios
Comunicación	Llamadas internas	API / Mensajería

🧠 3. Principios fundamentales

1. **Autonomía** → Cada servicio debe poder desarrollarse y desplegarse solo.
2. **Desacoplamiento** → Comunicación vía contratos estables (API REST o eventos).
3. **Propiedad de datos** → Cada microservicio gestiona su propia base.
4. **Escalabilidad independiente** → Aumenta instancias solo donde haga falta.
5. **Resiliencia** → Si un servicio falla, los demás deben seguir funcionando.
6. **Observabilidad** → Cada servicio debe registrar, medir y reportar su estado.

⚙️ 4. Arquitectura típica de microservicios



La comunicación puede ser:

- **Síncrona:** vía HTTP/REST o gRPC.
- **Asíncrona:** vía RabbitMQ, Kafka, Azure Service Bus.

5. Diseño de contratos estables

Cada microservicio debe definir su **contrato público** (API o evento).
Ejemplo de contrato REST:

GET /api/pedidos/{id}

```
{
  "id": 102,
  "clienteId": 4,
  "total": 120.50,
  "estado": "Enviado"
}
```

➡️ El contrato es la frontera; el interior puede cambiar libremente.

---

6. Comunicación entre microservicios

- **REST (HTTP):** simple, universal, pero más lento.
- **gRPC:** binario y eficiente, ideal para alto rendimiento.
- **Mensajería (RabbitMQ/Kafka):** para integración asíncrona y eventos.

Ejemplo:
El servicio de **Pedidos** publica un evento:

```
{
  "evento": "PedidoCreado",
  "pedidoId": 102,
  "clienteId": 4
}
```

El servicio de **Notificaciones** lo consume y envía un correo.

🧩 7. Base de datos por servicio

Cada microservicio posee su propia base.
No hay consultas cruzadas directas; la sincronización se hace por eventos o APIs.

Servicio	Base	Tecnología
Pedidos	pedidos_db	SQL Server
Clientes	clientes_db	PostgreSQL
Logística	logistica_db	MongoDB

➡️ Evita el "acoplamiento de datos": el error más común al migrar desde un monolito.

🧠 8. Beneficios del enfoque

✅ Escalabilidad granular
✅ Resiliencia ante fallos
✅ Libertad tecnológica
✅ Despliegue independiente
✅ Mejor alineación con equipos ágiles

⚠️ 9. Desventajas y desafíos

⚠️ Complejidad en la comunicación
⚠️ Configuración distribuida
⚠️ Dificultad de debugging
⚠️ Coordinación de despliegues
⚠️ Observabilidad más compleja

El arquitecto debe equilibrar el **valor del desacoplamiento** con el **coste operativo**.

🧩 10. Cuándo aplicar microservicios

✅ Tienes equipos grandes y especializados.
✅ El monolito crece sin control.
✅ Se necesita despliegue continuo e independiente.
✅ Existen módulos con picos de carga distintos.
❌ No los uses para proyectos pequeños o simples: la complejidad no compensa.

✅ Objetivo

Comprender los **principios de diseño, ventajas y desafíos** de los microservicios, y cuándo aplicarlos para lograr sistemas flexibles, resilientes y escalables.

Capítulo 27 – Comunicación entre Servicios (HTTP, RabbitMQ, Kafka)

"En un sistema distribuido, la verdadera arquitectura no está en los servicios, sino en sus conversaciones."

🧭 Introducción

Los microservicios no existen aislados.
Su poder surge cuando **colaboran** entre sí para cumplir una función de negocio.
Como arquitecto, debes decidir **cómo y cuándo se comunican**:

- **Síncrono (HTTP, gRPC)** → inmediato, directo.
- **Asíncrono (RabbitMQ, Kafka)** → desacoplado, resiliente.

Ambos enfoques pueden coexistir dentro de la misma arquitectura.

⚙️ 1. Comunicación síncrona: HTTP y REST

El método más común.
Permite que un servicio invoque directamente a otro usando un endpoint.

📘 Ejemplo en .NET (HttpClientFactory)

```
builder.Services.AddHttpClient("ClientesService", client =>
{
    client.BaseAddress = new Uri("https://clientes-api/");
});
```

Uso:

```
public class PedidoService
{
    private readonly HttpClient _http;

    public PedidoService(IHttpClientFactory factory)
    {
        _http = factory.CreateClient("ClientesService");
```

```
    }

    public async Task<Cliente?> ObtenerCliente(int id)
    {
        var resp = await _http.GetAsync($"api/clientes/{id}");
        if (!resp.IsSuccessStatusCode) return null;
        return await resp.Content.ReadFromJsonAsync<Cliente>();
    }
}
```

✅ Simple, legible y perfecto para respuestas rápidas.
⚠️ Depende de la red; si un servicio falla, puede bloquear a otros.

🧩 2. Comunicación síncrona de alto rendimiento: gRPC

gRPC usa **Protocol Buffers** en lugar de JSON, lo que reduce tamaño y mejora velocidad.
Ideal para **microservicios internos o de alta frecuencia**.

📦 Instalación

```
dotnet add package Grpc.AspNetCore
dotnet add package Grpc.Tools
```

🧠 Definición de contrato (.proto)

```
syntax = "proto3";

service Pedidos {
  rpc ObtenerPedido (PedidoRequest) returns (PedidoResponse);
}

message PedidoRequest { int32 id = 1; }
message PedidoResponse { int32 id = 1; string estado = 2; double total = 3; }
```

⚙️ Implementación en el servidor

```
public class PedidosService : Pedidos.PedidosBase
{
   public override Task<PedidoResponse> ObtenerPedido(PedidoRequest req,
ServerCallContext ctx)
   {
      return Task.FromResult(new PedidoResponse { Id = req.Id, Estado = "Enviado", Total =
120 });
   }
}
```

🔗 Cliente gRPC

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Pedidos.PedidosClient(channel);
var respuesta = await client.ObtenerPedidoAsync(new PedidoRequest { Id = 10 });
```

✅ Rápido, eficiente y seguro.
⚠️ Requiere compatibilidad en clientes (ideal para backend interno).

🐇 3. Comunicación asíncrona: RabbitMQ

RabbitMQ implementa el modelo **publish/subscribe** (mensajería).
Perfecto para eventos, colas de trabajo y comunicación desacoplada.

📦 Instalación

```
dotnet add package RabbitMQ.Client
```

🧩 Productor

```
var factory = new ConnectionFactory() { HostName = "localhost" };
using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.QueueDeclare("pedidos", false, false, false, null);
```

```
var mensaje = Encoding.UTF8.GetBytes("Pedido creado: 102");
channel.BasicPublish("", "pedidos", null, mensaje);
```

🧩 **Consumidor**

```
var factory = new ConnectionFactory() { HostName = "localhost" };
using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.QueueDeclare("pedidos", false, false, false, null);

var consumer = new EventingBasicConsumer(channel);
consumer.Received += (s, e) =>
{
    var body = e.Body.ToArray();
    Console.WriteLine($"Mensaje recibido: {Encoding.UTF8.GetString(body)}");
};

channel.BasicConsume("pedidos", true, consumer);
Console.ReadLine();
```

✅ Tolerante a fallos y sin bloqueo.
⚠️ No garantiza orden ni entrega inmediata.

🧱 4. Comunicación basada en eventos: Kafka

Apache Kafka se usa para **streaming y procesamiento de datos en tiempo real**.
Ideal cuando los microservicios deben reaccionar a eventos.

📦 Instalación

```
dotnet add package Confluent.Kafka
```

🧩 Productor Kafka

```
var config = new ProducerConfig { BootstrapServers = "localhost:9092" };
using var producer = new ProducerBuilder<Null, string>(config).Build();

await producer.ProduceAsync("pedidos-topic",
    new Message<Null, string> { Value = "Pedido #120 creado" });
```

🧩 Consumidor Kafka

```
var config = new ConsumerConfig
{
    BootstrapServers = "localhost:9092",
    GroupId = "pedido-consumer",
    AutoOffsetReset = AutoOffsetReset.Earliest
};

using var consumer = new ConsumerBuilder<Ignore, string>(config).Build();
consumer.Subscribe("pedidos-topic");

while (true)
{
    var cr = consumer.Consume();
    Console.WriteLine($"Evento: {cr.Message.Value}");
}
```

✅ Escalable, confiable y preparado para Big Data.
⚠️ Requiere infraestructura robusta.

🔗 5. Cuándo usar cada tecnología

Escenario	Tecnología	Tipo
Solicitudes rápidas o CRUD	HTTP/REST	Síncrono
Comunicación interna de alto rendimiento	gRPC	Síncrono
Eventos entre servicios o colas de trabajo	RabbitMQ	Asíncrono
Flujos de datos masivos / auditoría	Kafka	Asíncrono

🧠 6. Buenas prácticas arquitectónicas

- Diseña **contratos claros** (OpenAPI, .proto, esquemas JSON).
- Implementa **reintentos y circuit breakers** (Polly).
- Usa **trazabilidad distribuida** (correlation ID).
- Define **nombres únicos de colas y topics** por dominio.
- Documenta las dependencias entre servicios.
- Evita ciclos de comunicación directa (usa un broker).

✅ Objetivo

Dominar los modelos **síncronos y asíncronos** de comunicación en microservicios .NET, sabiendo elegir entre **REST, gRPC, RabbitMQ y Kafka** según el tipo de carga, acoplamiento y necesidad de escalabilidad.

Capítulo 28 – Docker y Kubernetes con .NET

"Contenerizar no es solo empacar código; es empacar confiabilidad."

🧭 Introducción

Antes, desplegar una aplicación .NET implicaba servidores dedicados, configuraciones manuales y dependencias frágiles.
Con **Docker**, cada servicio se ejecuta en su propio **contenedor aislado**, idéntico en cualquier entorno.
Kubernetes, por su parte, orquesta estos contenedores, los escala y garantiza su disponibilidad.

⚙️ 1. ¿Qué es Docker?

Docker es una tecnología que **empaqueta aplicaciones y dependencias** en contenedores.
Un contenedor es una instancia ligera que comparte el kernel del sistema, pero se comporta como si fuera una máquina independiente.

Concepto	Descripción
Imagen	Plantilla con la app y entorno
Contenedor	Ejecución de una imagen
Dockerfile	Script para construir imágenes
Registry	Repositorio de imágenes (Docker Hub, ACR)

🧩 2. Crear un Dockerfile para .NET

Ejemplo para una API .NET 8:

```
# Etapa 1: Compilación
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY . .
RUN dotnet restore
RUN dotnet publish -c Release -o /app
```

```
# Etapa 2: Ejecución
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build /app .
EXPOSE 8080
ENTRYPOINT ["dotnet", "Pedidos.Api.dll"]
```

📦 **Construir y ejecutar**

```
docker build -t pedidos-api .
docker run -d -p 8080:8080 pedidos-api
```

Acceso: http://localhost:8080/swagger

✅ Portable, reproducible y aislado.
⚠️ Cada microservicio debe tener su propio Dockerfile.

⚙️ **3. Docker Compose (multi-contenedor)**

Permite levantar varios servicios juntos (API, base de datos, cache…).

```
version: '3.9'
services:
  pedidos:
    image: pedidos-api
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - sqlserver

  sqlserver:
    image: mcr.microsoft.com/mssql/server:2022-latest
    environment:
```

```
    SA_PASSWORD: "Password123!"
    ACCEPT_EULA: "Y"
  ports:
    - "1433:1433"
```

```
docker-compose up -d
```

➡️ Ideal para entornos locales o pruebas integradas.

🛞 4. Introducción a Kubernetes

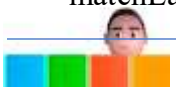
Kubernetes (K8s) es un **orquestador de contenedores**: gestiona el ciclo de vida, escalado, red y balanceo.
Sus componentes clave:

Elemento	Función
Pod	Unidad mínima (uno o varios contenedores)
Deployment	Controla versiones y actualizaciones
Service	Expone los pods en red
Ingress	Gestiona acceso HTTP/HTTPS externo
ConfigMap / Secret	Configuración y credenciales

🧩 5. Desplegar un microservicio .NET en Kubernetes

📄 **deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pedidos-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
```



```
    app: pedidos
  template:
    metadata:
      labels:
        app: pedidos
    spec:
      containers:
      - name: pedidos-api
        image: pedidos-api:latest
        ports:
        - containerPort: 8080
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: pedidos-service
spec:
  selector:
    app: pedidos
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

Despliegue

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Comprobar:

```
kubectl get pods
```

kubectl get services

✅ Kubernetes levantará los pods, balanceará tráfico y reiniciará los fallidos automáticamente.

⚡ 6. Configuración y secretos

Evita credenciales en el código. Usa ConfigMap y Secret:

kubectl create configmap appsettings --from-file=appsettings.Production.json

kubectl create secret generic sql-cred --from-literal=SA_PASSWORD=Password123!

Y en el deployment.yaml:

```
env:
- name: ConnectionStrings__SqlServer
  valueFrom:
    secretKeyRef:
      name: sql-cred
      key: SA_PASSWORD
```

🧠 7. Observabilidad y logs

- **kubectl logs [pod-name]** → Ver logs de un pod.
- **kubectl describe pod [name]** → Diagnóstico detallado.
- Integración con **Prometheus, Grafana o Azure Monitor**.
- Usa **Application Insights** para métricas dentro del contenedor.

🧩 8. Beneficios del enfoque Docker + K8s

✅ Entornos reproducibles y portátiles.
✅ Escalado automático por carga.
✅ Despliegues continuos sin downtime.
✅ Recuperación automática ante fallos.
✅ Integración directa con **Azure Kubernetes Service (AKS)** o AWS EKS.

⚠️ 9. Buenas prácticas del arquitecto

- Usa **imágenes oficiales y firmadas**.
- Mantén los contenedores **inmutables**.
- Versiona tus imágenes (v1.0, v1.1, etc.).
- Evita latest en producción.
- Configura límites de CPU/memoria (resources en YAML).
- Usa **Helm charts** para despliegues complejos.

✅ Objetivo

Contenerizar y orquestar aplicaciones .NET mediante **Docker y Kubernetes**, logrando entornos reproducibles, escalables y fácilmente desplegables en cualquier infraestructura cloud o local.

Capítulo 29 – Configuración Centralizada y Resiliencia (Polly)

"La resiliencia no es evitar los errores, sino continuar funcionando a pesar de ellos."

🧭 Introducción

En un sistema de microservicios, los fallos son inevitables: redes lentas, APIs caídas, bases inalcanzables…
Un arquitecto .NET debe diseñar servicios **resistentes**, capaces de recuperarse automáticamente.
A su vez, mantener configuraciones distribuidas en archivos locales es inviable: necesitamos una **fuente centralizada y segura**.

⚙️ 1. Configuración centralizada

Cuando cada microservicio tiene su propio appsettings.json, los cambios se vuelven difíciles de controlar.
La solución es **unificar la configuración** en un servicio central que todos consulten dinámicamente.

🔹 Opciones comunes

Solución	Descripción	Integración
Azure App Configuration	Servicio PaaS para almacenar claves globales	Integración nativa con .NET
Consul	Almacenamiento distribuido y descubrimiento de servicios	HashiCorp + .NET
etcd	Backend ligero usado en Kubernetes	API REST
Spring Cloud Config	Compatible con entornos híbridos	vía HTTP

🧩 2. Azure App Configuration (ejemplo)

📦 Instalación

dotnet add package Microsoft.Extensions.Configuration.AzureAppConfiguration

⚙️ Configuración en Program.cs

```
builder.Configuration.AddAzureAppConfiguration(options =>
{
```

```
    options.Connect("Endpoint=https://miappconfig.azconfig.io;Id=xxxx;Secret=yyyy")
           .Select(KeyFilter.Any, LabelFilter.Null);
});
```

Ahora tus valores (por ejemplo "AppSettings:ApiKey") se leen igual que en appsettings.json, pero pueden cambiar **en tiempo real sin reiniciar el servicio**.

🔐 3. Configuración jerárquica

Puedes usar etiquetas o prefijos para separar entornos:

Clave	Valor	Etiqueta
Pedidos:Timeout	30	Producción
Pedidos:Timeout	5	Desarrollo

Con LabelFilter eliges qué entorno cargar:

```
options.Select(KeyFilter.Any, "Producción");
```

➡️ Ideal para manejar **decenas de microservicios con configuraciones diferentes.**

🧠 4. Introducción a la resiliencia con Polly

Polly es una biblioteca de .NET que implementa patrones de resiliencia como:

- **Retry** (reintentos)
- **Circuit Breaker** (bloquear llamadas tras errores)
- **Timeout** (cancelar operaciones lentas)
- **Fallback** (respuestas alternativas)
- **Bulkhead** (limitar concurrencia)

📦 Instalación

```
dotnet add package Polly
dotnet add package Microsoft.Extensions.Http.Polly
```

⚙️ 5. Integración con HttpClientFactory

```
builder.Services.AddHttpClient("PedidosService", client =>
{
    client.BaseAddress = new Uri("https://pedidos-api/");
})
.AddTransientHttpErrorPolicy(p =>
    p.WaitAndRetryAsync(3, intento => TimeSpan.FromSeconds(Math.Pow(2, intento))))
.AddTransientHttpErrorPolicy(p =>
    p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

🧩 Explicación:

- 3 reintentos con espera exponencial (2, 4, 8 s).
- Si 5 fallos consecutivos → **Circuito abierto** 30 s (no se intenta más).

⚡ 6. Uso manual de políticas Polly

```
var policy = Policy
    .Handle<HttpRequestException>()
    .Retry(3, (ex, intento) =>
        Console.WriteLine($"Reintento #{intento}: {ex.Message}"));

await policy.ExecuteAsync(async () =>
{
    var resp = await _httpClient.GetAsync("api/pedidos");
    resp.EnsureSuccessStatusCode();
});
```

También puedes combinar políticas:

```
var policy = Policy.WrapAsync(retryPolicy, breakerPolicy, timeoutPolicy);
```

🧩 7. Patrón Circuit Breaker visual

[Estado Cerrado] → llamadas normales

↓ errores

[Estado Abierto] → corta llamadas temporalmente

↓ tiempo pasa

[Estado Semiabierto] → prueba una llamada

↓ éxito

[Vuelve a Cerrado]

➡️ Este patrón **protege recursos** y evita cascadas de fallos entre servicios.

🧱 8. Resiliencia complementaria

- Usa **Bulkhead** para limitar el número de peticiones simultáneas.
- Implementa **TimeoutPolicy** para evitar bloqueos prolongados.
- Configura **FallbackPolicy** para respuestas por defecto ("modo degradado").
- Registra métricas de fallos con **Application Insights**.
- Asegura logs claros de cada política aplicada.

🧩 9. Ejemplo de fallback combinado

```
var fallback = Policy<string>
    .Handle<Exception>()
    .FallbackAsync("Servicio no disponible temporalmente");

var retry = Policy
    .Handle<Exception>()
    .RetryAsync(2);

var final = Policy.WrapAsync(fallback, retry);
```

```
var respuesta = await final.ExecuteAsync(async () =>
{
    return await _httpClient.GetStringAsync("https://api.interna/estado");
});
```

🧠 10. Buenas prácticas del arquitecto

- Centraliza la configuración en servicios externos (Azure AppConfig, Consul).
- Versiona las claves y protege las sensibles en Azure Key Vault.
- Define políticas de resiliencia según criticidad del servicio.
- Evita reintentos infinitos: mejor fallar rápido.
- Registra siempre los errores de Polly.
- Integra alertas cuando el circuito esté "abierto" por demasiado tiempo.

✅ Objetivo

Diseñar microservicios **resilientes y autoconfigurables**, capaces de adaptarse a entornos cambiantes, tolerar fallos temporales y mantener la operación sin intervención manual.

Capítulo 30 – Observabilidad y Logging Distribuido

"No puedes mejorar lo que no puedes observar."
— Peter Drucker

🧭 Introducción

En un sistema distribuido, los logs locales ya no bastan.
Cada microservicio puede ejecutarse en su propio contenedor o incluso en otra nube.
Por eso, el arquitecto debe implementar **observabilidad integral**, que combine tres pilares:

1. **Logs** – qué ocurrió.
2. **Métricas** – qué tan bien ocurrió.
3. **Trazas** – cómo ocurrió (ruta de una petición).

⚙️ 1. Logging estructurado en .NET

El sistema de logging integrado permite registrar información con niveles y proveedores configurables.

📘 Ejemplo básico

```
private readonly ILogger<PedidosController> _logger;


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


[HttpGet]
public IActionResult Get()
{
   _logger.LogInformation("Consulta de pedidos iniciada a {hora}", DateTime.Now);
   return Ok("Pedidos consultados");
}
```

🔹 **Niveles de log**

Nivel	Uso
Trace	Diagnóstico muy detallado
Debug	Información de desarrollo
Information	Operaciones normales
Warning	Comportamientos inesperados
Error	Fallos manejables
Critical	Fallos graves del sistema

🧩 2. Logging distribuido con Serilog

Serilog permite generar logs estructurados (JSON) y enviarlos a archivos, bases o sistemas externos.

📦 Instalación

```
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Sinks.File
dotnet add package Serilog.Sinks.Seq
```

⚙️ Configuración

```
Log.Logger = new LoggerConfiguration()
    .Enrich.FromLogContext()
    .WriteTo.File("logs/app.log",  rollingInterval: RollingInterval.Day)
    .WriteTo.Seq("http://localhost:5341")
    .CreateLogger();

builder.Host.UseSerilog();
```

➡️ **Seq** o **Elastic Stack** (ELK) permiten analizar logs en tiempo real con filtros y dashboards.

⚙️ 3. Application Insights (Azure)

Application Insights (AI) ofrece monitoreo unificado: logs, métricas, excepciones, dependencias y trazas.

📦 Instalación

dotnet add package Microsoft.ApplicationInsights.AspNetCore

⚙️ Configuración

builder.Services.AddApplicationInsightsTelemetry(builder.Configuration["ApplicationInsights:ConnectionString"]);

En Azure Portal podrás visualizar:

- Latencia promedio por endpoint.
- Fallos por servicio.
- Mapa de dependencias.
- Trazas distribuidas entre microservicios.

🔗 4. OpenTelemetry (OTel)

OpenTelemetry es el estándar abierto para instrumentar aplicaciones.
Permite enviar trazas y métricas a distintos backends (Prometheus, Jaeger, Zipkin, Grafana Cloud…).

📦 Instalación

dotnet add package OpenTelemetry

dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol

dotnet add package OpenTelemetry.Extensions.Hosting

⚙️ Configuración en Program.cs

```
builder.Services.AddOpenTelemetry()
    .WithTracing(tracerProviderBuilder =>
        tracerProviderBuilder
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddEntityFrameworkCoreInstrumentation()
```

```
        .AddOtlpExporter(o => o.Endpoint = new Uri("http://localhost:4317")));
```

➡️ Cada request entre microservicios genera un **trace ID** que permite seguir la llamada completa a través del sistema.

🧠 5. Métricas personalizadas

Puedes crear métricas propias (latencia, número de pedidos, etc.):

```
var meter = new Meter("Pedidos.Metrics");
var counter = meter.CreateCounter<int>("pedidos_creados_total");


counter.Add(1, KeyValuePair.Create<string, object?>("estado", "confirmado"));
```

Estas métricas se pueden visualizar en **Prometheus**, **Grafana** o **Azure Monitor**.

🔍 6. Correlación de trazas

Agrega un **RequestId** o **CorrelationId** para unir logs de distintos servicios:

```
app.Use(async (ctx, next) =>
{
    ctx.Items["CorrelationId"] = Guid.NewGuid().ToString();
    await next();
});
```

Y luego en cada log:

```
_logger.LogInformation("Procesando pedido {@CorrelationId}", ctx.Items["CorrelationId"]);
```

➡️ Facilita el rastreo de una petición desde el Gateway hasta los microservicios internos.

⚡ 7. Dashboards y alertas

- **Grafana**: paneles personalizados con Prometheus.
- **Azure Application Insights**: alertas automáticas (por tiempo de respuesta, errores 5xx, etc.).
- **Kibana (Elastic)**: análisis textual avanzado de logs.

- **Seq**: monitoreo local rápido y visual.

🧱 8. Buenas prácticas de observabilidad

✅ Centraliza todos los logs.
✅ Usa formato estructurado (JSON).
✅ Asigna un ServiceName único a cada microservicio.
✅ Propaga traceId y spanId entre llamadas HTTP.
✅ No registres información sensible.
✅ Define retención y almacenamiento adecuados.
✅ Implementa alertas automáticas ante anomalías.

✅ Objetivo

Implementar **observabilidad distribuida y logging avanzado** en microservicios .NET, integrando Application Insights, Serilog y OpenTelemetry para detectar fallos, analizar rendimiento y mantener una trazabilidad completa del sistema.

Parte VII – Nube y DevOps

Capítulo 31 – Introducción a Azure

"La nube no es solo un lugar donde se ejecuta tu código, sino una plataforma donde tu arquitectura evoluciona."

Introducción

Microsoft Azure es un **ecosistema completo de servicios** que permiten ejecutar, escalar y administrar aplicaciones .NET sin preocuparte por la infraestructura.
Desde máquinas virtuales hasta microservicios, bases de datos, IA o pipelines de CI/CD, Azure es el entorno natural para cualquier arquitecto .NET moderno.

---

1. ¿Qué es Azure?

Azure es una **plataforma cloud de propósito general**, que combina servicios de infraestructura (IaaS), plataforma (PaaS) y software (SaaS).
Su ventaja principal es la **integración nativa con .NET, Visual Studio y GitHub**, lo que facilita el ciclo completo de desarrollo y despliegue.

---

2. Categorías principales de servicios

Categoría	Ejemplos	Propósito
Compute	App Service, Functions, Kubernetes (AKS), Virtual Machines	Ejecutar aplicaciones y servicios
Storage	Blob, File, Queue, Table	Almacenar datos estructurados o binarios
Database	SQL Database, Cosmos DB, PostgreSQL, Redis Cache	Persistencia de datos
Networking	VNet, Load Balancer, Application Gateway	Conectividad y balanceo
Security	Key Vault, Defender for Cloud, Azure AD	Protección y gestión de identidades
DevOps	Azure DevOps, GitHub Actions	Automatización del ciclo de vida

Categoría	Ejemplos	Propósito
Monitoring	Application Insights, Log Analytics	Observabilidad y alertas

---

🧩 3. Modelos de servicio en la nube

Modelo	Responsabilidad del proveedor	Responsabilidad del usuario
IaaS (Infraestructura)	Red, hardware, virtualización	Sistema operativo y app
PaaS (Plataforma)	Infraestructura + sistema operativo	Código y configuración
SaaS (Software)	Todo el stack completo	Solo uso del software

➡️ En .NET, lo habitual es usar **PaaS**, como App Service o Functions, para evitar la gestión de servidores.

🧠 4. Zonas, regiones y disponibilidad

Azure se distribuye globalmente en **regiones** (ej. *West Europe*, *East US*) con **Zonas de Disponibilidad** redundantes.

- **Region Pairing:** replicación automática entre dos regiones cercanas.
- **Availability Zone:** centros de datos físicos distintos dentro de una región.
- **SLA (Service Level Agreement):** define el nivel de disponibilidad garantizado (ej. 99.95%).

📍 Consejo: elige siempre regiones cercanas a tus usuarios y habilita replicación geográfica.

⚡ 5. Azure Portal, CLI y PowerShell

Tienes tres formas de gestionar recursos:

- **Azure Portal:** interfaz gráfica amigable.
- **Azure CLI:** comandos multiplataforma.
- **Azure PowerShell:** para administradores Windows.

📦 Ejemplo CLI

az login

az group create --name RG-Pedidos --location "westeurope"

az appservice plan create --name PlanPedidos --resource-group RG-Pedidos --sku B1

az webapp create --name pedidos-api --plan PlanPedidos --resource-group RG-Pedidos --runtime "DOTNET:8"

🔐 6. Identidad y seguridad: Azure AD

Azure Active Directory (hoy Entra ID) gestiona usuarios, roles y autenticación.
Permite:

- Iniciar sesión con **OAuth2, OpenID o SSO**.
- Controlar acceso a APIs (.NET con Microsoft.Identity).
- Integrarse con **Microsoft 365** o **GitHub Enterprise**.

➡️ Es la base de la autenticación moderna en ecosistemas Microsoft.

🧩 7. Azure Resource Manager (ARM)

ARM es la capa que describe, controla y automatiza recursos mediante **plantillas declarativas JSON**.

Ejemplo:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "resources": [
    {
      "type": "Microsoft.Web/sites",
      "apiVersion": "2022-09-01",
      "name": "pedidos-api",
      "location": "westeurope",
      "properties": { "serverFarmId": "PlanPedidos" }
```

```
    }
  ]
}
```

Puedes desplegar con:

```
az deployment group create --resource-group  RG-Pedidos --template-file template.json
```

➡️ Ideal para **infraestructura como código (IaC)**.

🔗 8. Integración natural con .NET

Visual Studio y .NET CLI permiten desplegar directamente a Azure:

```
dotnet publish -c Release
```

```
az webapp deploy --resource-group  RG-Pedidos --name pedidos-api --src-path ./bin/Release/net8.0/publish
```

Además, el SDK de Azure facilita acceso a Storage, Key Vault, AI, etc.:

```
var blobService = new BlobServiceClient(connectionString);
```

```
var container = blobService.GetBlobContainerClient("facturas");
```

```
await container.UploadBlobAsync("invoice.pdf", stream);
```

🧠 9. Costos y escalabilidad

- Usa **Pricing Calculator** para estimar costos.
- Configura **Auto Scaling** (App Service Plan o AKS).
- Activa **Budget Alerts** para evitar sobrecostes.
- Usa **Reserved Instances** si tu carga es constante.
- Considera **Azure Spot** para tareas de bajo costo (no críticas).

💡 10. Buenas prácticas del arquitecto cloud

- ✅ Usa **nombres consistentes** en todos los recursos.
- ✅ Agrupa por **Resource Group** (lógica y entorno).
- ✅ Automatiza con **Bicep o Terraform**.
- ✅ Aplica **etiquetas (tags)** para costos y ownership.
- ✅ Protege secretos con **Azure Key Vault**.
- ✅ Supervisa todo con **Application Insights + Log Analytics**.

✅ Objetivo

Comprender la **arquitectura base de Azure**, sus servicios principales y cómo integrarlos con proyectos .NET para diseñar soluciones escalables, seguras y automatizadas en la nube.

Capítulo 32 – Azure App Service, Functions y Storage

"En la nube moderna, el arquitecto diseña sistemas que escalan solos, se ejecutan siempre y cuestan solo cuando se usan."

🧭 Introducción

El arquitecto .NET debe dominar los servicios **PaaS** de Azure, donde Microsoft se encarga de la infraestructura y tú te concentras en el código.
App Service aloja tus aplicaciones web o APIs, Functions ejecuta tareas bajo demanda, y Storage guarda datos de forma segura y masiva.

⚙️ 1. Azure App Service

Azure App Service es un entorno totalmente administrado para **aplicaciones web, APIs REST y backends móviles**.
Permite desplegar directamente desde Visual Studio, GitHub o pipelines CI/CD.

📦 Creación desde CLI

az appservice plan create --name PlanWeb --resource-group RG-App --sku B1

az webapp create --name pedidos-api --plan PlanWeb --runtime "DOTNET:8" --resource-group RG-App

Luego puedes publicar directamente desde Visual Studio con **"Publicar en Azure App Service"**.

⚙️ Variables y configuración

Cada appsettings.json puede reemplazarse con configuraciones en el portal:

- ConnectionStrings
- API keys
- Logging Levels

Y puedes añadir un **Slot de despliegue** (por ejemplo "staging") para hacer pruebas sin afectar producción:

az webapp deployment slot create --name pedidos-api --resource-group RG-App --slot staging

➡️ Cuando confirmas que todo funciona, haces *Swap* entre *staging* y *production* sin downtime.

⚡ 2. Azure Functions: serverless .NET

Azure Functions ejecuta fragmentos de código bajo demanda.
Pagas solo por ejecución, no por servidor en línea.
Son perfectas para tareas **asíncronas, integraciones y automatización**.

📘 Ejemplo básico

```
[FunctionName("EnviarCorreoPedido")]

public static void Run([QueueTrigger("pedidos", Connection = "AzureWebJobsStorage")] string pedido)

{

    Console.WriteLine($"Correo enviado para: {pedido}");

}
```

📦 Creación con CLI

```
func init PedidoFunc --dotnet

cd PedidoFunc

func new --name EnviarCorreoPedido --template "QueueTrigger"

func start
```

🌐 Despliegue

```
func azure functionapp publish PedidoFuncApp
```

➡️ Puedes integrar con colas, blobs, HTTP, timers o incluso eventos de Cosmos DB.

🧱 3. Azure Storage

Azure Storage es el sistema de almacenamiento universal de Microsoft, con varias modalidades:

Tipo	Uso	Ejemplo
Blob	Archivos grandes (PDF, imágenes, backups)	BlobServiceClient
File Share	Carpetas accesibles por SMB	Mapear unidades en Windows
Queue	Mensajería ligera	Integración con Functions
Table	Datos NoSQL simples	Alternativa a Cosmos DB Lite

📘 Ejemplo con Blobs

```
var blobService = new BlobServiceClient(connectionString);
var container = blobService.GetBlobContainerClient("reportes");
await container.CreateIfNotExistsAsync();
await container.UploadBlobAsync("informe.pdf", File.OpenRead("informe.pdf"));
```

Y para leer:

```
var blobClient = container.GetBlobClient("informe.pdf");
await blobClient.DownloadToAsync("local_informe.pdf");
```

➡️ Ideal para almacenar archivos de usuario, logs o backups automáticos.

🧩 4. Integración App Service + Storage + Functions

Un flujo común:

1. El usuario sube un archivo a **Blob Storage**.
2. Se genera un evento que activa una **Azure Function**.
3. La Function procesa el archivo y actualiza la **API en App Service**.

Este patrón se denomina **Event-Driven Architecture** (arquitectura dirigida por eventos).

🧠 5. Monitorización y escalado automático

🔹 Autoescalado (App Service Plan)

Azure ajusta instancias según CPU, memoria o tráfico:

```
az monitor autoscale create \
  --resource-group RG-App \
  --resource pedidos-api \
  --min-count 1 --max-count 5 \
  --count 2
```

🔹 **Logs y diagnósticos**

- Application Insights integrado.
- "Live Metrics Stream" para ver tráfico en tiempo real.
- Registro de errores por slot o instancia.

⚡ **6. Autenticación y seguridad integradas**

App Service puede proteger tu API sin código:

- Conexión directa con **Azure AD / Entra ID**.
- OAuth2 y OpenID integrados.
- Restricción por IP o certificados.
- HTTPS forzado por defecto.

Y para Functions:

[Authorize] // si usas .NET Identity o JWT

🔗 **7. Casos de uso**

Servicio	Ideal para	Ejemplo
App Service	Webs, APIs, microservicios medianos	API REST .NET 8
Functions	Tareas por evento, temporizadores	Procesar colas o correos
Storage	Archivos, logs, backups	Subida de facturas, informes

💡 **8. Buenas prácticas del arquitecto**

✅ Usa App Service Slots para despliegues sin interrupción.
✅ Configura "Always On" para evitar latencia inicial.
✅ Protege Storage con SAS Tokens o Managed Identity.
✅ Mantén Functions ligeras y con dependencias mínimas.
✅ Centraliza logs con Application Insights.
✅ Automatiza el despliegue con Azure DevOps o GitHub Actions.

✅ Objetivo

Dominar los servicios **Azure App Service**, **Functions** y **Storage**, desplegando aplicaciones y tareas serverless .NET seguras, escalables y con monitoreo integrado.

Capítulo 33 – Bases de Datos y Seguridad en la Nube

"La nube no es insegura; la inseguridad está en las malas configuraciones."

🧭 Introducción

Azure ofrece una amplia gama de **servicios de bases de datos gestionadas**, tanto relacionales como NoSQL.
Estos servicios eliminan la carga del mantenimiento (backups, parches, alta disponibilidad) y se integran nativamente con la seguridad de Azure.
Como arquitecto, tu misión es elegir el tipo de base más adecuado y aplicar una **estrategia de seguridad de datos integral**.

⚙️ 1. Principales bases de datos en Azure

Tipo	Servicio	Ideal para	Características
Relacional	Azure SQL Database	Aplicaciones .NET clásicas	Alta disponibilidad, escalado, T-SQL completo
Open Source	Azure Database for PostgreSQL/MySQL	Proyectos multi-stack	Totalmente gestionadas
NoSQL	Cosmos DB	Microservicios globales	Baja latencia, replicación automática
En memoria	Azure Cache for Redis	Caching distribuido	Sub-milisegundos de respuesta
Analítica	Synapse Analytics	Big Data, BI	Integración con Power BI y Data Lake

---

🧩 2. Azure SQL Database

Azure SQL es el heredero cloud de SQL Server, pero **totalmente administrado**.
No necesitas instalar, actualizar ni hacer backups manuales.

📦 Creación rápida con CLI

az sql server create --name sqlpedidos --resource-group RG-Data --location westeurope --admin-user admin --admin-password P@ssw0rd!

az sql db create --resource-group RG-Data --server sqlpedidos --name pedidosdb --service-objective S0

📘 Conexión desde .NET

var conn = new SqlConnection("Server=tcp:sqlpedidos.database.windows.net;Database=pedidosdb;User Id=admin;Password=P@ssw0rd!;");

await conn.OpenAsync();

🧱 3. Cosmos DB (NoSQL global)

Cosmos DB es la opción para microservicios distribuidos, IoT o APIs de alto rendimiento. Soporta varios modelos: **SQL, MongoDB, Cassandra, Gremlin**.

📦 Ejemplo en .NET

var client = new CosmosClient("<connection-string>");

var db = await client.CreateDatabaseIfNotExistsAsync("PedidosDB");

var container = await db.Database.CreateContainerIfNotExistsAsync("Pedidos", "/clienteId");

await container.Container.CreateItemAsync(new { id = "101", clienteId = "C01", total = 120.50 });

➡️ Replicación automática entre regiones y lectura a milisegundos.

⚡ 4. Seguridad de datos en Azure

🔒 a) Cifrado en reposo

Todos los datos se cifran automáticamente con **Transparent Data Encryption (TDE)**.

🔒 b) Cifrado en tránsito

Usa **TLS 1.2+** en las cadenas de conexión:

Encrypt=True;TrustServerCertificate=False;

🔒 c) Managed Identity

Permite a las apps autenticarse sin almacenar contraseñas.

```
var credential = new DefaultAzureCredential();
var client = new SecretClient(new Uri("https://mi-keyvault.vault.azure.net/"), credential);
```

🧠 5. Azure Key Vault: gestión de secretos

Azure Key Vault guarda contraseñas, cadenas de conexión y certificados de forma segura.

📦 Creación

```
az keyvault create --name kvpedidos --resource-group RG-Data --location westeurope
az keyvault secret set --vault-name kvpedidos --name "SqlConnection" --value "Server=tcp:sqlpedidos.database.windows.net;Database=pedidosdb;"
```

⚙️ Acceso desde .NET

```
builder.Configuration.AddAzureKeyVault(
    new Uri("https://kvpedidos.vault.azure.net/"),
    new DefaultAzureCredential());
```

Ahora puedes acceder:

```
var connectionString = builder.Configuration["SqlConnection"];
```

➡️ No vuelvas a guardar secretos en appsettings.json.

🧩 6. Seguridad de red

- **Private Endpoints:** acceso privado a BD desde redes internas.
- **Firewall Rules:** restringe por IP o red virtual.
- **VNet Integration:** comunicación interna entre servicios sin salir a Internet.
- **Defender for SQL:** detección de anomalías y ataques de inyección.

⚙️ 7. Copias de seguridad y recuperación

Azure SQL mantiene **backups automáticos** (7-35 días según plan).
Cosmos DB ofrece **restore por punto en el tiempo (PITR)**.
Además, puedes exportar datos a **Blob Storage** para retención extendida.

```
az sql db export --server sqlpedidos --name pedidosdb --admin-user admin --admin-password P@ssw0rd! --storage-key-type StorageAccessKey --storage-key "clave..." --storage-uri "https://mibucket.blob.core.windows.net/backups/pedidos.bacpac"
```

🧩 8. Auditoría y control de acceso

- **Azure AD Authentication:** usuarios gestionados por roles.
- **Row-Level Security (RLS):** controla qué filas puede ver cada usuario.
- **Auditing + Log Analytics:** registro completo de operaciones SQL.
- **Customer Managed Keys (CMK):** tus propias claves de cifrado.

💡 9. Buenas prácticas del arquitecto

✅ Nunca expongas credenciales en texto plano.
✅ Usa Key Vault y Managed Identity siempre que sea posible.
✅ Aplica "principio de menor privilegio" en bases y roles.
✅ Habilita cifrado en tránsito y en reposo.
✅ Usa alertas de Defender for SQL y monitoreo de consultas lentas.
✅ Configura backups automáticos y restauración probada.

✅ Objetivo

Diseñar bases de datos **seguras, cifradas y resilientes** en Azure, integrando la autenticación sin secretos y asegurando la privacidad de los datos en reposo y tránsito.

Capítulo 34 – Azure DevOps y GitHub Actions

"Automatizar no es solo ahorrar tiempo, es eliminar el riesgo humano del proceso."

🧭 Introducción

El desarrollo moderno se basa en **CI/CD (Integración y Despliegue Continuos)**.
Cada vez que un desarrollador hace un *commit*, el sistema ejecuta pruebas, compila, crea artefactos y despliega automáticamente.
Azure DevOps y GitHub Actions son las dos herramientas más usadas en el ecosistema .NET para lograrlo.

⚙️ 1. ¿Qué es CI/CD?

- **Integración Continua (CI)** → Cada cambio en el código se construye y prueba automáticamente.
- **Despliegue Continuo (CD)** → Los artefactos validados se publican automáticamente en entornos (Dev, QA, Prod).

➡️ El objetivo: **entregas rápidas, seguras y confiables.**

🧩 2. Azure DevOps: componentes clave

Componente	Función
Repos	Control de versiones (Git)
Pipelines	Automatización CI/CD
Boards	Gestión de tareas y backlog
Artifacts	Repositorio de paquetes NuGet o npm
Test Plans	Ejecución de pruebas automatizadas

📘 Ejemplo: pipeline básico en YAML

```
trigger:
  - main

pool:
  vmImage: 'windows-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '8.0.x'

- script: dotnet build --configuration Release
  displayName: 'Compilar solución'

- script: dotnet test --no-build
  displayName: 'Ejecutar pruebas'

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
```

➡️ Cada commit en main dispara la compilación y pruebas automáticas.

3. Despliegue automático (CD) en Azure

Puedes agregar un segundo pipeline o tarea para desplegar:

```
- task: AzureWebApp@1
  inputs:
    azureSubscription: 'ConexiónAzure'
    appType: 'webApp'
    appName: 'pedidos-api'
    package: '$(System.DefaultWorkingDirectory)/**/*.zip'
```

Este paso toma el artefacto compilado y lo publica en **Azure App Service** automáticamente.

---

4. GitHub Actions: alternativa moderna

GitHub Actions permite definir flujos CI/CD directamente dentro del repositorio GitHub.
Cada flujo se define en un archivo .yml dentro de .github/workflows.

Ejemplo: flujo CI/CD para .NET 8

```
name: CI/CD .NET 8

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Instalar .NET
      uses: actions/setup-dotnet@v4
```

```
    with:
      dotnet-version: '8.0.x'

  - name: Compilar
    run: dotnet build --configuration Release

  - name: Ejecutar pruebas
    run: dotnet test --no-build

  - name: Publicar en Azure
    uses: azure/webapps-deploy@v3
    with:
      app-name: 'pedidos-api'
      publish-profile: ${{ secrets.AZURE_PUBLISH_PROFILE }}
      package: './bin/Release/net8.0/publish'
```

➡️ GitHub Actions se integra directamente con **Azure App Service**, **AKS** o **Functions**.
Solo necesitas guardar tus credenciales o *publish profile* en **GitHub Secrets**.

🧠 5. Secretos y seguridad en pipelines

Nunca guardes contraseñas o tokens en texto plano.
Usa:

- **Azure Key Vault** integrado con DevOps.
- **GitHub Secrets** (Settings → Secrets → Actions).
- **Managed Identities** en despliegues automatizados.

Ejemplo de uso seguro en YAML:

```
env:
  CONNECTION_STRING: ${{ secrets.SQL_CONN }}
```

🧱 6. Estrategia multi-entorno

Define entornos por etapas:

Build → Test → Staging → Production

Cada paso requiere aprobación:

```
stages:
- stage: Deploy_Prod
  dependsOn: Build
  jobs:
  - deployment: WebApp
    environment: 'Production'
    strategy:
      runOnce:
        deploy:
          steps:
          - script: echo "Desplegando en producción"
```

➡️ Ideal para **validaciones manuales** y **control de versiones por entorno**.

⚡ 7. Integración con pruebas automáticas

Puedes agregar pruebas unitarias, de integración o carga:

```
- script: dotnet test --logger trx --results-directory $(Build.ArtifactStagingDirectory)
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testResultsFiles: '**/*.trx'
```

➡️ Los resultados aparecerán en el dashboard de Azure DevOps o en la pestaña *Actions* de GitHub.

🧩 8. Monitoreo y retroalimentación

Una vez desplegado, vincula tu pipeline con **Application Insights** o **Log Analytics**.
Así podrás visualizar:

- Errores por versión.
- Tiempos de despliegue.
- Impacto de los commits.

💡 9. Buenas prácticas del arquitecto

✅ Versiona todos los pipelines en YAML (infraestructura como código).
✅ Usa variables y plantillas reutilizables.
✅ Ejecuta pruebas antes de cada despliegue.
✅ Integra alertas en Teams o correo ante fallos.
✅ Revisa logs de compilación y despliegue.
✅ Configura entornos separados por rama (dev, staging, prod).

✅ Objetivo

Automatizar el **ciclo completo de desarrollo, pruebas y despliegue** de aplicaciones .NET mediante **Azure DevOps** o **GitHub Actions**, logrando entregas seguras, consistentes y sin intervención manual.

Capítulo 35 – CI/CD y Despliegues Automatizados

"La verdadera entrega continua no es rapidez… es confianza."

🧭 Introducción

En este capítulo diseñaremos un **pipeline de CI/CD robusto para .NET** que va desde el *commit* hasta producción.
El objetivo es eliminar los pasos manuales, validar automáticamente la calidad del código y desplegar de forma **segura, controlada y reversible**.

⚙️ 1. Flujo general de un pipeline profesional

Commit → Build → Test → Package → Deploy → Monitor

Cada etapa debe tener objetivos y validaciones específicas:

Etapa	Objetivo	Validaciones
Build	Compilar código	Errores de compilación
Test	Garantizar calidad	Pruebas unitarias, cobertura
Package	Crear artefactos firmados	Integridad y versión
Deploy	Publicar versión	Configuración correcta
Monitor	Validar salud post-despliegue	Logs, métricas, rollback

---

🧩 2. Pipeline CI/CD completo (.NET + Azure)

Archivo: .azure-pipelines.yml

```
trigger:
  branches:
    include:
      - main

variables:
  buildConfiguration: 'Release'
```

```
  webAppName: 'pedidos-api'
  artifactName: 'drop'

stages:
- stage: Build
  jobs:
  - job: BuildJob
    pool:
      vmImage: 'windows-latest'
    steps:
    - task: UseDotNet@2
      inputs:
        packageType: 'sdk'
        version: '8.0.x'

    - script: dotnet restore
      displayName: 'Restaurar dependencias'

    - script: dotnet build --configuration $(buildConfiguration)
      displayName: 'Compilar solución'

    - script: dotnet test --no-build --configuration $(buildConfiguration)
      displayName: 'Ejecutar pruebas unitarias'

    - task: DotNetCoreCLI@2
      inputs:
        command: 'publish'
        publishWebProjects: true
```

```
        arguments: '--configuration $(buildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
        zipAfterPublish: true

    - task: PublishBuildArtifacts@1
      inputs:
        pathToPublish: '$(Build.ArtifactStagingDirectory)'
        artifactName: '$(artifactName)'

- stage: Deploy
  dependsOn: Build
  jobs:
  - deployment: DeployJob
    environment: 'Production'
    strategy:
      runOnce:
        deploy:
          steps:
          - download: current
            artifact: '$(artifactName)'

          - task: AzureWebApp@1
            inputs:
              azureSubscription: 'ConexiónAzure'
              appName: '$(webAppName)'
              package: '$(Pipeline.Workspace)/$(artifactName)/**/*.zip'
```

➡️ Este flujo:

1. Compila, prueba y empaqueta el proyecto.
2. Publica el artefacto.
3. Despliega automáticamente en **Azure App Service**.

🗲 3. GitHub Actions equivalente

Archivo: .github/workflows/cicd.yml

```
name: CI/CD .NET 8 to Azure

on:
  push:
    branches: [ "main" ]

jobs:
  build-deploy:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4

    - name: Setup .NET
      uses: actions/setup-dotnet@v4
      with:
        dotnet-version: '8.0.x'

    - name: Restore dependencies
      run: dotnet restore
```

```
- name: Build
  run: dotnet build --configuration Release

- name: Test
  run: dotnet test --no-build --verbosity normal

- name: Publish
  run: dotnet publish -c Release -o ./publish

- name: Deploy to Azure
  uses: azure/webapps-deploy@v3
  with:
    app-name: 'pedidos-api'
    publish-profile: ${{ secrets.AZURE_PUBLISH_PROFILE }}
    package: './publish'
```

✅ Compatible con App Service, AKS o Azure Functions.

🧠 4. Versionado automático

Usa **GitVersion** o etiquetas Git para marcar versiones:

```
git tag v1.0.0
```

En YAML:

```
variables:
  version: '1.0.$(Build.BuildId)'
```

Y en el nombre del artefacto:

```
artifactName: 'drop_$(version)'
```

🧩 5. Rollback automático

Si un despliegue falla o degrada rendimiento:

- Mantén **slots de despliegue** ("staging" y "production").
- Despliega en *staging* → prueba → **swap**.
- Si hay fallos → swap inverso = rollback inmediato.

az webapp deployment slot swap --name pedidos-api --slot staging --target-slot production

➡️ Todo sin downtime.

🔍 6. Pruebas post-despliegue

Agrega un paso que valide la salud del servicio:

```
- script: |
    curl -s -o /dev/null -w "%{http_code}" https://pedidos-api.azurewebsites.net/health
  displayName: "Verificar endpoint de salud"
```

Si el código no es 200 → marca el despliegue como fallido.

⚙️ 7. Integración con monitoreo

Conecta Application Insights:

```
- task: AzureAppServiceSettings@1
  inputs:
    azureSubscription: 'ConexiónAzure'
    appName: 'pedidos-api'
    appSettings: |
      -key APPINSIGHTS_INSTRUMENTATIONKEY -value $(APPINSIGHTS_KEY)
```

Así cada despliegue genera telemetría inmediata (errores, tiempos de carga, rendimiento).

🧱 8. Estrategia multi-entorno

Define entornos progresivos:

Desarrollo → QA → Staging → Producción

Cada entorno con su propio recurso en Azure y variables:

```
variables:
  devApp: 'pedidos-api-dev'
  prodApp: 'pedidos-api'
```

Y validaciones manuales antes de producción:

```
environments:
  - name: Production
    protectionRules:
      requiredReviewers:
        - admin@example.com
```

💡 9. Buenas prácticas del arquitecto DevOps

- ✅ Un pipeline por aplicación, no por entorno.
- ✅ Mantén pipelines versionados junto al código.
- ✅ Configura pruebas automáticas antes del despliegue.
- ✅ Implementa slots y rollback instantáneo.
- ✅ Usa variables seguras y secretos externos.
- ✅ Registra cada despliegue en Application Insights.
- ✅ Aplica alertas de fallos post-deploy.

✅ Objetivo

Diseñar un **pipeline CI/CD .NET completamente automatizado**, que compile, pruebe, despliegue, verifique y supervise sin intervención manual, garantizando entregas continuas, seguras y reversibles.

Parte VIII – Seguridad, Escalabilidad y Rendimiento

Comenzamos la **Parte VIII – Seguridad, Escalabilidad y Rendimiento**, donde el arquitecto .NET asume una nueva responsabilidad: **proteger, fortalecer y acelerar** sus soluciones en producción.
Aquí aprenderás a blindar tus APIs, manejar identidades, controlar accesos y defender tus datos frente a ataques comunes.

Capítulo 36 – Seguridad en APIs y Datos

"No hay arquitectura sólida sin seguridad desde el diseño."

Introducción

En la nube y en entornos distribuidos, la superficie de ataque crece.
El arquitecto .NET debe garantizar que **cada API, base de datos y flujo de información esté protegido**.
Esto implica autenticación, autorización, cifrado, validación, políticas y auditoría.

---

1. Principios de seguridad en APIs

1. **Cero confianza (Zero Trust):** nadie se considera seguro por defecto.
2. **Defensa en profundidad:** múltiples capas de protección.
3. **Menor privilegio:** cada servicio o usuario solo accede a lo necesario.
4. **Auditoría constante:** todo acceso se registra y revisa.
5. **Cifrado total:** en tránsito y en reposo.

---

2. Autenticación y autorización en .NET

Autenticación con JWT (JSON Web Tokens)

Usada para autenticar usuarios y microservicios.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
```

```
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidIssuer = "https://auth.miapp.com",
        ValidAudience = "https://api.miapp.com",
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes("clave-segura-super-larga"))
    };
});
```

Y en el controlador:

```
[Authorize]
[HttpGet("pedidos")]
public IActionResult GetPedidos() => Ok("Acceso autorizado");
```

- **Roles y políticas**

```
[Authorize(Roles = "Admin")]
public IActionResult SoloAdmins() => Ok("Bienvenido, administrador");
```

También puedes definir políticas personalizadas:

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("MayorEdad", policy =>
        policy.RequireClaim("Edad", "18"));
});
```

3. Protección ante ataques comunes

Ataque	Prevención
SQL Injection	Usa parámetros en EF o Dapper, nunca concatenes strings.
XSS (Cross-Site Scripting)	Escapa contenido HTML en vistas o respuestas.
CSRF (Cross-Site Request Forgery)	Usa [ValidateAntiForgeryToken] en formularios o tokens en API.
Brute Force / DDoS	Implementa rate limiting y API Gateway.
Directory Traversal	Valida rutas y evita acceso directo al sistema de archivos.

Rate limiting (límite de peticiones)

Desde .NET 8 puedes usar **Rate Limiting Middleware** nativo:

```
builder.Services.AddRateLimiter(options =>
{
    options.AddFixedWindowLimiter("default", opt =>
    {
        opt.Window = TimeSpan.FromSeconds(10);
        opt.PermitLimit = 5; // Máximo 5 peticiones cada 10 seg
        opt.QueueLimit = 2;
    });
});

app.UseRateLimiter();
```

➡️ Protege tus endpoints de abusos o bots.

🔒 4. Cifrado de datos sensibles

🔹 En tránsito

Todos los servicios deben usar HTTPS/TLS 1.2+:

```
app.UseHttpsRedirection();
```

🔹 En reposo

Cifra datos manualmente si los guardas fuera de Azure SQL o Key Vault:

```
using var aes = Aes.Create();
aes.Key = key;
aes.IV = iv;
```

🔹 En configuración

Nunca pongas claves en appsettings.json; usa **Azure Key Vault** o **User Secrets**.

🧠 5. Seguridad de cabeceras HTTP

Agrega políticas con el paquete **NWebsec** o el middleware nativo:

```
app.Use(async (ctx, next) =>
{
    ctx.Response.Headers.Add("X-Content-Type-Options", "nosniff");
    ctx.Response.Headers.Add("X-Frame-Options", "DENY");
    ctx.Response.Headers.Add("Referrer-Policy", "no-referrer");
    ctx.Response.Headers.Add("X-XSS-Protection", "1; mode=block");
    await next();
});
```

➡️ Recomendado para APIs públicas o frontends expuestos.

🔗 6. Seguridad entre microservicios

- Usa **mTLS (mutual TLS)** para validar identidad entre servicios.
- Firma mensajes o eventos con claves compartidas.
- Usa **Managed Identities** en Azure para conexión segura a recursos (sin contraseñas).
- Aplica **Network Security Groups (NSG)** y **Private Endpoints** para aislar tráfico interno.

🧩 7. Auditoría de accesos

Registra toda acción crítica (creación, borrado, login, cambios de permisos):

```
_logger.LogInformation("Usuario {user} eliminó pedido {pedidoId} a las {fecha}",
                       user, pedidoId, DateTime.UtcNow);
```

Y envía esos logs a **Application Insights** o **Log Analytics** para correlación y alertas.

⚙️ 8. Pruebas de seguridad

- **OWASP ZAP** o **Burp Suite**: escaneo automático de vulnerabilidades.
- **Dependency Check**: analiza librerías con CVEs conocidas.
- **Azure Security Center / Defender for Cloud**: auditoría continua del entorno.

💡 9. Buenas prácticas del arquitecto

✅ "Security by Design": la seguridad se diseña desde el inicio.
✅ Aplica MFA (autenticación multifactor) en todos los entornos.
✅ Revisa dependencias con vulnerabilidades.
✅ Automatiza el análisis de seguridad en CI/CD.
✅ Implementa políticas de contraseñas fuertes y expiración.
✅ Monitorea intentos de acceso y alertas anómalas.

✅ Objetivo

Diseñar sistemas .NET **seguros, auditables y resistentes**, aplicando autenticación moderna, cifrado, políticas de acceso y defensas frente a amenazas comunes.

Capítulo 37 – Escalabilidad Horizontal y Vertical

"El software bien diseñado no solo funciona… también crece."

🧭 Introducción

Toda aplicación exitosa debe estar preparada para **crecer**.
La escalabilidad es la capacidad de un sistema para **mantener su rendimiento** al aumentar la demanda.
Existen dos grandes enfoques: **escalar verticalmente** (mejor hardware) y **escalar horizontalmente** (más instancias).

⚙️ 1. Escalabilidad vertical (Scale Up)

Consiste en **aumentar la potencia** de un solo servidor o instancia:

- Más CPU, memoria o almacenamiento.
- Sin cambios en el código.

📘 Ejemplo (Azure App Service)

Cambiar el plan B1 → P2V2:

```
az appservice plan update --name PlanPedidos --resource-group RG-App --sku P2V2
```

✅ Ventajas

- Simple de implementar.
- Sin cambios de arquitectura.
- Ideal para sistemas monolíticos.

⚠️ Desventajas

- Costo creciente.
- Límite físico (máquina finita).
- No mejora tolerancia a fallos.

⚙️ 2. Escalabilidad horizontal (Scale Out)

Implica **añadir más instancias** que trabajan en paralelo.
Requiere que la aplicación sea **stateless** (sin estado compartido en memoria).

📘 Ejemplo

En App Service:

```
az monitor autoscale create \
  --resource pedidos-api \
  --resource-group RG-App \
  --min-count 2 --max-count 10 \
  --count 3
```

Azure se encargará de crear o eliminar instancias según CPU o tráfico.

🧩 3. Stateless: la clave del escalado

Evita guardar datos de sesión o usuario en memoria del servidor.

❌ Mal ejemplo:

```
HttpContext.Session["Usuario"] = "Oscar";
```

✅ Buen ejemplo:

Usar **Azure Redis Cache** o **SQL Session State**:

```
services.AddDistributedRedisCache(options =>
{
    options.Configuration = "miRedis.redis.cache.windows.net";
});
```

➡️ Esto permite que cualquier instancia atienda cualquier usuario.

4. Balanceo de carga (Load Balancing)

Distribuye el tráfico entre múltiples instancias.
Azure lo gestiona automáticamente con **Application Gateway** o **Azure Front Door**.

Tipos de balanceo:

- **Round Robin:** asigna solicitudes por turno.
- **Least Connections:** prioriza instancias menos cargadas.
- **Geo-balancing:** dirige al servidor más cercano.

az network front-door create --name frontPedidos --resource-group RG-App --accepted-protocols Https

---

5. Escalado automático (Auto Scaling)

Permite crecer o reducir recursos según demanda.
Criterios comunes:

- CPU > 70% durante 5 min → +1 instancia.
- CPU < 30% durante 10 min → -1 instancia.

También puede basarse en colas o métricas personalizadas (número de pedidos, conexiones, etc.).

Evita pagar por capacidad ociosa.

---

6. Escalabilidad en bases de datos

Escalar verticalmente:

Mejor tier (DTU/vCore).

az sql db update --name pedidosdb --service-objective S3

Escalar horizontalmente:

Usar **sharding** o **read replicas**.
Cosmos DB lo hace automáticamente con *partition keys*:

"partitionKey": { "paths": ["/clienteId"], "kind": "Hash" }

🍬 7. Cachés y colas: aliados del rendimiento

Redis → reduce carga de BD con resultados cacheados.
Azure Queue / Service Bus → desacopla procesos pesados.

Ejemplo con caching:

```
if (!_cache.TryGetValue("pedidos", out pedidos))
{
    pedidos = _db.Pedidos.ToList();
    _cache.Set("pedidos", pedidos, TimeSpan.FromMinutes(5));
}
```

🧱 8. Escalabilidad en microservicios

Cada microservicio puede escalar de forma independiente según su carga:

- Pedidos (CPU alta) → 5 instancias.
- Facturación (poco uso) → 2 instancias.

Usa **Kubernetes (AKS)** para control granular:

```
kubectl scale deployment pedidos-api --replicas=5
```

🧩 9. Patrones de escalabilidad .NET

Patrón	Descripción	Ejemplo
CQRS	Separar lectura y escritura	API con endpoints distintos
Event Sourcing	Estado derivado de eventos	Microservicios auditables
Circuit Breaker	Evita sobrecargar servicios caídos	Polly
Bulkhead	Aislar recursos para evitar propagación de fallos	Servicios independientes
Cache-aside	Leer de cache antes que de BD	Redis o MemoryCache

---

💡 10. Buenas prácticas del arquitecto

- ✅ Diseña tus apps como *stateless*.
- ✅ Escala horizontal antes que vertical.
- ✅ Usa colas y cachés para aliviar carga.
- ✅ Mide siempre antes de escalar.
- ✅ Automatiza reglas de autoescalado.
- ✅ Prueba la resiliencia bajo carga real.
- ✅ Asegura balanceo y replicación global.

✅ Objetivo

Diseñar arquitecturas .NET **escalables y elásticas**, capaces de crecer o reducirse dinámicamente según la demanda, sin sacrificar rendimiento ni disponibilidad.

Capítulo 38 – Caching Distribuido y Balanceadores

"El rendimiento no siempre viene de procesar más rápido, sino de repetir menos."

🧭 Introducción

El caching y el balanceo de carga son estrategias clave para garantizar que las aplicaciones .NET **respondan rápido, escalen sin esfuerzo y se mantengan disponibles** incluso bajo alta demanda.
Un arquitecto debe dominar cómo, cuándo y dónde cachear, y cómo distribuir el tráfico globalmente.

⚙️ 1. ¿Por qué usar caching?

- **Reduce latencia:** evita repetir operaciones costosas.
- **Ahorra recursos:** menos consultas a base de datos.
- **Incrementa disponibilidad:** respuesta rápida aunque haya picos o fallos temporales.
- **Escala mejor:** las peticiones se resuelven localmente o desde memoria compartida.

🧩 2. Tipos de caching

Tipo	Ubicación	Ejemplo	Ideal para
En memoria (MemoryCache)	Local en cada instancia	IMemoryCache	Apps pequeñas o únicas
Distribuido (Redis, SQL)	Compartido entre instancias	IDistributedCache	Apps escaladas o en clúster
Output Cache / Response Cache	En API Gateway o CDN	Azure Front Door, Cloudflare	Respuestas estáticas o GET
Edge Cache (CDN)	En nodos globales	Azure CDN	Archivos estáticos y contenido multimedia

---

⚙️ 3. Implementación de caching en .NET

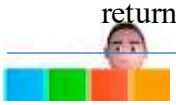
🔹 MemoryCache

```
services.AddMemoryCache();


public class PedidoService
{
    private readonly IMemoryCache _cache;
    private readonly AppDbContext _db;

    public PedidoService(IMemoryCache cache, AppDbContext db)
    {
        _cache = cache;
        _db = db;
    }

    public IEnumerable<Pedido> ObtenerPedidos()
    {
        return _cache.GetOrCreate("pedidos", entry =>
```



```
        {
            entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5);
            return _db.Pedidos.ToList();
        });
    }
}
```

➡️ Simple y rápido, pero no compartido entre instancias.

🔹 **Redis (Distributed Cache)**

Redis es el estándar en caching distribuido.
Azure Cache for Redis es totalmente gestionado, con alta disponibilidad y replicación.

Configuración

```
services.AddStackExchangeRedisCache(options =>
{
    options.Configuration =
"miredis.redis.cache.windows.net:6380,password=MiClaveSSL,ssl=True";
});
```

Uso

```
await _cache.SetStringAsync("totalPedidos", "120",
    new DistributedCacheEntryOptions { AbsoluteExpirationRelativeToNow =
TimeSpan.FromMinutes(10)  });


var total = await _cache.GetStringAsync("totalPedidos");
```

➡️ Ideal para microservicios, APIs escaladas y datos compartidos.

4. Estrategias de expiración

Estrategia	Descripción	Ejemplo
Absolute Expiration	Caduca tras cierto tiempo	5 minutos
Sliding Expiration	Se renueva con cada acceso	Sesiones activas
Cache-Aside	Solo cachea tras consultar base de datos	Patrón clásico
Write-Through	Actualiza cache al escribir en BD	Requiere sincronización
Invalidate on Update	Elimina entradas afectadas tras cambio	Evita datos obsoletos

---

5. Balanceadores de carga (Load Balancers)

El balanceo de carga distribuye el tráfico entre múltiples instancias o regiones.
Azure ofrece varios niveles de balanceo según la capa:

Nivel	Servicio Azure	Capa	Uso
Nivel 4 (TCP)	Azure Load Balancer	Red	VMs, contenedores
Nivel 7 (HTTP/HTTPS)	Application Gateway	Aplicación	APIs, Webs
Global	Azure Front Door	CDN + Gateway	Tráfico mundial

Ejemplo: Application Gateway

Balanceo inteligente con SSL, firewall y health checks:

```
az network application-gateway create \
  --name appGatewayPedidos \
  --resource-group  RG-App \
  --sku WAF_v2 \
  --frontend-port 443 \
  --http-settings-cookie-based-affinity Disabled
```

➡️ Azure se encarga de enrutar tráfico a instancias activas de forma transparente.

🔹 Health Checks

Los balanceadores realizan **pruebas de salud periódicas**:

- /health → debe devolver **HTTP 200**.
- Si falla → la instancia se retira del pool automáticamente.

```
app.MapGet("/health", () => Results.Ok("Healthy"));
```

🧩 6. CDN (Content Delivery Network)

Azure CDN replica contenido estático en nodos de todo el mundo:

- Reduce latencia global.
- Disminuye carga de backend.
- Mejora experiencia del usuario.

```
az cdn endpoint create --resource-group RG-App --profile-name cdnPedidos --origin pedidos-api.azurewebsites.net --name staticPedidos
```

➡️ Ideal para imágenes, vídeos o recursos front-end (React/Vue/Angular).

🔐 7. Seguridad y consistencia en caching

- Usa **SSL/TLS** en conexiones Redis.
- Define **políticas de expiración corta** para datos sensibles.
- Implementa **cache busting** tras operaciones CRUD.
- Configura **replicación activa** en Redis para alta disponibilidad.

⚙️ 8. Monitoreo y métricas

Supervisa:

- *Cache Hit Ratio* (ideal > 90%).
- Latencia de lectura/escritura.
- Fallos de conexión.
- Consumo de memoria en Redis.

Azure Monitor y Application Insights permiten crear alertas automáticas cuando bajan los aciertos de caché o sube la latencia.

💡 9. Buenas prácticas del arquitecto

- ✅ Usa Redis o MemoryCache según tu escala.
- ✅ No cachees datos confidenciales sin cifrado.
- ✅ Mantén tiempos de expiración realistas.
- ✅ Centraliza claves de caché con prefijos (user:123).
- ✅ Aplica balanceo global con Azure Front Door.
- ✅ Mide la eficiencia del caching antes y después.

✅ Objetivo

Implementar **caching distribuido y balanceadores inteligentes** que reduzcan latencia, alivien la base de datos y garanticen alta disponibilidad en arquitecturas .NET modernas.

Capítulo 39 – Pruebas de Carga y *Performance Tuning*

"No puedes mejorar lo que no mides." — Peter Drucker

🧭 Introducción

Un sistema puede funcionar bien con pocos usuarios, pero **colapsar ante picos de tráfico** si no fue diseñado y probado correctamente.
El arquitecto .NET debe asegurar que cada API, base de datos y microservicio pueda **mantener tiempos de respuesta predecibles**, incluso bajo estrés.

⚙️ 1. Tipos de pruebas de rendimiento

Tipo	Objetivo	Herramientas
Carga (Load Test)	Evaluar rendimiento bajo carga normal	k6, JMeter, Azure Load Testing
Estrés (Stress Test)	Detectar el punto de ruptura	JMeter, Locust
Soak (Endurance Test)	Probar estabilidad a largo plazo	Azure Load Testing
Spike Test	Medir reacción ante picos súbitos	k6, Gatling
Benchmark	Comparar configuraciones o versiones	BenchmarkDotNet

---

⚡ 2. Herramientas recomendadas

🔹 Azure Load Testing

Servicio nativo para simular miles de usuarios concurrentes desde Azure.

```
az load test create \
  --name testPedidos \
  --resource-group RG-Perf \
  --load-test-config "url=https://api.miapp.com/pedidos, duration=5m, users=200"
```

Muestra métricas de:

- Latencia promedio

- Errores HTTP
- Throughput (requests/segundo)
- CPU y memoria del backend

🔹 **k6 (open source, local o CI/CD)**

Archivo test.js:

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  vus: 50, // usuarios virtuales
  duration: '2m',
};

export default function () {
  let res = http.get('https://api.miapp.com/pedidos');
  check(res, { 'status 200': (r) => r.status === 200 });
  sleep(1);
}
```

Ejecución:

```
k6 run test.js
```

➡️ Ideal para integrarlo en GitHub Actions o Azure Pipelines.

🧩 3. Métricas clave que debes observar

Métrica	Descripción	Umbral recomendado
Latency (p95/p99)	Tiempo de respuesta más lento	< 500 ms
Throughput	Peticiones procesadas por segundo	Escalable según carga
CPU/Memory Usage	Consumo de recursos	< 75% sostenido
Error Rate	% de errores o timeouts	< 1%
DB Queries/sec	Actividad en BD	Monitorear índices y bloqueos

⚙️ 4. Diagnóstico con Application Insights

Azure Application Insights permite detectar cuellos de botella:

- *Request duration* por endpoint
- *Dependencia lenta* (SQL, HTTP, Redis)
- *Exception rate* y *failure traces*
- *Live Metrics Stream* en tiempo real

```
builder.Services.AddApplicationInsightsTelemetry();
```

Y desde el portal:
Performance → Slowest Operations → Investigate Bottleneck

🔧 5. *Performance Tuning* en .NET

🔹 **Optimiza consultas LINQ**

❌ Ineficiente:

```
var pedidos = _db.Pedidos.ToList().Where(p => p.Total > 1000);
```

✅ Correcto:

```
var pedidos = _db.Pedidos.Where(p => p.Total > 1000).ToList();
```

🔹 **Evita asignaciones innecesarias**

Usa tipos struct o Span<T> cuando sea posible en cálculos intensivos.

🔹 **Usa IAsyncEnumerable en streams largos**

```
await foreach (var pedido in _db.Pedidos.AsAsyncEnumerable())
    Procesar(pedido);
```

- **Controla la serialización**

Usa System.Text.Json (más rápido que Newtonsoft):

```
builder.Services.AddControllers()
    .AddJsonOptions(opt => opt.JsonSerializerOptions.PropertyNamingPolicy =
JsonNamingPolicy.CamelCase);
```

---

6. Ajustes del entorno

- **Pooling de conexiones SQL**: activa en ConnectionString
  Pooling=true; Max Pool Size=100;
- **Compression (Gzip/Brotli)**: reduce tamaño de respuesta.
- **Response Caching**: evita recalcular respuestas idénticas.
- **ThreadPool optimizado**: .NET lo ajusta automáticamente, pero puedes pre-calentar con tareas asíncronas iniciales.

---

7. Escenarios de tuning avanzados

- **Parallel.ForEachAsync** para tareas I/O intensivas.
- **BenchmarkDotNet** para medir código crítico:

```
[MemoryDiagnoser]
public class SumaBenchmark
{
    [Benchmark]
    public int LinqSum() => Enumerable.Range(1, 1000).Sum();
}
```

- **GC tuning** para servicios intensivos:

```
COMPlus_gcServer=1
COMPlus_gcConcurrent=1
```

🔍 8. Performance Budget

Define objetivos medibles:

- Tiempo máximo por endpoint.
- Consumo máximo de CPU/memoria.
- Límite de conexiones abiertas.

➡️ Cada release debe validarse contra este presupuesto antes del despliegue.

💡 9. Buenas prácticas del arquitecto

✅ Mide antes de optimizar (no asumas).
✅ Automatiza pruebas de carga en CI/CD.
✅ Usa Application Insights para telemetría.
✅ Evita optimizaciones prematuras.
✅ Aísla y prueba cuellos de botella.
✅ Documenta resultados y mejoras.

✅ Objetivo

Implementar **estrategias de medición y ajuste de rendimiento** en arquitecturas .NET, garantizando sistemas estables, rápidos y escalables incluso bajo alta carga.

Cerramos la **Parte VIII – Seguridad, Escalabilidad y Rendimiento** con un tema clave para la confiabilidad y cumplimiento normativo: **la auditoría y la trazabilidad**.
En este capítulo aprenderás cómo diseñar sistemas .NET que registren, rastreen y expliquen cada acción relevante, sin afectar el rendimiento ni la privacidad.

Capítulo 40 – Auditoría y Trazabilidad

"Un sistema sin trazabilidad es una caja negra; y las cajas negras no inspiran confianza."

🧭 Introducción

La trazabilidad no solo sirve para detectar errores, sino también para **demostrar qué ocurrió, cuándo, quién lo hizo y por qué**.
En entornos empresariales y regulados (finanzas, salud, industria pública), la auditoría es tan importante como la funcionalidad.
Un arquitecto .NET debe garantizar que cada acción crítica quede registrada de forma **segura, legible y verificable.**

⚙️ 1. Diferencia entre logging, trazabilidad y auditoría

Concepto	Propósito	Ejemplo
Logging	Registro técnico del sistema	"Error en conexión SQL."
Trazabilidad	Seguimiento de flujos y transacciones	"Pedido #345 creado → validado → pagado."
Auditoría	Evidencia formal de cambios o accesos	"Usuario admin eliminó cliente 102 el 10/05/2025 12:00."

➡️ Todo sistema robusto debería tener **los tres niveles activos**.

🧩 2. Diseño de una arquitectura trazable

1. **Interceptar acciones** → Middleware, filtros o eventos.
2. **Registrar contexto** → usuario, IP, endpoint, timestamp, entidad afectada.
3. **Almacenar logs** en formato estructurado (JSON).
4. **Centralizar en un sistema seguro** (App Insights, SQL, Elasticsearch).
5. **Correlacionar eventos** con IDs únicos (*Correlation ID*).

⚙️ 3. Implementación en .NET

🔹 Middleware de auditoría

```
public class AuditoriaMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<AuditoriaMiddleware> _logger;

    public AuditoriaMiddleware(RequestDelegate next, ILogger<AuditoriaMiddleware> logger)
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var usuario = context.User?.Identity?.Name ?? "Anónimo";
        var endpoint = context.Request.Path;
        var metodo = context.Request.Method;
        var ip = context.Connection.RemoteIpAddress?.ToString();
        var id = Guid.NewGuid();

        _logger.LogInformation("AUDIT [{id}] Usuario: {usuario}, Acción: {metodo} {endpoint},
IP: {ip}, Hora: {hora}",
            id, usuario, metodo, endpoint, ip, DateTime.UtcNow);

        context.Response.Headers.Add("X-Correlation-ID", id.ToString());
        await _next(context);
```

```
    }
}
```

Y en Program.cs:

```
app.UseMiddleware<AuditoriaMiddleware>();
```

➡️ Cada petición genera un ID único de auditoría visible en la respuesta.

🧠 4. Correlación entre servicios

En sistemas distribuidos, cada servicio debe **propagar el Correlation ID**:

```
client.DefaultRequestHeaders.Add("X-Correlation-ID", correlationId);
```

En Application Insights:

- Usa OperationId para agrupar logs por transacción.
- Puedes seguir un flujo completo entre microservicios y bases de datos.

🗄️ 5. Almacenamiento estructurado de logs

Evita los TXT planos; usa formatos analíticos:

- **JSON estructurado** (ideal para ElasticSearch o Application Insights).
- **Tablas SQL con columnas clave** (usuario, acción, fecha, entidad, resultado).
- **Azure Monitor / Log Analytics** para consultas Kusto (KQL).

Ejemplo KQL:

```
traces
| where message contains "AUDIT"
| project timestamp, customDimensions.Usuario, message
| order by timestamp desc
```

🔒 6. Seguridad en los registros

- **No registres contraseñas ni tokens.**
- Cifra información sensible (p.ej. correos, DNIs).
- Usa roles o permisos para acceder a los logs.
- Define tiempos de retención (p.ej. 6 meses).
- Almacena auditorías críticas en sistemas inmutables (Write-Once).

⚡ 7. Auditoría en base de datos

🔹 Entity Framework Interceptors

Intercepta operaciones SaveChanges:

```
public override int SaveChanges()
{
    var entradas = ChangeTracker.Entries()
        .Where(e => e.State == EntityState.Modified || e.State == EntityState.Added || e.State == EntityState.Deleted);

    foreach (var e in entradas)
    {
        _auditoria.Add(new LogAuditoria
        {
            Entidad = e.Entity.GetType().Name,
            Operacion = e.State.ToString(),
            Fecha = DateTime.UtcNow,
            Usuario = Thread.CurrentPrincipal.Identity?.Name
        });
    }

    return base.SaveChanges();
```

```
}
```

➡️ Cada cambio queda trazado con entidad, tipo de operación y usuario.

🧩 8. Auditoría y cumplimiento

Según el tipo de organización, puedes necesitar cumplir con normativas como:

- **GDPR / RGPD** (Europa): protección de datos personales.
- **ISO 27001**: seguridad de la información.
- **SOX / PCI-DSS / HIPAA**: sectores financieros y sanitarios.

👉 El arquitecto debe garantizar trazabilidad **sin vulnerar privacidad** (seudonimización y control de acceso).

🧱 9. Visualización y alertas

- Usa **Application Insights Dashboard** o **Grafana** para monitorear actividad.
- Configura alertas ante eventos sospechosos:
  - Eliminación masiva.
  - Accesos repetidos fallidos.
  - Cambios en roles o privilegios.
- Exporta reportes automáticos a Power BI o Excel.

💡 10. Buenas prácticas del arquitecto

✅ Define un modelo estándar de auditoría para todo el sistema.
✅ Usa Correlation ID en todas las peticiones y logs.
✅ Centraliza y protege los registros.
✅ Aplica retención controlada según legislación.
✅ No mezcles logs funcionales y de auditoría.
✅ Automatiza reportes y alertas.
✅ Prueba auditorías con escenarios reales.

✅ Objetivo

Diseñar sistemas .NET **trazables, auditables y conformes con estándares de seguridad y cumplimiento**, capaces de registrar con precisión cada acción relevante sin comprometer el rendimiento ni la confidencialidad.

Parte IX – Soft Skills del Arquitecto

Comenzamos la **Parte IX – Soft Skills del Arquitecto**, una sección fundamental para pasar de ser un *buen técnico* a un **líder técnico completo**.
Aquí veremos cómo dirigir equipos, compartir conocimiento, comunicar decisiones y mantener el rumbo técnico de una organización.

Capítulo 41 – Liderazgo Técnico y Mentoring

"Un arquitecto no solo diseña sistemas, también construye personas."

Introducción

El liderazgo técnico es la capacidad de **inspirar, guiar y elevar** el nivel de un equipo de desarrollo.
El arquitecto .NET no debe limitarse a decidir tecnologías, sino que debe **ayudar a otros a entenderlas, aplicarlas y mejorarlas**.
Ser líder técnico no se trata de tener todas las respuestas, sino de **crear un entorno donde todos las encuentren juntos.**

---

1. Qué es el liderazgo técnico

El liderazgo técnico combina tres dimensiones:

1. **Visión técnica:** entender el panorama y las tendencias.
2. **Comunicación efectiva:** traducir complejidad en claridad.
3. **Mentoría y crecimiento:** potenciar las habilidades del equipo.

El arquitecto actúa como un **puente entre la estrategia y la ejecución**, asegurando que el software avance alineado con los objetivos del negocio.

---

2. Roles del arquitecto líder

Rol	Descripción
Mentor	Guía a otros desarrolladores a crecer profesionalmente.
Facilitador	Elimina obstáculos técnicos y promueve autonomía.

Rol	Descripción
Visionario	Define la dirección tecnológica del producto.
Evangelista	Comunica buenas prácticas y cultura de calidad.
Mediador	Equilibra entre velocidad, deuda técnica y calidad.

🧠 3. Mentoring: enseñar con propósito

El mentoring técnico busca **transmitir conocimiento de manera estructurada**.
Algunos principios prácticos:

- **Modela con el ejemplo:** tus decisiones son la guía.
- **Crea oportunidades de aprendizaje:** revisiones de código, talleres internos, pair programming.
- **Ofrece feedback constructivo:** específico, empático y accionable.
- **Celebra los logros:** el reconocimiento fortalece la motivación.

👉 Un buen mentor forma futuros mentores.

⚙️ 4. Cómo liderar sin imponer

El liderazgo técnico se gana por **influencia**, no por jerarquía.

Técnicas efectivas:

- **Escucha activa:** primero entiende, luego responde.
- **Argumenta con datos:** métricas, benchmarks, pruebas.
- **Comparte la decisión:** involucra al equipo en elecciones técnicas.
- **Da autonomía:** confía, pero supervisa con propósito.

Ejemplo:

En lugar de "vamos a usar Redis porque lo digo yo",
di "Redis reduce nuestra latencia 40% según las pruebas, y se integra sin reescribir código."

🧩 5. Construcción de equipos técnicos sólidos

Un arquitecto debe fomentar equipos:

- **Multidisciplinarios:** backend, frontend, QA, DevOps, seguridad.
- **Colaborativos:** comunicación abierta y cultura de revisión.
- **Autónomos:** capaces de decidir dentro de su ámbito.
- **Aprendices constantes:** que compartan conocimiento.

📘 **Herramientas para cohesionarlos:**

- Reuniones técnicas semanales (Tech Talks).
- Revisión cruzada de código.
- Wiki o portal interno de arquitectura.
- Pair programming en tareas críticas.

🔁 6. Gestión del cambio técnico

Todo cambio genera resistencia.
El arquitecto debe comunicarlo como **una evolución, no una imposición**.

Pasos:

1. Explica el **por qué** (beneficio tangible).
2. Demuestra con **prototipos** o resultados previos.
3. Capacita al equipo antes de exigir el cambio.
4. Acompaña en la transición.

➡️ Liderar el cambio con empatía evita fricciones y aumenta la adopción.

🧩 7. Cultura de calidad y responsabilidad

Un líder técnico promueve:

- **Revisiones de código obligatorias.**
- **Pruebas automatizadas desde el inicio.**
- **Documentación clara y viva.**
- **Respeto por los estándares del equipo.**

La cultura técnica **no se impone, se demuestra**: la calidad se contagia.

🧠 8. Desarrollo personal del líder

El arquitecto también debe **seguir aprendiendo**:

- Participar en comunidades (.NET Conf, Azure Days, Dev Talks).
- Mantener certificaciones actualizadas.
- Leer y escribir documentación técnica.
- Desarrollar habilidades de coaching y comunicación.

➡️ Liderar también implica **humildad para seguir siendo aprendiz.**

💬 9. Frases guía del líder técnico

- "No es mi idea, es nuestra solución."
- "Tu error de hoy será tu aprendizaje de mañana."
- "El código se borra, pero las relaciones permanecen."
- "Ser arquitecto no significa saber más, sino ayudar mejor."

💡 10. Buenas prácticas del arquitecto-líder

✅ Escucha y comunica con empatía.
✅ Promueve mentoría cruzada en el equipo.
✅ Evita decisiones unilaterales.
✅ Celebra el progreso técnico.
✅ Defiende la calidad sin ser dogmático.
✅ Fomenta la documentación compartida.
✅ Sé ejemplo de profesionalismo y aprendizaje continuo.

✅ Objetivo

Desarrollar las habilidades humanas que convierten a un arquitecto .NET en un **líder técnico inspirador**, capaz de guiar equipos hacia la excelencia técnica, la colaboración y la mejora continua.

Continuamos con un aspecto que diferencia a un arquitecto profesional de un mero implementador: **la capacidad de documentar sus decisiones técnicas y el porqué detrás de ellas.**
En este capítulo aprenderás cómo usar **ADR (Architectural Decision Records)** y técnicas modernas de documentación para mantener una arquitectura viva, comprensible y trazable.

Capítulo 42 – Documentación y Decisiones Arquitectónicas (ADR)

"La documentación no es un recuerdo, es un mapa para el futuro."

🧭 Introducción

En sistemas complejos, la memoria humana no basta.
Los equipos cambian, las tecnologías evolucionan y los motivos detrás de ciertas decisiones se olvidan.
Los **Architectural Decision Records (ADR)** permiten capturar el razonamiento, las alternativas y los impactos de cada decisión relevante, creando una historia arquitectónica clara y auditable.

⚙️ 1. ¿Qué es un ADR?

Un **ADR (Architectural Decision Record)** es un documento breve que describe **una decisión técnica significativa** dentro de un proyecto.

Cada ADR responde cuatro preguntas:

1. ¿Qué problema resolvimos?
2. ¿Qué opciones evaluamos?
3. ¿Qué decisión tomamos?
4. ¿Por qué la elegimos?

📘 Ejemplo de ADR

ADR-005: Uso de Redis como caché distribuido

Contexto

Necesitamos mejorar la velocidad de lectura de datos frecuentes en la API.

Decisión

Implementar Azure Cache for Redis como capa de caching distribuido.

Alternativas consideradas

- MemoryCache (no compartido entre instancias)
- Redis local (no gestionado)
- Azure Cache for Redis ✅

Consecuencias

+ Reducción de latencia (~40%)
+ Mejora de escalabilidad
- Costo mensual adicional

➡️ Cada ADR se guarda en /docs/adr/ADR-005-redis-cache.md.

🧩 2. Por qué usar ADRs

Beneficio	Descripción
Trazabilidad	Puedes entender por qué una decisión fue tomada.
Transparencia	Todo el equipo conoce el razonamiento técnico.
Estandarización	Las decisiones se registran con formato uniforme.
Onboarding rápido	Los nuevos miembros entienden la arquitectura sin preguntar.
Prevención de errores repetidos	Evita rediscutir lo que ya fue evaluado.

⚙️ 3. Estructura recomendada

ADR-###: [Título descriptivo]

Contexto

[Problema o situación que motiva la decisión.]

Decisión

[Solución adoptada.]

Alternativas consideradas

[Resumen de opciones descartadas.]

Consecuencias

[Impactos positivos y negativos.]

Estado

[Propuesto / Aceptado / Reemplazado / Obsoleto]

Fecha

[DD/MM/AAAA]

👉 Mantén los ADR **cortos, claros y accionables**.

🧠 4. Cuándo registrar un ADR

Registra un ADR cuando:

- Se adopta una **nueva tecnología o framework.**
- Se cambia un patrón arquitectónico.
- Se definen políticas de **seguridad, autenticación o datos.**
- Se decide una integración clave (por ejemplo, Kafka vs RabbitMQ).
- Se modifica un componente crítico de rendimiento o escalabilidad.

No es necesario registrar pequeñas decisiones de implementación.

🧱 5. Cómo versionar y mantener ADRs

- Guarda cada ADR como archivo Markdown (.md) en un repositorio Git.
- Usa numeración secuencial (ADR-001, ADR-002, etc.).
- Marca los obsoletos con:
  ## Estado: Reemplazado por ADR-009
- Revisa ADRs cada 6-12 meses.
- Incluye referencias cruzadas entre decisiones relacionadas.

Ejemplo:

"Esta decisión reemplaza a ADR-003: Uso de SQL Server on-premises."

🧩 6. Herramientas útiles

- **adr-tools (CLI):** crea y administra ADRs fácilmente.
- adr new "Usar PostgreSQL en lugar de SQL Server"
- **Markdownlint:** verifica formato uniforme.
- **Docs-as-Code (con MkDocs o Docusaurus):** genera portales web de documentación técnica.
- **PlantUML o Mermaid:** añade diagramas embebidos.

⚡ 7. Documentación viva

La documentación técnica **no debe morir después del diseño**.
Usa un modelo *Docs-as-Code*:

- Documenta junto al código (mismo repositorio).
- Actualiza ADRs en cada *pull request*.
- Requiere revisión técnica igual que el código.

➡️ La arquitectura evoluciona con el software, no aparte de él.

🔍 8. Diagramas y vistas arquitectónicas

Acompaña tus ADRs con diagramas que expliquen el sistema:

- **C4 Model (Context, Container, Component, Code)**.
- **Secuencia (UML o Mermaid)** para flujos críticos.
- **Infraestructura (IaC)** para despliegues en Azure.

Ejemplo de C4 con Mermaid:

```
graph TD
  User --> API
  API --> Redis
  API --> SQL
```

💬 9. Comunicación efectiva de decisiones

Un ADR no sustituye el diálogo.
El arquitecto debe **explicar las decisiones al equipo**:

- En reuniones de arquitectura.
- Durante revisiones de código.
- A través de wikis o dashboards técnicos.

La claridad en la comunicación evita conflictos y refuerza el compromiso con la arquitectura.

💡 10. Buenas prácticas del arquitecto documental

- ✅ Documenta decisiones, no solo resultados.
- ✅ Usa un formato estándar y breve.
- ✅ Versiona ADRs junto al código.
- ✅ Manténlos actualizados y revisados.
- ✅ Acompaña cada ADR con un diagrama simple.
- ✅ Fomenta que todos los roles técnicos puedan proponer ADRs.
- ✅ Revisa la documentación en cada release.

✅ Objetivo

Adoptar una **cultura de documentación arquitectónica viva**, usando ADRs y modelos visuales que preserven el conocimiento técnico, respalden decisiones y faciliten la evolución controlada del sistema.

Entramos en una de las habilidades más decisivas del arquitecto moderno: **la comunicación efectiva**.
No basta con diseñar arquitecturas brillantes; hay que **transmitirlas con claridad**, adaptando el mensaje tanto a desarrolladores como a directivos.
Este capítulo te mostrará cómo hacerlo con estrategia, empatía y precisión.

Capítulo 43 – Comunicación con Equipos y *Stakeholders*

"El arquitecto es el traductor entre la tecnología y el negocio."

🧭 Introducción

El arquitecto .NET se mueve entre dos mundos:

- El **técnico**, donde se habla de APIs, patrones, microservicios y rendimiento.
- El **estratégico**, donde se discuten costos, tiempos y riesgos.

Tu papel es lograr que ambos lados **se entiendan y colaboren**, garantizando que la arquitectura técnica apoye los objetivos del negocio.

⚙️ 1. Los tres niveles de comunicación del arquitecto

Nivel	Audiencia	Enfoque	Ejemplo
Técnico	Desarrolladores, DevOps, QA	Profundidad y precisión	"Usaremos CQRS y RabbitMQ para desacoplar eventos."
Táctico	Líderes de equipo, PMs	Coordinación y impacto	"Esta decisión reduce el tiempo de despliegue en un 40%."
Estratégico	Dirección, negocio, clientes	Valor y riesgo	"La migración a Azure permitirá escalar sin costos adicionales de hardware."

➡️ Un arquitecto competente domina los tres registros y sabe cuándo usarlos.

🧩 2. Comunicación con equipos técnicos

La comunicación interna es el cimiento de un proyecto estable.
Tu función no es dar órdenes, sino **alinear y guiar.**

📘 Estrategias:

- Usa **diagramas C4 o UML** en lugar de solo palabras.
- Explica las **decisiones con contexto** ("por qué" > "cómo").
- Promueve revisiones técnicas colectivas.
- Usa canales claros (Teams, Slack, Confluence, GitHub Wiki).
- Evita jergas innecesarias y documenta acuerdos.

Ejemplo:

"Optamos por Azure Service Bus en lugar de Kafka por su mejor integración con nuestros entornos PaaS."

💡 3. Comunicación con *stakeholders* (negocio)

El lenguaje técnico no sirve con ejecutivos: hay que **traducir arquitectura en valor**.

🔹 Técnicas efectivas:

- **Habla en resultados:** rendimiento, ahorro, rapidez, seguridad.
- **Visualiza con indicadores:** dashboards, diagramas simples, ROI.
- **Sé conciso:** tres ideas por reunión son más efectivas que treinta.
- **Muestra progresos, no solo tareas:** versiones, métricas, impacto real.

Ejemplo:

"Implementar caching distribuido reducirá el tiempo de carga promedio de la app de 4 a 1 segundo, mejorando la satisfacción de usuario en un 30%."

🧠 4. Adaptar el mensaje según el perfil

Audiencia	Necesita saber	Evita
Desarrollador	Cómo implementar	Lenguaje de negocio
CTO / Gerente	Impacto y costo	Detalles de código

Audiencia	Necesita saber	Evita
Cliente final	Beneficio directo	Terminología técnica
QA / Seguridad	Riesgos y controles	Aspectos de UX

➡️ La empatía es clave: **no todos tienen tu mapa mental**.

🗂️ 5. Estructura de presentación de una arquitectura

Una comunicación clara sigue este esquema:

1. **Contexto** – Qué problema existe.
2. **Propuesta** – Qué solución técnica se plantea.
3. **Impacto** – Cómo mejora el sistema o negocio.
4. **Alternativas** – Qué se descartó y por qué.
5. **Riesgos y mitigación** – Cómo se controlarán.
6. **Plan de implementación** – Cronograma o fases.

📊 Ejemplo visual:

- Diagrama C4 (overview).
- Tabla de decisiones (resumen ADR).
- Indicadores de valor (rendimiento, disponibilidad, costos).

⚙️ 6. Comunicación durante crisis técnicas

Cuando algo falla, el arquitecto debe **mantener la calma y liderar la claridad**.

Pasos:

1. Explica el problema sin culpas ni tecnicismos excesivos.
2. Identifica impacto real y alcance.
3. Propón plan inmediato (rollback, fix, workaround).
4. Comunica avances periódicos hasta la resolución.
5. Documenta la causa raíz y aprendizajes (Postmortem).

➡️ La transparencia genera confianza incluso en momentos difíciles.

🧩 7. Herramientas para comunicar mejor

- **Miro / Draw.io / Whimsical:** diagramas colaborativos.
- **Confluence / Notion:** documentación viva.
- **Azure DevOps Wiki / GitHub Projects:** seguimiento técnico.
- **Power BI / Application Insights Dashboards:** visualización de métricas.
- **PlantUML / Mermaid:** documentación versionable junto al código.

🔁 8. Fomentar cultura de comunicación abierta

Un arquitecto debe crear espacios donde todos puedan opinar:

- Reuniones técnicas abiertas.
- Canales de discusión sin jerarquías.
- "Demo Days" para mostrar avances técnicos.
- Retroalimentación continua y constructiva.

➡️ Equipos que comunican bien **fallan menos y aprenden más rápido.**

🧩 9. Comunicación interdepartamental

La arquitectura impacta a más que al área técnica:

- **Seguridad:** validación de riesgos.
- **Infraestructura:** despliegues y redes.
- **Finanzas:** presupuestos cloud.
- **UX:** rendimiento y experiencia final.

👉 El arquitecto debe integrar todas las visiones sin perder coherencia técnica.

💡 10. Buenas prácticas del arquitecto comunicador

- ✅ Traduce lo técnico en valor.
- ✅ Usa métricas y visuales, no párrafos.
- ✅ Escucha más de lo que hablas.
- ✅ Ajusta el lenguaje según la audiencia.
- ✅ Evita tecnicismos cuando no aportan claridad.
- ✅ Sé transparente ante problemas y decisiones.
- ✅ Documenta acuerdos en un solo repositorio.

✅ Objetivo

Desarrollar la habilidad de **comunicar arquitectura como un puente entre tecnología y negocio**, adaptando el mensaje a cada audiencia y fortaleciendo la colaboración, la confianza y la toma de decisiones técnicas alineadas.

Cerramos la **Parte IX – Soft Skills del Arquitecto** con un tema que define la madurez técnica de cualquier sistema: **la gestión de la deuda técnica y la evolución sostenible del producto**.
Un arquitecto que ignora la deuda técnica compromete el futuro del proyecto; uno que la gestiona con inteligencia, garantiza su longevidad.

Capítulo 44 – Gestión de Deuda Técnica y Evolución del Producto

"Toda decisión técnica es una deuda: si no la pagas con tiempo, la pagarás con dolor."

🧭 Introducción

La **deuda técnica** surge cuando se prioriza la velocidad sobre la calidad: código duplicado, falta de pruebas, arquitectura rígida o dependencias obsoletas.
No siempre es mala: a veces es necesaria para cumplir plazos.
El problema es **no reconocerla ni planificar su pago**.

El arquitecto .NET debe equilibrar tres fuerzas:

- **Tiempo** (entregas rápidas)
- **Calidad** (código mantenible)
- **Costo** (recursos y riesgos)

⚙️ 1. Qué es la deuda técnica

Analogía con las finanzas:

"Tomar una decisión rápida hoy es como pedir un préstamo; obtienes resultados inmediatos, pero pagarás intereses en mantenimiento, errores o refactorización."

Ejemplo:

- Copiar código en lugar de crear un método reutilizable.
- No implementar pruebas unitarias por falta de tiempo.
- Aplazar actualizaciones de .NET o librerías.

➡️ Cada una acumula "intereses" con el tiempo.

🧩 2. Tipos de deuda técnica

Tipo	Descripción	Ejemplo
De diseño	Estructura deficiente del sistema	Lógica de negocio en controladores
De código	Falta de estándares o duplicación	Funciones demasiado largas
De pruebas	Cobertura insuficiente	Sin tests automáticos
De dependencias	Librerías o frameworks desactualizados	.NET 5 sin soporte
De documentación	Falta de claridad o ADRs	Decisiones sin registrar
De infraestructura	Despliegues manuales o inseguros	Sin CI/CD ni contenedores

---

⚙️ 3. Identificación y visibilidad

La deuda técnica debe **hacerse visible y medible**.
Herramientas útiles:

- **SonarQube / SonarCloud** → analiza complejidad, duplicación y vulnerabilidades.
- **Application Insights + Logs** → detecta cuellos de rendimiento.
- **Azure Boards o Jira** → crear tickets etiquetados como "deuda técnica".
- **ADR y revisiones de código** → registrar compromisos técnicos.

Ejemplo de ticket:

[DEUDA] Refactorizar módulo de autenticación.

Impacto: Seguridad y mantenimiento.

Prioridad: Alta.

Estimado: 2 días.

🧱 4. Priorización de la deuda

No toda deuda debe pagarse de inmediato.
Usa una matriz de impacto:

Impacto / Urgencia	Alta	Media	Baja
Alta	Abordar ya	Planificar en sprint próximo	Documentar
Media	Planificar	Evaluar con métricas	Aplazar
Baja	Monitorear	Ignorar temporalmente	Documentar

➡️ Prioriza según riesgo al negocio y al rendimiento.

🧩 5. Integrar la deuda en la planificación ágil

Cada sprint debería incluir **tiempo dedicado a deuda técnica** (10–20%).

- Define tareas de refactorización como *User Stories*.
- Documenta mejoras esperadas (rendimiento, legibilidad, estabilidad).
- Usa *Definition of Done* para exigir calidad:
  "Sin warnings, con pruebas unitarias, documentación actualizada."

📘 Ejemplo:

"Durante este sprint optimizaremos consultas EF Core para reducir el tiempo de carga de 2,5s a 1s."

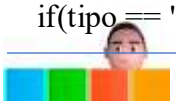
🔥 6. Refactorización continua

El mejor momento para reducir deuda técnica es **mientras trabajas en el código**.
Principios:

- Pequeños pasos, sin romper funcionalidad.
- Apoyarse en pruebas automatizadas.
- Mantener cobertura constante.
- Revisar impacto de rendimiento.

Ejemplo:

```
// Antes
if(tipo == "admin") { /* ... */ } else if(tipo == "user") { /* ... */ }
```



```
// Después
RolStrategyFactory.Obtener(tipo).Ejecutar();
```

🔁 7. Evolución del producto y versionado

El producto debe evolucionar sin perder estabilidad.
Estrategias:

- **SemVer (Versionado Semántico):**
  MAJOR.MINOR.PATCH → 2.1.3
  - Cambios mayores → ruptura.
  - Menores → nuevas funciones.
  - Parches → correcciones.
- **Feature Flags:**
  Activar nuevas funciones gradualmente.
- **Blue-Green / Canary Deployments:**
  Probar versiones nuevas sin afectar producción.

🧠 8. Indicadores de salud técnica

Monitorea periódicamente:

- % de cobertura de pruebas.
- Número de bugs repetidos.
- Tiempo medio de despliegue (MTTD).
- Complejidad ciclomática promedio.
- Tasa de deuda técnica (SonarQube metric).

👉 Estos datos ayudan a justificar inversión en refactorización.

🔒 9. Cultura de responsabilidad técnica

El arquitecto debe promover:

- Revisión de código obligatoria.
- Documentación de deuda y compromisos.
- Sesiones retrospectivas técnicas.
- Alineación con negocio: "invertir en calidad hoy evita fallos mañana."

➡️ La deuda técnica no es solo problema de TI, sino del producto completo.

💡 10. Buenas prácticas del arquitecto

✅ Visibiliza la deuda y cuantifícala.
✅ Prioriza según impacto, no por gusto técnico.
✅ Asigna tiempo fijo para mantenimiento.
✅ Fomenta refactorización continua.
✅ Usa métricas objetivas.
✅ Alinea deuda técnica con valor de negocio.
✅ Mantén la arquitectura flexible y evolutiva.

✅ Objetivo

Gestionar la deuda técnica como parte natural del ciclo de vida del software, equilibrando innovación y estabilidad, y garantizando que los sistemas .NET evolucionen de forma sostenible, predecible y controlada.

🧠 Parte X – Casos Prácticos

Llegamos a la **Parte X – Casos Prácticos**, donde aplicaremos todo lo aprendido en ejemplos reales de arquitectura .NET.
Comenzamos con uno de los proyectos empresariales más representativos: el diseño de un **ERP modular en .NET**, una arquitectura que integra contabilidad, inventario, clientes, ventas y más, bajo principios modernos de desacoplamiento y escalabilidad.

Capítulo 45 – Arquitectura de un ERP Modular en .NET

"La modularidad es la columna vertebral de toda arquitectura sostenible."

🧭 Introducción

Un **ERP (Enterprise Resource Planning)** centraliza los procesos clave de una empresa.
Diseñarlo en .NET requiere pensar en:

- Separación clara por dominios funcionales.
- Escalabilidad y mantenibilidad a largo plazo.
- Comunicación entre módulos con bajo acoplamiento.
- Integración con servicios internos y externos.

⚙️ 1. Objetivos del diseño

- **Modularidad:** cada área (Ventas, Inventario, Finanzas) debe ser independiente.
- **Extensibilidad:** permitir añadir módulos sin reescribir los existentes.
- **Reutilización:** servicios compartidos (usuarios, auditoría, logging).
- **Escalabilidad:** soportar múltiples instancias y ubicaciones.
- **Seguridad:** autenticación centralizada (Identity + JWT).

🧩 2. Estructura general del ERP

```
ERP.NET/
├── Core/              → Entidades, interfaces, utilidades comunes
├── Modules/
│   ├── Inventario/
│   ├── Ventas/
│   ├── Finanzas/
│   ├── RRHH/
│   └── CRM/
├── Infrastructure/    → Persistencia (EF Core, repositorios)
├── API/               → Controladores REST, endpoints públicos
├── AuthService/       → IdentityServer / JWT
├── Gateway/           → API Gateway y balanceo
└── Shared/            → Logging, configuración, eventos comunes
```

Cada módulo es **autónomo**, con sus propias entidades y servicios, pero comparte el *core* común.

⚙️ 3. Patrones arquitectónicos aplicados

Patrón	Función
Clean Architecture	Separación entre dominio, aplicación e infraestructura.
CQRS	Lecturas y escrituras separadas para escalabilidad.
Mediator Pattern	Comunicación desacoplada entre casos de uso.
Repository + Unit of Work	Abstracción del acceso a datos.
Event Bus (RabbitMQ o Azure Service Bus)	Comunicación entre módulos.

📘 4. Ejemplo: módulo de Inventario

Dominio:

```
public class Producto
{
    public Guid Id { get; set; }
    public string Nombre { get; set; }
    public int Stock { get; set; }
    public decimal Precio { get; set; }
}
```

Caso de uso (Application Layer):

```
public record ActualizarStockCommand(Guid ProductoId, int Cantidad) : IRequest<bool>;

public class ActualizarStockHandler : IRequestHandler<ActualizarStockCommand, bool>
{
    private readonly IProductoRepository _repo;
    public async Task<bool> Handle(ActualizarStockCommand request, CancellationToken ct)
    {
        var producto = await _repo.GetByIdAsync(request.ProductoId);
        producto.Stock += request.Cantidad;
        await _repo.SaveChangesAsync();
        return true;
    }
}
```

Infraestructura (EF Core):

```
public class ProductoRepository : IProductoRepository
{
    private readonly InventarioDbContext _ctx;
    public Task<Producto> GetByIdAsync(Guid id) => _ctx.Productos.FindAsync(id).AsTask();
```

```
}
```

🔐 5. Autenticación y seguridad centralizada

Usa **IdentityServer o Azure AD B2C**:

- Autenticación con JWT.
- Autorización basada en roles y claims.
- Tokens reutilizables entre módulos.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.Authority = "https://login.miempresa.com";
        options.Audience = "erp_api";
    });
```

⚡ 6. Comunicación entre módulos

- **Sincrónica (HTTP / REST):** ideal para consultas rápidas.
- **Asíncrona (mensajería):** para eventos de negocio, usando *Event Bus*.

Ejemplo:

Cuando se registra una venta, el módulo "Ventas" emite un evento ProductoVendido, que el módulo "Inventario" consume para descontar stock.

```
await _eventBus.PublishAsync(new ProductoVendidoEvent(productoId, cantidad));
```

🧠 7. Integración con otros sistemas

- Exportación a Excel, Power BI o APIs de contabilidad externa.
- Webhooks para sincronización con CRM.
- APIs abiertas (Swagger + OpenAPI).

📊 8. Observabilidad y auditoría

- **Application Insights:** tiempos de respuesta y errores.
- **Logging estructurado:** con Serilog + ElasticSearch.
- **Auditoría por módulo:** quién, cuándo y qué modificó.

🧩 9. Despliegue y CI/CD

- Cada módulo se despliega como **servicio independiente** (Docker).
- Azure DevOps o GitHub Actions para *build/test/deploy*.
- Pipeline único con plantillas YAML por módulo.

```
jobs:
- job: build_inventario
  steps:
   - task: DotNetCoreCLI@2
     inputs:
       command: 'build'
       projects: 'Modules/Inventario/*.csproj'
```

💡 10. Buenas prácticas del arquitecto ERP

✅ Diseña módulos con límites de dominio claros.
✅ Implementa un bus de eventos común.
✅ Centraliza autenticación y auditoría.
✅ Evita dependencias directas entre módulos.
✅ Usa interfaces y contratos compartidos.
✅ Versiona APIs por módulo.
✅ Aplica métricas y monitoreo continuo.

✅ Objetivo

Diseñar una **arquitectura ERP modular en .NET** escalable, mantenible y extensible, capaz de integrar múltiples dominios empresariales mediante patrones modernos, servicios distribuidos y una comunicación fluida y segura.

Pasamos al caso práctico estrella en cloud: construir una **plataforma SaaS multi-tenant en Azure con microservicios** y CI/CD. Aquí unimos arquitectura, seguridad, costos y operación continua.

Capítulo 46 – Plataforma SaaS en Azure con Microservicios

"Un buen SaaS no solo escala en usuarios; escala en inquilinos, equipos y cambios."

🧭 Visión general

Objetivo: entregar una plataforma **multi-tenant** (varios clientes/"inquilinos") con **aislamiento de datos**, **escala elástica** y **operación automatizada**.

Pilares:

- **Tenancy**: identificación del inquilino y aislamiento de datos.
- **Arquitectura**: microservicios desacoplados, API Gateway, eventos.
- **Cloud nativo**: App Service / AKS, Storage, Service Bus, Key Vault.
- **Seguridad**: Entra ID (Azure AD), OAuth2/OIDC, RBAC y secretos gestionados.
- **Operación**: CI/CD, observabilidad, facturación/uso, control de costos.

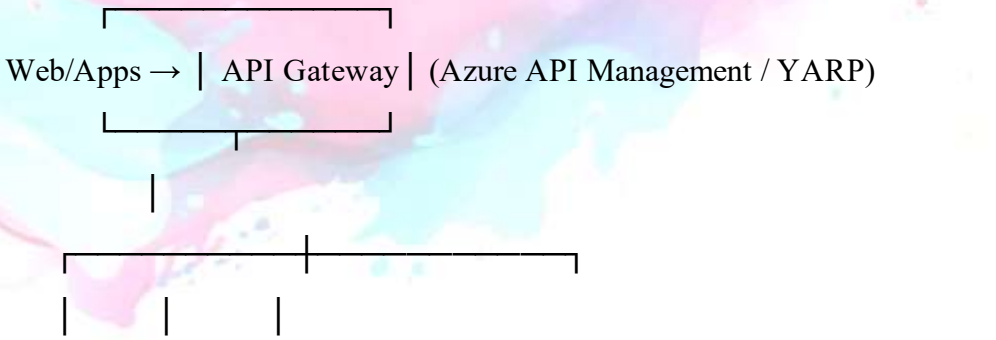
---

1) Modelo de *multi-tenancy* (elige según tu SLA/costo)

Modelo	Aislamiento	Coste	Complejidad	Cuándo usar
Shared DB, shared schema	Bajo	Bajo	Bajo	SaaS con SLAs moderados y muchos tenants pequeños
Shared DB, schema por tenant	Medio	Medio	Medio	Requisitos de partición/backup por tenant
DB por tenant (Azure SQL/Cosmos)	Alto	Alto	Alto	Tenants grandes, fuerte compliance o ruido entre vecinos
Workload por tenant (servicios dedicados)	Máximo	Máximo	Alto	Aislamiento estricto/sector regulado

Para comenzar: **DB compartida + partición por TenantId** y migrar *gradualmente* a *schema/DB per tenant* para clientes premium.

2) Topología de referencia (Azure)



- **Key Vault**: secretos, certificados, claves de cifrado.
- **App Config**: *feature flags* y config por entorno/tenant.
- **Front Door / CDN**: latencia global, WAF y TLS.
- **Application Insights + Log Analytics**: observabilidad y costos.

---

3) Resolución de inquilino (*tenant resolution*)

Tres enfoques comunes:

1. **Subdominio**: tenantA.midominio.com
2. **Path/claim**: api.midominio.com/t/tenantA/...
3. **Claim en el token** (tid, tenant_id)

Middleware de resolución:

```
public class TenantContext
{
   public string TenantId { get; init; } = default!;
}
```

```
public class TenantResolverMiddleware
{
   private readonly RequestDelegate _next;
   public TenantResolverMiddleware(RequestDelegate next) => _next = next;

   public async Task InvokeAsync(HttpContext ctx)
   {
      var tenant = ctx.User.FindFirst("tenant_id")?.Value
            ?? ctx.Request.Headers["X-Tenant"].FirstOrDefault()
            ?? ctx.Request.Host.Host.Split('.').First(); // subdominio
      if (string.IsNullOrWhiteSpace(tenant)) { ctx.Response.StatusCode = 400; return; }

      ctx.Items[nameof(TenantContext)] = new TenantContext { TenantId =
tenant.ToLowerInvariant() };
      await _next(ctx);
   }
}
app.UseMiddleware<TenantResolverMiddleware>();
```

---

4) Aislamiento de datos con EF Core

Filtro global por tenant:

```
public class AppDbContext : DbContext
{
   private readonly TenantContext _tenant;
   public AppDbContext(DbContextOptions<AppDbContext> opt, IHttpContextAccessor acc) :
base(opt)
      => _tenant = (TenantContext)acc.HttpContext!.Items[nameof(TenantContext)]!;
```

```
    protected override void OnModelCreating(ModelBuilder mb)
    {
        mb.Entity<Pedido>().HasQueryFilter(p => p.TenantId == _tenant.TenantId);
    }

    public override Task<int> SaveChangesAsync(CancellationToken  ct = default)
    {
        foreach (var e in ChangeTracker.Entries<ITenantEntity>().Where(e  => e.State ==
EntityState.Added))
            e.Entity.TenantId = _tenant.TenantId;
        return base.SaveChangesAsync(ct);
    }
}
```

Conexión por tenant (DB per tenant):

```
public class TenantDbFactory : IDbContextFactory<AppDbContext>
{
    private readonly IConfiguration _cfg; private readonly IHttpContextAccessor _acc;
    public TenantDbFactory(IConfiguration c, IHttpContextAccessor a){ _cfg=c; _acc=a; }

    public AppDbContext CreateDbContext()
    {
        var tenant = ((TenantContext)_acc.HttpContext!.Items[nameof(TenantContext)]!).TenantId;
        var cs = _cfg.GetConnectionString($"Sql_{tenant}") ??
_cfg.GetConnectionString("Sql_Default");
        var opt = new DbContextOptionsBuilder<AppDbContext>().UseSqlServer(cs).Options;
        return new AppDbContext(opt, _acc);
    }
}
```

En **Azure SQL** puedes usar **Elastic Pools** para múltiples DBs y controlar costos.

5) Seguridad y control de acceso

- **Autenticación**: Azure Entra ID / B2C (OIDC).
- **Autorización**: *scopes* y *roles* por tenant (claims role, tenant_id).
- **mTLS/Managed Identity** entre microservicios.
- **Rate limiting** por tenant en API Gateway.
- **Key Vault**: secretos/CMK; **Private Endpoints** para bases de datos.

---

6) Comunicación y eventos

- **HTTP/gRPC** para lecturas y operaciones rápidas.
- **Service Bus / Event Grid** para **eventos multi-tenant** (p.ej. PedidoCreado).
- **Outbox pattern** para confiabilidad entre DB y cola.
- **Dead-Letter Queues** para mensajes problemáticos por tenant.

---

7) Escalabilidad y performance

- **Stateless** + **Redis** para sesiones/caché per-tenant: claves tenant:{id}:…
- **Autoescalado** por métricas (CPU/cola/latencia).
- **Partition keys**: Cosmos DB "/tenantId".
- **CDN/Front Door** para contenido estático y caché de respuestas GET.

---

8) Medición de uso y facturación (*metering & billing*)

- Registra **unidades de consumo**: llamadas API, GB almacenados, jobs ejecutados.
- Métricas por tenant_id en **App Insights/Log Analytics**.
- Tabla de *usage* agregada por día/tenant/servicio.
- Planes **Básico/Pro/Enterprise** activados con **feature flags** (Azure App Configuration).
- Genera informes y **webhooks** de uso para el sistema de cobro.

---

9) CI/CD multi-entorno y *blue-green*

- Pipelines YAML (Azure DevOps / GitHub Actions).
- **Infra as Code** con **Bicep/Terraform** (App Service/AKS, SQL/Cosmos, Service Bus, Key Vault, App Config).
- *Slots* **staging→production** + **pruebas de humo** + *swap*.
- **Canary** por porcentaje o por lista de tenants (header/feature flag).

---

10) Observabilidad por tenant

- **CorrelationId** + tenant_id en cada log/traza.
- Tableros (Workbooks / Grafana) con: latencia p95, tasa de errores, throughput por tenant.
- Alertas específicas (ej.: *error rate* > 1% para tenant=enterpriseX).
- **SLO/SLI** por plan (p. ej., Enterprise con SLO 99.9%).

---

11) Cost control y *FinOps*

- **Tags** en recursos (env=prod, svc=catalog, tenantClass=premium).
- **Budgets** y alertas de costo en suscripciones.
- Uso de **Elastic Pools**, **autoscale** y **reservas** donde convenga.
- Automatiza *scale-down* nocturno para entornos no productivos.

---

12) Checklist de *readiness* SaaS

- Resolución de tenant (dominio/claim/header).
- Aislamiento de datos (filtro global / DB por tenant).
- Auth OIDC + roles/scopes por tenant.
- API Gateway (límite y cuotas por plan).
- Observabilidad con tenant_id.
- CI/CD con *blue-green* y *feature flags*.
- Facturación/uso y planes comerciales.
- DR/Backups por tenant crítico.

Ejemplo breve: cuota por tenant en API

```
builder.Services.AddRateLimiter(opt =>
{
   opt.AddPolicy("per-tenant", http =>
   {
      var tenant = ((TenantContext)http.Items[nameof(TenantContext)]!).TenantId;
      return RateLimitPartition.GetFixedWindowLimiter(tenant, _ => new
FixedWindowRateLimiterOptions
      {
         Window = TimeSpan.FromMinutes(1),
         PermitLimit = tenant.StartsWith("pro") ? 1200 : 300,
         QueueLimit = 50
      });
   });
});
app.UseRateLimiter();
```

Buenas prácticas clave

- Diseña **primero** para multi-tenant (no lo “añadas luego”).
- Mantén **contratos estables** y versionados.
- Aísla **hard** a clientes regulados/premium.
- No mezcles secretos en código; usa **Managed Identity + Key Vault**.
- Todo **medido y trazado** por tenant_id.
- Automatiza **todo**: IaC, CI/CD, pruebas de humo, *rollbacks*.

✅ **Objetivo**

Diseñar y operar una **plataforma SaaS multi-tenant en Azure** con microservicios .NET, asegurando **aislamiento, seguridad, escalabilidad, observabilidad y control de costos** desde el día uno.

Cerramos el borde externo de la plataforma SaaS: **cómo entrar y con qué permisos**.
En este capítulo diseñamos un **API Gateway** con **autenticación centralizada** para unificar entrada, políticas, cuotas y tokens en todo el ecosistema.

Capítulo 47 – API Gateway y Autenticación Centralizada

"Un buen gateway convierte un enjambre de servicios en una sola puerta, segura y predecible."

🧭 Introducción

En microservicios, exponer cada servicio directamente aumenta complejidad, latencia y riesgos.
El **API Gateway** centraliza **enrutamiento, seguridad, cuotas, versionado y observabilidad**.
Con **autenticación centralizada** (OIDC/OAuth2) emitimos y validamos **tokens** coherentes para todo el sistema.

⚙️ 1) Opciones de API Gateway en .NET/Azure

Opción	Cuándo usar	Ventajas	Consideraciones
Azure API Management (APIM)	SaaS/Enterprise, multientorno, monetización	Políticas declarativas, developer portal, cuotas, versionado	Coste por uso, IaC recomendado
YARP (Yet Another Reverse Proxy)	Auto-hosted, control total, bajo costo	.NET nativo, muy flexible, rápido	Debes construir portal, cuotas y analítica
NGINX/Envoy	Multi-stack, alta performance	Ecosistema maduro, mTLS	Operación e integración por tu cuenta

Recomendación: **APIM** para SaaS en Azure (rápido, seguro, con portal). **YARP** para control fino o entornos híbridos.

🧩 2) Enrutamiento y agregación (APIM + YARP)

APIM – política básica de *forwarding*

```
<policies>
  <inbound>
    <base />
    <set-backend-service base-url="https://pedidos-api.internal"/>
    <rewrite-uri template="/api/v1/pedidos/{id}" />
  </inbound>
  <backend><base /></backend>
  <outbound><base /></outbound>
</policies>
```

YARP – appsettings.json

```
{
  "ReverseProxy": {
    "Routes": {
      "pedidos": {
        "ClusterId": "pedidosCluster",
        "Match": { "Path": "/pedidos/{**catch-all}" }
      }
    },
    "Clusters": {
      "pedidosCluster": {
        "Destinations": { "d1": { "Address": "https://pedidos-api.internal/" } }
      }
    }
  }
}
```

```
builder.Services.AddReverseProxy().LoadFromConfig(builder.Configuration.GetSection("ReverseProxy"));

app.MapReverseProxy();
```

➡️ El gateway puede **agregar** respuestas (BFF), reducir *chattiness* y estabilizar contratos externos.

🔐 3) Autenticación centralizada (OIDC/OAuth2)

Flujo típico

1. Cliente → Gateway (sin token)
2. Gateway redirige a **IdP** (Entra ID / B2C / Auth0 / IdentityServer)
3. Usuario se autentica → IdP emite **ID Token** + **Access Token (JWT)**
4. Cliente envía Authorization: Bearer <token> al Gateway
5. Gateway valida y **inyecta claims** al backend

Scopes/roles definen permisos de acceso; **claims** (p. ej. tenant_id, role) definen contexto.

🔐 4) Validación de tokens en el Gateway

APIM – política de validación JWT

```
<validate-jwt header-name="Authorization" failed-validation-httpcode="401" require-scheme="Bearer">
  <openid-config url="https://login.microsoftonline.com/{tenant}/v2.0/.well-known/openid-configuration" />
  <required-audiences>
    <audience>erp_api</audience>
  </required-audiences>
  <required-claims>
    <claim name="scp">
      <value>pedidos.read</value>
    </claim>
  </required-claims>
```

</validate-jwt>

YARP + Duende/Identity (validación en backend)

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
 .AddJwtBearer(o =>
 {
   o.Authority = "https://login.contoso.com";
   o.Audience  = "erp_api";
   o.TokenValidationParameters.RoleClaimType  = "role";
 });
app.UseAuthentication(); app.UseAuthorization();
```

Centraliza en el gateway siempre que sea posible; los servicios internos pueden **confiar en encabezados** ya validados (pero valida de nuevo en servicios críticos).

🧠 5) Multitenancy en el borde

- **Resolución de tenant** por **subdominio**, **ruta** o **claim** en el token.
- APIM/YARP agrega encabezado X-Tenant al backend:

```
<set-header name="X-Tenant" exists-action="override">
  <value>@(Jwt.GetClaim("tenant_id"))</value>
</set-header>
```

➡️ El backend aplica **filtro global** por TenantId.

📏 6) Políticas: rate limiting, cuotas y seguridad

APIM – rate limit por suscripción/tenant

```
<rate-limit calls="300" renewal-period="60" />
<quota calls="10000" renewal-period="86400" />
```

APIM – sanitizar y proteger

```
<set-header name="X-Frame-Options" exists-action="override"><value>DENY</value></set-header>
<ip-filter action="allow">
```

```
  <address-range from="10.0.0.0" to="10.255.255.255" />
</ip-filter>
<check-header name="Content-Type" failed-check-httpcode="415">
  <value>application/json</value>
</check-header>
```

YARP – *rate limiting* en .NET 8

```
builder.Services.AddRateLimiter(o =>
  o.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext, string>(ctx =>
    RateLimitPartition.GetFixedWindowLimiter(
      key: ctx.Request.Headers["X-Tenant"].ToString(),
      factory: _ => new FixedWindowRateLimiterOptions { Window =
TimeSpan.FromMinutes(1),  PermitLimit = 300 })));
app.UseRateLimiter();
```

---

7) Versionado y *blue/green* en el Gateway

- Expón /v1, /v2 en rutas/policies (APIM **versions & revisions**).
- Publica **revisiones** y haz **swap** gradual (canary) por porcentaje o por **producto/tenant**.
- Mantén **contratos estables**; depreca con fecha y plan claro.

---

8) API Keys vs OAuth2

- **API Keys**: útiles para integraciones *machine-to-machine* simples o *webhooks*.
- **OAuth2 (client credentials)**: preferido para M2M con **scopes** y expiración.
- Mezcla ambos: **API Key** para *throttling* + **JWT** para autorización granular.

🧰 9) Portal de desarrolladores

Si expones APIs a terceros:

- **APIM Developer Portal**: suscripción, *try it*, documentación automatizada (OpenAPI), planes y cuotas.
- Versiona especificaciones **OpenAPI** y genera SDKs.
- Pruebas de *sandbox* por tenant/plan.

🛡️ 10) mTLS y seguridad interna

- **mTLS** entre Gateway ↔ Servicios críticos (cert cliente).
- **Private Link / VNet Integration** para cerrar accesos públicos.
- **WAF (APIM/Front Door)** para OWASP Top 10.
- **Key Vault** para certificados rotados automáticamente.

📊 11) Observabilidad desde el borde

- **Correlation-ID** generado en Gateway y propagado:
  X-Correlation-ID → logs, trazas y métricas de backend.
- Métricas clave: **p95/p99**, **rate limit hits**, **5xx por ruta**, **errores de validación JWT**.
- Exporta a **Application Insights / Log Analytics** y crea alertas.

✅ Checklist de Arquitecto

- Gateway elegido (APIM/YARP) con IaC.
- Validación OIDC/JWT y *claims* obligatorios.
- Resolución y propagación de **tenant**.
- Rate limit / cuotas por plan.
- Versionado y revisiones controladas.
- mTLS/Private Endpoints para tráfico interno.
- Portal dev (si APIs públicas).
- Telemetría y alertas en el borde.

💡 Buenas prácticas

- ✅ Autenticación/autorización **en el borde**; defensa en profundidad en servicios.
- ✅ Mantén políticas **declarativas** (APIM) y versionadas en Git.
- ✅ No expongas servicios internos; usa red privada + Gateway.
- ✅ Tokens **cortos** + *refresh tokens* donde aplique.
- ✅ *Feature flags* para rutas/políticas por plan/tenant.
- ✅ Test de contrato (Pact, Dredd) en CI/CD del Gateway.

🎯 Objetivo

Diseñar un **frente único y seguro** con API Gateway + autenticación centralizada, aplicando **políticas, cuotas, multitenancy y observabilidad** para exponer microservicios de forma profesional.

Cerramos los casos prácticos instrumentando la plataforma **end-to-end** con métricas, trazas y paneles accionables usando **Application Insights** (AI) y, cuando aplique, **OpenTelemetry** para estandarizar la instrumentación.

Capítulo 48 – Monitoreo y Logging Distribuido con Application Insights

"Operar sin telemetría es conducir de noche con las luces apagadas."

🧭 Objetivo del capítulo

Diseñar una observabilidad **unificada** para APIs y microservicios .NET (en App Service/AKS), con:

- **Logs estructurados**, **métricas** (p. ej., latencia p95/p99), **trazas distribuidas** (correlation/operation id).
- **Dashboards** de salud, **alertas** proactivas y **workbooks** por tenant/servicio.
- Integración con **OpenTelemetry** para portabilidad.

---

1) Habilitar Application Insights en .NET

Instalación & wire-up

```
dotnet add package Microsoft.ApplicationInsights.AspNetCore
```

```
// Program.cs
builder.Services.AddApplicationInsightsTelemetry(
    builder.Configuration["ApplicationInsights:ConnectionString"]);
```

Consejo: activa el muestreo adaptable (reduce coste, mantiene representatividad):

```
// appsettings.json
{
  "ApplicationInsights": {
    "ConnectionString": "<tu-connection-string>",
    "EnableAdaptiveSampling": true
  }
}
```

2) Trazas distribuidas (requests, dependencias, SQL, HTTP)

AI captura automáticamente:

- **Requests** (endpoints).
- **Dependencies** (SQL, HTTP, Redis, Service Bus).
- **Exceptions** (stack trace).

Agrega **propiedades personalizadas** (tenant, userId, featureFlag) para filtrar:

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;

public class TelemetryEnricher : ITelemetryInitializer
{
    private readonly IHttpContextAccessor _acc;
    public TelemetryEnricher(IHttpContextAccessor acc) => _acc = acc;

    public void Initialize(Microsoft.ApplicationInsights.Channel.ITelemetry t)
    {
        if (_acc.HttpContext is null) return;
        var tenant = _acc.HttpContext.Items["TenantContext"] as TenantContext;
        t.Context.GlobalProperties["tenant_id"] = tenant?.TenantId ?? "unknown";
        t.Context.GlobalProperties["env"] =
Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")!;
    }
}
// Program.cs
builder.Services.AddSingleton<ITelemetryInitializer, TelemetryEnricher>();
```

Propaga **Correlation-ID** en el borde (API Gateway) y en llamadas internas (HttpClient):

```
builder.Services.AddHttpClient("svc")
    .AddHttpMessageHandler(() => new CorrelationHandler());
```

```
public class CorrelationHandler : DelegatingHandler
{
   protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage req,
CancellationToken ct)
   {
      var id = Activity.Current?.TraceId.ToString() ?? Guid.NewGuid().ToString();
      req.Headers.TryAddWithoutValidation("X-Correlation-ID", id);
      return base.SendAsync(req, ct);
   }
}
```

---

3) Logs estructurados con Serilog + AI

```
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Sinks.ApplicationInsights
Log.Logger = new LoggerConfiguration()
  .Enrich.FromLogContext()
  .WriteTo.ApplicationInsights(
     services.GetRequiredService<TelemetryConfiguration>(),
     TelemetryConverter.Traces)
  .CreateLogger();

builder.Host.UseSerilog();
Uso:
_logger.LogInformation("AUDIT: {Action} on {Entity} by {User} (tenant {Tenant})",
    "Delete", "Pedido", userId, tenantId);
```

4) OpenTelemetry + AI (opcional, recomendado)

Estandariza la instrumentación y exporta a AI (u otros backends):

```
dotnet add package OpenTelemetry.Extensions.Hosting
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.Http
dotnet add package OpenTelemetry.Instrumentation.SqlClient
dotnet add package OpenTelemetry.Exporter.AzureMonitor
builder.Services.AddOpenTelemetry()
  .WithTracing(tp => tp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddSqlClientInstrumentation(o => o.SetDbStatementForText = true)
    .AddAzureMonitorTraceExporter())
  .WithMetrics(mp => mp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddRuntimeInstrumentation()
    .AddAzureMonitorMetricExporter());
```

Ventaja: si cambias de backend (Jaeger/OTLP/Grafana), no reescribes tu app.

5) Consultas Kusto (KQL) útiles

Requests lentos (p95) por ruta:

```
requests
| summarize p95_duration = percentiles(duration, 95) by name
| order by p95_duration desc
```

Errores por servicio y tenant:

```
requests
| where success == false
| summarize errores=count() by cloud_RoleName, tostring(customDimensions.tenant_id)
| order by errores desc
```

Dependencias que fallan (SQL/HTTP):

```
dependencies
| where success == false
| summarize fails=count(), p95=percentiles(duration,95) by target, type, name
| order by fails desc
```

Trazabilidad por Correlation-ID:

```
let cid = "abcd-1234..."; // X-Correlation-ID o operation_Id
union requests, dependencies, traces, exceptions
| where operation_Id == cid or tostring(customDimensions["X-Correlation-ID"]) == cid
| order by timestamp asc
```

Uso por tenant (SaaS metering):

```
requests
| summarize calls=count(), p95=percentiles(duration,95) by
tostring(customDimensions.tenant_id), bin(timestamp, 1h)
| order by timestamp desc
```

6) Dashboards y Workbooks (lo esencial)

Crea un **Workbook** con estas secciones:

1. **Overview**
   - Availability, TPS, errores (4xx/5xx), p95/p99.
2. **Top endpoints**
   - Latencia y tasa de errores por ruta.
3. **Dependencias**
   - SQL/HTTP fallidas y más lentas.
4. **SaaS**
   - Métricas por tenant_id (consumo, latencia, errores).
5. **Infra (AKS/App Service)**
   - CPU/Memoria/Pod restarts (vía Container insights).
6. **Liberaciones**
   - Versión desplegada (custom property release en cada request).

---

7) Alertas proactivas

Configura reglas (Metric/Log Alerts):

- **Latencia p95 > 800ms** en requests/duration durante 5 min.
- **Error rate > 1%** en producción.
- **DeadLetter en colas** (Service Bus) > 0.
- **CPU > 80%** sostenido 10 min (App Service/AKS).
- **Incremento de 5xx** por endpoint/tenant.

Acciones → Teams/Email/Webhook/ITSM y **auto-scale** si aplica.

Ejemplo de alerta (consulta KQL):

```
requests
| where timestamp > ago(5m)
| summarize errorRate = 100.0 * countif(success==false) / count()
| where errorRate > 1
```

---

8) *Release markers* y *feature flags*

Anota la versión en cada request para correlacionar fallos con despliegues:

```
app.Use(async (ctx, next) =>
{
    Activity.Current?.SetTag("release", builder.Configuration["Release:Version"]);
    await next();
});
```

Con **Azure App Configuration + Feature Flags**, envía el flag activo como propiedad para evaluar impacto.

---

9) Buenas prácticas de observabilidad

- **Nombra** cloud_RoleName por microservicio (envíalo en AppSettings).
- **No loguees PII** ni secretos (GDPR).
- Usa **muestreo** (adaptive) en alto tráfico.
- Propaga **operation_Id / traceId** en todo el flujo.
- Separa **logs de auditoría** de logs técnicos si el volumen es alto.
- Define **SLO/SLI** (p. ej., disponibilidad 99.9%, p95 < 400ms) y alerta por brechas.
- Automatiza recursos AI por IaC y **workbooks versionados en Git**.

---

10) Checklist de cierre

- AI habilitado en todos los servicios (prod/stg/dev).
- Correlation y tenant_id enriquecidos en telemetría.
- Dashboards/Workbooks por servicio y globales.
- Alertas de latencia, error rate, dependencias y colas.
- OpenTelemetry para trazas/métricas portables.
- Cost control: muestreo y retención adecuada.
- Postmortems con KQL guardados como *queries* reutilizables.

🎯 Resultado

Una plataforma .NET **observable de extremo a extremo**, con **AI + OTel** para ver lo que sucede, **resolver antes de que impacte** y **aprender de cada incidente**.

🏁 Epílogo – La evolución hacia arquitecturas inteligentes

"El arquitecto del futuro no diseña solo software, diseña ecosistemas que aprenden."

Durante décadas, el rol del arquitecto .NET se centró en construir aplicaciones estables, seguras y mantenibles.
Hoy, su misión ha evolucionado: debe crear **sistemas que se adaptan**, **aprenden** y **dialogan con la nube**.

La arquitectura ya no termina en el diagrama; se extiende hasta el usuario, el dato y la decisión. Cada servicio que diseñamos, cada pipeline que automatizamos, y cada métrica que medimos forma parte de un **organismo digital** que debe ser resiliente, ético y humano.

En el camino de este libro, pasamos de entender el *framework* a comprender la *visión*:

- De escribir código, a **diseñar contextos de colaboración**.
- De optimizar consultas, a **optimizar experiencias**.
- De pensar en servidores, a **pensar en ecosistemas conectados**.

El arquitecto moderno combina **razón técnica** con **intuición estratégica**.
Comprende que el código es un medio, no un fin; y que la tecnología solo tiene sentido si **sirve al propósito del negocio y de las personas**.

En un mundo donde la inteligencia artificial se integra en cada capa —desde la infraestructura hasta el análisis predictivo—, la arquitectura se vuelve **cognitiva**.
Los sistemas .NET del mañana no solo responderán a eventos: **anticiparán necesidades**.

💡 Consejos para seguir aprendiendo

- **Actualízate constantemente:** .NET, Azure, DevOps y AI evolucionan cada trimestre.
- **Estudia patrones modernos:** Event Sourcing, DDD, CQRS, Clean y Serverless.
- **Aprende de la comunidad:** GitHub, blogs técnicos, conferencias y proyectos open source.
- **Experimenta con IA:** integra modelos GPT o Copilot en flujos de desarrollo.
- **Escribe y enseña:** la mejor forma de entender arquitectura es explicarla.

🌐 Recursos recomendados

- Microsoft Learn – Arquitectura Cloud y .NET
- Azure Architecture Center
- The .NET Foundation
- OpenTelemetry.io
- Awesome .NET Architecture Repos

🧭 Palabras finales

Ser arquitecto .NET no es dominar un lenguaje, sino **dominar la capacidad de conectar piezas**: personas, procesos y tecnología.
Tu código construye hoy lo que otros mantendrán mañana.
Diseña con claridad, comunica con empatía y decide con propósito.

Porque el mejor arquitecto no solo crea software que funciona, sino **software que trasciende**.

🏆 Tu siguiente paso

Este libro ha sido un mapa.
Ahora tú eres el explorador.
Lleva tus ideas a la práctica, diseña tu propio estilo de arquitectura y comparte lo que aprendas.
Porque el conocimiento que no se comparte **no construye legado**, y el mejor arquitecto es aquel que deja sistemas —y personas— mejores que cuando empezó.

@oscardelacuesta
Palentino.es — "De Desarrollador a Arquitecto .NET"