BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 0

# Empieza aquí: cómo funciona el software

Este capítulo es la puerta de entrada al curso. Da por hecho que has usado aplicaciones y sitios web, pero que nunca has programado, y explica desde cero qué es el software, cómo representa y procesa los datos, cómo se organiza el código, cómo se comunican las computadoras a través de las redes, dónde se almacenan los datos, cómo colaboran sistemas independientes mediante API, qué son los servidores y la nube, y cómo los equipos construyen y lanzan software. Los ingenieros con experiencia pueden hojearlo como un vocabulario compartido; todos los demás deberían leerlo antes del Capítulo 1, porque cada capítulo posterior se apoya en estas ideas.

bewiseapps.com/es/software-architecture-wise/foundations/ · Versión 2026-10-26. Solo información educativa, no asesoramiento profesional.

**DE UN VISTAZO**

**Aprenderás a**

- Explicar qué ocurre, paso a paso, cuando se ejecuta un programa, desde el código fuente hasta el procesador, pasando por el sistema operativo.
- Distinguir las estructuras de datos habituales y usar la notación O grande (Big-O) para predecir cuánto se ralentiza una operación a medida que crecen los datos.
- Describir cómo las funciones, los módulos, los objetos y las interfaces dividen el código, y la diferencia entre una biblioteca y un framework.
- Seguir lo que ocurre cuando alguien escribe una dirección web, nombrando DNS, TCP, TLS y HTTP y lo que aporta cada uno.
- Comparar los archivos, las bases de datos relacionales y las principales familias NoSQL, y explicar qué garantiza una transacción.
- Explicar qué es una API, cómo funciona una llamada REST sobre HTTP y cuándo un mensaje asíncrono encaja mejor que una llamada síncrona.
- Distinguir los servidores físicos, las máquinas virtuales, los contenedores y los principales modelos de servicio en la nube.
- Seguir una sola acción de un usuario a través de todas las capas de un sistema real, y reconocer las decisiones que forman su arquitectura.

**Antes de empezar**

Ninguno. Basta con estar familiarizado con el uso de aplicaciones y sitios web.

**Tiempo**

Unas 3 horas de lectura; 45 minutos para la prueba.

**Términos clave**

Proceso · Compilador e intérprete · Estructura de datos · Notación O grande (Big-O) · Interfaz · Framework · Dependencia · Dirección IP y puerto · DNS · HTTP y HTTPS · Latencia · Base de datos y transacción · API · Contenedor y nube

## Por qué empezar aquí

La arquitectura de software trata de las grandes decisiones de un sistema de software: cuáles son sus partes principales, cómo se comunican entre sí, dónde residen los datos y qué cualidades, como la velocidad, la fiabilidad y la seguridad, debe lograr el diseño. Esas decisiones se toman con un vocabulario propio: procesos e hilos, índices y cachés, interfaces y dependencias, HTTP y DNS, transacciones, API, contenedores y pipelines. Los arquitectos usan estas palabras con precisión, y buena parte del curso depende de saber qué significa realmente cada una y qué cuesta.

Este capítulo aporta ese vocabulario. Está escrito para alguien que nunca ha programado, y cada sección comienza con una breve analogía cotidiana antes de explicar el mecanismo real. También pretende ser lo bastante preciso como para que un ingeniero con experiencia que lo lea esté de acuerdo con cada afirmación. Cuando un tema merece mucha más profundidad, el capítulo lo presenta y remite a la sección posterior que lo trata a fondo.

## Un programa en una computadora

Las tres primeras secciones se quedan dentro de una sola máquina. Qué es el software explica la diferencia entre el hardware y el software, cómo el código fuente escrito por personas se convierte en instrucciones que un procesador puede ejecutar (mediante compiladores, intérpretes y máquinas virtuales), qué hace el sistema operativo y por qué los datos en memoria desaparecen al reiniciar mientras que los datos en disco se conservan. Termina siguiendo un programa en Python de nueve líneas a lo largo de cada paso de su vida.

Datos y algoritmos muestra cómo las computadoras representan los números y el texto, y presenta las estructuras de datos de uso diario: listas, diccionarios, conjuntos, colas, pilas, árboles y grafos. Con la notación O grande (Big-O), explica por qué un diseño para encontrar a un cliente sigue siendo rápido con un millón de clientes mientras que otro se hunde, y por qué existen los índices y las cachés. Organizar el código sube después de las funciones a los módulos, los objetos y las interfaces, explica la diferencia entre una biblioteca y un framework, advierte del costo oculto de las dependencias y ofrece un primer vistazo al acoplamiento y la cohesión, las dos ideas que recorren todo el diseño de software.

## Muchas computadoras trabajando juntas

Las tres secciones siguientes salen de la máquina única. Cómo se comunican las computadoras explica los paquetes, las direcciones IP y los puertos, TCP y UDP, cómo el DNS convierte nombres en direcciones y cómo funcionan las solicitudes y respuestas HTTP, protegidas con TLS. También señala algo que sorprende a mucha gente: la latencia tiene un límite inferior fijado por la velocidad de la luz, así que un viaje de ida y vuelta entre Londres y Sídney nunca puede ser rápido, por mucho ancho de banda que compres.

Almacenar datos trata de los archivos y las bases de datos: tablas, claves y consultas SQL, índices, transacciones y las garantías ACID (ilustradas con una transferencia bancaria que nunca debe quedarse a medias), y las principales familias NoSQL y cuándo encaja cada una. API e integración explica cómo cooperan sistemas independientes: una API como contrato, REST y JSON sobre HTTP, claves y tokens, webhooks, la diferencia entre esperar una respuesta y enviar un mensaje a través de una cola, y por qué los cambios incompatibles en una API perjudican a todos los que dependen de ella.

## Dónde se ejecuta el software y cómo se construye

Servidores y la nube muestra qué es realmente un servidor, cómo las máquinas virtuales y los contenedores permiten que muchas cargas de trabajo compartan una sola máquina física, y qué venden los proveedores de nube: desde alquilar máquinas sin más hasta ejecutar tu código solo cuando llega un evento. Presenta las regiones, las zonas de disponibilidad, las redes de distribución de contenidos y las dos formas de afrontar el crecimiento: una máquina más grande o más máquinas. Cómo se construye el software pasa de la tecnología a la práctica: requisitos, control de versiones con Git, la pirámide de pruebas, entornos, integración y entrega continuas, formas ágiles de trabajar y DevOps. Termina siguiendo un toque en "Hacer pedido" en una aplicación de reparto de comida a través de cada parte del capítulo, desde el teléfono hasta el DNS, HTTPS, un balanceador de carga, servicios, una transacción de base de datos, un proveedor de pagos, una cola y una notificación push.

## La idea que debes llevarte

Cada sistema de este capítulo está construido con partes que ocultan sus detalles tras un límite acordado: el sistema operativo tras las llamadas al sistema, un módulo tras su interfaz, un servidor web tras HTTP, una base de datos tras SQL, un proveedor de pagos tras su API, una plataforma en la nube tras sus servicios. Los límites son lo que permite construir sistemas grandes sin entender cada detalle, y es en ellos donde están los compromisos importantes: una llamada a función es rápida y fiable, mientras que una llamada a través de la red es más lenta y puede fallar; una dependencia ahorra esfuerzo pero introduce riesgo; una caché ahorra tiempo pero puede servir datos obsoletos.

Decidir dónde van esos límites, qué los cruza y qué compromisos aceptar es la arquitectura de software. Cada sección termina con una breve nota sobre por qué su tema importa para la arquitectura, y el Capítulo 1 comienza exactamente donde termina este.

SECCIÓN 0.1 DE 0.8 · PRINCIPIANTE · 15 MIN DE LECTURA

# Programas, código y cómo los ejecuta una computadora

**LA IDEA EN BREVE**

Una computadora es como la cocina de un restaurante. El chef (el procesador) solo puede seguir recetas escritas, y las sigue al pie de la letra, palabra por palabra. La encimera (la memoria) contiene lo que el chef está usando en este momento y se despeja cada noche; la despensa (el almacenamiento) guarda las cosas de forma permanente. El software es el conjunto de recetas. Las personas escriben las recetas en un idioma que pueden leer, y un traductor las convierte al idioma propio del chef, que es muy sencillo. El jefe de cocina (el sistema operativo) decide qué receta recibe la atención del chef y evita que los puestos se estorben entre sí.

**LO QUE NECESITAS SABER**

- El hardware es la máquina física (CPU, RAM, almacenamiento, tarjeta de red); el software es el conjunto de instrucciones y datos que le dicen qué hacer.
- Un programa es una lista de instrucciones precisas; la computadora hace exactamente lo que dice, así que los errores en las instrucciones se convierten en bugs.
- Los programadores escriben código fuente en lenguajes de alto nivel, que los compiladores, los intérpretes o las máquinas virtuales de bytecode convierten en código máquina que la CPU puede ejecutar.
- El sistema operativo reparte la CPU, la memoria, los archivos y la red entre los programas y aplica la seguridad; los programas le piden servicios mediante llamadas al sistema.
- Un programa en ejecución es un proceso con su propia memoria; los hilos dentro de un proceso comparten esa memoria, lo cual es útil pero puede causar bugs de concurrencia.
- La RAM es rápida pero pierde su contenido al reiniciar; todo lo que no deba perderse tiene que escribirse en un almacenamiento duradero, normalmente una base de datos.

## Hardware y software

Una computadora es una máquina que sigue instrucciones muy deprisa. Las partes físicas que podrías tocar son el hardware; las instrucciones que les dicen a esas partes qué hacer son el software. Una computadora portátil, un teléfono, el servidor que ejecuta el sitio web de tu banco y el chip dentro de un termostato inteligente son todos computadoras construidas con los mismos cuatro tipos de componentes.

La CPU (unidad central de procesamiento, o procesador) ejecuta las instrucciones. Un procesador moderno tiene varios *núcleos*, cada uno capaz de procesar su propio flujo de instrucciones, y cada núcleo ejecuta miles de millones de pasos sencillos por segundo. La RAM (memoria de acceso aleatorio, normalmente llamada simplemente *memoria*) contiene las instrucciones y los datos con los que la CPU está trabajando en este momento. Es rápida, pero volátil: si se corta la corriente, su contenido desaparece. El **almacenamiento** (una unidad de estado sólido o un disco duro giratorio) conserva los archivos cuando no hay corriente, pero acceder a él es mucho más lento. Una **tarjeta de red** (por cable o Wi-Fi) envía y recibe datos hacia y desde otras computadoras.

Piensa en la cocina de un restaurante. El chef es la CPU: el único que realmente cocina. La encimera es la RAM: todo lo que el chef está usando en este momento está dispuesto allí, al alcance de la mano, pero se despeja al final de la noche. La despensa y el congelador son el almacenamiento: grandes, permanentes y a un paseo de distancia. La ventanilla que da al comedor es la tarjeta de red, por la que llegan los pedidos y salen los platos. Las recetas son el software.

**Las cuatro partes que tiene toda computadora**

| Parte | Qué hace | Analogía de la cocina | Qué ocurre si es el cuello de botella |
|---|---|---|---|
| **CPU (procesador)** | Ejecuta instrucciones: aritmética, comparaciones, movimiento de datos | El chef | El trabajo se acumula; las páginas y los informes se vuelven lentos aunque sobre memoria |
| **RAM (memoria)** | Contiene los programas en ejecución y los datos que están usando | La encimera | El sistema empieza a trasladar datos al disco y se ralentiza bruscamente, o se detienen programas |
| **Almacenamiento (SSD o disco)** | Guarda archivos y bases de datos de forma permanente | La despensa y el congelador | Guardar y cargar datos se vuelve lento; un disco lleno puede detener una base de datos |
| **Tarjeta de red** | Envía y recibe datos a otras máquinas | La ventanilla que da al comedor | Las solicitudes esperan para enviarse o recibirse; los usuarios notan retrasos |

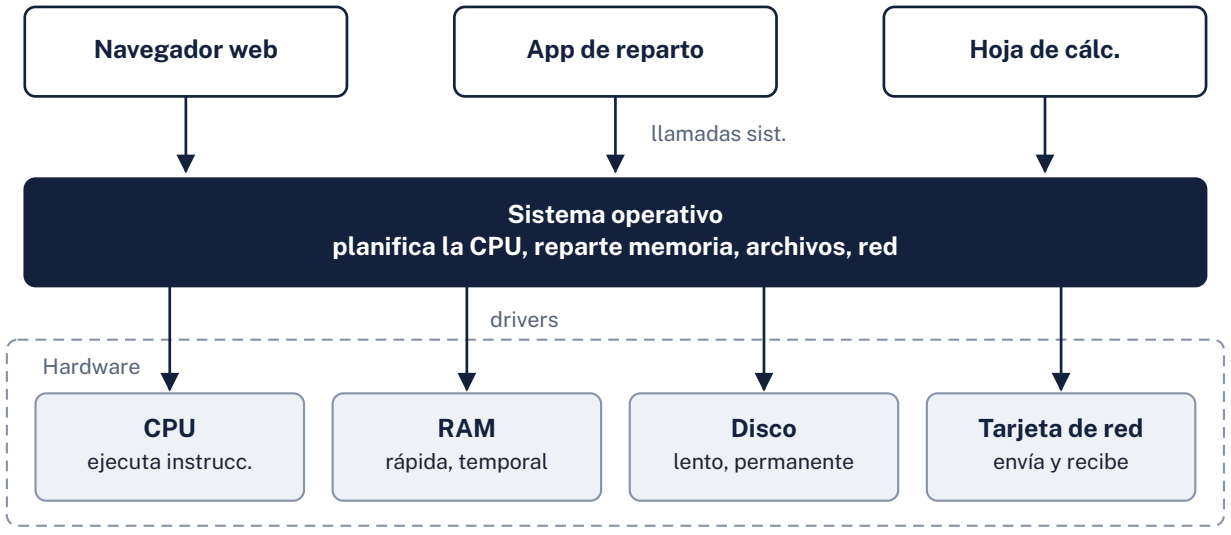
## Qué es un programa

Un programa es una lista de instrucciones precisas que una computadora sigue para hacer un trabajo: calcular una factura, mostrar una página web, enviar un pago. La palabra que importa es *precisas*. Un cocinero humano puede arreglárselas con "sal al gusto"; una computadora no. Hace exactamente lo que se le dice, exactamente en el orden en que se le dice, y nada más. Si una instrucción dice que se divida entre un número que resulta ser cero, la computadora no se encoge de hombros y sigue adelante: el programa se detiene con un error, a menos que el programador haya escrito más instrucciones que digan qué hacer en ese caso.

En el nivel más bajo, la CPU repite un ciclo mientras está encendida: obtener la siguiente instrucción de la memoria, decodificar lo que significa, ejecutarla y pasar a la siguiente. Cada instrucción es diminuta: sumar dos números, comparar dos valores, copiar un valor de un lugar de la memoria a otro, saltar a otra instrucción si una comparación resultó verdadera. Todo lo que hace una computadora, desde reproducir una película hasta entrenar un modelo de IA, se construye con miles de millones de estos pequeños pasos.

**Software** es el término más amplio: los programas junto con los datos, la configuración y la documentación que necesitan. Una *aplicación* (o app) es software que hace algo útil para una persona, como una aplicación bancaria. El *software de sistema*, como el sistema operativo que se describe más abajo, existe principalmente para ejecutar otro software.

**Capas: las aplicaciones, el sistema operativo y el hardware**




Las aplicaciones nunca manejan el hardware directamente: se lo piden al sistema operativo, que reparte el hardware entre ellas de forma justa y segura.

## El código fuente y los lenguajes de programación

Las CPU solo entienden código máquina: instrucciones codificadas como números, específicas de una familia de procesadores. Los chips de Intel y AMD de la mayoría de las computadoras portátiles y servidores usan una familia (x86-64); los chips de los teléfonos, de los Mac recientes y de muchos servidores en la nube usan otra (Arm). El código máquina que funciona en una no funciona en la otra. Escribir código máquina a mano es lento, propenso a errores y casi nunca se hace hoy en día.

En su lugar, las personas escriben código fuente en un **lenguaje de programación**: una notación definida con precisión que se lee en parte como el inglés y en parte como las matemáticas. Python, JavaScript, Java, C#, Go, Rust y C son ejemplos muy usados. Se llaman lenguajes de *alto nivel* porque permiten a un programador decir "suma los precios de este carrito" en una o dos líneas en lugar de decenas de instrucciones máquina, y porque ocultan los detalles de la CPU concreta. El código fuente es texto plano, guardado en archivos, y es lo que los programadores leen, escriben, revisan y guardan en el control de versiones (un sistema que conserva el historial de cada cambio, explicado en la última sección de este capítulo).

Los lenguajes se diferencian en lo que facilitan. Python se valora por su legibilidad y domina el trabajo de ciencia de datos e IA; JavaScript es el lenguaje que ejecutan todos los navegadores web; Java y C# son habituales en los grandes sistemas empresariales; Go es popular para los servicios de red; C y Rust se usan donde cada microsegundo y cada byte importan, como en los sistemas operativos y las bases de datos. Ningún lenguaje es el mejor en todo, y elegir uno es en parte una decisión técnica y en parte una cuestión de qué habilidades tiene el equipo.

## Del código fuente a un programa en ejecución: compiladores, intérpretes y máquinas virtuales

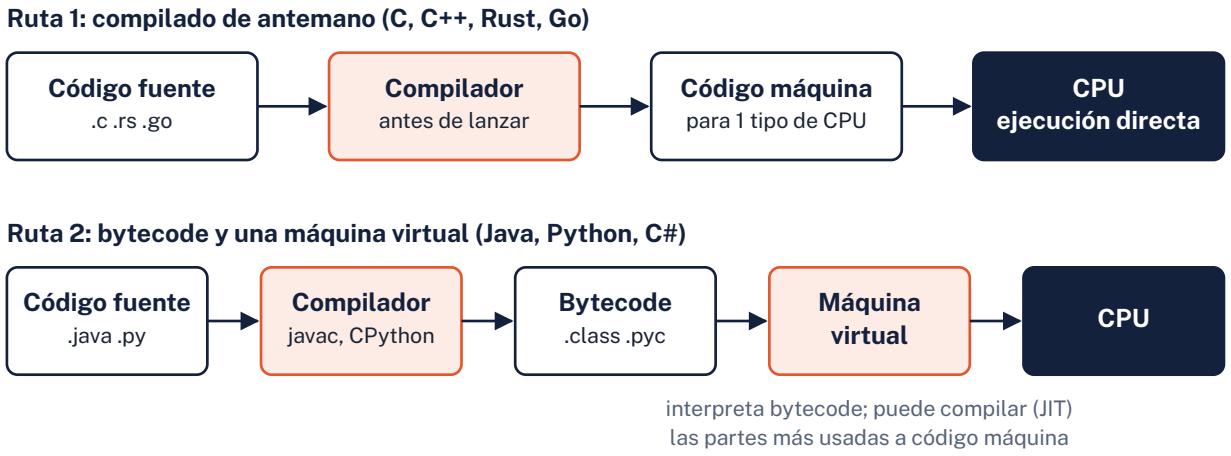
El código fuente tiene que convertirse en algo que la CPU pueda ejecutar. Hay tres enfoques generales, y los lenguajes reales a menudo los combinan.

Un compilador es un programa que traduce todo el código fuente de un programa a otra forma antes de que se ejecute. Los lenguajes como C, C++, Rust y Go suelen compilarse *de antemano* a código máquina para una CPU y un sistema operativo concretos. El resultado es un archivo *ejecutable* que arranca rápido y funciona deprisa, pero hay que compilar una versión distinta para cada tipo de máquina que quieras admitir, y el compilador rechaza el programa si encuentra ciertos tipos de error.

Un intérprete es un programa que lee el código fuente (o algo muy parecido) y va ejecutando sus instrucciones sobre la marcha, paso a paso. La interpretación facilita probar cosas y ejecutar el mismo programa en cualquier máquina que tenga el intérprete, a costa de la velocidad.

Muchos lenguajes populares se sitúan en un punto intermedio. El código fuente de Java lo compila una herramienta llamada `javac` a bytecode: instrucciones compactas para una computadora imaginaria e idealizada, en lugar de para una CPU real. La **Máquina Virtual de Java** (JVM) ejecuta después ese bytecode en cualquier máquina real en la que esté instalada, y así es como un mismo programa Java funciona sin cambios en Windows, macOS y Linux. La JVM observa qué partes del programa se ejecutan con más frecuencia y las compila a código máquina mientras el programa está en marcha, una técnica llamada compilación *just-in-time* (JIT). C# en .NET funciona de forma parecida. La implementación estándar de Python, CPython, también compila tus archivos `.py` a bytecode (guardando en caché el bytecode de los módulos importados en archivos `.pyc`) y después ejecuta ese bytecode en su propio bucle de intérprete.

**Dos rutas del código fuente a la CPU**




Compilar de antemano produce código máquina para un tipo de CPU; compilar a bytecode permite que un mismo archivo se ejecute en cualquier lugar donde haya instalada una máquina virtual adecuada.

ATENCIÓN

### Dos significados de "máquina virtual"

Las máquinas virtuales de lenguaje descritas aquí (la JVM, el runtime de .NET, el intérprete CPython) son programas que ejecutan bytecode. Son distintas de las máquinas virtuales que alquilan los proveedores de nube, que imitan una computadora completa para que un servidor físico pueda dividirse en varios. Ambas se llaman máquinas virtuales; el segundo tipo se explica en Servidores y la nube.

**Comparación de los enfoques**

| Enfoque | Lenguajes típicos | Ventajas | Costos |
|---|---|---|---|
| **Compilación de antemano a código máquina** | C, C++, Rust, Go | Arranque y ejecución rápidos; un único archivo autónomo que desplegar | Hay que compilar una versión distinta por CPU y sistema operativo; el ciclo de editar y probar es más lento |
| **Bytecode más una máquina virtual con JIT** | Java, Kotlin, C# | Una misma compilación funciona en muchas plataformas; el JIT puede hacer muy rápidos los servicios que se ejecutan durante mucho tiempo | Hay que instalar el runtime; arranque más lento y memoria adicional mientras el JIT entra en calor |
| **Bytecode más un intérprete** | Python (CPython), Ruby | Rápido de escribir y probar; muy portable | Suele ser más lento en cálculos pesados; las partes críticas para la velocidad a menudo se escriben en C y se llaman desde Python |
| **Compilación JIT en el navegador o el servidor** | JavaScript (V8 en Chrome y Node.js) | Funciona en cualquier lugar donde funcione un navegador; lo bastante rápido para la mayor parte del trabajo web | El rendimiento puede ser menos predecible que el del código compilado |

## La función del sistema operativo

Cuando enciendes una computadora, el primer programa importante que se inicia es el sistema operativo (SO): Windows, macOS, Linux, Android o iOS. La mayoría de los servidores de internet ejecutan Linux. El sistema operativo tiene cuatro funciones principales.

Primero, **reparte la CPU**. Una computadora portátil puede tener unas cuantas decenas de programas en marcha pero solo un puñado de núcleos, así que el SO alterna cada núcleo entre los programas muchas veces por segundo, lo bastante rápido como para que todos parezcan ejecutarse a la vez. Segundo, **gestiona la memoria**: da a cada programa su propia región e impide que un programa lea o sobrescriba la de otro. Tercero, proporciona **archivos y dispositivos**: los programas piden al SO que abra un archivo, dibuje en la pantalla o envíe datos por la red, y el SO, mediante pequeños programas llamados *drivers*, se ocupa del hardware concreto. Cuarto, **aplica la seguridad**: usuarios, permisos y la regla de que una aplicación no puede leer tus fotos a menos que se lo permitas.

Los programas hacen estas solicitudes mediante *llamadas al sistema*, un conjunto fijo de servicios que ofrece el SO. Este es el primer ejemplo del curso de una idea arquitectónica importante: una capa que oculta los detalles engorrosos tras un conjunto estable de solicitudes. Quien escribe una aplicación de reparto de comida no necesita saber de qué marca es el chip de red del teléfono.

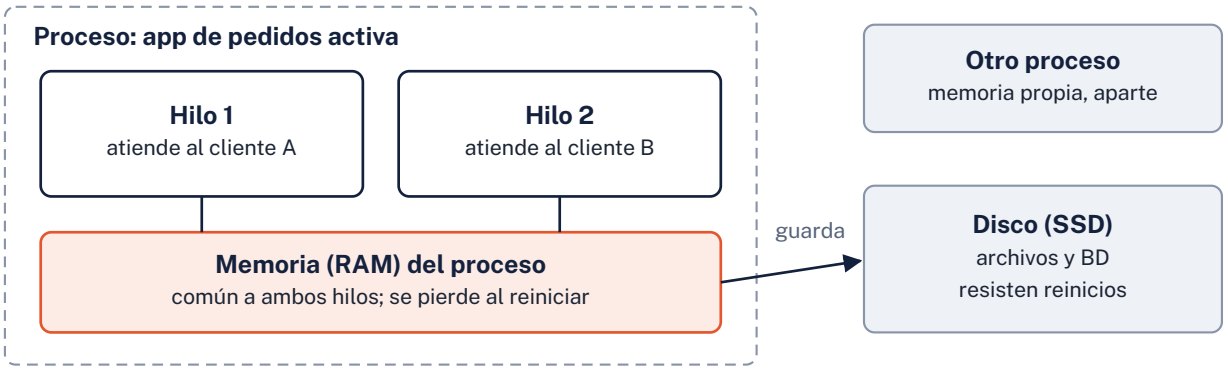
## Procesos, hilos, memoria y disco

Un archivo de programa guardado en el disco no es más que instrucciones almacenadas. Cuando lo inicias, el SO crea un proceso: una instancia en ejecución del programa con su propia memoria privada, su propia identidad y su propia parte del tiempo de CPU. Abre dos ventanas del navegador y puede que tengas varios procesos; inicia un servidor web y puede ejecutarse como un solo proceso o como muchos.

Dentro de un proceso puede haber varios hilos. Un hilo es una secuencia de instrucciones que se está ejecutando. Un proceso con varios hilos puede hacer varias cosas a la vez, por ejemplo atender las solicitudes de dos clientes al mismo tiempo, y todos sus hilos comparten la memoria del proceso. Compartirla es cómodo pero arriesgado: si dos hilos modifican el mismo dato en el mismo momento sin coordinarse, el resultado puede ser erróneo de formas que solo aparecen de vez en cuando. Estos *bugs de concurrencia* están entre los más difíciles de encontrar, y esa es una de las razones por las que muchos sistemas prefieren procesos separados, o diseños en los que cada solicitud se atiende de forma independiente.

La diferencia entre la memoria y el disco explica gran parte del diseño de software. Acceder a los datos en RAM es rápido, aproximadamente una décima de microsegundo, pero desaparecen cuando el proceso termina, falla o la máquina se reinicia. Los datos en disco se conservan, pero acceder a ellos lleva mucho más tiempo: decenas de microsegundos o más en una unidad de estado sólido, y varios milisegundos en un disco giratorio. Por eso, cualquier información que no deba perderse, como un pedido o un pago, tiene que escribirse en un almacenamiento duradero, normalmente a través de una base de datos, antes de que el sistema le diga al cliente "listo".

**Un proceso, sus hilos, su memoria y el disco**




Los hilos de un proceso comparten su memoria; los procesos separados no; solo lo que se guarda en disco sobrevive a un reinicio.

```
PRICE_PER_ITEM = 12.50   # pounds
VAT_RATE = 0.20          # 20 per cent

def total_with_vat(quantity):
    subtotal = quantity * PRICE_PER_ITEM
    return subtotal * (1 + VAT_RATE)

quantity = int(input("How many items? "))
print("Total to pay:", round(total_with_vat(quantity), 2))
```

Un pequeño programa en Python: calcular el precio de un pedido con IVA incluido

EJEMPLO PRÁCTICO

## Ejemplo práctico: qué ocurre cuando se ejecuta el programa

Supón que el código anterior está guardado como `order.py` y escribes `python order.py`, y después introduces 3.

| | |
|---|---|
| **1. Inicio** | El sistema operativo crea un proceso nuevo, le asigna memoria e inicia dentro de él el intérprete de Python (que a su vez es un programa compilado). |
| **2. Compilación** | CPython lee `order.py` del disco, comprueba que es Python válido y lo compila a bytecode. Un paréntesis o unos dos puntos que falten se notificarían aquí como un *error de sintaxis*, antes de que se ejecute nada. |
| **3. Definición** | El intérprete ejecuta el bytecode desde el principio. Guarda 12.5 y 0.2 en la memoria con los nombres `PRICE_PER_ITEM` y `VAT_RATE`, y registra la función `total_with_vat` sin ejecutarla todavía. |
| **4. Entrada** | `input()` pide al SO que lea del teclado. El proceso espera, sin usar CPU, hasta que escribes 3 y pulsas Intro. `int()` convierte el texto "3" en el número 3. |
| **5. Cálculo** | La función se ejecuta: 3 por 12.5 es 37.5, y después 37.5 por 1.2 es 45.0. Cada paso es un puñado de instrucciones de bytecode, y cada una de ellas se convierte en muchas instrucciones máquina en la CPU. |
| **6. Salida** | `print()` pide al SO que muestre `Total to pay: 45.0` en la pantalla. |
| **7. Fin** | El programa llega a su última línea, el proceso termina y el SO recupera su memoria. No se guardó nada en el disco, así que el resultado se pierde a menos que lo hayas copiado. |

Incluso este programa de nueve líneas depende del sistema operativo para la memoria, el teclado y la pantalla, y de un runtime del lenguaje para traducirlo para la CPU.

Si escribieras `three` en lugar de 3, `int()` fallaría con un *error en tiempo de ejecución* y el programa se detendría: el código nunca dijo qué hacer con una entrada incorrecta.

El programa guarda el dinero en un número de coma flotante para ser breve. Los sistemas financieros reales guardan peniques enteros o usan un tipo decimal, por razones que explica la sección siguiente.

## Bugs: cuando las instrucciones están mal

Un bug es un error en el software que hace que se comporte de forma distinta a la prevista. Como las computadoras siguen las instrucciones al pie de la letra, casi todos los bugs son una diferencia entre lo que el programador quería decir y lo que realmente escribió, o una situación en la que nadie pensó.

Conviene distinguir tres tipos. Un **error de sintaxis** rompe la gramática del lenguaje (un paréntesis que falta), así que el compilador o el intérprete se niega a ejecutar el programa; estos se detectan de inmediato. Un **error en tiempo de ejecución** ocurre mientras el programa se ejecuta, como dividir entre cero o intentar leer un archivo que no existe; el programa suele detenerse con un mensaje de error. Un **error lógico** es el más peligroso: el programa se ejecuta sin problemas y da la respuesta equivocada, por ejemplo cobrando el IVA dos veces. Nada falla de forma visible, así que solo las pruebas, una revisión cuidadosa o un cliente descontento lo pondrán al descubierto.

Los equipos profesionales cuentan con que habrá bugs y construyen defensas: pruebas automatizadas, revisión de código, lanzamientos graduales y una monitorización que detecta cuándo aumentan las tasas de error. Muchas de estas defensas son arquitectónicas, y aparecen a lo largo de todo este curso.

DE DÓNDE VIENE

### La polilla en la máquina

Los ingenieros ya llamaban "bugs" a los defectos mucho antes de que existieran las computadoras; Thomas Edison usaba la palabra en la década de 1870. La historia más conocida de la informática data del 9 de septiembre de 1947, cuando los operadores de la computadora Harvard Mark II encontraron una polilla atrapada en uno de sus relés, la pegaron con cinta en el cuaderno de registro y escribieron "First actual case of bug being found" ("primer caso real de bicho encontrado"). El chiste funciona precisamente porque el término ya se usaba. Grace Hopper, que trabajó en la máquina y más tarde influyó en los primeros lenguajes de programación, popularizó la historia.

PARA PROFUNDIZAR

### Para profundizar: puedes ver el bytecode

La biblioteca estándar de Python incluye un módulo llamado `dis` que muestra el bytecode de cualquier función. Ejecutar `dis.dis(total_with_vat)` sobre el ejemplo anterior enumera instrucciones como `LOAD_FAST` (obtener una variable local), `LOAD_GLOBAL` (obtener un nombre del nivel del módulo), `BINARY_OP` (multiplicar o sumar) y `RETURN_VALUE`. Las instrucciones exactas cambian entre versiones de Python, y por eso los archivos `.pyc` están ligados a la versión que los generó.

## Por qué esto importa para la arquitectura

La arquitectura trata de las grandes decisiones que son costosas de cambiar, y varias de ellas se basan en las ideas de esta sección. La elección del lenguaje y del runtime afecta al tiempo de arranque, al uso de memoria, a qué bibliotecas puedes usar y a quién puedes contratar. La división entre memoria y disco decide qué debe almacenarse de forma duradera y cómo se recupera un sistema después de un fallo. Los procesos y los hilos son las unidades que se ejecutan, fallan y escalan: un capítulo posterior plantea si un sistema debe ser un solo proceso o muchos que cooperan. Y el diseño en capas del sistema operativo, que oculta el hardware tras un conjunto estable de solicitudes, es el primer ejemplo de un principio que volverás a encontrar como ocultación de información.

La sección siguiente examina aquello con lo que realmente trabajan los programas: los datos, y las estructuras y algoritmos que hacen que trabajar con ellos sea rápido o lento.

COMPRUEBA LO QUE HAS ENTENDIDO

**▼ ¿Por qué desaparecen los datos de un programa cuando falla, a menos que se hayan guardado?**

Mientras un programa se ejecuta, sus datos están en la RAM, que pertenece al proceso y se borra cuando el proceso termina o la máquina se reinicia. Solo sobreviven los datos escritos en un almacenamiento como un disco o una base de datos.

**▼ ¿Cuál es la diferencia entre un compilador y un intérprete?**

Un compilador traduce todo el programa a otra forma (código máquina o bytecode) antes de que se ejecute; un intérprete ejecuta las instrucciones a medida que las lee. Muchos lenguajes, como Java y Python, compilan primero a bytecode y después ejecutan ese bytecode en una máquina virtual.

▼ **¿Por qué un programa Java compilado en una computadora portátil con Windows suele poder ejecutarse sin cambios en un servidor Linux?**

Java se compila a bytecode, no a código máquina para una CPU concreta. Cualquier máquina que tenga instalada una Máquina Virtual de Java puede ejecutar ese bytecode.

▼ **¿Por qué los programas piden al sistema operativo que lea un archivo o envíe datos, en lugar de controlar ellos mismos el disco o la tarjeta de red?**

El sistema operativo reparte el hardware de forma segura entre los programas, aplica los permisos y oculta los detalles de cada dispositivo tras las llamadas al sistema, de modo que las aplicaciones no necesitan saber en qué hardware se están ejecutando.

**FUENTES**


- The Python Tutorial (Python documentation): https://docs.python.org/3/tutorial/index.html
- Python glossary: bytecode, interpreter and related terms: https://docs.python.org/3/glossary.html
- dis: disassembler for Python bytecode: https://docs.python.org/3/library/dis.html
- The Java Virtual Machine Specification, Java SE 21 Edition: https://docs.oracle.com/javase/specs/jvms/se21/html/index.html

SECCIÓN 0.2 DE 0.8 · PRINCIPIANTE · 16 MIN DE LECTURA

# Datos, estructuras de datos y algoritmos, y por qué la velocidad depende de ellos

**LA IDEA EN BREVE**

Imagina que buscas una carta en una oficina de correos. Si las cartas están amontonadas, tienes que revisarlas una por una, y un montón más grande tarda más. Si están clasificadas en casilleros por código postal, vas directo al casillero correcto, haya las cartas que haya. Las estructuras de datos son las formas de organizar los datos (montones, casilleros, colas, pilas), y los algoritmos son los métodos para trabajar con ellas. Elegirlos bien decide si un sistema sigue siendo rápido a medida que crece.

**LO QUE NECESITAS SABER**

- Las computadoras lo almacenan todo como bits; un byte son ocho bits, y los tipos de datos (enteros, números de coma flotante, booleanos, cadenas) indican cómo leerlos.
- Los números de coma flotante son aproximados, así que el dinero debe almacenarse como enteros de la unidad más pequeña o con un tipo decimal.
- El texto se almacena como caracteres Unicode codificados como bytes, normalmente en UTF-8, donde un carácter puede ocupar de uno a cuatro bytes.
- Cada estructura de datos, como las listas, los diccionarios, los conjuntos, las colas, las pilas, los árboles y los grafos, hace rápidas operaciones distintas.
- La notación O grande (Big-O) describe cómo crece el trabajo de un algoritmo con los datos: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$.
- Los índices y las cachés cambian espacio adicional (y el trabajo de mantenerlos al día) por respuestas más rápidas.

## Todo son bits

Las computadoras lo almacenan todo como bits: dígitos binarios que valen 0 o 1. Físicamente, un bit puede ser una diminuta carga eléctrica, una zona magnética de un disco o un pulso de luz en una fibra, pero para el software es simplemente uno de dos valores. Ocho bits forman un byte, que puede contener $2^8 = 256$ patrones distintos, de 00000000 a 11111111.

A partir de ahí, los tamaños van creciendo. Un kilobyte son unos mil bytes, un megabyte alrededor de un millón, un gigabyte alrededor de mil millones y un terabyte alrededor de un billón. (En sentido estricto, los prefijos estándar kilo, mega y giga significan potencias de 1000; la memoria suele medirse en potencias de 1024, y para ellas existen las unidades kibibyte,

mebibyte y gibibyte. La diferencia rara vez importa para el diseño, pero explica por qué un disco de "1 TB" muestra menos espacio en algunos gestores de archivos.) Una página de texto plano ocupa unos pocos kilobytes, una foto de un teléfono unos pocos megabytes y una hora de video en alta definición unos pocos gigabytes.

Los bits no significan nada por sí solos. El patrón 01000001 podría ser el número 65, la letra A, parte de un color de una foto o parte de una instrucción para la CPU. Lo que da significado a los bits es el acuerdo sobre cómo leerlos, que es la idea de un tipo de datos.

## Tipos de datos: números, texto y verdadero o falso

Un tipo de datos le dice a un programa qué clase de valor es un dato y, por tanto, qué operaciones tienen sentido. Todos los lenguajes de programación tienen unos cuantos tipos básicos, y los de Python son representativos.

Los **enteros** son números sin decimales, como 3, 0 o -42, que se usan para contar cosas: artículos en un carrito, asientos en un vuelo. Los **números de coma flotante** (floats), como 12.5 o 0.2, representan los decimales de forma aproximada, en binario. Como muchos decimales no tienen una forma binaria exacta, `0.1 + 0.2` en Python da `0.30000000000000004`. Eso no es problema en física o en gráficos, pero es incorrecto para el dinero, y por eso los sistemas financieros almacenan los importes como números enteros de peniques o céntimos, o usan un tipo decimal (Python tiene `decimal.Decimal`). Los **booleanos** tienen solo dos valores, `True` y `False`, y registran hechos de sí o no, como "¿se ha pagado este pedido?". Las **cadenas** son texto, como el nombre de un cliente. Los sistemas reales añaden más tipos por encima: fechas y horas, dinero y datos binarios como las imágenes.

Los tipos importan más allá de un solo programa. Cuando dos sistemas intercambian datos, deben ponerse de acuerdo tanto en los tipos como en los valores: ¿un código postal es un número o texto (texto: algunos empiezan por letras y algunos códigos numéricos empiezan por cero)? ¿Una fecha va en orden día-mes o mes-día, y en qué zona horaria? Muchos bugs de integración costosos son desacuerdos sobre tipos.

## El texto y UTF-8

El texto se almacena asignando un número a cada carácter y guardando después esos números como bytes. El estándar internacional **Unicode** asigna un número, llamado punto de código, a bastante más de cien mil caracteres de los sistemas de escritura del mundo, además de símbolos y emojis. Después, una *codificación* decide cómo esos números se convierten en bytes. Hoy, la codificación dominante, en la web y en la mayoría de los sistemas nuevos, es UTF-8.

UTF-8 usa un número variable de bytes por carácter: un byte para las letras básicas del inglés, los dígitos y la puntuación (así que el texto que solo usa estos caracteres es idéntico, byte a byte, en UTF-8 y en la codificación ASCII, mucho más antigua), dos bytes para caracteres como `é`, tres para caracteres como `€` y la mayoría de los caracteres chinos, japoneses y coreanos, y cuatro para los emojis y algunas escrituras menos comunes. Esto tiene una consecuencia práctica: el número de caracteres de una cadena y el número de bytes que ocupa pueden ser distintos. Una columna de base de datos limitada a 50 bytes puede no ser capaz de contener un nombre de 50 caracteres escrito en griego.

El bug de texto clásico son los caracteres ilegibles (por ejemplo `CafÃ©` en lugar de `Café`), que aparecen cuando unos bytes escritos en una codificación se leen como si fueran de otra. La solución es fácil de enunciar: usa UTF-8 en todas partes, y dilo explícitamente siempre que el texto cruce un límite, como un archivo, una base de datos o un mensaje de red.

## Estructuras de datos: formas de organizar los datos

Los programas rara vez manejan un solo valor a la vez. Una estructura de datos es una forma de organizar muchos valores en memoria para que las operaciones que necesitas sean rápidas. La elección correcta puede marcar la diferencia entre una página que carga al instante y otra que no llega a cargar nunca. Cada una tiene una analogía cotidiana.

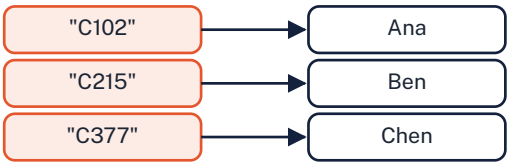
Una **lista** (llamada *array* en muchos lenguajes) mantiene los elementos en orden, como una lista de la compra numerada. Obtener el quinto elemento es instantáneo, porque la computadora puede calcular dónde está. Encontrar un elemento por su valor obliga a revisar los elementos uno por uno. Un diccionario (también llamado *mapa* o *tabla hash*) guarda valores bajo claves únicas, como un guardarropa: entregas el resguardo C377 y recibes el abrigo correcto de inmediato, sin mirar todos los abrigos. Un **conjunto** es una colección sin duplicados, como una lista de invitados, y responde rápidamente a "¿está esta persona en ella?".

Una **cola** funciona como la fila ante el mostrador de una oficina de correos: el primero que entra es el primero que sale. Los sistemas usan colas para el trabajo pendiente, como los pedidos que esperan a la cocina. Una **pila** funciona como un montón de platos: el último que entra es el primero que sale. El botón Atrás de tu navegador usa una: la página que visitaste más recientemente es la primera a la que vuelves. Un **árbol** tiene una única raíz con ramas por debajo, como un árbol genealógico o las carpetas de tu computadora; los árboles también son la forma en que las bases de datos mantienen los datos ordenados para buscar rápido. Un **grafo** son elementos (nodos) unidos por enlaces (aristas) con cualquier patrón, como un mapa de carreteras o una red social, y responde a preguntas como "¿cuál es la ruta más corta?" o "¿quiénes son amigos de mis amigos?".

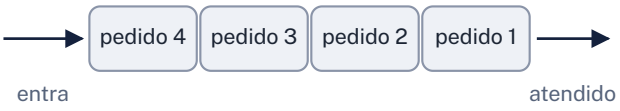
## Seis estructuras de datos comunes

**Lista (array): elementos en orden, por posición**

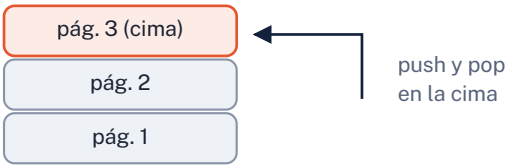
**Diccionario (mapa): valor por clave**



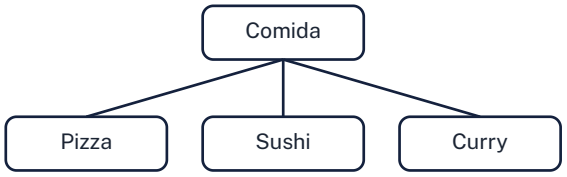

**Cola: sale primero el primero**



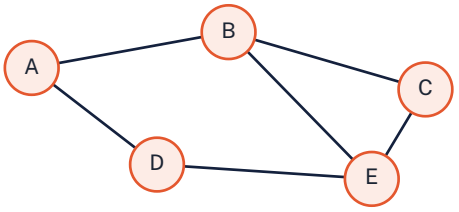

**Pila: sale primero el último**




**Árbol: una raíz, cada elemento tiene hijos**




**Grafo: elementos enlazados libremente**




Cada estructura organiza el mismo tipo de datos de forma distinta, lo que hace rápidas preguntas distintas.

### Para qué sirve cada estructura

| Estructura | Rápida en | Lenta en | Uso típico |
|---|---|---|---|
| **Lista / array** | Obtener el elemento de una posición; añadir al final | Encontrar un elemento por su valor (hay que recorrerla); insertar cerca del principio | Un carrito ordenado de artículos; una página de resultados de búsqueda |
| **Diccionario / mapa** | Encontrar, añadir o eliminar por clave | Todo lo que necesite los elementos en orden | Clientes por ID; ajustes por nombre; una caché |
| **Conjunto** | Comprobar si un elemento pertenece; eliminar duplicados | Mantener un orden (la mayoría de los conjuntos no tienen ninguno) | Qué usuarios ya han votado; etiquetas únicas |
| **Cola** | Añadir al final y sacar por el principio | Acceder a los elementos del medio | Pedidos que esperan a la cocina; tareas que esperan a un worker |
| **Pila** | Añadir y sacar por arriba | Acceder a los elementos de más abajo | Historial de deshacer; el botón Atrás; seguimiento de las llamadas a funciones |
| **Árbol (ordenado)** | Encontrar por valor y listar un rango en orden | Más complejo de mantener equilibrado | Índices de bases de datos; carpetas; menús |
| **Grafo** | Seguir relaciones | Los grafos muy grandes necesitan un almacenamiento especializado | Rutas y mapas; redes sociales; recomendaciones |

```
basket = ["pizza", "cola", "pizza"]           # list: ordered, duplicates allowed
print(basket[0])                               # by position: "pizza"

customers = {"C102": "Ana", "C377": "Chen"}     # dictionary: key to value
print(customers["C377"])                       # by key: "Chen"

voted = {"ana@example.com", "ben@example.com"}   # set: no duplicates
print("ana@example.com" in voted)              # True

from collections import deque
kitchen = deque(["order 1", "order 2"])         # queue
kitchen.append("order 3")                      # joins at the back
print(kitchen.popleft())                       # served first: "order 1"
```

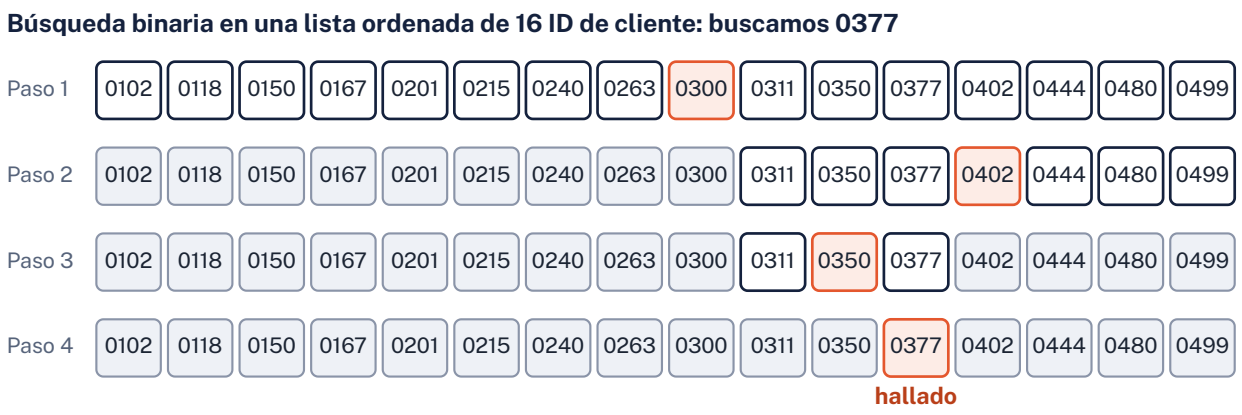
Las mismas estructuras en Python

## Algoritmos

Un algoritmo es un método preciso, paso a paso, para resolver un problema y que siempre termina: como una receta, pero sin margen de interpretación. Ordenar una lista de nombres, encontrar la ruta más corta entre dos direcciones, comprobar si una contraseña coincide y decidir qué anuncio mostrar son tareas que realizan los algoritmos.

Normalmente hay más de un algoritmo para el mismo problema, y pueden diferir enormemente en la cantidad de trabajo que hacen. Piensa en buscar un nombre en una guía telefónica impresa. Podrías empezar en la página uno y leer cada entrada hasta encontrarlo: correcto, pero lento. O podrías abrir la guía por la mitad, ver si el nombre va antes o después de esa página, descartar la mitad que no puede contenerlo y repetir. Como la guía está ordenada, cada consulta reduce a la mitad las páginas restantes. Este segundo método se llama **búsqueda binaria**, y en una guía de un millón de nombres necesita unas 20 consultas en lugar de hasta un millón.

**Búsqueda binaria: dividir el problema a la mitad**




Resaltado: el elemento central comparado en cada paso. Gris: descartado. Quedan: 16, 7, 3, 1

En una lista ordenada, cada comparación descarta la mitad de lo que queda, así que 16 elementos necesitan como mucho 4 o 5 consultas.

## Big-O: cómo crece el trabajo con los datos

Los ingenieros describen cómo crece el costo de un algoritmo usando la notación O grande (Big-O). Ignora el tiempo exacto en una máquina concreta y plantea una sola pregunta: si la cantidad de datos, llamada *n*, se hace diez o mil veces mayor, ¿cuánto trabajo más hay? Describe el peor caso salvo que se indique otra cosa, e ignora los factores constantes, lo cual es a la vez su fortaleza (capta la forma del crecimiento) y su debilidad (dos algoritmos O(n) pueden diferir en la práctica en un factor de diez).

**O(1)**, tiempo constante: el trabajo no depende de n. Buscar una clave en un diccionario es O(1) de media. **O(log n)**, logarítmico: el trabajo crece un paso cada vez que n se duplica, como en la búsqueda binaria. **O(n)**, lineal: el doble de datos, el doble de trabajo, como al recorrer una lista. **O(n log n)**: un poco peor que lineal; los buenos algoritmos de ordenación de uso general trabajan a este ritmo. **O($n^2$)**, cuadrático: el doble de datos, cuatro veces el trabajo. Suele aparecer cuando, para cada elemento, un programa recorre todos los demás elementos, por ejemplo al comparar cada cliente con todos los demás para encontrar duplicados.

**Cuántos pasos básicos, aproximadamente, a medida que crecen los datos**

| n (elementos) | O(1) | O(log n) | O(n) | O(n log n) | O($n^2$) |
|---|---|---|---|---|---|
| **10** | 1 | unos 3 | 10 | unos 33 | 100 |
| **1,000** | 1 | unos 10 | 1,000 | unos 10 000 | 1,000,000 |
| **1,000,000** | 1 | unos 20 | 1,000,000 | unos 20 000 000 | 1,000,000,000,000 |

Lee con atención la última columna. A mil millones de pasos sencillos por segundo, un algoritmo de un millón al cuadrado necesita un billón de pasos: alrededor de un cuarto de hora de cálculo puro y, en la práctica, mucho más. El mismo trabajo hecho con un método O(n log n) lleva una pequeña fracción de segundo. Un código que parecía ir bien en las pruebas con 1000 clientes puede quedar paralizado con un millón, y ninguna cantidad de hardware adicional salvará por mucho tiempo a un algoritmo cuadrático.

EJEMPLO PRÁCTICO

## Ejemplo práctico: encontrar a un cliente entre un millón

Una aplicación de reparto de comida almacena 1 000 000 de clientes. Cada vez que llega un pedido, tiene que encontrar al cliente por su ID. Compara dos diseños, suponiendo que cada comprobación tarda 0.1 microsegundos.

| | |
|---|---|
| **Diseño A: una lista** | `for c in customers: if c.id == wanted: ...` recorre la lista hasta encontrar una coincidencia. |
| **Comprobaciones por búsqueda (A)** | Hasta 1 000 000; de media, unas 500 000. |
| **Tiempo por búsqueda (A)** | Unas 500 000 x 0.1 microsegundos = 50 milisegundos. |
| **Con 200 pedidos por segundo (A)** | 200 x 50 ms = 10 segundos de trabajo de CPU cada segundo: el servidor no da abasto, así que las solicitudes se acumulan en cola y agotan su tiempo de espera. |
| **Diseño B: un diccionario con el ID como clave** | `customers_by_id[wanted]` calcula dónde está la entrada y va directamente a ella. |
| **Comprobaciones por búsqueda (B)** | Más o menos 1, haya mil clientes o un millón. |
| **Tiempo por búsqueda (B)** | Muy por debajo de un microsegundo. |
| **Costo del diseño B** | Memoria adicional para el diccionario, y el trabajo de construirlo y mantenerlo al día cada vez que se añade o modifica un cliente. |

Los dos diseños dan respuestas idénticas; solo cambia la estructura de datos, y eso convierte el resultado de un fracaso en un margen holgado.

Las bases de datos ofrecen la misma elección. Sin un índice, encontrar una fila por una columna obliga a recorrer la tabla; con un índice, bastan unos pocos pasos a través de un árbol ordenado. La sección sobre el almacenamiento de datos vuelve sobre esto.

**Recorrido frente a búsqueda directa**

Un recorrido hace un trabajo proporcional al número de elementos; un diccionario va directamente a la ranura correcta.

## Espacio frente a tiempo

El diccionario del ejemplo práctico es más rápido porque usa memoria adicional para organizar los datos de antemano. Este es uno de los compromisos más habituales de la informática: a menudo puedes comprar velocidad con espacio, o ahorrar espacio gastando tiempo.

Un índice son datos adicionales, guardados junto a los datos principales, que responden rápidamente a un tipo concreto de pregunta, como el índice al final de un libro. Acelera las lecturas, pero ocupa almacenamiento y debe actualizarse en cada escritura, así que insertar en una tabla con diez índices es lento. Una caché guarda una copia de la respuesta a una pregunta costosa en algún lugar rápido, como la memoria, para que la siguiente persona que pregunte la obtenga de inmediato. Las cachés son la razón de que la página de un producto popular cargue rápido aunque la base de datos que hay detrás esté ocupada, pero plantean una pregunta difícil: cuando cambian los datos subyacentes, ¿con qué rapidez debe cambiar también la copia? Mostrar un precio obsoleto durante unos segundos puede ser aceptable; mostrar un saldo bancario obsoleto quizá no.

La compresión es el intercambio contrario: gastar tiempo de CPU para comprimir los datos en menos bytes, lo que ahorra almacenamiento y tiempo de transferencia por la red.

ATENCIÓN

### Ideas erróneas habituales

- **"Big-O te dice lo rápido que se ejecuta el código."** Te dice cómo crece el costo. Con diez elementos, un método $O(n^2)$ puede ser más rápido que uno ingenioso O(n log n); Big-O importa a medida que n se hace grande.
- **"Un hardware más rápido arregla los algoritmos lentos."** Una máquina el doble de rápida solo permite a un algoritmo cuadrático procesar unas 1.4 veces más datos en el mismo tiempo.
- **"Las búsquedas en un diccionario siempre son instantáneas."** Son O(1) de media, pero usan más memoria que una lista y no ofrecen ningún orden, así que "todos los clientes cuyo nombre empieza por C" sigue necesitando un recorrido o una estructura ordenada.
- **"Los números de coma flotante sirven para el dinero."** Los errores de redondeo se acumulan a lo largo de millones de transacciones. Usa enteros de la unidad más pequeña, o un tipo decimal.

PARA PROFUNDIZAR

### Para profundizar: medias, peores casos y hashing

Un diccionario logra O(1) de media pasando cada clave por una *función hash*, que convierte la clave en un número que elige una ranura. Dos claves pueden caer en la misma ranura (una *colisión*); las implementaciones lo gestionan, y en un caso patológico en el que colisionan muchas claves, las búsquedas se degradan hacia O(n). No es solo teoría: hay atacantes que han enviado a servidores web parámetros de solicitud diseñados para colisionar, y por eso los lenguajes modernos, Python incluido, aleatorizan el hashing de sus cadenas. Los índices de las bases de datos suelen usar un árbol equilibrado llamado árbol B (B-tree), que ofrece búsquedas O(log n) y, a diferencia de una tabla hash, también admite rangos y orden.

## Por qué esto importa para la arquitectura

Los arquitectos rara vez escriben algoritmos de ordenación, pero toman constantemente decisiones cuyas consecuencias son exactamente estas. ¿Qué preguntas tendrá que responder rápidamente el sistema, y tienen los datos los índices necesarios para responderlas? ¿Cuánto crecerán los datos en tres años, y qué partes del diseño son lineales o peores ante ese crecimiento? ¿Dónde compensa la caché su complejidad, y hasta qué punto pueden estar obsoletos los datos en caché? Cuando el volumen de datos se multiplica por diez, ¿qué componente falla primero?

Son preguntas sobre rendimiento y escalabilidad, que el Capítulo 2 precisa, y sobre el uso de cachés y la elección de base de datos, que el Capítulo 7 trata a fondo. La sección siguiente pasa de los datos al código, y a cómo se organiza el código para que las personas puedan entenderlo y modificarlo.

**COMPRUEBA LO QUE HAS ENTENDIDO**

**▼ ¿Por qué `0.1 + 0.2` no da exactamente 0.3 en la mayoría de los lenguajes?**

Los números de coma flotante se almacenan en binario, y muchas fracciones decimales, como 0.1 y 0.2, no tienen una forma binaria exacta, así que aparecen pequeños errores de redondeo.

**▼ ¿Qué estructura de datos encaja con un botón de "deshacer" en un editor de texto, y por qué?**

Una pila: cada acción se coloca encima, y deshacer saca primero la más reciente (el último que entra es el primero que sale).

**▼ Si un proceso O(n) tarda un segundo con 100 000 elementos, ¿cuánto tardará aproximadamente con 1 000 000?**

Unos diez segundos, porque el trabajo lineal crece en proporción a los datos.

**▼ ¿Qué pagas a cambio de la velocidad que te da un índice de base de datos?**

Almacenamiento adicional, y trabajo adicional en cada inserción, actualización y eliminación para mantener el índice al día.

**FUENTES**


- Python tutorial: data structures: https://docs.python.org/3/tutorial/datastructures.html
- Python wiki: time complexity of built-in operations: https://wiki.python.org/moin/TimeComplexity
- Unicode HOWTO (Python documentation): https://docs.python.org/3/howto/unicode.html
- RFC 3629: UTF-8, a transformation format of ISO 10646: https://www.rfc-editor.org/rfc/rfc3629

SECCIÓN 0.3 DE 0.8 · PRINCIPIANTE · 16 MIN DE LECTURA

# Funciones, módulos, objetos, interfaces, bibliotecas y frameworks

**LA IDEA EN BREVE**

Piensa en un restaurante grande. En lugar de un solo cocinero que lo hace todo, hay estaciones (parrilla, repostería, ensaladas), cada una con su propio personal, sus herramientas y sus recetas. Los pedidos pasan de una estación a otra por una ventanilla con un sistema de comandas, así que la parrilla no necesita saber cómo trabaja el repostero. El código se organiza de la misma manera: pequeñas tareas con nombre (funciones) se agrupan en estaciones (módulos), y las estaciones se relacionan entre sí solo mediante comandas acordadas (interfaces). Así, una estación puede cambiar sus métodos sin alterar al resto de la cocina.

**LO QUE NECESITAS SABER**

- Una función es un fragmento de código reutilizable y con nombre, que tiene entradas y una salida; permite escribir una regla una sola vez y cambiarla en un único lugar.
- Los módulos y los paquetes agrupan código relacionado y separan una pequeña parte pública de los detalles privados, que pueden cambiar libremente.
- Las clases reúnen los datos con los métodos que actúan sobre ellos; el encapsulamiento mantiene los datos privados para que las reglas se cumplan siempre.
- Una interfaz es un contrato entre partes; el código que depende de una interfaz y no de una implementación puede cambiar de implementación con facilidad.
- Tú llamas a una biblioteca; un framework llama a tu código. Esta inversión de control hace que los frameworks sean potentes, pero más difíciles de sustituir.
- Las dependencias, que obtienen gestores de paquetes como pip y npm, ahorran esfuerzo, pero traen código transitivo, riesgos de seguridad y trabajo de mantenimiento.
- Una buena estructura significa bajo acoplamiento y alta cohesión, lo que mantiene barato el cambio y permite que los equipos trabajen de forma independiente.

## Por qué organizar el código

Un pequeño script de veinte líneas se puede leer de principio a fin en un minuto. Un sistema real, como una plataforma de reparto de comida, puede tener cientos de miles o millones de líneas escritas por decenas de personas a lo largo de muchos años. Nadie puede tener todo eso

en la cabeza. La única forma de trabajar en un sistema así es dividirlo en partes que se puedan entender, probar y cambiar de una en una, con reglas claras sobre cómo pueden usarse unas a otras.

Los lenguajes de programación nos ofrecen para ello una escalera de bloques de construcción, de menor a mayor: funciones, módulos y paquetes, clases y objetos, y las interfaces entre ellos. Por encima del código que escribimos nosotros están las bibliotecas y los frameworks escritos por otros. Esta sección sube esa escalera. Esas mismas ideas, aplicadas a mayor escala, son el objeto de la arquitectura de software.

## Funciones: poner nombre a una tarea y reutilizarla

Una función es un fragmento de código con nombre que hace una sola tarea. Recibe *entradas* (llamadas parámetros o argumentos), hace un trabajo y normalmente devuelve una *salida* (su valor de retorno). Piensa en una cafetera: pones granos, agua y eliges una bebida; la máquina hace su trabajo; sale un café. No necesitas saber cómo funciona la bomba para usarla.

Las funciones aportan tres beneficios. **Reutilización**: el cálculo del IVA se escribe una vez y se llama desde el proceso de pago, la factura y la pantalla de reembolsos. **Legibilidad**: `total_with_vat(basket)` dice qué está pasando sin mostrar cómo. **Cambio en un único lugar**: cuando cambia el tipo de IVA, cambia una función, no veinte copias de la misma operación repartidas por el código (copias que pronto acabarían siendo distintas).

Una buena función hace una sola cosa, tiene un nombre que dice lo que hace, recibe como parámetros las entradas que necesita en lugar de ir a buscar datos ocultos, y es lo bastante corta como para leerla de un vistazo. Una función que solo calcula su salida a partir de sus entradas, sin cambiar nada más, se llama función *pura*; las funciones puras son especialmente fáciles de probar, porque las mismas entradas siempre dan la misma salida.

```
VAT_RATE = 0.20

def total_with_vat(prices_in_pence):
    """Return the total of a basket in pence, including VAT."""
    subtotal = sum(prices_in_pence)
    return round(subtotal * (1 + VAT_RATE))

checkout_total = total_with_vat([1250, 399])   # 1979
invoice_total = total_with_vat([5000])         # 6000
```

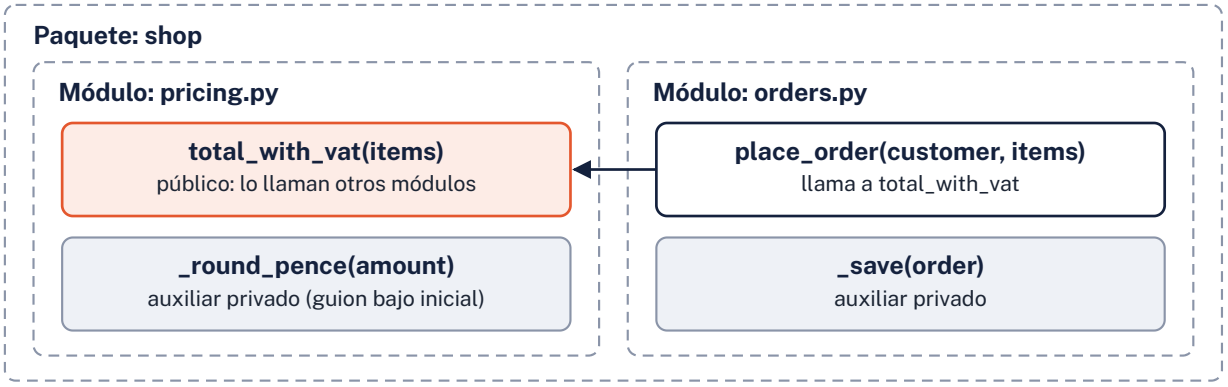
Una función, reutilizada en varios lugares

## Módulos y paquetes: agrupar el código relacionado

Un módulo agrupa funciones y datos relacionados en una unidad con su propio nombre. En Python, un módulo es simplemente un archivo `.py`; `pricing.py` podría contener todo lo relacionado con precios, descuentos e impuestos. Otro archivo lo usa escribiendo `import pricing`. Un **paquete** es una carpeta de módulos relacionados, así que una tienda podría tener un paquete `shop` que contenga los módulos `pricing`, `orders` y `customers`. Otros lenguajes usan palabras distintas (Java tiene paquetes, C# tiene espacios de nombres y ensamblados, JavaScript tiene módulos), pero la idea es la misma.

Un módulo es como un departamento de una organización. El departamento de finanzas tiene un mostrador de atención público donde otros pueden hacer solicitudes ("por favor, emitan esta factura") y procesos internos que nadie de fuera necesita ver. Del mismo modo, los módulos separan una pequeña parte **pública**, pensada para que la usen otros módulos, de los detalles **privados**. Python marca los nombres privados por convención con un guion bajo inicial, como `_round_pence`; Java y C# imponen la privacidad con palabras clave como `private`. El valor de esta división es que la parte privada se puede reescribir libremente sin romper nada a nadie, siempre que la parte pública cumpla sus promesas.

**Funciones, módulos y un paquete**




Las funciones están en módulos; los módulos afines forman un paquete; desde fuera solo se usan los nombres públicos

El módulo orders usa solo la función pública del módulo pricing, nunca su auxiliar privado.

## Clases y objetos: los datos con su comportamiento

Muchos lenguajes permiten reunir los datos con las funciones que trabajan sobre ellos. Una **clase** es la plantilla; un **objeto** es una cosa concreta hecha a partir de ella. Una clase `BankAccount` podría decir que toda cuenta tiene un titular y un saldo, y que admite depósitos y retiros. La cuenta de cada cliente es entonces un objeto distinto creado a partir de esa clase. Las funciones que pertenecen a una clase se llaman *métodos*. (Consulta la entrada del glosario sobre clase y objeto).

La idea más útil aquí para la arquitectura es el encapsulamiento: un objeto mantiene sus datos privados y solo deja que otro código los cambie a través de sus métodos. Si cualquiera pudiera asignar `account.balance` directamente, nada impediría que código de otra parte dejara un saldo en negativo o se saltara el registro de auditoría. Con el encapsulamiento, el método `withdraw` es la única puerta, y puede hacer cumplir las reglas cada vez. La analogía es un cajero de banco: los clientes no pueden entrar en la cámara acorazada, pero pueden pedírselo al cajero, que comprueba las reglas antes de entregar el dinero.

La programación orientada a objetos tiene mucho más (herencia, polimorfismo, patrones de diseño), y los profesionales no se ponen de acuerdo sobre cuánto de ello es útil: algunos lenguajes, como Go, omiten deliberadamente la herencia, y muchos equipos prefieren un estilo más funcional. Sin embargo, el encapsulamiento es valioso en cualquier estilo.

```
class BankAccount:
    def __init__(self, owner):
        self.owner = owner
        self._balance = 0          # private by convention

    def deposit(self, pence):
        if pence ⩽ 0:
            raise ValueError("Deposit must be positive")
        self._balance += pence

    def withdraw(self, pence):
        if pence > self._balance:
            raise ValueError("Insufficient funds")
        self._balance -= pence

    def balance(self):
        return self._balance

acct = BankAccount("Ana")
acct.deposit(10000)
acct.withdraw(2500)
print(acct.balance())   # 7500
```

Encapsulamiento: el saldo solo puede cambiar mediante métodos que hacen cumplir las reglas

## Interfaces: contratos entre partes

Una interfaz es el conjunto acordado de operaciones que una parte de un sistema ofrece a las demás: sus nombres, qué entradas reciben, qué devuelven y qué puede salir mal. Es un contrato. Quien llama confía en el contrato, no en cómo se hace el trabajo. Un enchufe es una interfaz: cualquier aparato con la clavija adecuada funciona, y al aparato no le importa si la electricidad procede del viento, del gas o de la energía nuclear.

En el código, una interfaz permite cambiar una implementación por otra. Supón que una tienda cobra hoy los pagos con tarjeta a través de un proveedor de pagos, pero puede cambiar de proveedor el año que viene. Si el código del proceso de pago llama directamente a las funciones propias del proveedor, por todas partes, cambiar de proveedor obliga a reescribir el proceso de pago. Si, en cambio, la tienda define su propia interfaz pequeña `PaymentGateway` con una sola operación, `charge(amount, card_token)`, y luego escribe un adaptador por proveedor, el proceso de pago depende solo de la interfaz y cambiar de proveedor significa escribir un adaptador nuevo. El mismo truco facilita las pruebas: en ellas, una pasarela falsa que nunca contacta con un banco real puede sustituir a la verdadera.

Las interfaces son la idea más importante de esta sección para los arquitectos. Cada límite de un sistema, desde la firma de una función hasta una API web que usan miles de otras empresas, es una interfaz. Las buenas son pequeñas, estables y ocultan las decisiones que probablemente cambiarán; ese principio, llamado ocultación de información, lo formuló David Parnas en 1972 y se trata en el Capítulo 3.

```
from typing import Protocol

class PaymentGateway(Protocol):
    def charge(self, amount_pence: int, card_token: str) → bool: ...

class ProviderAGateway:
    def charge(self, amount_pence, card_token):
        ...  # call Provider A's web API
        return True

class FakeGateway:                      # used in tests
    def charge(self, amount_pence, card_token):
        return amount_pence < 100_000   # pretend large payments fail

def checkout(basket_total, card_token, gateway: PaymentGateway):
    return "paid" if gateway.charge(basket_total, card_token) else "declined"
```

Una interfaz (aquí un Protocol de Python) con dos implementaciones intercambiables

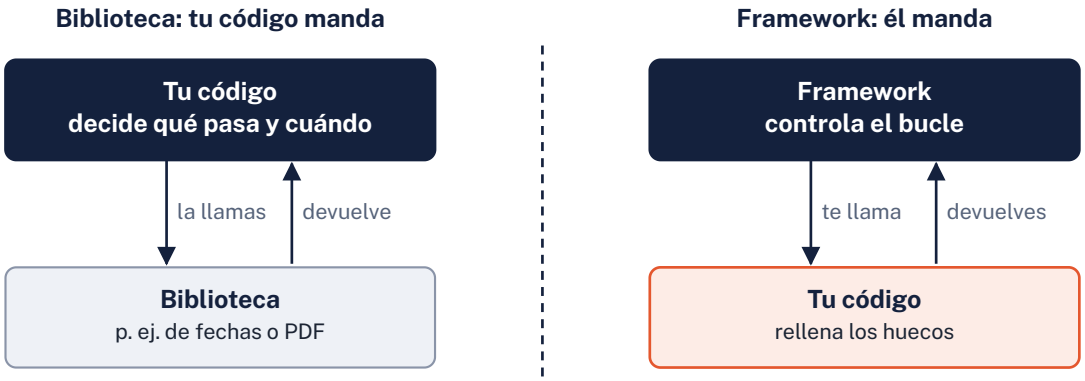
## Bibliotecas y frameworks: código escrito por otros

Casi nadie escribe todo desde cero. Una biblioteca es código reutilizable que ha escrito otra persona y al que tu código llama cuando lo necesita: una biblioteca para manejar fechas y zonas horarias, para generar PDF, para comunicarse con una base de datos. Tú sigues al mando; la biblioteca es una herramienta que tomas y dejas.

Un framework es diferente. Proporciona el esqueleto general de una aplicación y decide cuándo ocurren las cosas; tú rellenas los huecos con tu propio código, y el framework llama a ese código en los momentos adecuados. Un framework web como Django (Python), Spring (Java), Ruby on Rails o Express (JavaScript) recibe cada solicitud web entrante, determina cuál de tus funciones debe atenderla, llama a esa función y convierte su resultado en una página web o en datos. La forma corta de recordar la diferencia: *tú llamas a una biblioteca; un framework llama a tu código*.

Esa inversión se llama inversión de control. Martin Fowler atribuye a Ralph Johnson y Brian Foote el uso del término en 1988, y la idea suele resumirse como el principio de Hollywood: "No nos llames, nosotros te llamaremos". Una franquicia de restaurantes es una buena analogía: la franquicia aporta el local, el sistema de pedidos, el horario de apertura y las reglas; el franquiciado aporta la cocina en los huecos que la franquicia le deja.

**Quién llama a quién**




La dirección del control es la verdadera diferencia entre una biblioteca y un framework.

**Biblioteca o framework**

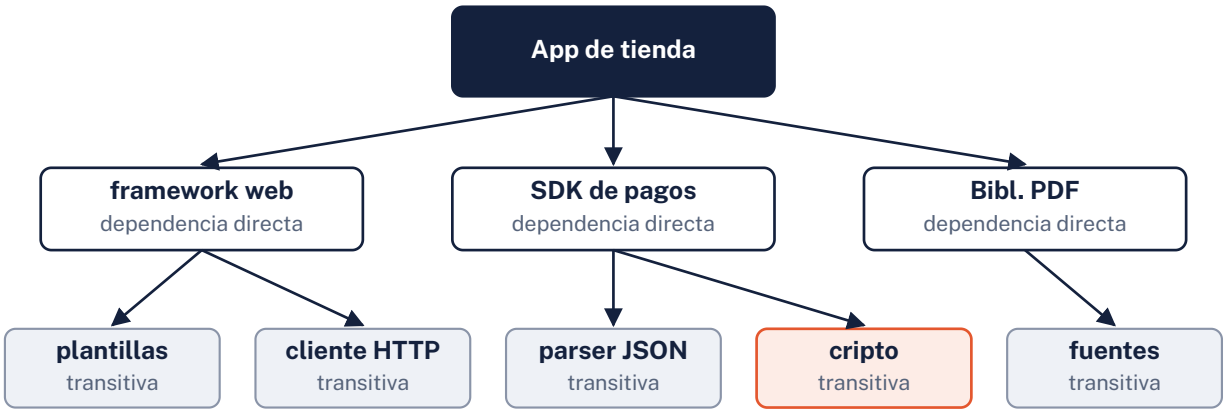
| | Biblioteca | Framework |
|---|---|---|
| **Quién controla el flujo** | Tu código | El framework (inversión de control) |
| **Ejemplos** | Una biblioteca de fechas, un generador de PDF, un parser de JSON | Django, Spring, Ruby on Rails, Angular |
| **Facilidad para sustituirlo más adelante** | Normalmente moderada: cambiar los lugares que la llaman | Normalmente difícil: la estructura del framework da forma a toda la aplicación |
| **Principal beneficio** | Te ahorra escribir tú mismo una capacidad | Te ahorra diseñar toda la estructura; convenciones coherentes en todo el equipo |
| **Principal riesgo** | Una dependencia más que mantener | Dependencia (lock-in) de su forma de trabajar y de su ciclo de actualizaciones |

## Dependencias y gestores de paquetes

Cada biblioteca o framework que necesita tu código es una dependencia. Los proyectos modernos tienen muchas, así que usan un gestor de paquetes para obtenerlas: `pip` para Python (desde el Python Package Index, PyPI) y `npm` para JavaScript (desde el registro de npm) son los más conocidos, con equivalentes como Maven para Java y NuGet para .NET. Un proyecto enumera lo que necesita, a menudo con rangos de versiones, y el gestor de paquetes descarga esos paquetes y, lo que es crucial, los paquetes de los que dependen *ellos*. Estas dependencias indirectas, *transitivas*, suelen superar muchas veces en número a las directas.

Las versiones suelen numerarse como MAJOR.MINOR.PATCH según una convención llamada versionado semántico: un cambio en el número mayor indica un cambio incompatible; en el número menor, nuevas funciones que no deberían romper nada a los usuarios existentes; y en el número de parche, correcciones de errores. Los proyectos también guardan un *lockfile* (archivo de bloqueo), como `package-lock.json` en npm, que registra la versión exacta de cada paquete instalado, para que la máquina de cada desarrollador y la compilación de producción usen el mismo código.

**Dependencias directas y transitivas**




Elegiste tres paquetes; dependes de ocho. Un defecto en cualquiera (por ejemplo, cripto) es un defecto en tu app.

La mayor parte del código que ejecuta tu aplicación lo eligieron las decisiones de otras personas.

Las dependencias ahorran un esfuerzo enorme, pero cada una es código que no escribiste tú y que se ejecuta con todos los permisos del tuyo. Dos incidentes muy conocidos muestran los riesgos. En marzo de 2016, un desarrollador retiró sus paquetes de npm a raíz de una disputa, entre ellos `left-pad`, una función diminuta de unas once líneas para rellenar texto; miles de proyectos que dependían de él, directa o indirectamente, dejaron de compilarse de repente hasta que npm lo restauró. En diciembre de 2021 se reveló una grave vulnerabilidad de seguridad, apodada Log4Shell, en Log4j 2, una biblioteca de logging para Java muy utilizada; organizaciones de todo el mundo pasaron semanas localizando cada sistema que la contenía, a menudo enterrada varios niveles por debajo en sus dependencias.

### Lista de comprobación: antes de añadir una dependencia

Preguntas que se hace un equipo cuidadoso antes de confiar en el paquete de otra persona.

- [ ] **¿Lo necesitamos?**
  Unas pocas líneas de código propio pueden salir más baratas con el tiempo que un paquete al que hay que seguir y actualizar.
- [ ] **¿Se mantiene?**
  Mira los lanzamientos recientes, los problemas de seguridad abiertos y cuántas personas lo mantienen. Un paquete abandonado se convierte en tu problema.
- [ ] **¿Qué trae consigo?**
  Comprueba cuántas dependencias transitivas añade; cada una es más código en el que confiar.
- [ ] **¿Es compatible la licencia?**
  Las licencias de código abierto son diferentes; algunas imponen obligaciones cuando distribuyes tu software.
- [ ] **¿Están las versiones fijadas y analizadas?**
  Usa un lockfile y una herramienta automática que avise cuando una vulnerabilidad conocida afecte a una versión que usas.
- [ ] **¿Podríamos sustituirlo?**
  Envuelve las dependencias importantes detrás de una interfaz propia, para que un cambio de proveedor afecte a un solo lugar.

## Por qué importa la estructura: el cambio y los equipos

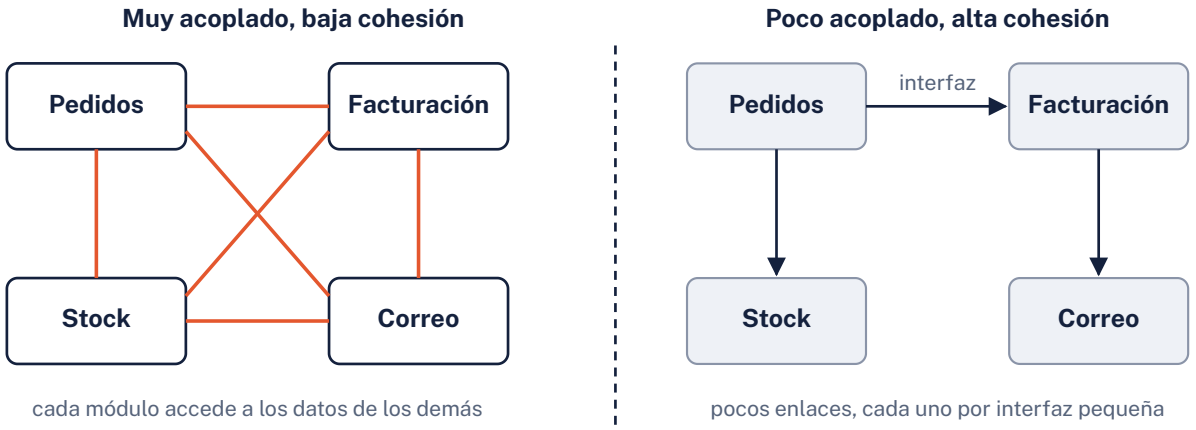
El código se lee y se cambia muchas más veces de las que se escribe por primera vez. La mayor parte del costo de un software a lo largo de su vida se dedica a cambiarlo: corregir errores, añadir funciones, adaptarse a nuevas leyes y a nuevos socios. La estructura decide lo caro que resulta. En un sistema bien organizado, un cambio en la forma de calcular el IVA afecta a un módulo. En uno mal organizado, afecta a cuarenta archivos de cinco equipos distintos, y algo que no tiene nada que ver se rompe.

Dos ideas, descritas por primera vez por Larry Constantine y sus colegas en el movimiento del diseño estructurado de finales de la década de 1960 y de la de 1970, recogen lo que significa "bien organizado". El acoplamiento es cuánto depende una parte de los detalles de otra: en un sistema muy acoplado, cambiar una parte obliga a cambiar otras. La cohesión es lo bien que

encaja entre sí el contenido de una parte: un módulo muy cohesivo hace una tarea clara. El objetivo es **bajo acoplamiento y alta cohesión**: partes que hacen bien una tarea cada una y que se comunican entre sí mediante interfaces pequeñas y estables.

La estructura también importa para las personas. Si dos equipos tienen que editar el mismo módulo para la mayor parte de su trabajo, se esperarán y se estorbarán constantemente. Si cada equipo es responsable de módulos cohesivos con interfaces claras, pueden trabajar y lanzar de forma independiente. Melvin Conway observó en 1968 que los sistemas tienden a reflejar la estructura de comunicación de las organizaciones que los construyen, una observación conocida como ley de Conway que se analiza en el Capítulo 3.

**El acoplamiento de un vistazo**




Los mismos cuatro módulos, conectados de dos formas: cada enlace adicional es otro camino por el que puede propagarse un cambio.

ATENCIÓN

## Errores comunes

- **Copiar y pegar en lugar de usar una función.** La lógica duplicada acaba siendo distinta, así que la misma regla da respuestas diferentes en lugares diferentes.
- **Todo público.** Si cada detalle de un módulo se puede usar desde fuera, cada detalle se convierte en algo de lo que depende otro código, y nada se puede cambiar con seguridad.
- **Código del framework por todas partes.** Cuando las reglas de negocio están enredadas con llamadas al framework, actualizar o sustituir el framework significa reescribir el negocio.
- **Dependencias sin examinar.** Añadir un paquete para una tarea trivial trae consigo todo su árbol de dependencias, sus errores y sus riesgos de seguridad.

## Por qué esto importa para la arquitectura

La arquitectura es en gran parte la misma actividad que esta sección, a mayor escala y con más en juego: decidir cuáles son las partes principales de un sistema, de qué se encarga cada una, qué interfaces las conectan y en qué dirección apuntan las dependencias. Un módulo dentro de un programa y un servicio desplegado por separado al otro lado de una red son, ambos, partes con interfaces; las cuestiones de acoplamiento, cohesión y ocultación de información se aplican a los dos.

El Capítulo 3 precisa estos principios, empezando por la modularidad y el acoplamiento y la cohesión, y el Capítulo 8 vuelve a las dependencias como cuestión de seguridad en la seguridad de la cadena de suministro. A continuación, el código sale de la computadora única: la siguiente sección explica cómo se comunican las computadoras entre sí.

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **¿Por qué es mejor llamar a una sola función `total_with_vat` desde tres lugares que escribir el cálculo tres veces?**

Porque así la regla está en un único lugar: cuando cambia el tipo de IVA, un solo cambio corrige a todos los que la llaman, y las copias no pueden acabar siendo distintas.

▼ **¿Qué impide el encapsulamiento que haga otro código con un objeto de cuenta bancaria?**

Impide que otro código cambie el saldo directamente, de modo que todo cambio tiene que pasar por métodos como withdraw, que hacen cumplir las reglas.

▼ **Un framework web llama a tu función cada vez que llega una solicitud para `/orders`. ¿Cómo se llama esta forma de organizarlo?**

Inversión de control: es el framework, no tu código, quien controla el flujo y llama a tu código cuando hace falta.

▼ **¿Para qué sirve un lockfile, como `package-lock.json`?**

Registra la versión exacta de cada paquete instalado, incluidos los indirectos, para que la máquina de cada desarrollador y cada compilación de producción usen exactamente el mismo código.

**FUENTES**


- Python tutorial: modules and packages: https://docs.python.org/3/tutorial/modules.html

- Python tutorial: classes: https://docs.python.org/3/tutorial/classes.html
- Martin Fowler: Inversion of Control: https://martinfowler.com/bliki/InversionOfControl.html
- Semantic Versioning 2.0.0: https://semver.org/
- npm documentation: package-lock.json: https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json

SECCIÓN 0.4 DE 0.8 · PRINCIPIANTE · 18 MIN DE LECTURA

# Redes, internet y la web: IP, DNS, HTTP y el modelo cliente-servidor

**LA IDEA EN BREVE**

Enviar datos por internet es como enviar correo postal. Tu mensaje se divide en paquetes, cada uno con la dirección del remitente y la del destinatario (direcciones IP) y un número de oficina (un puerto), y unos centros de clasificación (routers) los van pasando. Los nombres como shop.example se buscan en una enorme libreta de direcciones compartida (DNS) para encontrar la dirección real. Las cartas siguen un formato fijo (HTTP): una solicitud como "por favor, envíame el producto 42" y una respuesta con un código que dice cómo ha ido. HTTPS mete la carta en un sobre sellado e inviolable.

**LO QUE NECESITAS SABER**

- Los datos cruzan las redes en paquetes, que los routers reenvían salto a salto usando la dirección IP del destino; un número de puerto identifica el programa en esa máquina.
- TCP ofrece una conexión fiable y ordenada tras un handshake; UDP envía paquetes sueltos sin garantías, pero con menos retraso.
- DNS convierte nombres en direcciones IP mediante una jerarquía de servidores raíz, de dominio de nivel superior y autoritativos, con mucha caché controlada por los TTL.
- HTTP es un protocolo de solicitud y respuesta sin estado: un método (GET, POST, PUT, DELETE), una ruta, cabeceras y un cuerpo, a los que se responde con un código de estado (2xx, 3xx, 4xx, 5xx).
- HTTPS es HTTP sobre TLS, que cifra el tráfico, detecta manipulaciones y usa certificados firmados por autoridades de confianza para demostrar la identidad del servidor.
- En el modelo cliente-servidor, los clientes inician las solicitudes y los servidores guardan los datos y las reglas compartidos; los servidores nunca deben confiar ciegamente en los clientes.
- La latencia (retraso) y el ancho de banda (capacidad) son cosas distintas; la latencia está limitada por la velocidad de la luz, así que la distancia y los viajes de ida y vuelta importan.

## Redes y paquetes

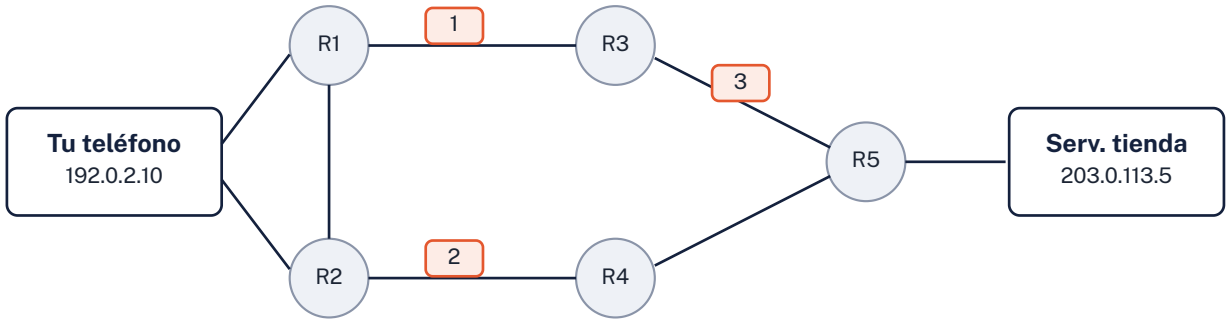
Casi ningún software moderno funciona solo. Una app de reparto de comida en tu teléfono se comunica con los servidores de la empresa, que se comunican con las tabletas de los restaurantes, un proveedor de pagos, un servicio de mapas y un servicio de notificaciones. Esta

sección explica cómo funciona esa comunicación, de abajo arriba.

Una **red** es un conjunto de computadoras conectadas para poder intercambiar datos, por cable, wifi, señal de telefonía móvil o enlace de fibra óptica. **Internet** es una red de redes: millones de redes con propietarios distintos (banda ancha doméstica, operadores de telefonía móvil, universidades, proveedores de nube) que han acordado transportar el tráfico de las demás usando reglas comunes llamadas *protocolos*.

Los datos no viajan como un flujo continuo. Se cortan en paquetes: trozos de normalmente hasta unos 1500 bytes, cada uno etiquetado con su origen y su destino, como un paquete postal con la dirección del remitente y la del destinatario. Unos dispositivos llamados **routers** pasan los paquetes de una red a otra, y cada uno elige un buen siguiente salto, igual que los centros de clasificación hacen avanzar los paquetes postales hacia su destino. Distintos paquetes del mismo mensaje pueden tomar rutas diferentes, llegar desordenados u ocasionalmente perderse, y la computadora que los recibe los vuelve a montar. Dividir los datos en paquetes permite que millones de conversaciones compartan los mismos cables de forma equitativa, y que el tráfico rodee un enlace roto.

**Los paquetes encuentran su propio camino**




Un mensaje se divide en paquetes numerados; cada uno encuentra su camino entre routers y se vuelve a montar al final

Ningún cable único conecta tu teléfono con el servidor; los paquetes saltan entre routers que pertenecen a muchas redes distintas.

## Direcciones: direcciones IP y puertos

Para que un paquete llegue, el destino necesita una dirección. Cada dispositivo de internet se alcanza mediante una dirección IP (IP significa Internet Protocol, protocolo de internet). La versión antigua, IPv4, escribe las direcciones como cuatro números del 0 al 255, por ejemplo `192.0.2.10`; admite unos 4.3 mil millones de direcciones, que no bastan para cada teléfono, computadora portátil y sensor del mundo. La versión más reciente, IPv6, usa direcciones mucho más largas escritas en hexadecimal (base 16, con los dígitos del 0 al 9 y las letras de la a a la f), por ejemplo `2001:db8::1`, y tiene espacio para una cantidad prácticamente ilimitada

de dispositivos. Ambas se usan en paralelo. (Las direcciones de ejemplo de esta sección proceden de rangos reservados para documentación, así que no pertenecen a ninguna máquina real).

Una dirección IP lleva un paquete a la computadora correcta, pero una computadora ejecuta muchos programas a la vez. Un número de puerto, del 0 al 65 535, indica qué programa debe recibirlo. Si la dirección IP es la dirección postal de un edificio de oficinas, el puerto es el número de la oficina. Los servicios conocidos tienen puertos estándar: 443 para HTTPS, 80 para HTTP simple, 53 para DNS, 5432 para la base de datos PostgreSQL. Un programa servidor *escucha* en un puerto, a la espera de conexiones; cuando tu navegador contacta con un sitio web, se conecta al puerto 443 de la dirección IP del sitio.

## TCP y UDP: dos formas de entregar

Por encima de IP hay dos protocolos de transporte principales, que deciden cuánto cuidado se pone en la entrega. TCP (Transmission Control Protocol) da a dos programas un flujo de bytes fiable y ordenado. Antes de enviar ningún dato, las dos partes establecen una conexión con un *handshake* de tres pasos. Cada paquete va numerado; el receptor confirma lo que ha llegado; lo que falta se vuelve a enviar; y todo se entrega al programa en el orden correcto. TCP también reduce la velocidad cuando la red está congestionada. Es como un servicio de correo certificado con seguimiento: fiable, pero con cierta sobrecarga.

UDP (User Datagram Protocol) simplemente envía paquetes sueltos, llamados datagramas, sin handshake, sin confirmaciones, sin reenvíos y sin orden. Es como enviar postales: barato y rápido, pero puede que algunas no lleguen. Eso es justo lo adecuado cuando un paquete que llega tarde no sirve de nada. En una videollamada, un fragmento de sonido que llega medio segundo tarde es peor que uno que nunca llega, así que las llamadas, las transmisiones en directo y los juegos en línea usan UDP y toleran pequeños huecos. Las consultas DNS, que son preguntas y respuestas diminutas, también usan tradicionalmente UDP.

**TCP y UDP comparados**

| | TCP | UDP |
|---|---|---|
| **Establecimiento de la conexión** | Sí: un handshake antes de que fluyan los datos | Ninguno: simplemente envía |
| **Entrega** | Garantizada: los paquetes perdidos se reenvían | Mejor esfuerzo: las pérdidas no se reparan |
| **Orden** | Se conserva | No garantizado |
| **Sobrecarga y retraso** | Mayor | Menor |
| **Usos habituales** | Páginas web y API (HTTP/1.1 y HTTP/2), correo electrónico, transferencia de archivos, conexiones a bases de datos | Llamadas de voz y video, transmisión en directo, juegos en línea, DNS |

PARA PROFUNDIZAR

### Para profundizar: QUIC y HTTP/3

La versión más reciente del protocolo de la web, HTTP/3 (publicada como RFC 9114 en 2022), no funciona sobre TCP. Funciona sobre QUIC (RFC 9000, 2021), un protocolo de transporte construido sobre UDP que vuelve a implementar por sí mismo la entrega fiable y combina el establecimiento de la conexión con el del cifrado, lo que ahorra viajes de ida y vuelta. Demuestra que la línea entre el "TCP fiable" y el "UDP no fiable" tiene que ver con lo que garantiza la capa de transporte, y que la fiabilidad puede construirse en una capa superior cuando eso sirve mejor a la aplicación.

## DNS: la libreta de direcciones de internet

Las personas recuerdan nombres como `shop.example`; las redes enrutan por direcciones IP. El sistema de nombres de dominio (DNS, Domain Name System) traduce lo uno en lo otro. Es un directorio mundial y distribuido: ninguna computadora contiene todos los nombres. En su lugar, la responsabilidad se reparte según los puntos del nombre, leídos de derecha a izquierda. Los servidores **raíz** saben quién es responsable de cada dominio de nivel superior, como `.com` o `.uk`; esos servidores de dominio de nivel superior saben qué servidores de nombres **autoritativos** son responsables de cada dominio registrado bajo ellos; y los servidores autoritativos contienen los registros reales, como la dirección IP de `www.shop.example`. La zona raíz la atienden 13 identidades de servidor con nombre, gestionadas por 12 organizaciones independientes mediante bastante más de mil instancias de servidor repartidas por todo el mundo.

Tu dispositivo no suele recorrer esta cadena por sí mismo. Pregunta a un **resolvedor recursivo**, gestionado normalmente por tu proveedor de internet o por un servicio público, que hace el trabajo pesado y recuerda (guarda en caché) las respuestas durante un periodo fijado por el propietario del dominio, llamado TTL (time to live, tiempo de vida). Como las respuestas se guardan en caché en muchos niveles, la mayoría de las consultas se responden en unos pocos milisegundos sin llegar siquiera a la raíz. Esa misma caché explica por qué un cambio en un registro DNS puede tardar minutos u horas en verse en todas partes.

**La resolución DNS paso a paso (sin nada en caché)**

1. **Preguntar al resolvedor**
   Tu dispositivo pregunta a su resolvedor recursivo configurado: ¿cuál es la dirección IP de `www.shop.example`?

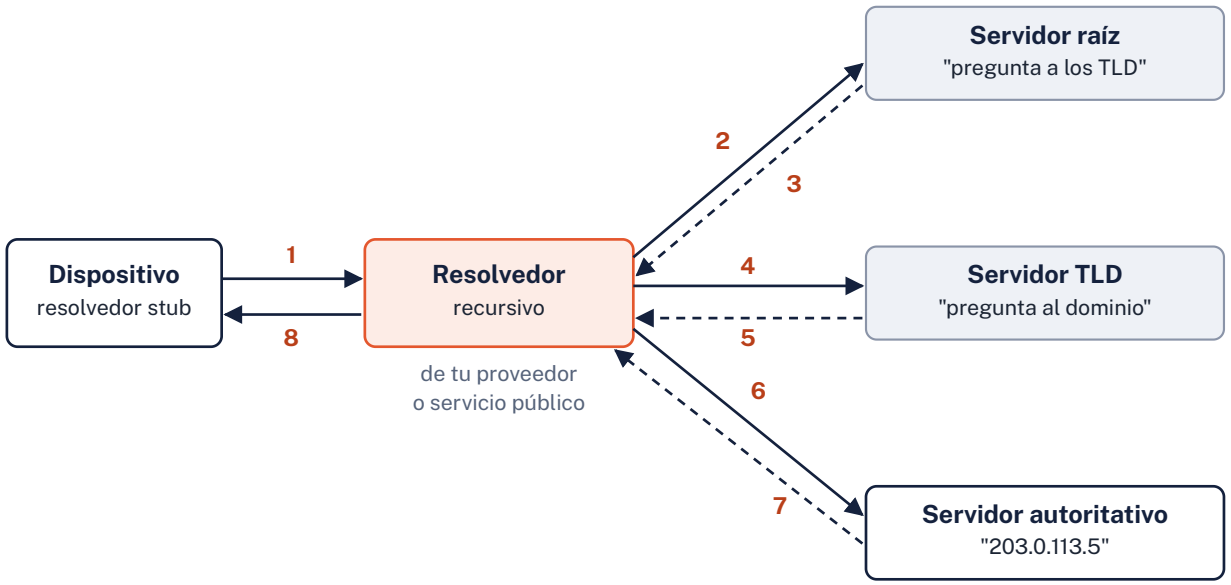
2. **Preguntar a un servidor raíz**
   El resolvedor pregunta a un servidor raíz, que no sabe la respuesta pero responde con los servidores responsables del dominio de nivel superior.

3. **Preguntar al servidor del dominio de nivel superior**
   El resolvedor pregunta a uno de esos servidores, que responde con los servidores de nombres autoritativos de `shop.example`.

4. **Preguntar al servidor autoritativo**
   El resolvedor pregunta al servidor autoritativo, que responde con el registro: por ejemplo, `203.0.113.5`, válido durante 300 segundos.

5. **Responder y recordar**
   El resolvedor devuelve la dirección a tu dispositivo, y ambos la guardan en su caché hasta que vence el TTL, así que la siguiente consulta es casi instantánea.

**Quién responde a una pregunta DNS**




Las flechas continuas son preguntas y las discontinuas, respuestas; gracias a la caché, la mayoría de las consultas reales se saltan los pasos 2 a 7.

## HTTP: solicitudes y respuestas

Una vez que un programa puede llegar a otro, los dos necesitan un lenguaje común para decir lo que quieren. En la web, ese lenguaje es HTTP (Hypertext Transfer Protocol). Sigue un patrón sencillo: el cliente envía una **solicitud**; el servidor devuelve una **respuesta**. Cada solicitud indica un **método** (qué hacer), una **ruta** (sobre qué), algunas **cabeceras** (información adicional en forma de pares nombre-valor) y, a veces, un **cuerpo** (los datos que se envían). Cada respuesta lleva un código de estado, cabeceras y normalmente un cuerpo: una página web, una imagen o datos en un formato como JSON (un formato de texto sencillo para datos estructurados, que se explica en la sección sobre las API).

Los métodos más comunes son `GET` (obtener algo, sin cambiarlo), `POST` (enviar algo nuevo, como hacer un pedido), `PUT` (sustituir algo en una dirección conocida, como actualizar una dirección de entrega) y `DELETE` (eliminar algo). GET, PUT y DELETE se definen como *idempotentes*: enviar dos veces la misma solicitud tiene el mismo efecto que enviarla una vez, lo que hace seguro reintentarlas tras un fallo de red. POST no lo es, y por eso un reintento descuidado puede hacer un pedido dos veces. Las cabeceras llevan cosas como el tipo de contenido (`Content-Type: application/json`), el idioma que prefiere el usuario, las cookies (pequeños datos que un sitio pide al navegador que guarde y devuelva con las solicitudes posteriores) y las credenciales, e instrucciones de caché.

HTTP es *sin estado*: cada solicitud se entiende por sí sola, y el servidor no tiene por qué recordar las anteriores. Las sesiones con usuario identificado se construyen por encima, normalmente porque el cliente envía una cookie o un token con cada solicitud. La ausencia de estado es una de las razones por las que la web escala tan bien, porque cualquiera de muchos servidores idénticos puede atender cualquier solicitud.

```
POST /api/orders HTTP/1.1
Host: shop.example
Content-Type: application/json
Authorization: Bearer eyJhbGciOi...

{"product_id": 42, "quantity": 2}

HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/orders/9917

{"order_id": 9917, "status": "received", "total_pence": 3000}
```

Un intercambio HTTP realista: hacer un pedido y, después, la respuesta del servidor

**Códigos de estado por familia**

| Familia | Significado | Ejemplos habituales |
|---|---|---|
| **2xx** | Éxito | 200 OK; 201 Created (se ha creado algo nuevo); 204 No Content |
| **3xx** | Redirección: busca en otro sitio | 301 Moved Permanently; 302 Found (traslado temporal); 304 Not Modified (tu copia en caché sigue siendo válida) |
| **4xx** | Error del cliente: la solicitud era incorrecta | 400 Bad Request; 401 Unauthorized (sin sesión iniciada); 403 Forbidden (con sesión iniciada, pero sin permiso); 404 Not Found; 429 Too Many Requests |
| **5xx** | Error del servidor: la solicitud era correcta, pero el servidor falló | 500 Internal Server Error; 502 Bad Gateway; 503 Service Unavailable; 504 Gateway Timeout |

La división entre 4xx y 5xx importa en la práctica. Un 4xx dice "no te limites a reintentar; corrige la solicitud". Un 5xx dice "el problema está de nuestro lado"; reintentar más tarde puede funcionar, y hay que avisar al equipo de operaciones cuando aumenta la tasa de 5xx.

## HTTPS y TLS: privacidad e identidad

El HTTP simple envía todo como texto legible, así que cualquiera que esté en el camino, como alguien conectado al mismo wifi de una cafetería, podría leerlo o alterarlo. HTTPS es HTTP enviado a través de un canal protegido creado por **TLS** (Transport Layer Security; la versión actual, TLS 1.3, se publicó como RFC 8446 en 2018). TLS aporta tres cosas: *confidencialidad* (quien escucha a escondidas solo ve datos codificados), *integridad* (se detectan las manipulaciones) y *autenticación* (de verdad estás hablando con `shop.example`, no con un impostor).

La autenticación se basa en **certificados**. El certificado de un sitio es un documento digital que dice, en esencia, "esta clave pública pertenece a `shop.example`", firmado por una **autoridad de certificación** (CA) en la que tu navegador o tu teléfono ya confían. Funciona como un pasaporte: lo emite una autoridad en la que ambas partes confían, se comprueba frente a su titular y solo es válido hasta su fecha de caducidad. Durante el handshake de TLS, el servidor presenta su certificado, el cliente comprueba la firma, el nombre y las fechas, y las dos partes acuerdan claves secretas nuevas para cifrar el resto de la conversación. Si falla alguna comprobación, el navegador muestra una advertencia.

TLS protege los datos *en tránsito* entre dos puntos. No protege los datos una vez almacenados, ni demuestra que el sitio sea honrado, solo que es quien dice ser. Hoy HTTPS es la norma para todo el tráfico web, no solo para los pagos, y los certificados a menudo se emiten gratis y se renuevan automáticamente.

## El modelo cliente-servidor

La mayoría de los sistemas de este curso siguen el modelo cliente-servidor. Un **cliente** (un navegador, una app móvil, otro programa) inicia la conversación enviando una solicitud. Un **servidor** espera solicitudes y las responde. El servidor guarda lo que debe ser compartido y fiable: el catálogo de productos, los pedidos de todos, las reglas sobre precios y reembolsos. El cliente se ocupa de la pantalla del usuario y de lo que este introduce.

En un restaurante, los comensales son los clientes y la cocina es el servidor: los comensales inician los pedidos; la cocina tiene los ingredientes y el oficio, y atiende a muchos comensales a la vez. Una consecuencia es que el servidor nunca debe confiar ciegamente en el cliente. El usuario puede modificar una app de su teléfono, así que los precios, los permisos y los niveles de stock deben comprobarse en el servidor, por mucho cuidado que ponga también la app al comprobarlos.

A menudo un servidor también es cliente: el servidor web de la tienda es cliente de su base de datos y del proveedor de pagos. Los sistemas reales son cadenas y redes de relaciones de este tipo, como muestra el diagrama de secuencia de más abajo.

## Latencia y ancho de banda

Dos números describen lo rápida que se siente una red, y a menudo se confunden. El ancho de banda es cuántos datos puede transportar una conexión por segundo, por ejemplo 100 megabits por segundo: el ancho de la tubería. La latencia es cuánto tarda un dato en ir de un extremo al otro, y suele medirse como **tiempo de ida y vuelta** (RTT, round-trip time) en milisegundos: la longitud de la tubería. Ayuda una analogía con una autopista: añadir carriles (ancho de banda) deja pasar más vehículos por hora, pero no acorta el trayecto de ningún vehículo concreto (latencia).

La latencia tiene un mínimo fijado por la física. La luz en la fibra óptica viaja a unos 200 000 km por segundo, alrededor de dos tercios de su velocidad en el vacío. Londres y Sídney están a unos 17 000 km de distancia en línea recta sobre la superficie del globo, así que incluso una fibra perfecta a lo largo de esa línea necesitaría unos 85 ms de ida y 170 ms de ida y vuelta. Los cables reales no siguen el camino más corto y los routers añaden retraso, de modo que los viajes de ida y vuelta medidos rondan normalmente los 250 ms. Ninguna mejora puede eliminar eso. Si cargar una página desde Sídney requiere diez viajes de ida y vuelta secuenciales a un servidor en Londres, el usuario espera dos segundos y medio, por muy rápida que sea su banda ancha.

EJEMPLO PRÁCTICO

## Ejemplo práctico: un servidor en Londres, un cliente en Sídney

Una tienda aloja todo en Londres. Un cliente en Sídney abre la página de un producto a través de una nueva conexión HTTPS. Supón un viaje de ida y vuelta de 250 ms e ignora el tiempo de procesamiento del servidor.

| | |
|---|---|
| **Consulta DNS (sin caché)** | Puede que el resolvedor necesite varias consultas; pongamos 1 viaje de ida y vuelta a servidores cercanos: unos 50 ms |
| **Handshake de TCP** | 1 viaje de ida y vuelta a Londres: 250 ms |
| **Handshake de TLS 1.3** | 1 viaje de ida y vuelta: 250 ms |
| **Solicitar la página HTML** | 1 viaje de ida y vuelta: 250 ms |
| **Solicitar el CSS y los scripts que necesita la página** | Al menos 1 viaje de ida y vuelta más: 250 ms |
| **Total antes de que aparezca algo útil** | Unos 1050 ms, y el ancho de banda apenas influye |
| **El mismo cliente, con el contenido servido desde Sídney** | Viaje de ida y vuelta de quizá 10 ms: los mismos cinco pasos tardan unos 50 ms |

En las páginas pequeñas domina la latencia, no el ancho de banda. Mejorar la banda ancha del cliente de 50 a 500 megabits por segundo apenas cambiaría el resultado.

Los arquitectos reducen la latencia recortando viajes de ida y vuelta (reutilizando conexiones, combinando solicitudes) y acercando el contenido a los usuarios, por ejemplo con redes de distribución de contenidos, que se tratan en Servidores y la nube.

GUÍA DE DECISIÓN

### Guía de decisión: ¿una experiencia de red lenta se debe a la latencia o al ancho de banda?

Un primer diagnóstico aproximado cuando los usuarios se quejan de que algo va lento por la red.

**Q1** **¿También son lentas las solicitudes pequeñas (una llamada breve a una API, una página diminuta)?**

- **Sí** La causa probable es la latencia. Comprueba la distancia hasta el servidor y cuántos viajes de ida y vuelta secuenciales hace la página o la app.
- **No** Pasa a la siguiente pregunta.

**Q2** **¿Solo son lentas las transferencias grandes (videos, descargas grandes, informes extensos)?**

- **Sí** El límite probable es el ancho de banda, en la conexión del usuario o en la del servidor. Plantéate la compresión, archivos más pequeños o una red de distribución de contenidos.
- **No** Pasa a la siguiente pregunta.

**Q3** **¿El retraso es el mismo desde todas las ubicaciones?**

- **Sí** Puede que la red no sea el problema en absoluto: revisa el procesamiento en el servidor, las consultas a la base de datos y los algoritmos (consulta la sección sobre datos y algoritmos).
- **No** Un retraso que depende de la ubicación apunta a la latencia y al enrutamiento: atiende a los usuarios desde un lugar más cercano a donde están.

## Qué ocurre cuando escribes una dirección web

Todas estas piezas se combinan cada vez que visitas un sitio web. Esta es la secuencia cuando escribes `https://shop.example/products/42` y pulsas Intro, simplificada pero exacta.

#### De Intro a la página

1. **Analizar la dirección**
   El navegador divide la URL en el esquema (`https`, por tanto puerto 443 y TLS), el nombre de host (`shop.example`) y la ruta (`/products/42`).

2 **Encontrar la dirección IP**
El navegador consulta su propia caché, luego la del sistema operativo, y después pregunta al resolvedor recursivo, que consulta el DNS como se ha descrito antes y devuelve, por ejemplo, `203.0.113.5`.

3 **Abrir una conexión TCP**
El navegador y el servidor intercambian los tres mensajes del handshake para establecer una conexión fiable con el puerto 443.

4 **Establecer TLS**
El servidor presenta su certificado; el navegador comprueba que es válido para `shop.example` y que lo ha firmado una autoridad de confianza; las dos partes acuerdan las claves de cifrado.

5 **Enviar la solicitud HTTP**
El navegador envía `GET /products/42` con cabeceras como los idiomas aceptados y las cookies que tenga para el sitio.

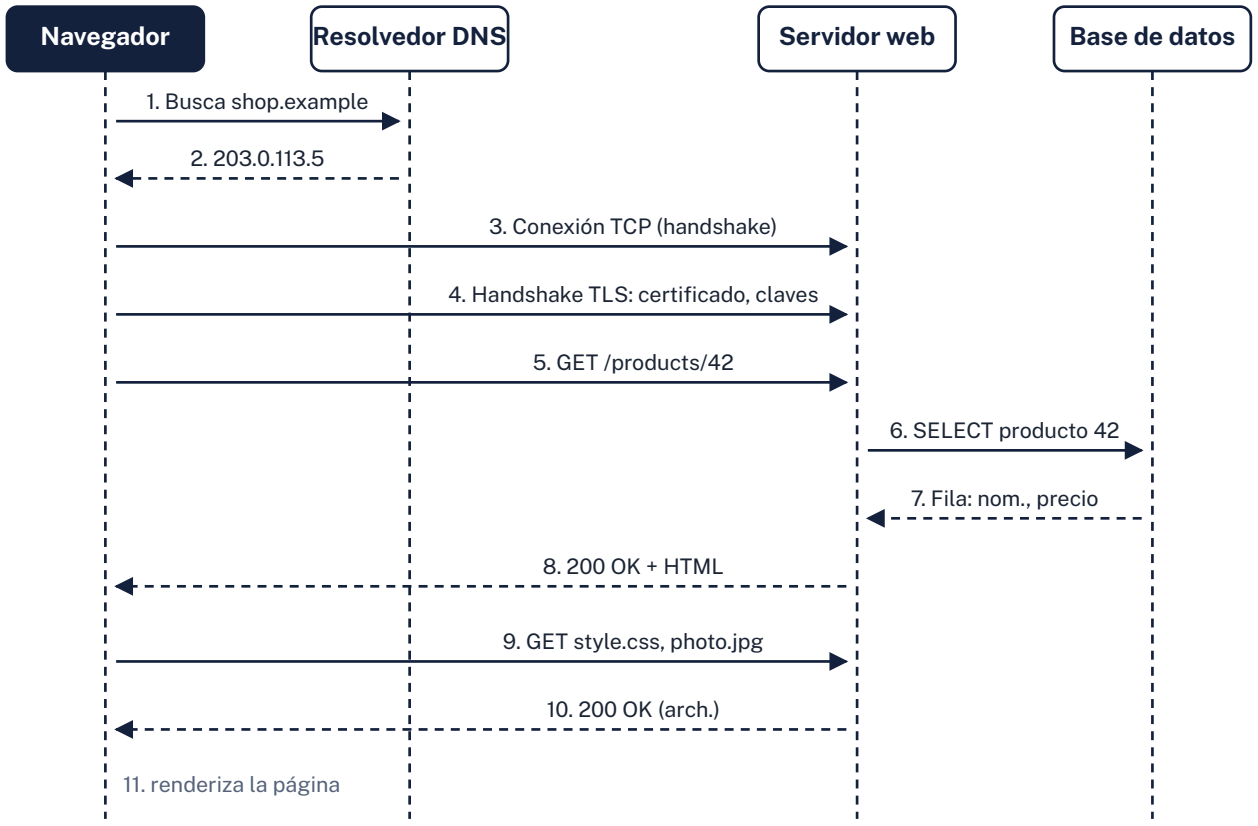
6 **El servidor hace su trabajo**
El software del servidor recibe la solicitud y ejecuta el código de la tienda, que consulta en una base de datos el producto 42, y construye una página HTML.

7 **Recibir la respuesta**
El servidor responde `200 OK` con el HTML en el cuerpo y cabeceras como `Content-Type: text/html` y reglas de caché.

8 **Obtener el resto**
El HTML hace referencia a hojas de estilo, scripts e imágenes; el navegador también los solicita, a menudo muchos a la vez por la misma conexión.

9 **Renderizar**
El navegador compone la página y la dibuja en pantalla; después, los scripts pueden hacer más solicitudes, por ejemplo para obtener reseñas.

**Diagrama de secuencia: carga de la página de un producto**




El tiempo avanza hacia abajo; las flechas continuas son solicitudes y las discontinuas, respuestas.

**ATENCIÓN**

## Ideas erróneas habituales

- **"Internet es rápido, así que la distancia no importa".** El ancho de banda ha crecido enormemente; la velocidad de la luz, no. Cada viaje de ida y vuelta a un servidor lejano cuesta tiempo real.
- **"HTTPS significa que el sitio es seguro".** Significa que la conexión es privada y que el sitio es quien dice ser. Un sitio fraudulento puede tener un certificado válido.
- **"Un 404 significa que el servidor está caído".** Un 404 es el servidor funcionando correctamente y diciendo que lo que se ha pedido no existe. Un servidor caído normalmente no da ninguna respuesta, o devuelve un 5xx desde algo que tiene delante.
- **"Cambiar el DNS es instantáneo".** Las respuestas en caché siguen vigentes hasta que vence su TTL, así que los cambios se propagan poco a poco.

## Por qué esto importa para la arquitectura

En cuanto un sistema se reparte entre más de una computadora, cada interacción entre sus partes se convierte en una llamada de red, con latencia, posibilidad de fallo y necesidad de seguridad. Una llamada a una función dentro de un programa tarda nanosegundos y nunca se pierde; una llamada a través de una red tarda milisegundos, a veces mucho más, y a veces nunca vuelve. Muchos de los problemas más difíciles de la arquitectura se derivan de esa diferencia, y por eso el Capítulo 6 empieza con las falacias de la computación distribuida, como la creencia de que la latencia es cero.

Los protocolos de esta sección también reaparecen como bloques de construcción: HTTP sustenta las API web de la siguiente parte de este capítulo y los estilos de comunicación del Capítulo 6, mientras que TLS y los certificados vuelven en comunicación segura y zero trust. Situar los servidores cerca de los usuarios forma parte de la arquitectura multirregión y en el borde.

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **¿Por qué necesita un paquete un número de puerto además de una dirección IP?**

La dirección IP lo lleva a la computadora correcta, pero esa computadora ejecuta muchos programas; el puerto indica qué programa debe recibirlo, como el servidor web que escucha en el 443.

▼ **¿Qué le dice al cliente un código de estado 3xx?**

Que lo que pidió está en otro lugar o no ha cambiado: por ejemplo, 301 significa que se ha trasladado de forma permanente, y 304, que la copia en caché del cliente sigue siendo válida.

▼ **¿Cuál es la diferencia entre una respuesta 401 y una 503?**

401 es un error del cliente: a la solicitud le faltan credenciales válidas. 503 es un error del servidor: el servicio no puede atender la solicitud temporalmente.

▼ **¿Por qué ninguna mejora de la red puede bajar a 10 ms un viaje de ida y vuelta entre Londres y Sídney?**

La luz en la fibra recorre unos 200 000 km por segundo, así que el viaje de ida y vuelta de aproximadamente 34 000 km tarda como mínimo unos 170 ms.

**FUENTES**

- MDN: An overview of HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview
- MDN: HTTP response status codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
- RFC 9110: HTTP Semantics: https://www.rfc-editor.org/rfc/rfc9110
- RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3: https://www.rfc-editor.org/rfc/rfc8446
- Root Server Technical Operations Association: https://root-servers.org/

SECCIÓN 0.5 DE 0.8 · PRINCIPIANTE · 18 MIN DE LECTURA

# Archivos y bases de datos: almacenamiento relacional y no relacional, consultas y transacciones

## LA IDEA EN BREVE

Una base de datos es como una biblioteca con un bibliotecario cuidadoso. En lugar de que cada programa meta papeles en su propio cajón, todos los datos van a un único lugar que mantiene un catálogo (un índice), comprueba que cada registro cumple las reglas, se asegura de que dos personas no puedan cambiar lo mismo a la vez y nunca pierde un cambio que ha prometido conservar. La mayoría de las bases de datos guardan los datos en tablas, como hojas de cálculo con reglas estrictas, y se les hacen preguntas en un lenguaje llamado SQL.

## LO QUE NECESITAS SABER

- Los archivos sirven para documentos completos, como imágenes y logs; las bases de datos sirven para datos estructurados que muchos usuarios leen y modifican a la vez, porque el DBMS se encarga de la concurrencia, las reglas, la búsqueda rápida y la recuperación tras una caída.
- Las bases de datos relacionales guardan los datos en tablas de filas y columnas; una clave primaria identifica cada fila y una clave foránea enlaza una fila con una fila de otra tabla.
- SQL es declarativo: indicas el resultado que quieres, como un SELECT filtrado o un JOIN entre tablas, y el planificador de consultas de la base de datos decide cómo obtenerlo.
- Un índice permite a la base de datos encontrar filas en O(log n) pasos en lugar de recorrer toda la tabla, a costa de más almacenamiento y escrituras más lentas.
- Una transacción hace que varios cambios tengan éxito o fallen juntos; ACID (atomicidad, consistencia, aislamiento, durabilidad) describe sus garantías, como en una transferencia bancaria que nunca puede quedarse a medias.
- Las familias NoSQL (documental, clave-valor, de columnas anchas y de grafos) se adaptan a formas de datos y patrones de acceso concretos; muchos sistemas usan una base de datos relacional más una o varias de ellas.
- Las copias de seguridad solo se demuestran restaurándolas, las réplicas no protegen contra los errores y las bases de datos gestionadas en la nube se encargan de la operación, pero no del diseño del esquema ni de las consultas.

## Por qué los programas necesitan un lugar donde guardar los datos

Un programa en ejecución mantiene sus datos de trabajo en memoria (RAM), que es rápida pero lo olvida todo cuando el programa se detiene o la máquina se reinicia (consulta qué es el software). Todo lo que deba sobrevivir a un reinicio, como la cuenta de un cliente, un pedido o una cita en un hospital, tiene que escribirse en un almacenamiento que conserve su contenido sin corriente: un disco duro o una unidad de estado sólido. Los datos que sobreviven de esta forma se llaman *persistentes*, y decidir cómo persistirlos es una de las primeras decisiones de cualquier sistema y de las que más duran.

La forma más sencilla de persistir datos es un archivo: una secuencia de bytes con nombre que el sistema operativo guarda en disco. Los archivos son excelentes para cosas que por naturaleza son documentos completos: imágenes, PDF, videos, configuración, logs, informes exportados. Los sistemas suelen guardarlos en un *almacenamiento de objetos* en la nube (un servicio que guarda archivos por nombre) y registrar en otro lugar solo una referencia.

Los archivos se vuelven un problema cuando muchos usuarios leen y modifican a la vez pequeños fragmentos de datos estructurados. Imagina una tienda que guarda todos sus pedidos en un único archivo de hoja de cálculo. Dos clientes pagan en el mismo instante: ¿qué cambio prevalece? El servidor se cae a mitad del guardado: ¿el archivo queda mitad antiguo, mitad nuevo? Un agente de soporte quiere los pedidos de un cliente: ¿tiene el programa que leer el archivo entero? Alguien escribe un número de cliente que no existe: ¿quién se lo impide? Cada programa que usara el archivo tendría que resolver cada uno de estos problemas, y de la misma manera.

## Qué es una base de datos

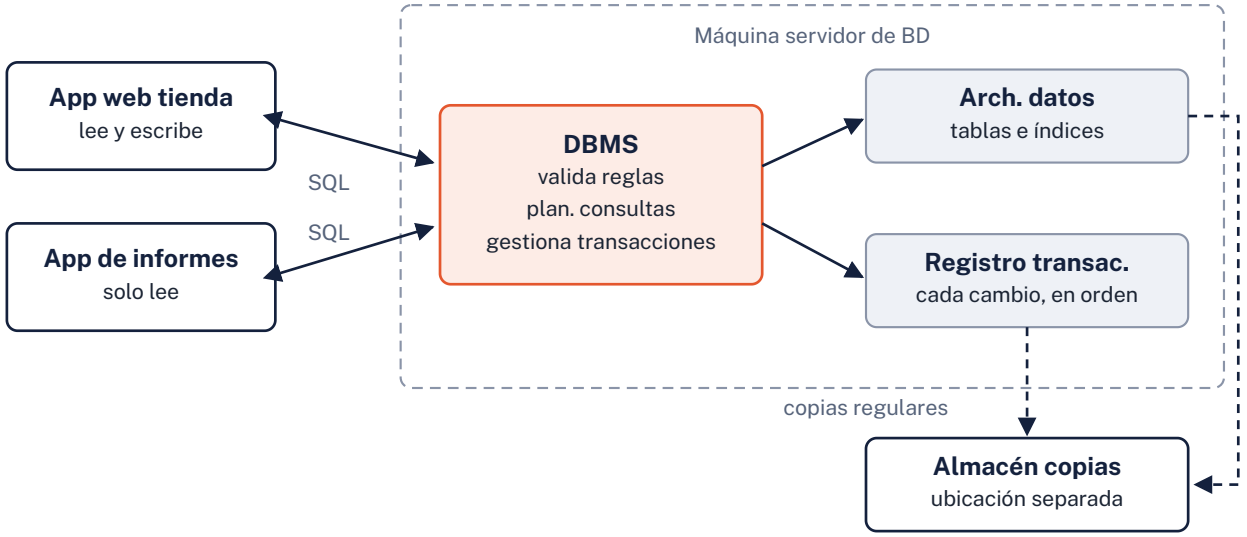
Una base de datos es una colección organizada de datos, y el programa que se ocupa de ella es un sistema de gestión de bases de datos (DBMS), como PostgreSQL, MySQL, Microsoft SQL Server, Oracle Database o SQLite. A menudo se dice "la base de datos" para ambas cosas. El DBMS resuelve los problemas de los archivos una sola vez, para todos: permite que muchos usuarios trabajen a la vez sin pisarse, encuentra los datos rápidamente, rechaza los datos que incumplen las reglas y vuelve a un estado correcto tras una caída.

La analogía es una biblioteca. Un montón de papeles sobre una mesa funciona si eres su único usuario. Una biblioteca tiene un catálogo, reglas sobre dónde va cada cosa, un registro de quién se ha llevado qué y un bibliotecario que no deja que dos personas se lleven el mismo ejemplar. Las aplicaciones no tocan las estanterías; se lo piden al bibliotecario.

En la mayoría de los sistemas empresariales, el DBMS se ejecuta como un proceso servidor propio, a menudo en su propia máquina, y las aplicaciones se comunican con él a través de la red (consulta cómo se comunican las computadoras). SQLite es una excepción muy conocida:

es una biblioteca que se ejecuta dentro de la aplicación y guarda toda la base de datos en un único archivo, por eso se encuentra en teléfonos, navegadores y pequeñas herramientas.

**Dónde viven realmente los datos**




Las aplicaciones nunca escriben directamente en los archivos de la base de datos: envían solicitudes al DBMS, que escribe los archivos de datos y un registro, y un proceso aparte copia ambos al almacenamiento de copias de seguridad.

## Bases de datos relacionales: tablas, claves y relaciones

El tipo de base de datos más utilizado es la base de datos relacional, basada en un modelo publicado por Edgar F. Codd, de IBM, en 1970. Los datos viven en **tablas**. Cada tabla describe un tipo de cosa (clientes, pedidos, productos). Cada **fila** es una instancia de esa cosa (un cliente), y cada **columna** es un dato sobre ella (nombre, dirección de correo electrónico, fecha de alta), con un tipo declarado, como entero, texto, fecha o decimal.

Cada tabla tiene una clave primaria: una columna (o combinación de columnas) cuyo valor es único para cada fila y nunca está vacío, como `customer_id`. Una clave foránea es una columna de una tabla que contiene la clave primaria de una fila de otra tabla. Un pedido tiene una columna `customer_id`, que indica qué cliente lo hizo. Declararla como clave foránea le indica al DBMS que rechace un pedido para el cliente 999 si no existe ningún cliente 999: esto se llama *integridad referencial*.

El conjunto completo de tablas, columnas, tipos, claves y reglas es el esquema. El esquema es un contrato: cada programa que usa la base de datos puede confiar en que un pedido siempre tiene un cliente válido y un precio que es un número. Cambiar un esquema más adelante, cuando millones de filas y varios programas dependen de él, es posible pero exige cuidado, y por eso la evolución del esquema tiene su propia sección en el Capítulo 7.

Los diseños relacionales intentan guardar cada dato una sola vez. La dirección de correo electrónico del cliente vive en una fila de la tabla de clientes, no copiada en cada pedido. Esto se llama *normalización*. Significa que una actualización ocurre en un solo lugar y no puede dejar atrás copias contradictorias. El precio de este orden es que responder a una pregunta a menudo obliga a combinar tablas, que es para lo que sirven los joins.

## Hacer preguntas con SQL

SQL (Structured Query Language, que se pronuncia "ese-cu-ele" o, en inglés, "sequel") es el lenguaje que se usa para trabajar con bases de datos relacionales. Se desarrolló en IBM en la década de 1970 y se convirtió en norma ANSI e ISO a finales de la década de 1980; todos los DBMS relacionales importantes lo hablan, cada uno con pequeñas diferencias de dialecto. SQL es *declarativo*: describes el resultado que quieres, no los pasos para encontrarlo. El *planificador de consultas* del DBMS elige los pasos, por ejemplo si leer una tabla entera o usar un índice.

Un join combina filas de dos o más tablas haciendo coincidir columnas, normalmente una clave foránea con la clave primaria a la que remite. La segunda consulta de abajo lista cada línea de los pedidos de un cliente junto con el nombre del producto, usando tres tablas a la vez.

```
-- One customer's orders since 1 October, newest first
SELECT order_id, status, placed_at
FROM orders
WHERE customer_id = 42
  AND placed_at ⩾ '2026-10-01'
ORDER BY placed_at DESC;

-- Each order line with the product's name (a join)
SELECT o.order_id, p.name, i.quantity, i.unit_price
FROM orders AS o
JOIN order_items AS i ON i.order_id = o.order_id
JOIN products AS p ON p.product_id = i.product_id
WHERE o.customer_id = 42;
```

Dos consultas sobre el esquema de la tienda que se muestra más adelante en esta sección: un SELECT filtrado y un JOIN de tres tablas.

`INSERT`, `UPDATE` y `DELETE` modifican datos, y `CREATE TABLE` define el esquema. El código de las aplicaciones suele enviar SQL a través de un *mapeador objeto-relacional* (ORM), una biblioteca que convierte filas en objetos. Los ORM ahorran escritura, pero ocultan el SQL que realmente se envía; un problema de rendimiento clásico es un ORM que lanza sin avisar una consulta por fila en lugar de un único join.

## Índices: el índice del libro, no el libro entero

Sin ayuda, la única forma que tiene la base de datos de encontrar todos los pedidos del cliente 42 es leer cada fila de la tabla de pedidos y comprobarla una por una: un recorrido completo de la tabla, que es O(n) en la notación de datos y algoritmos. Un índice de base de datos es una estructura de datos adicional y ordenada que relaciona los valores de una columna con las filas que los contienen, igual que el índice al final de un libro relaciona palabras con números de página. La mayoría de las bases de datos relacionales usan por defecto un *árbol B* (B-tree), un árbol ancho, poco profundo y siempre equilibrado que encuentra un valor en O(log n) pasos.

Las claves primarias se indexan automáticamente. Los demás índices se añaden de forma deliberada, para las columnas por las que las consultas filtran, combinan u ordenan: `CREATE INDEX idx_orders_customer ON orders (customer_id);`. Los índices no son gratis. Cada inserción, actualización o borrado debe actualizar también cada índice de la tabla, y cada índice ocupa espacio en disco y memoria. Una tabla con quince índices lee rápido y escribe despacio. Elegir índices es un compromiso guiado por las consultas que el sistema ejecuta realmente, lo que constituye un primer ejemplo del compromiso entre espacio y tiempo que atraviesa toda la arquitectura.

EJEMPLO PRÁCTICO

### Ejemplo práctico: encontrar los pedidos de un cliente

Una tienda tiene 10 millones de filas en su tabla de pedidos. La pantalla de atención al cliente pide todos los pedidos de un cliente, unas 20 filas.

| | |
|---|---|
| **Sin índice** | Recorrido completo de la tabla: la base de datos lee y comprueba las 10 000 000 de filas para encontrar las 20 que coinciden. |
| **Con un índice de árbol B sobre customer_id** | Cada página del índice contiene cientos de claves, así que el árbol solo tiene tres o cuatro niveles de profundidad. La base de datos lee unas pocas páginas del índice y después recupera las 20 filas que coinciden. |
| **Crecimiento** | Si la tabla crece hasta 100 millones de filas, el recorrido hace diez veces más trabajo; la búsqueda en el índice añade como mucho un nivel más. |
| **Costo pagado en otra parte** | Cada pedido nuevo escribe ahora en la tabla y en el índice. El uso de disco crece en el tamaño del índice. |

Esto es O(n) frente a O(log n), de la Sección 0.2: a menudo, la diferencia entre una respuesta en segundos y una en milisegundos. Muchas quejas por consultas lentas en sistemas jóvenes se resuelven con el índice adecuado, no con hardware más grande.

## Transacciones y ACID: la transferencia bancaria

Mover 100 libras de la cuenta de Alice a la de Bob requiere dos cambios: restar 100 a Alice y sumar 100 a Bob. Si el servidor se cae después del primer cambio y antes del segundo, 100 libras han desaparecido. Si dos transferencias desde la cuenta de Alice se ejecutan a la vez, ambas podrían ver un saldo de 150 y ambas podrían tener éxito, dejándola en descubierto. Una transacción agrupa varias operaciones en una unidad de trabajo que o bien se completa por entero (*commit*) o bien no tiene ningún efecto (*rollback*).

Las garantías que ofrece una transacción se resumen como ACID, un acrónimo acuñado por Theo Härder y Andreas Reuter en 1983, a partir de trabajos anteriores de Jim Gray:

| Propiedad | Qué promete | En la transferencia bancaria |
|---|---|---|
| **Atomicidad** | Todos los cambios de la transacción se producen, o no se produce ninguno. | Nunca se carga dinero a Alice sin abonárselo a Bob. |
| **Consistencia** | Una transacción lleva la base de datos de un estado válido a otro; las reglas, como las claves y las comprobaciones, nunca quedan incumplidas. | Una regla según la cual los saldos no pueden bajar de cero rechaza una transferencia que Alice no puede pagar. |
| **Aislamiento** | Las transacciones que se ejecutan a la vez no ven el trabajo a medio terminar de las demás; el resultado es como si se hubieran ejecutado una tras otra (en la medida que fije el nivel de aislamiento). | Dos transferencias simultáneas desde la cuenta de Alice no pueden basarse ambas en el mismo saldo antiguo. |
| **Durabilidad** | Una vez que la base de datos confirma un commit, el cambio sobrevive a una caída o a un corte de corriente. | Cuando la app dice "Transferencia completada", el dinero se ha movido de verdad, porque el cambio ya está en el registro en disco. |

```
import sqlite3

conn = sqlite3.connect("bank.db")
# The accounts table was created with CHECK (balance ⩾ 0)

def transfer(from_id, to_id, amount):
    try:
        with conn:  # one transaction: commit on success, rollback on error
            debit = conn.execute(
                "UPDATE accounts SET balance = balance - ? WHERE id = ?",
                (amount, from_id))
            credit = conn.execute(
                "UPDATE accounts SET balance = balance + ? WHERE id = ?",
                (amount, to_id))
            if debit.rowcount ≠ 1 or credit.rowcount ≠ 1:
                raise ValueError("No such account")  # forces a rollback
        return "Transfer complete"
    except (sqlite3.Error, ValueError):
        return "Transfer refused: no account was changed"
```

Una transferencia con el módulo sqlite3 integrado en Python. El bloque with hace commit si ambas actualizaciones tienen éxito y hace rollback si alguna falla o si una cuenta no existe.

Fíjate en la comprobación de `rowcount` (el número de filas que ha cambiado cada actualización): para la base de datos, una actualización que no coincide con ninguna cuenta no es un error, así que el propio código debe tratarla como tal. Hay otros dos detalles importantes más allá de este ejemplo. Los marcadores `?` pasan los valores por separado del texto SQL, que es la defensa estándar contra la *inyección SQL*, un ataque en el que la entrada del usuario se cuela en una consulta como si fuera código (el Capítulo 8 vuelve sobre esto). Y la promesa de durabilidad se cumple gracias a un *registro de escritura anticipada* (write-ahead log): el DBMS anota cada cambio en un registro en disco al que solo se añaden datos antes de confirmar el commit, de modo que tras una caída puede repetir o deshacer trabajo hasta llegar a un estado correcto.

El aislamiento es la propiedad más sutil. El aislamiento completo (llamado *serializable*) es seguro pero limita cuánto trabajo puede ejecutarse en paralelo, así que las bases de datos ofrecen *niveles de aislamiento* más débiles que cambian algo de protección por velocidad, y muchas usan por defecto un nivel más débil. Qué anomalías permite cada nivel se trata en transacciones y aislamiento. Por ahora, la idea clave es que una sola base de datos ofrece garantías fuertes y bien conocidas dentro de una transacción, y que esas garantías son mucho más difíciles de conseguir cuando los datos están repartidos entre varias bases de datos o servicios.

## NoSQL: otras formas de datos

Las bases de datos relacionales no son la única opción. NoSQL es un nombre genérico, popularizado en un encuentro de 2009 en San Francisco, para las bases de datos que no usan el modelo relacional. Surgieron para manejar volúmenes muy grandes repartidos entre muchas máquinas, formas de datos flexibles y patrones de acceso que las tablas atienden con dificultad. El nombre dice lo que no son, así que abarca familias bastante distintas.

**Las cuatro principales familias NoSQL**

| Familia | Cómo guarda los datos | Encaja bien con | Ejemplos |
|---|---|---|---|
| **Base de datos documental** | Cada registro es un documento autónomo, normalmente similar a JSON, que puede anidar listas y subobjetos. Los documentos de una misma colección no tienen por qué compartir exactamente los mismos campos. | Datos que se leen y escriben como un todo: un catálogo de productos con atributos variados, un perfil de usuario, el contenido de una página web. | MongoDB, Couchbase, Firestore |
| **Almacén clave-valor** | Un diccionario gigante: una clave única se asocia a un valor en cuyo interior la base de datos no mira. La búsqueda es solo por clave. | Búsquedas simples muy rápidas: sesiones, carritos de la compra, cachés, feature flags. | Redis, Amazon DynamoDB, etcd |
| **Almacén de columnas anchas** | Filas identificadas por una clave, cada una con muchas columnas, agrupadas y repartidas (particionadas) entre muchas máquinas. Las tablas se diseñan en torno a las consultas que se van a ejecutar. | Enormes volúmenes de escrituras, como lecturas de sensores, logs de eventos e historiales de mensajería, entre muchos servidores y centros de datos. | Apache Cassandra, Apache HBase, Google Bigtable |
| **Base de datos de grafos** | Nodos (cosas) y aristas (relaciones), ambos con propiedades, como en los grafos de la Sección 0.2. | Preguntas sobre conexiones: amigos de amigos, redes de fraude, planificación de rutas, recomendaciones. | Neo4j, Amazon Neptune |

Los profesionales no se ponen de acuerdo sobre la opción por defecto. Un grupo dice que hay que empezar con una base de datos relacional salvo que haya un motivo claro para no hacerlo: SQL es ampliamente conocido, las transacciones y los joins son potentes, y los sistemas relacionales modernos también guardan bien documentos JSON. El otro grupo dice que hay que elegir desde el principio el almacén que encaje con el patrón de acceso, porque un

almacén documental o clave-valor puede ser más sencillo y escalar de forma más barata con la carga de trabajo adecuada. Ambos coinciden en que muchos sistemas reales usan más de un tipo, cada uno donde encaja, lo que se llama *persistencia políglota*. La sección del Capítulo 7 sobre cómo elegir una base de datos ofrece una forma estructurada de decidir.

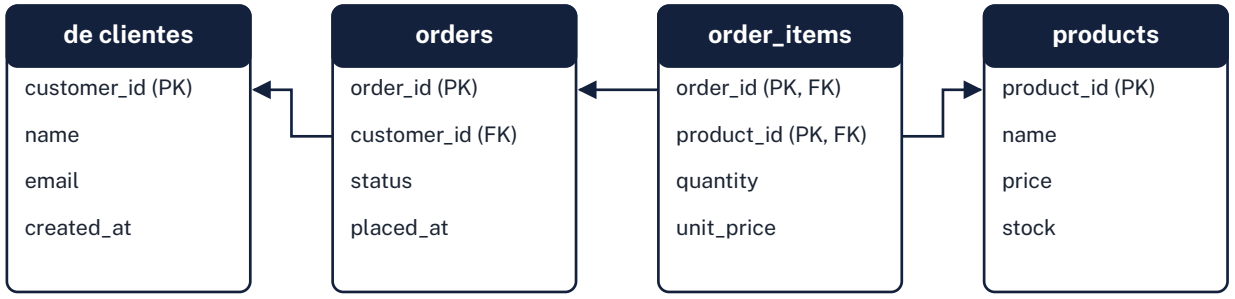
**ATENCIÓN**

### "Sin esquema" no significa "no hay esquema"

Una base de datos documental aceptará hoy un documento de cliente con `email` y mañana uno con `emailAddress`. Los datos siguen teniendo una forma; ahora la impone, o no, cada programa que los lee, en lugar de la base de datos. Los equipos que se saltan la reflexión sobre el esquema en los almacenes NoSQL suelen redescubrirla más tarde, en forma de código lleno de casos especiales para los documentos antiguos.

## Ejemplo práctico: el esquema de una tienda en línea

Piensa en una pequeña tienda en línea. Necesita saber quiénes son sus clientes, qué vende y quién ha pedido qué. Un primer intento podría ponerlo todo en una sola tabla con una fila por pedido y las columnas `product1`, `product2`, `product3`. Eso se rompe en cuanto alguien pide cuatro cosas, y copia el nombre y la dirección del cliente en cada pedido.




Cada flecha va de una clave foránea (FK) a la clave primaria (PK) a la que remite.

Un esquema de tienda normalizado: cada dato se guarda una sola vez y las claves foráneas registran las relaciones.

EJEMPLO PRÁCTICO

### Ejemplo práctico: por qué el esquema tiene cuatro tablas

El esquema anterior responde a las preguntas de la tienda sin copiar datos.

| | |
|---|---|
| **de clientes** | Una fila por persona. Cambiar una dirección de correo electrónico cambia una fila. |
| **orders** | Una fila por compra. `customer_id` es una clave foránea, así que cada pedido pertenece a un cliente real. Un cliente tiene muchos pedidos: una relación *uno a muchos*. |
| **products** | Una fila por artículo a la venta, con el precio y el nivel de existencias actuales. |
| **order_items** | Un pedido contiene muchos productos y un producto aparece en muchos pedidos: una relación *muchos a muchos*. Las bases de datos relacionales la representan con una tabla de enlace, con una fila por producto y pedido. |
| **Copia deliberada: unit_price** | El precio se copia en `order_items` cuando se hace el pedido. No es un error de normalización: el pedido debe registrar lo que el cliente pagó realmente, aunque el precio del catálogo cambie la semana que viene. |
| **Hacer un pedido** | Insertar el pedido, insertar sus artículos y reducir las existencias de cada producto ocurren en una sola transacción, de modo que una caída nunca puede dejar un pedido sin artículos ni unas existencias que no se hayan reducido. |

Un buen diseño de esquema parte de las preguntas que se hace el negocio y de los datos que nunca debe perder, no de las pantallas de la primera versión de la app.

La misma tienda en una base de datos documental podría guardar cada pedido como un documento con sus artículos anidados dentro, lo que facilita leer un pedido entero de una vez pero dificulta preguntas como "ventas totales por producto".

## Copias de seguridad, recuperación y bases de datos gestionadas

Los discos fallan, la gente ejecuta el `DELETE` equivocado, el software tiene errores y hay atacantes que cifran los datos para pedir un rescate. Una copia de seguridad es una copia de los datos guardada en un lugar separado, a partir de la cual se puede restaurar la base de datos. Hay tres hábitos profesionales importantes. Una copia de seguridad que nunca se ha restaurado es una esperanza, no una copia de seguridad, así que los equipos ensayan las restauraciones y miden cuánto tardan. Las copias se guardan en más de un lugar (una regla práctica habitual es tener tres copias, en dos tipos de soporte, una fuera de las instalaciones). Y una *réplica*, una segunda base de datos que copia continuamente la primera, protege contra el

fallo de una máquina, pero no contra los errores: un `DELETE` erróneo llega a la réplica en milisegundos. Muchas bases de datos pueden combinar una copia de seguridad completa periódica con el registro de transacciones guardado para restaurar a cualquier momento, lo que se llama *recuperación a un punto en el tiempo*.

Administrar bien una base de datos (aplicar parches, ajustarla, replicarla, hacer copias de seguridad y restaurarla a las tres de la mañana) es un trabajo especializado que nunca termina, así que los proveedores de nube venden bases de datos gestionadas: Amazon RDS, Azure SQL Database y Google Cloud SQL son ejemplos. El proveedor se encarga de las máquinas, las actualizaciones de software, las copias de seguridad automáticas y el failover a una copia en reserva; el cliente sigue siendo responsable del esquema, las consultas, los índices, los permisos de acceso y el costo. Los servicios gestionados se tratan de forma más amplia en servidores y la nube.

## Por qué esto importa para la arquitectura

Los datos suelen durar más que el código que los rodea: las aplicaciones se reescriben cada pocos años, mientras que los registros de clientes y los historiales médicos se conservan durante décadas. Las decisiones sobre dónde viven los datos, qué tipo de base de datos los guarda, qué programa es su propietario y qué garantías necesitan son, por tanto, de las más costosas de revertir. Varias grandes cuestiones de arquitectura surgen directamente de esta sección: si cada servicio debe tener su propia base de datos (propiedad de los datos), qué base de datos sirve para cada trabajo (cómo elegir una base de datos) y cómo mantener los datos correctos cuando una acción de negocio afecta a varias bases de datos, donde ya no se dispone de una única transacción ACID (sagas). Cuando un arquitecto pregunta "¿dónde está la fuente de verdad de esto?", esta sección es el motivo.

**COMPRUEBA LO QUE HAS ENTENDIDO**

**▼ ¿Por qué un archivo de hoja de cálculo es un mal lugar para guardar los pedidos de una tienda en línea cuando muchos clientes compran a la vez?**

Varios escritores simultáneos pueden sobrescribirse entre sí, una caída a mitad de un guardado puede corromper el archivo, encontrar los pedidos de un cliente obliga a leerlo todo y nada impone reglas como que cada pedido tenga un cliente real. Un DBMS resuelve los cuatro problemas.

**▼ En el esquema de la tienda, ¿qué hace que customer_id en la tabla orders sea una clave foránea, y qué te aporta?**

Contiene la clave primaria de una fila de la tabla customers y está declarada como referencia a ella. Así, la base de datos rechaza los pedidos de clientes que no existen.

**▼ ¿Por qué no indexar simplemente todas las columnas de todas las tablas?**

Cada índice debe actualizarse en cada inserción, actualización y borrado, y ocupa espacio. Demasiados índices hacen que las escrituras sean lentas y el almacenamiento caro, así que los índices se eligen según las consultas que se ejecutan realmente.

**▼ ¿Qué propiedad ACID se incumple si una caída deja a Alice con el cargo hecho pero a Bob sin el abono?**

La atomicidad: los dos cambios deberían haberse producido juntos o no producirse en absoluto.

**▼ Un equipo dice: "tenemos una réplica en otro centro de datos, así que no necesitamos copias de seguridad". ¿Qué se les ha pasado por alto?**

Una réplica copia los errores y la corrupción con la misma fidelidad que los datos buenos, así que un borrado accidental o un ataque de ransomware llega a ella en cuestión de momentos. Las copias de seguridad, idealmente con recuperación a un punto en el tiempo, protegen contra eso.

**FUENTES**


- PostgreSQL documentation: Transactions (tutorial): https://www.postgresql.org/docs/current/tutorial-transactions.html
- PostgreSQL documentation: Indexes, introduction: https://www.postgresql.org/docs/current/indexes-intro.html
- PostgreSQL documentation: Joins between tables: https://www.postgresql.org/docs/current/tutorial-join.html
- Martin Fowler: NosqlDefinition: https://martinfowler.com/bliki/NosqlDefinition.html

SECCIÓN 0.6 DE 0.8 · PRINCIPIANTE · 17 MIN DE LECTURA

# Las API, JSON y cómo trabajan juntos sistemas separados

**LA IDEA EN BREVE**

Una API es como la carta de un restaurante y su camarero. La carta te dice exactamente qué puedes pedir y qué vas a recibir; el camarero lleva tu pedido a la cocina y te trae el plato. Nunca entras en la cocina, y la cocina puede cambiar su forma de cocinar sin que te des cuenta, siempre que los platos de la carta sigan siendo los mismos. El software usa las API de la misma manera: un programa le pide algo a otro en un formato acordado y recibe una respuesta acordada.

**LO QUE NECESITAS SABER**

- Una API es un contrato entre piezas de software: qué se puede solicitar, con qué entradas y qué se devuelve, mientras que la implementación que hay detrás permanece oculta y libre para cambiar.
- Las API en proceso son llamadas a funciones dentro de un programa; las API web cruzan una red, así que son más lentas y pueden fallar de formas en que no puede fallar una llamada a una función.
- Las API REST exponen recursos en URL, actúan sobre ellos con métodos HTTP (GET, POST, PUT, PATCH, DELETE), comunican los resultados con códigos de estado y normalmente intercambian JSON.
- Las claves de API identifican a la aplicación que llama; los tokens de acceso son credenciales de corta duración, a menudo en nombre de un usuario; ambos deben viajar solo por HTTPS y mantenerse fuera del código fuente.
- Un webhook es el proveedor llamando a una URL del cliente cuando ocurre un evento; los receptores deben verificar las firmas y gestionar los duplicados.
- Las llamadas síncronas esperan una respuesta; los mensajes asíncronos pasan por una cola, de modo que emisor y receptor no necesitan estar disponibles al mismo tiempo.
- Los cambios incompatibles perjudican a los clientes, así que las API se versionan, y los límites de tasa (HTTP 429) protegen a los proveedores; cada API de terceros es una dependencia que hay que aislar y tener en cuenta al planificar.

## Qué es una API

Una API (interfaz de programación de aplicaciones) es la forma definida en que una pieza de software le pide a otra que haga algo. Enumera qué se puede solicitar, qué hay que proporcionar, qué se devuelve y qué puede salir mal. La palabra importante es *definida*: una API

es un contrato. El proveedor promete mantenerla funcionando; el consumidor promete usar solo lo que describe.

Un restaurante es la analogía habitual. La carta enumera lo que puedes pedir y lo que lleva cada plato; el camarero lleva tu pedido a la cocina y te trae el resultado. No entras en la cocina, y no necesitas saber si el chef cocina con gas o con inducción. La cocina puede cambiar su equipamiento, su personal y sus proveedores a voluntad, siempre que los platos de la carta sigan llegando tal como se describen. La carta es la API, la cocina es la implementación, y la separación entre ambas es lo que permite que las dos partes cambien de forma independiente.

Es la misma idea que las interfaces de organizar el código, aplicada a todas las escalas. También es la razón por la que a los arquitectos les importan tanto las API: una API es el punto donde el trabajo de un equipo se encuentra con el de otro, y una vez que otros dependen de ella, cambiarla es costoso.

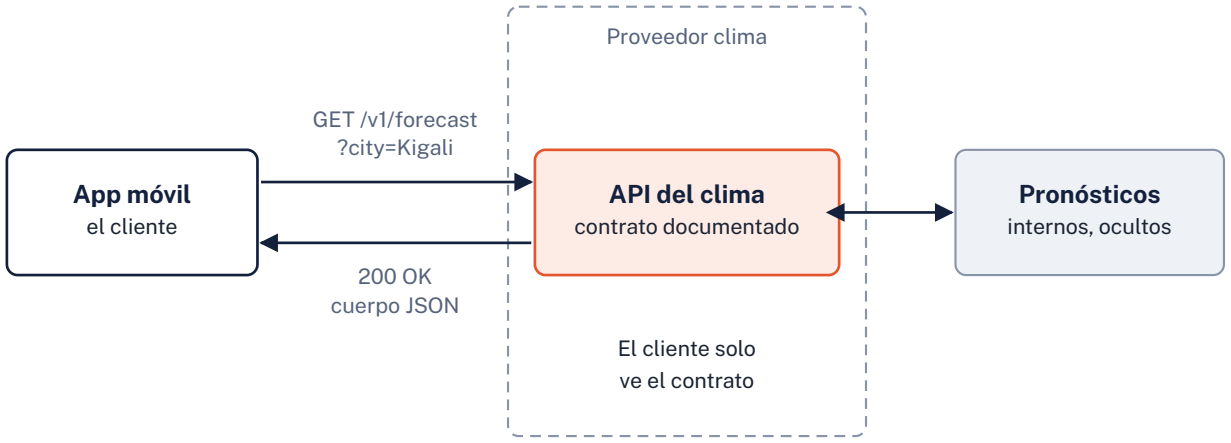
## API en proceso y API web

Algunas API se llaman dentro de un mismo programa en ejecución. Cuando un código Python llama a `json.loads(text)`, usa la API del módulo `json` de la biblioteca estándar: una llamada a una función que tarda nanosegundos y solo falla si la entrada es incorrecta. Las funciones y clases públicas de una biblioteca son su API en proceso.

Una API web se llama a través de una red, normalmente con HTTP (consulta cómo se comunican las computadoras). Quien llama y el proveedor pueden estar en máquinas distintas, estar escritos en lenguajes distintos y pertenecer a organizaciones distintas. Una app de banca para dispositivos móviles se comunica con la API web del banco; los servidores del banco se comunican con la API de una red de tarjetas; una tienda se comunica con la API de un proveedor de pagos.

La diferencia no es superficial. Una llamada de red es entre miles y millones de veces más lenta que una llamada a una función, y puede fallar de formas en que no puede fallar una llamada a una función: la red pierde la solicitud, el servidor está sobrecargado, la respuesta llega después de que quien llama se haya rendido, o la solicitud tuvo éxito pero la respuesta se perdió, de modo que quien llama no puede saber si el pago se realizó. Gran parte del diseño de sistemas distribuidos, que se trata en el Capítulo 6, consiste en convivir con estos fallos.

**Una llamada a una API web**




El cliente solo ve el contrato de la API; cómo guarda y calcula los pronósticos el proveedor permanece oculto detrás de él.

## REST: recursos, URL y verbos

La mayoría de las API web actuales siguen el estilo llamado REST (Representational State Transfer), descrito por Roy Fielding en su tesis doctoral de 2000. Fielding definió REST como un conjunto de restricciones para grandes sistemas en red como la web; en el uso cotidiano, "una API REST" significa algo más sencillo: una API HTTP organizada en torno a **recursos**.

Un **recurso** es algo que la API expone: un cliente, un pedido, un pronóstico. Cada uno tiene una URL, como `/orders/1042`. Una colección de ellos también tiene una URL, como `/orders`. El cliente actúa sobre los recursos con los métodos HTTP estándar, lee el código de estado del resultado e intercambia datos en el cuerpo, normalmente como JSON. Por tanto, los mismos pocos verbos sirven para cualquier recurso, lo que hace que las API REST bien diseñadas sean predecibles.

**Un recurso, los verbos estándar**

| Solicitud | Significado | Estado habitual en caso de éxito |
|---|---|---|
| `GET /orders?customer=42` | Lista los pedidos, filtrados por un parámetro de consulta. Las lecturas nunca cambian nada. | 200 OK |
| `GET /orders/1042` | Obtiene un pedido. | 200 OK (o 404 Not Found si no existe ese pedido) |
| `POST /orders` | Crea un pedido nuevo a partir del JSON del cuerpo de la solicitud. | 201 Created, con la URL del nuevo pedido |
| `PUT /orders/1042` | Sustituye el pedido 1042 por la representación enviada. | 200 OK o 204 No Content |
| `PATCH /orders/1042` | Cambia solo algunos campos, como la dirección de entrega. | 200 OK |
| `DELETE /orders/1042` | Cancela o elimina el pedido. | 204 No Content |

Los códigos de estado transmiten un significado según el cual puede actuar un programa cliente. Un código **4xx** dice que el cliente ha hecho algo mal: 400 para una solicitud mal formada, 401 cuando faltan las credenciales o no son válidas, 403 cuando se sabe quién llama pero no tiene permiso, 404 cuando el recurso no existe, 429 cuando ha enviado demasiadas solicitudes. Un código **5xx** dice que el servidor ha fallado: 500 para un error inesperado, 503 cuando no está disponible temporalmente. La distinción importa porque reintentar sin cambios una solicitud 4xx volverá a fallar, mientras que reintentar un 503 un poco más tarde puede tener éxito.

Los datos viajan como JSON (JavaScript Object Notation), un formato de texto normalizado como RFC 8259. Tiene solo unos pocos elementos básicos: objetos (pares nombre-valor entre llaves), arrays (listas entre corchetes), cadenas entre comillas dobles, números, `true`, `false` y `null`. A pesar del nombre, todos los lenguajes habituales pueden leerlo y escribirlo, y por eso se convirtió en la lengua común de las API web. Existen otros estilos que responden a necesidades distintas: GraphQL permite al cliente especificar exactamente qué campos quiere, y gRPC usa un formato binario compacto para llamadas rápidas entre servicios; el Capítulo 6 los compara.

EJEMPLO PRÁCTICO

## Ejemplo práctico: llamar a una API del clima

Una app de viajes quiere el pronóstico de mañana para Kigali. El proveedor (aquí un api.example.com inventado) documenta un *endpoint* (una URL que acepta un tipo concreto de solicitud) y entrega a la app una clave de API.

| | |
|---|---|
| **1. Leer el contrato** | La documentación dice: `GET /v1/forecast` con los parámetros de consulta `city` y `days`; envía la clave en una cabecera `Authorization`; las respuestas son JSON; límite de 60 solicitudes por minuto. |
| **2. Enviar la solicitud** | La app envía la solicitud HTTP que se muestra abajo, por HTTPS, para que la clave y los datos viajen cifrados. |
| **3. Comprobar el estado** | 200 significa que el cuerpo contiene un pronóstico. 401 significa que la clave es incorrecta; 429 significa que hay que ir más despacio; 503 significa que hay que volver a intentarlo más tarde. |
| **4. Analizar el JSON** | La app lee `days[0].max_c` y `days[0].rain_chance` e ignora los campos que no necesita. |
| **5. Gestionar el fallo** | Si no llega ninguna respuesta en, por ejemplo, 3 segundos, la app muestra el último pronóstico guardado en caché en lugar de una pantalla en blanco. |

Cada paso depende del contrato, no de cómo calcula los pronósticos el proveedor. El proveedor puede pasarse mañana a servidores nuevos o a un nuevo modelo meteorológico sin que la app cambie.

El paso 5 es el que olvidan los principiantes: cualquier llamada a través de una red puede ser lenta o fallar, así que quien llama debe decidir qué hacer cuando eso ocurra.

```
GET /v1/forecast?city=Kigali&days=2 HTTP/1.1
Host: api.example.com
Authorization: Bearer sk_test_4f9a...
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  "city": "Kigali",
  "units": "metric",
  "days": [
    {"date": "2026-10-23", "min_c": 16, "max_c": 27, "rain_chance": 0.6},
    {"date": "2026-10-24", "min_c": 16, "max_c": 26, "rain_chance": 0.4}
  ]
}
```

La solicitud y la respuesta del ejemplo práctico.

```
import os
import requests

API_KEY = os.environ["WEATHER_API_KEY"]   # never hard-code secrets

resp = requests.get(
    "https://api.example.com/v1/forecast",
    params={"city": "Kigali", "days": 2},
    headers={"Authorization": f"Bearer {API_KEY}"},
    timeout=3,                # give up after 3 seconds
)
resp.raise_for_status()       # raises an error for 4xx and 5xx
tomorrow = resp.json()["days"][0]
print(tomorrow["max_c"], "degrees, rain chance", tomorrow["rain_chance"])
```

La misma llamada desde Python con la popular biblioteca requests, incluidos un timeout y una comprobación del estado.

## ¿Quién llama? Claves de API y tokens

La mayoría de las API necesitan saber quién llama, para facturarle, limitarle y decidir qué puede ver. Conviene separar dos ideas desde el principio. La *autenticación* responde a "¿quién eres?"; la *autorización* responde a "¿qué tienes permitido hacer?"

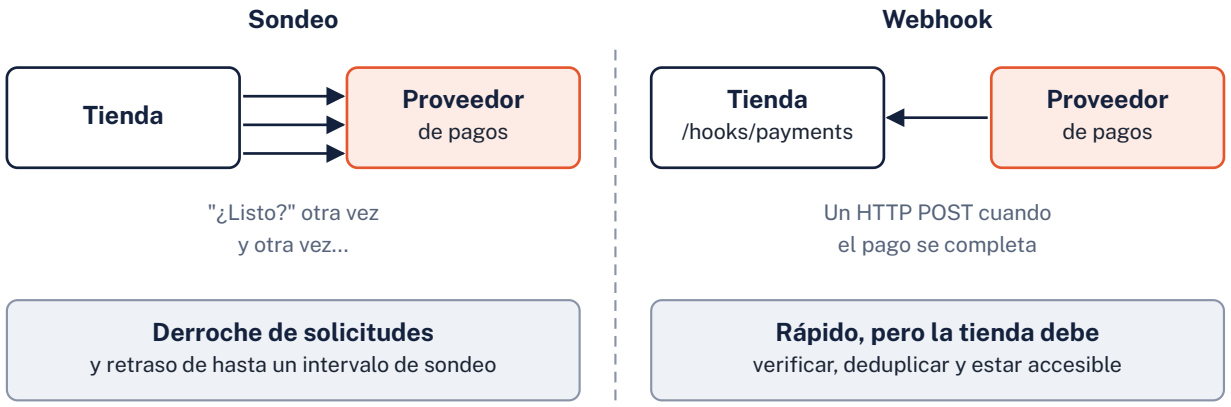
Una clave de API es una cadena secreta larga que identifica a la aplicación que llama y se envía con cada solicitud, normalmente en una cabecera. Es sencilla, e identifica a un programa en lugar de a una persona. Un token de acceso es una credencial de corta duración que se emite después de que alguien inicia sesión, a menudo en nombre de un usuario concreto y con

permisos concretos ("leer el calendario de este usuario, pero no su correo electrónico"); normalmente se envía como `Authorization: Bearer <token>`. Los tokens caducan, así que uno robado solo es útil durante un tiempo limitado. Los protocolos estándar que hay detrás, OAuth 2.0 y OpenID Connect, se tratan en identidad y acceso.

Se use lo que se use, las reglas son las mismas: envía las credenciales solo por HTTPS, mantenlas fuera del código fuente y de las apps para dispositivos móviles, de donde cualquiera puede extraerlas, guárdalas en un gestor de secretos o en la configuración del entorno, y asegúrate de poder revocarlas y sustituirlas rápidamente.

## Webhooks: la API te devuelve la llamada

A veces el cliente necesita saber cuándo ocurre algo en el lado del proveedor: un pago se completa, se entrega un paquete, se firma un documento. El cliente podría *sondear* (polling), preguntando "¿ya ha ocurrido?" cada pocos segundos, lo que desperdicia solicitudes y aun así se entera tarde. Un webhook invierte el sentido: el cliente registra una URL propia y el proveedor envía un HTTP POST a esa URL cuando se produce el evento. El término lo acuñó Jeff Lindsay en 2007.



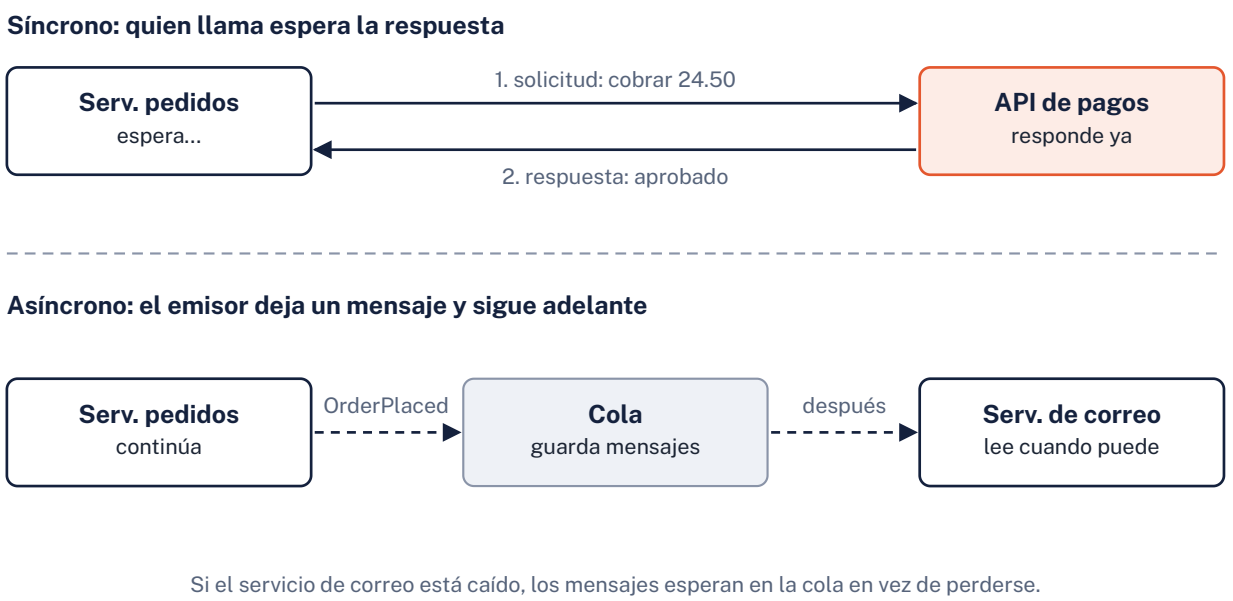

El sondeo pregunta una y otra vez; un webhook espera a que se lo digan, a costa de nuevas responsabilidades para el receptor.

Los webhooks traen responsabilidades. El endpoint receptor está en la internet pública, así que debe comprobar que cada llamada procede realmente del proveedor, normalmente verificando una firma criptográfica en una cabecera. Los proveedores reintentan cuando el receptor no responde, así que el mismo evento puede llegar dos veces y el receptor debe reconocer los duplicados (procesar un duplicado no debe hacer que el pedido se envíe dos veces). Y los eventos pueden llegar desordenados.

## Esperar una respuesta, o no: síncrono y asíncrono

Todo lo visto hasta ahora ha sido síncrono: quien llama envía una solicitud y espera la respuesta antes de continuar, como en una llamada telefónica. Es fácil de razonar, y es la opción correcta cuando quien llama necesita de verdad la respuesta en ese momento, como "¿se ha aprobado este pago con tarjeta?". Su punto débil es que quien llama solo está tan disponible y es tan rápido como todo aquello de lo que está esperando.

La alternativa es la mensajería asíncrona, más parecida a dejar una carta en un buzón. El emisor pone un mensaje en una cola de mensajes y sigue con lo suyo. El receptor toma los mensajes de la cola cuando está listo. Si el receptor va lento o está caído durante una hora, los mensajes esperan en lugar de fallar. Cuando la tienda confirma un pedido, enviar el correo electrónico de confirmación no tiene por qué bloquear la pantalla del cliente; un mensaje que diga `OrderPlaced` puede ir a una cola para que el servicio de correo lo gestione un momento después.




Si el servicio de correo está caído, los mensajes esperan en la cola en vez de perderse.

Una llamada síncrona ata a quien llama a la disponibilidad de quien recibe la llamada; una cola los desacopla en el tiempo.

La asincronía no es gratis. El emisor no sabe de inmediato si el trabajo ha tenido éxito, el sistema se vuelve más difícil de rastrear y de probar, y los mensajes pueden entregarse más de una vez. Decidir qué interacciones deben ser síncronas y cuáles asíncronas es una de las decisiones centrales de los sistemas distribuidos, tratada a fondo en comunicación y API y en colas y back pressure.

GUÍA DE DECISIÓN

### Guía de decisión: ¿llamar ahora o enviar un mensaje?

Una guía inicial para una interacción entre dos partes de un sistema. El Capítulo 6 la afina.

**Q1** **¿Necesita quien llama el resultado antes de poder responder a su propio usuario (por ejemplo, si se ha aprobado la tarjeta)?**

- **Sí** Usa una llamada síncrona, con un timeout y un plan para cuando el otro lado vaya lento o esté caído.
- **No** Pasa a la siguiente pregunta.

**Q2** **¿Puede hacerse el trabajo unos segundos o minutos más tarde sin que nadie lo note (enviar un correo electrónico, actualizar un informe)?**

- **Sí** Envía un mensaje asíncrono a través de una cola, para que un receptor lento o caído no bloquee a quien llama.
- **No** Pasa a la siguiente pregunta.

**Q3** **¿Es lento el trabajo (de segundos a horas), como generar una exportación grande?**

- **Sí** Acepta la solicitud de inmediato (por ejemplo, con 202 Accepted), haz el trabajo desde una cola y avisa al cliente cuando esté terminado mediante un webhook o una URL de estado que pueda consultar.
- **No** Una llamada síncrona sencilla suele bastar; mantenla rápida y fija un timeout.

## Convivir con las API de otros: SDK, versiones y límites

Muchos proveedores publican un SDK (kit de desarrollo de software): una biblioteca para cada lenguaje popular que envuelve las llamadas HTTP en funciones normales, de modo que un desarrollador escribe `client.refunds.create(payment_id)` en lugar de construir las solicitudes a mano. Los SDK ahorran tiempo y se ocupan de detalles como los reintentos y las firmas, pero son otra dependencia que hay que mantener al día.

Una vez que una API tiene usuarios, cambiarla se vuelve delicado. Añadir un campo opcional nuevo a una respuesta suele ser seguro. Cambiar el nombre de un campo, eliminarlo, cambiar su tipo o su significado, o hacer obligatoria una entrada opcional es un cambio incompatible (breaking change): todos los clientes que dependían del comportamiento anterior se rompen, a menudo en producción y a menudo para un proveedor que no puede ver ni actualizar esos clientes (piensa en las apps para dispositivos móviles que los usuarios nunca actualizan). Por eso los proveedores asignan una versión a sus API, por ejemplo `/v1/` y `/v2/` en la URL o una

cabecera de versión, mantienen la versión anterior en funcionamiento durante un periodo publicado y anuncian las *obsolescencias* (avisos de que una función se va a retirar) con mucha antelación. La observación de Hyrum Wright, conocida como ley de Hyrum, añade una advertencia: con suficientes usuarios, alguien dependerá de cada comportamiento observable de una API, incluidos los que la documentación nunca prometió.

Los proveedores también se protegen con un límite de tasa: un tope como 100 solicitudes por minuto y por clave. Las solicitudes que superan el límite se rechazan con el estado 429 Too Many Requests, a menudo con una cabecera `Retry-After` que indica cuándo volver a intentarlo. Los clientes deben respetarlo, repartir sus solicitudes, guardar en caché las respuestas que no cambian a menudo y retroceder (back off) cuando se les indique.

## Usar API de terceros: construir o comprar

Las aplicaciones modernas se ensamblan tanto como se escriben. Una app de reparto de comida podría usar la API de un proveedor de pagos para cobrar con tarjeta, una API de mapas para direcciones y rutas, una API de correo electrónico o SMS para los recibos y un servicio de notificaciones push para avisar de que "tu repartidor está cerca". Cada una ahorra meses de trabajo especializado y aporta una experiencia (seguridad de tarjetas, datos cartográficos globales, entregabilidad del correo electrónico) que pocos equipos podrían igualar.

Cada una aporta también una dependencia que el equipo no controla: las caídas del proveedor se convierten en tus caídas, sus cambios de precio se convierten en tus costos, su tratamiento de los datos pasa a formar parte de tu historia de cumplimiento normativo, y abandonarlo puede obligar a reescribir código. Los profesionales gestionan esto de forma deliberada. Aíslan cada proveedor detrás de una pequeña interfaz propia, de modo que solo una parte del código conozca la API del proveedor; fijan timeouts y deciden qué ocurre cuando el proveedor está caído; vigilan la página de estado y el registro de cambios del proveedor; y anotan lo que costaría cambiar de proveedor.

## Por qué esto importa para la arquitectura

La arquitectura trata en gran medida de las partes de un sistema y de los contratos entre ellas, y las API son esos contratos hechos realidad. Una API que es fácil de usar correctamente y difícil de romper permite a los equipos trabajar en paralelo y cambiar libremente su funcionamiento interno; una API con fugas o inestable acopla a todos con todos. La elección entre llamadas síncronas y mensajes asíncronos determina cómo se propagan los fallos. El Capítulo 3 explica los principios de las buenas interfaces (ocultación de información y abstracción), y el Capítulo 6 muestra cómo las decisiones sobre API y mensajería hacen triunfar o fracasar a los sistemas distribuidos.

### COMPRUEBA LO QUE HAS ENTENDIDO

**▼ En la analogía del restaurante, ¿qué corresponde a la API y qué a la implementación?**

La carta (y la forma en que el camarero toma los pedidos) es la API: la lista acordada de lo que se puede pedir y de lo que se devuelve. La cocina es la implementación, libre para cambiar siempre que se cumplan las promesas de la carta.

**▼ ¿Cuál es la diferencia entre una respuesta 404 y una 503, y cuál merece la pena reintentar más tarde?**

404 Not Found dice que el recurso solicitado no existe, un problema del lado del cliente que reintentar no resolverá. 503 Service Unavailable dice que el servidor no puede responder temporalmente, así que reintentar tras una espera puede tener éxito.

**▼ ¿Por qué el receptor de un webhook debe estar preparado para recibir el mismo evento dos veces?**

Los proveedores reintentan cuando no reciben a tiempo una respuesta de éxito, aunque la primera entrega se haya procesado en realidad, así que los duplicados son normales. El receptor debe reconocer un evento que ya ha gestionado y no volver a actuar sobre él.

**▼ ¿Añadir un campo opcional nuevo a una respuesta JSON es normalmente un cambio incompatible?**

Normalmente no, porque los clientes bien hechos ignoran los campos que no reconocen. Eliminar un campo o cambiarle el nombre, o cambiar su tipo, sí es un cambio incompatible.

### FUENTES


- Roy Fielding (2000), Architectural Styles and the Design of Network-based Software Architectures, Chapter 5: REST: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format: https://www.rfc-editor.org/rfc/rfc8259
- MDN Web Docs: 429 Too Many Requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/429
- Stripe documentation: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Hyrum's Law: https://www.hyrumslaw.com/

SECCIÓN 0.7 DE 0.8 · PRINCIPIANTE · 18 MIN DE LECTURA

# Servidores, sistemas operativos, máquinas virtuales, contenedores y servicios en la nube

## LA IDEA EN BREVE

Un servidor es simplemente una computadora cuya tarea es responder solicitudes, alojada en un gran edificio lleno de otros servidores llamado centro de datos. En lugar de comprar los suyos, la mayoría de las organizaciones alquilan hoy capacidad de cómputo a proveedores de nube, igual que los hogares compran electricidad a la red en vez de tener su propio generador. Puedes alquilar una computadora virtual completa, una plataforma lista para tu código o un software terminado, y pagar solo por lo que usas. Cuanto más alquilas ya hecho, menos tienes que mantener, y menos controlas.

## LO QUE NECESITAS SABER

- Un servidor es cualquier computadora, o programa, que espera solicitudes y las responde; la mayoría de los servidores de internet usan Linux y están en centros de datos construidos para garantizar energía, refrigeración y seguridad.
- Un hipervisor permite que una máquina física ejecute varias máquinas virtuales, cada una con su propio sistema operativo huésped y aislada de las demás.
- Un contenedor empaqueta una aplicación con sus bibliotecas y su configuración y se ejecuta como un proceso aislado que comparte el kernel del anfitrión, por lo que es más ligero y arranca más rápido que una VM, pero está menos aislado.
- La computación en la nube (NIST SP 800-145) significa acceso por red, bajo demanda, en autoservicio y con pago por uso, a recursos compartidos y elásticos.
- IaaS, PaaS, serverless (FaaS) y SaaS delegan progresivamente más partes de la pila al proveedor, cambiando control por menos trabajo operativo; el cliente sigue siendo siempre responsable de sus datos y su configuración.
- Las regiones son áreas geográficas aisladas; las zonas de disponibilidad son grupos de centros de datos separados dentro de una región; las CDN sirven contenido estático desde ubicaciones en el borde cercanas a los usuarios.
- Escalar verticalmente significa una máquina más grande; escalar horizontalmente significa más máquinas detrás de un balanceador de carga, lo que permite crecer más y sobrevivir a fallos, pero solo si el software está diseñado para ello.

## Un servidor es simplemente una computadora que sirve

Un servidor es una computadora cuya tarea es esperar solicitudes y responderlas: páginas web, llamadas a una API, consultas a una base de datos, archivos. La palabra describe una función, no un tipo especial de máquina. Una computadora portátil que ejecuta una pequeña aplicación web es un servidor mientras responde solicitudes; una máquina montada en un rack de un centro de datos es un servidor porque no hace otra cosa. La misma palabra se usa también para el programa que sirve (un "servidor web" como nginx), así que el contexto importa.

El hardware de un servidor tiene las mismas piezas que cualquier computadora (consulta qué es el software): procesadores, memoria, almacenamiento y tarjetas de red. Lo que cambia es el énfasis. Los servidores suelen tener muchos núcleos de procesador, grandes cantidades de memoria, fuentes de alimentación y discos redundantes para que una pieza averiada no detenga la máquina, administración remota para que nadie tenga que estar en la sala, y ninguna pantalla ni teclado. Están hechos para funcionar durante años sin apagarse.

La mayoría de los servidores de internet ejecutan Linux u otro sistema operativo tipo Unix; Windows Server es habitual en organizaciones que dependen de productos de Microsoft. Un sistema operativo de servidor hace las mismas tareas que el de tu computadora portátil (ejecutar procesos, gestionar la memoria y los archivos, comunicarse con la red), pero normalmente se administra mediante una línea de comandos de texto a través de una conexión remota, y se configura para poder reconstruirlo automáticamente en lugar de a mano.

## Centros de datos: donde viven los servidores

Los servidores se guardan en un centro de datos: un edificio diseñado en torno a sus necesidades. Miles de máquinas se alojan en racks, refrigeradas por aire o por líquido, porque convierten casi toda su electricidad en calor. La energía llega por más de un suministro, respaldada por baterías durante los primeros segundos de un corte y después por generadores diésel. Las conexiones de red provienen de varios proveedores. Las puertas exigen tarjeta de acceso y, a menudo, huella dactilar.

Una organización puede tener su propio centro de datos o alquilar espacio en racks en el de otra (lo que se llama *colocation*), y antes de 2006, aproximadamente, esa era casi la única opción. Gestionar tus propios servidores (*on-premises*, en instalaciones propias) significa comprar máquinas meses antes de necesitarlas, adivinar cuántas, pagarlas tanto si están ocupadas como si están inactivas, y emplear a personas que cambien discos averiados de noche. Ese costo y esa rigidez son el problema que la nube se propuso resolver.

## Máquinas virtuales: muchas computadoras dentro de una

Un servidor típico pasa la mayor parte del tiempo inactivo si ejecuta una sola aplicación. Una máquina virtual (VM) permite que una máquina física se haga pasar por varias. Cada VM cree que es una computadora completa con sus propios procesadores, memoria, disco y tarjeta de red, y ejecuta su propio sistema operativo, llamado *huésped*. Una capa de software llamada hipervisor reparte el hardware real entre las VM y las mantiene separadas, de modo que un fallo o un ataque en una no llegue a las demás. Algunos hipervisores se ejecutan directamente sobre el hardware (VMware ESXi, Microsoft Hyper-V, KVM, que forma parte de Linux); otros se ejecutan como una aplicación sobre un sistema operativo normal (VirtualBox).

La virtualización cambió la economía de los servidores. Una sola máquina puede llenarse de VM de distintos equipos o clientes. Una VM es un conjunto de archivos, así que se puede copiar, respaldar, trasladar a otra máquina y crear en minutos, en lugar de pedirla y esperar a que llegue. Ten en cuenta que este es un uso de "máquina virtual" distinto del de las VM de lenguaje, como la Java Virtual Machine mencionada en qué es el software: aquellas ejecutan programas; estas imitan computadoras completas.

## Contenedores: enviar la aplicación con todo lo que necesita

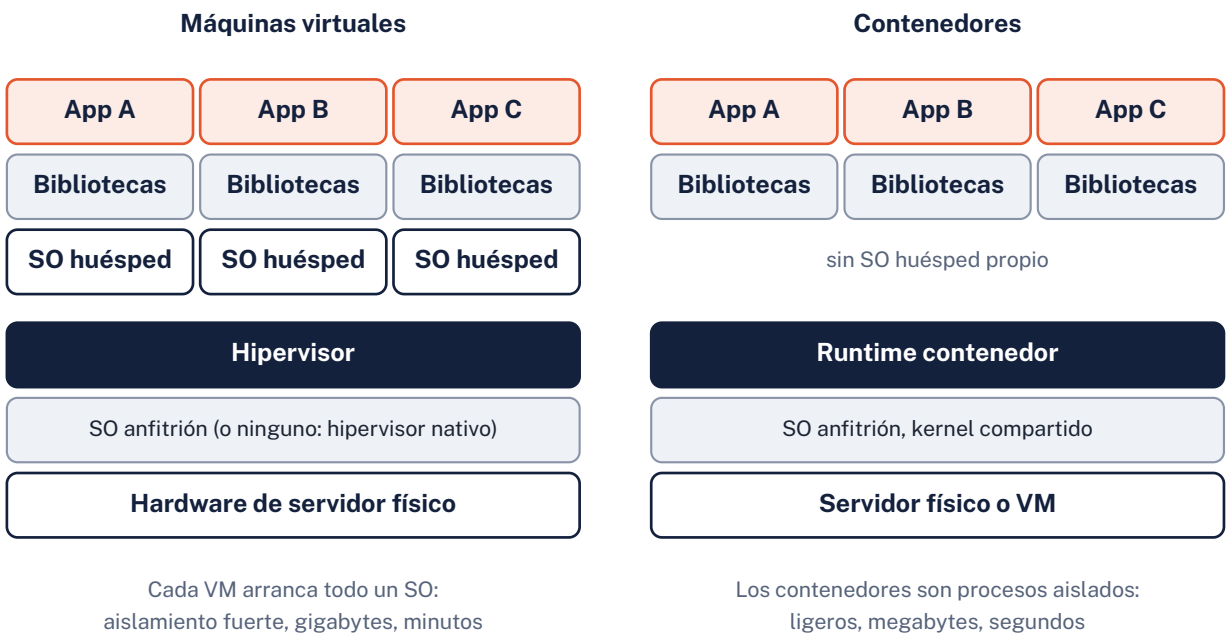
Todo desarrollador ha oído "en mi máquina funciona". El programa depende de una versión concreta del lenguaje, de bibliotecas concretas y de una configuración concreta, y el servidor tiene otras ligeramente distintas. Un contenedor resuelve esto empaquetando una aplicación junto con todo lo que necesita para ejecutarse (su runtime, sus bibliotecas, su configuración) en una unidad estándar que funciona igual en una computadora portátil, en un servidor de pruebas o en la nube. El nombre viene de los contenedores de transporte: cajas estándar que cualquier grúa, barco o camión puede manejar sin saber qué hay dentro.

El paquete es una imagen de contenedor: un archivo de solo lectura y con versión, construido a partir de una receta breve (con Docker, un `Dockerfile`). Una copia en ejecución de una imagen es un contenedor. Docker, lanzado en 2013, hizo esto práctico para los desarrolladores del día a día, y su formato de imagen es hoy un estándar abierto mantenido por la Open Container Initiative, de modo que otras herramientas pueden construir y ejecutar las mismas imágenes.

La diferencia clave con una VM es qué se comparte. Una VM imita el hardware y ejecuta un sistema operativo huésped completo. Un contenedor es un proceso aislado: comparte con otros contenedores el *kernel* del sistema operativo anfitrión (el núcleo del sistema operativo, que controla el hardware, la memoria y los procesos), y las funciones de aislamiento del kernel le dan su propia vista de los archivos, los procesos y la red. Por eso los contenedores arrancan en segundos o menos y ocupan megabytes en lugar de gigabytes, así que pueden ejecutarse

docenas donde cabrían unas pocas VM. El precio es un aislamiento más débil, porque todos los contenedores dependen del único kernel compartido; por eso los proveedores de nube suelen ejecutar los contenedores de sus clientes dentro de VM, combinando ambos.

**Máquinas virtuales y contenedores comparados**




Las VM virtualizan el hardware y cada una lleva un sistema operativo; los contenedores virtualizan el sistema operativo y comparten su kernel.

```
# Start from an image that already has Python installed
FROM python:3.12-slim
WORKDIR /app
# Bake the libraries into the image
COPY requirements.txt .
RUN pip install -r requirements.txt
# Add the application code
COPY . .
EXPOSE 8000
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Un Dockerfile completo para una pequeña API web en Python. Ejecutar docker build convierte esta receta en una imagen.

Ejecutar un contenedor es fácil. Ejecutar cientos en docenas de máquinas, reiniciar los que fallan, añadir copias cuando sube el tráfico y desplegar nuevas versiones sin tiempo de inactividad no lo es. Esa tarea la hace un orquestador, del que Kubernetes es el más conocido; se trata en contenedores y orquestación.

## La nube: alquilar capacidad de cómputo por minutos

La computación en la nube consiste en alquilar recursos de cómputo a través de internet, bajo demanda, a un proveedor que gestiona enormes centros de datos: Amazon Web Services (AWS), Microsoft Azure y Google Cloud son los más grandes, entre muchos otros. El Instituto Nacional de Estándares y Tecnología de EE. UU. (NIST) publicó en 2011 la definición que más se cita, la Special Publication 800-145. En ella enumera cinco características esenciales: autoservicio bajo demanda (creas los recursos tú mismo, desde un sitio web o una API, sin llamar a nadie), amplio acceso por red, agrupación de recursos (muchos clientes comparten el hardware del proveedor), elasticidad rápida (la capacidad crece y se reduce con rapidez) y servicio medido (pagas por lo que usas).

La analogía es la electricidad. Antes las fábricas tenían sus propios generadores; ahora se conectan a la red y pagan por kilovatio-hora. Con la nube, una startup puede alquilar cien servidores durante una tarde para probar un lanzamiento y dejar de pagar cuando los devuelve. Eso es el precio de pago por uso (pay-as-you-go). Convierte una gran compra inicial en un costo continuo, y premia los sistemas que pueden apagar capacidad cuando no se necesita. También crea un riesgo nuevo: costos que crecen sin que nadie lo note, porque nada impide que cualquiera cree recursos y se olvide de ellos.

## IaaS, PaaS, SaaS y serverless: ¿cuánto gestionas tú?

Los servicios en la nube se diferencian sobre todo en cuánta parte de la pila gestiona el proveedor por ti. El NIST definió tres modelos de servicio; un cuarto, serverless, se popularizó después de que AWS lanzara su servicio Lambda en 2014. Una forma popular de recordarlos es la pizza: puedes hacerla en casa, comprar una lista para hornear, pedirla a domicilio o comer fuera. Cada paso delega más trabajo en otra persona y te quita algunas opciones.

**Modelos de servicio en la nube comparados**

| Modelo | Gestionas tú | Gestiona el proveedor | Ejemplos | Analogía de la pizza |
|---|---|---|---|---|
| **En instalaciones propias (on-premises)** | Todo: edificio, hardware, SO, runtime, aplicación, datos | Nada | Tus propios servidores en tu propio centro de datos | Hecha en casa desde cero |
| **IaaS (infraestructura como servicio)** | Sistema operativo, runtime, aplicación, datos | Hardware, virtualización, red, edificio | Amazon EC2, Azure Virtual Machines, Google Compute Engine | Para hornear en casa: la cocinas en tu propio horno |
| **PaaS (plataforma como servicio)** | Código de la aplicación y datos | Todo lo que hay debajo, incluidos los parches del SO y el escalado | Heroku, Azure App Service, Google App Engine | A domicilio: tú pones la mesa y las bebidas |
| **Serverless / FaaS (función como servicio)** | Funciones individuales, activadas por eventos | Todos los servidores, con escalado a cero en reposo y facturación por llamada | AWS Lambda, Azure Functions, Google Cloud Run functions | Pizza por porciones, pagada por bocado |
| **SaaS (software como servicio)** | Tus datos, tus usuarios y su configuración | La aplicación completa | Microsoft 365, Gmail, Salesforce | Comer en un restaurante |

"Serverless" no significa que no haya servidores; significa que nunca los ves ni los gestionas. Subes una función, indicas qué eventos la activan (una solicitud HTTP, la llegada de un archivo, un mensaje en una cola) y el proveedor ejecuta tantas copias como hagan falta y cobra por ejecución y por milisegundo. Es barato para tráfico irregular o bajo y casi no requiere trabajo de operaciones. Sus costos aparecen en otros sitios: la primera llamada tras un periodo tranquilo puede ser lenta mientras el proveedor inicia la función (un *arranque en frío*), hay límites sobre cuánto tiempo puede ejecutarse una función, y con tráfico intenso y constante los costos pueden superar los de un servidor normal. El Capítulo 6 analiza a fondo las arquitecturas serverless.

Entre estos modelos se sitúan los servicios gestionados: bases de datos, colas de mensajes, cachés, almacenamiento de objetos, motores de búsqueda y muchos más, que el proveedor instala, ejecuta, parchea, respalda y escala. Usar una base de datos gestionada en lugar de

instalar una en una VM traslada al proveedor una gran cantidad de trabajo operativo. El compromiso, que se analiza en FinOps y dependencia del proveedor, es que cuantos más servicios específicos de un proveedor usa un sistema, más difícil es trasladarlo.

**ATENCIÓN**

### El modelo de responsabilidad compartida

Pasar a la nube no traslada toda la responsabilidad. Los proveedores declaran claramente que ellos protegen la propia nube (edificios, hardware, la capa de virtualización) y que el cliente protege lo que pone en ella: sus datos, sus permisos de acceso, su configuración y, en IaaS, el sistema operativo. Muchas filtraciones de datos en la nube muy publicitadas no fueron fallos del proveedor, sino almacenamiento mal configurado por el cliente y abierto a internet.

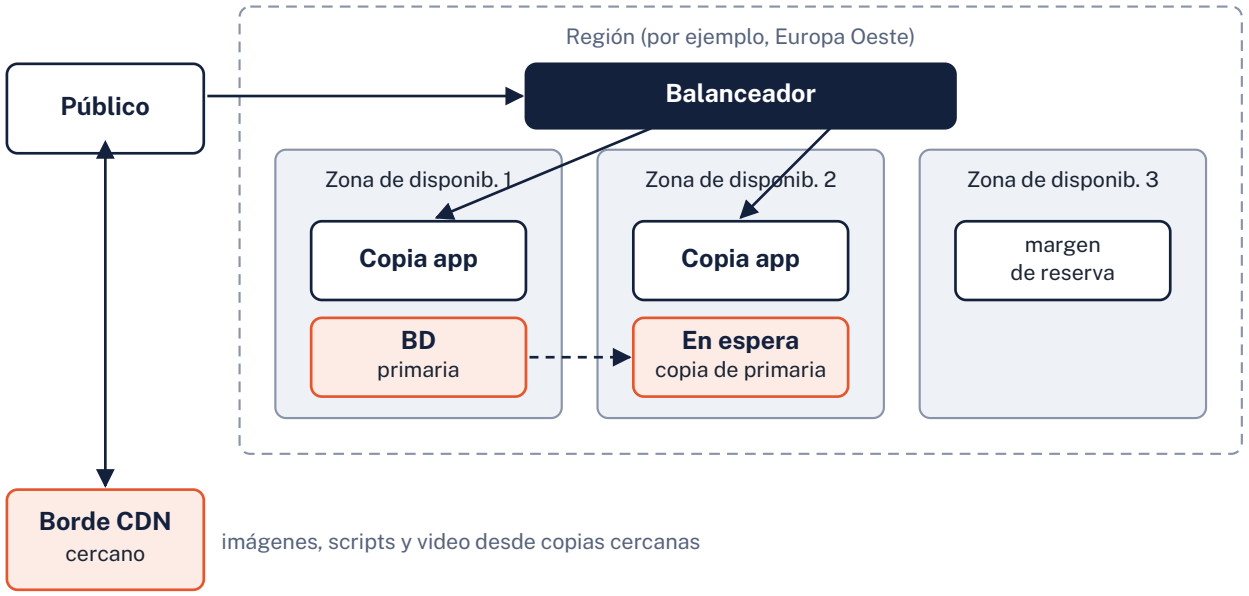
## Regiones, zonas de disponibilidad y redes de distribución de contenidos

Los proveedores de nube agrupan sus centros de datos por geografía. Una región es un área geográfica (como Irlanda, Fráncfort o Virginia) con su propio conjunto de centros de datos, aislado de otras regiones. Cada región se divide en varias zonas de disponibilidad: grupos físicamente separados de uno o más centros de datos, cada uno con energía, refrigeración y red independientes, normalmente a kilómetros de distancia entre sí, pero lo bastante cerca para tener conexiones rápidas. Un incendio, una inundación o un corte de energía debería afectar como máximo a una zona.

Esto da a los arquitectos un menú de protección. Ejecutar copias de una aplicación en dos o tres zonas de una región permite sobrevivir a la pérdida de un centro de datos con poca latencia adicional; ejecutarla en dos regiones permite sobrevivir a la pérdida de una región entera, pero hace mucho más difícil mantener los datos sincronizados. Las regiones también importan por la ley y por la latencia: las normas de protección de datos pueden exigir que los datos personales permanezcan en una jurisdicción concreta, y los usuarios reciben un servicio más rápido desde una región cercana, porque ninguna red puede superar la velocidad de la luz (consulta cómo se comunican las computadoras).

Una red de distribución de contenidos (CDN) lleva esto más lejos para el contenido que no cambia según el usuario, como imágenes, hojas de estilo, scripts y video. Guarda copias en servidores de cientos de ubicaciones, llamadas ubicaciones en el borde (edge locations), para que un usuario de Nairobi obtenga las fotos de productos de la tienda desde una ciudad cercana y no desde un centro de datos en Europa. La página carga más rápido y los servidores propios de la tienda atienden muchas menos solicitudes. Muchas CDN también absorben el tráfico de ataques y pueden ejecutar pequeños fragmentos de código en el borde.

**Dentro de una región**




Las copias de la aplicación en dos zonas de disponibilidad mantienen el servicio en marcha si una zona falla, mientras una CDN sirve el contenido estático desde ubicaciones cercanas a los usuarios.

## Escalado vertical y escalado horizontal

Cuando un sistema necesita manejar más trabajo, hay dos movimientos básicos. El escalado vertical (scaling up) significa una máquina más grande: más núcleos, más memoria. No requiere cambiar el software, que es su gran ventaja, pero hay una máquina más grande que se puede comprar, se vuelve desproporcionadamente cara cerca del límite y una sola máquina sigue siendo un único punto de fallo. El escalado horizontal (scaling out) significa más máquinas que se reparten el trabajo, normalmente detrás de un balanceador de carga que distribuye las solicitudes entre ellas y deja de enviar a las que fallan. Puede crecer mucho más y sobrevivir a fallos de máquinas, pero solo si el software está diseñado para ello: por ejemplo, un servidor no debe guardar la sesión de un usuario solo en su propia memoria, ni guardar los archivos subidos solo en su propio disco, si la siguiente solicitud del usuario puede ir a otro servidor.

La nube hace atractivo el escalado horizontal porque añadir una máquina es una llamada a una API, y el *autoescalado* puede añadir y quitar copias automáticamente según cambia la carga. Las bases de datos suelen ser la parte más difícil de escalar horizontalmente, y por eso el Capítulo 7 dedica una sección a la replicación y el particionamiento.

EJEMPLO PRÁCTICO

### Ejemplo práctico: alojar un sitio de venta de entradas

Una pequeña cadena de teatros vende entradas en línea. El tráfico es bajo la mayor parte de la semana, pero cuando sale a la venta un espectáculo popular, llegan diez veces más personas en cuestión de minutos.

| | |
|---|---|
| **Una VM alquilada (IaaS)** | Simple y barata. El equipo aplica los parches al SO. El día de la venta, la única máquina se ve desbordada y, si falla, el sitio deja de funcionar. |
| **VM más grande** | Escalar verticalmente da margen sin cambiar el código, pero el teatro paga la máquina grande toda la semana para cubrir unas pocas horas de mucha actividad. |
| **App en una PaaS, dos zonas, autoescalado** | Las copias de la app se ejecutan en dos zonas de disponibilidad detrás de un balanceador de carga. El día de la venta, la plataforma añade copias automáticamente y las retira después. Requiere que la app no guarde datos de sesión en la memoria local. |
| **Base de datos gestionada** | El proveedor ejecuta una base de datos primaria en una zona y otra en espera en otra zona, con copias de seguridad automáticas. El equipo sigue diseñando el esquema y los índices (consulta almacenar datos). |
| **CDN para imágenes y scripts** | Los planos de asientos y las fotos de los espectáculos se sirven desde ubicaciones en el borde, así que los servidores de la app dedican su esfuerzo a las reservas. |

El diseño final cuesta más de montar que una sola VM, pero menos de operar que una máquina permanentemente sobredimensionada, y sobrevive tanto a un pico de tráfico como a la pérdida de un centro de datos.

Cada paso cambió control por comodidad: el equipo eligió qué trabajo conservar y cuál delegar al proveedor.

## Por qué esto importa para la arquitectura

Dónde se ejecuta el software es hoy una decisión arquitectónica, no algo que se piensa al final. La elección entre VM, contenedores, plataformas y funciones determina el costo, la rapidez con la que el equipo puede lanzar versiones, cómo escala el sistema y qué pasa cuando falla un centro de datos. Las regiones y las zonas convierten los objetivos de disponibilidad en planes de despliegue concretos; los servicios gestionados cambian trabajo operativo por dependencia de un proveedor. El Capítulo 9 desarrolla las estrategias de escalado, y el Capítulo 10 trata los modelos de servicio y de despliegue en la nube, los contenedores y la infraestructura con profundidad profesional.

### COMPRUEBA LO QUE HAS ENTENDIDO

**▼ ¿Por qué los contenedores arrancan más rápido y ocupan menos espacio que las máquinas virtuales?**

Un contenedor no arranca su propio sistema operativo: es un proceso aislado que comparte el kernel del anfitrión y solo lleva la aplicación y sus bibliotecas. Una VM lleva y arranca un sistema operativo huésped completo.

**▼ Un equipo alquila una VM en la nube e instala en ella su propia base de datos. ¿Quién es responsable de aplicar los parches al sistema operativo?**

El equipo. Con IaaS, el proveedor gestiona el hardware y la virtualización, y el cliente gestiona el sistema operativo y todo lo que está por encima.

**▼ ¿Contra qué fallo protege ejecutar en dos zonas de disponibilidad, y contra qué no protege?**

Protege contra la pérdida de un centro de datos o de una zona (energía, incendio, red). No protege contra la pérdida de la región entera, ni contra errores de software o datos incorrectos, que afectan a todas las copias.

**▼ ¿Por qué una aplicación que funciona bien en un servidor podría comportarse mal al escalarla horizontalmente detrás de un balanceador de carga?**

Puede guardar estado, como las sesiones de usuario, en su propia memoria, así que cuando el balanceador de carga envía la siguiente solicitud a otra copia, esa copia no conoce al usuario.

### FUENTES


- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- Docker: What is a container?: https://www.docker.com/resources/what-container/
- AWS documentation: Regions and Availability Zones: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html
- Microsoft Learn: What are availability zones?: https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview
- MDN Web Docs: CDN (glossary): https://developer.mozilla.org/en-US/docs/Glossary/CDN

SECCIÓN 0.8 DE 0.8 · PRINCIPIANTE · 20 MIN DE LECTURA

# Requisitos, control de versiones, pruebas, entrega y el recorrido de una solicitud

## LA IDEA EN BREVE

Construir software es un deporte de equipo con una rutina cuidadosa. El equipo anota lo que necesitan los usuarios en forma de historias breves, guarda cada cambio del código en un historial compartido para que nada se pierda, pide a un colega que revise cada cambio y ejecuta pruebas automáticas que funcionan como una red de seguridad. Después, los cambios pasan por copias de práctica del sistema antes de llegar a la versión real que usan los clientes. Al final de la sección se sigue un solo toque del botón Hacer pedido a través de todas las partes de un sistema de reparto de comida.

## LO QUE NECESITAS SABER

- Los requisitos describen qué debe hacer un sistema y con qué calidad; las historias de usuario (Como..., quiero..., para...) con criterios de aceptación son una forma ligera y habitual de recoger las necesidades funcionales.
- Git registra cada cambio como commits; las ramas aíslan el trabajo en curso, y los pull requests aportan revisión de código y comprobaciones automáticas antes de fusionar en main.
- La pirámide de pruebas recomienda muchas pruebas unitarias rápidas, menos pruebas de integración y pocas pruebas de extremo a extremo, que son lentas.
- Los cambios pasan por los entornos de desarrollo, pruebas y staging antes de producción; cuanto más se parezca staging a producción, más problemas detecta.
- La integración continua verifica automáticamente cada cambio; la entrega continua mantiene el software siempre listo para lanzarse; el despliegue continuo lanza automáticamente cada cambio que supera las comprobaciones.
- Cascada, Scrum y Kanban organizan el trabajo de forma distinta; DevOps es una cultura de responsabilidad compartida para construir y operar el software.
- Una sola solicitud en un sistema real atraviesa DNS, HTTPS, un balanceador de carga, servicios, una transacción de base de datos, API de terceros y colas, y decidir cómo se organizan esas partes es arquitectura de software.

## De una idea a un requisito

El software empieza con una necesidad: un restaurante quiere recibir pedidos en línea, un hospital quiere que los pacientes reserven sus citas ellos mismos. Convertir esa necesidad en algo que un equipo pueda construir es el trabajo de los requisitos. Algunos requisitos describen qué debe hacer el sistema ("un cliente puede pagar con tarjeta"); otros describen con qué calidad debe hacerlo ("el pago se completa en menos de dos segundos para el 95% de los clientes"). El Capítulo 2 muestra que el segundo tipo, los atributos de calidad, impulsa la mayoría de las decisiones arquitectónicas.

Muchos equipos recogen las necesidades funcionales como historias de usuario: descripciones breves escritas desde el punto de vista del usuario. Kent Beck las introdujo en Extreme Programming a finales de la década de 1990, y un equipo de la empresa londinense Connextra ideó en 2001 la plantilla que hoy usa la mayoría:

```
As a hungry customer,
I want to reorder a meal I have ordered before,
so that I can eat sooner without searching the menu again.

Acceptance criteria
- My last 10 orders appear on the home screen.
- Tapping "Reorder" fills the basket with the same items.
- Items that are no longer available are flagged, not silently dropped.
```

Una historia de usuario con criterios de aceptación. Los criterios son lo que comprobarán los testers y las pruebas automáticas.

Una historia es deliberadamente pequeña e incompleta. Ron Jeffries la describió como una tarjeta, una conversación y una confirmación: la tarjeta es un recordatorio, los detalles se concretan conversando entre quienes quieren la funcionalidad y quienes la construyen, y los criterios de aceptación confirman cuándo está terminada. Los profesionales no se ponen de acuerdo sobre cuánto debe escribirse por adelantado. Los equipos de sectores regulados, como los dispositivos médicos y la aviación, deben mantener requisitos formales y trazables; los equipos de producto suelen preferir historias más ligeras que se refinan justo antes de empezar el trabajo. Ambas opciones son legítimas en su contexto.

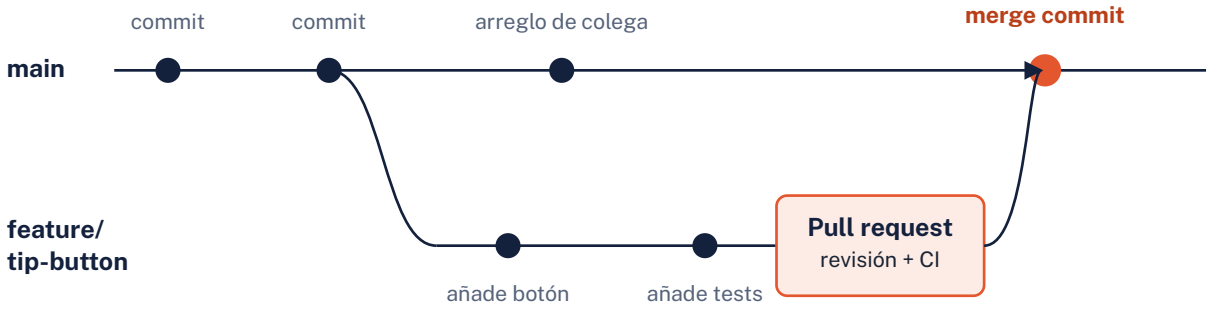
## Control de versiones con Git

El código fuente es el documento compartido más valioso del equipo, y muchas personas lo modifican cada día. El control de versiones registra cada cambio, quién lo hizo y por qué, permite trabajar en paralelo sin sobrescribir el trabajo de otros y hace posible restaurar

cualquier estado anterior. Hoy casi todos los equipos usan Git, creado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux, normalmente junto con un servicio de alojamiento como GitHub, GitLab o Bitbucket.

Unas pocas ideas cubren la mayor parte del uso diario. Un commit es una instantánea guardada de un conjunto de cambios, con un mensaje que lo explica ("Add tip button to checkout"). Una rama es una línea de trabajo separada: un desarrollador crea una rama a partir de `main`, hace commits en ella sin molestar a nadie y, al terminar, *fusiona* la rama de vuelta (merge), combinando los dos historiales. Si dos personas cambiaron las mismas líneas, Git informa de un *conflicto de fusión* que una persona debe resolver.

En los servicios de alojamiento, la fusión suele pasar por un pull request (GitLab lo llama merge request): una página web que muestra el cambio propuesto, donde se ejecutan comprobaciones automáticas y los colegas hacen la revisión de código, leyendo el cambio, haciendo preguntas y sugiriendo mejoras antes de aprobarlo. La revisión detecta defectos, pero sus mayores beneficios son difundir el conocimiento (al menos dos personas entienden cada cambio) y mantener la coherencia del diseño. Los arquitectos suelen participar en las revisiones de cambios que afectan a la estructura o a las interfaces.




La rama mantiene el trabajo en curso fuera de main hasta que se ha revisado y probado.

Rama, commit, revisión, fusión: el ciclo diario del control de versiones.

```
git switch -c feature/tip-button     # create a branch and move onto it
# ... edit files ...
git add checkout.py test_checkout.py
git commit -m "Add tip button to checkout"
git push -u origin feature/tip-button  # upload; then open a pull request
# after review and green checks, the branch is merged into main
```

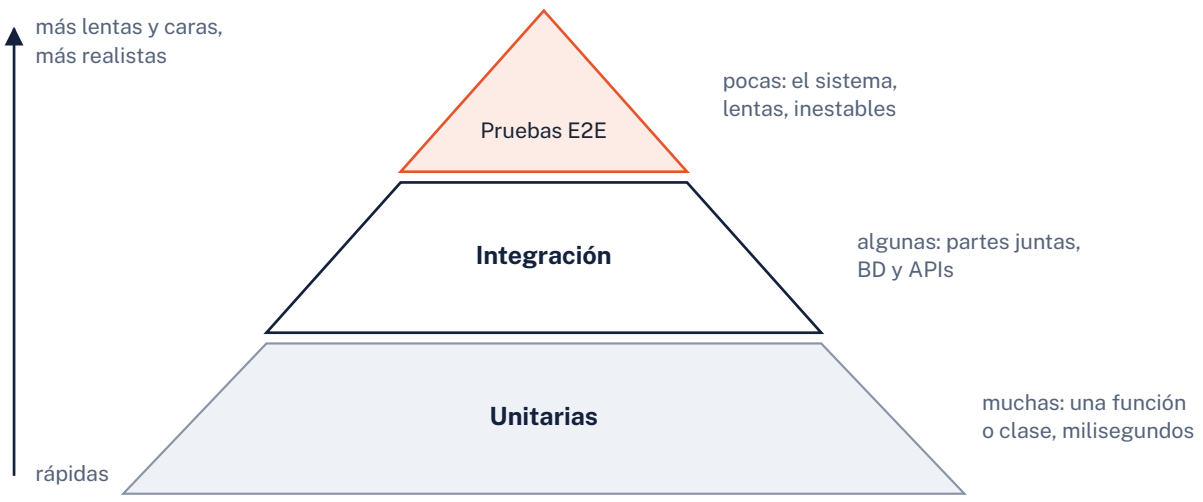
El ciclo diario de Git desde la línea de comandos.

Los equipos difieren en cómo usan las ramas. Algunos mantienen ramas de larga duración por funcionalidad o por versión; otros practican el *desarrollo basado en trunk*, fusionando cambios pequeños en `main` al menos una vez al día y ocultando las funcionalidades sin terminar tras interruptores llamados feature flags. La investigación del programa DORA asocia las ramas de corta duración con un mejor rendimiento de entrega, pero la elección correcta también depende de cómo lanza el equipo sus versiones.

## Pruebas: la pirámide

Cada cambio corre el riesgo de romper algo que antes funcionaba, lo que se llama una *regresión*. Las pruebas automáticas son programas que comprueban otros programas, así que pueden ejecutarse con cada cambio en minutos en lugar de a cargo de personas durante días. Se organizan en capas. Una prueba unitaria comprueba una pieza pequeña, como una función, de forma aislada, y se ejecuta en milisegundos. Una prueba de integración comprueba que las partes funcionan juntas: el código de pedidos con una base de datos real, o la tienda con el entorno de pruebas de un proveedor de pagos. Una prueba de extremo a extremo maneja el sistema completo como lo haría un usuario, por ejemplo tocando la app desde el menú hasta la confirmación del pedido.

La pirámide de pruebas de Mike Cohn, publicada en su libro de 2009 *Succeeding with Agile*, sugiere las proporciones: muchas pruebas unitarias, menos pruebas de integración y pocas pruebas de extremo a extremo. Las capas superiores son más realistas, pero más lentas, más caras de mantener y más propensas a fallos *intermitentes* (flaky), que van y vienen por motivos ajenos al cambio. Algunos profesionales defienden otra forma, por ejemplo más pruebas de integración en sistemas que son sobre todo pegamento entre servicios; el principio en el que todos coinciden es detectar la mayoría de los problemas con pruebas rápidas, fiables y focalizadas.




La forma guía las proporciones, no es una ley: que casi todas las pruebas sean rápidas y concretas.

Muchas pruebas unitarias rápidas, menos pruebas de integración y unas pocas de extremo a extremo dan una respuesta rápida sin perder realismo.

```
def delivery_fee(basket_total, distance_km):
    if basket_total ⩾ 25:
        return 0.0                      # free delivery on larger orders
    return round(1.50 + 0.40 * distance_km, 2)

def test_free_delivery_over_threshold():
    assert delivery_fee(30.00, 5) == 0.0

def test_fee_grows_with_distance():
    assert delivery_fee(10.00, 5) == 3.50
```

Dos pruebas unitarias escritas con pytest para una función de tarifa de envío. Se ejecutan en milisegundos y fijan las reglas.

## Entornos: de desarrollo a producción

El código pasa por varios entornos en su camino hacia los clientes, cada uno una copia separada del sistema con sus propios servidores, datos y configuración. El objetivo es encontrar los problemas donde son baratos e inofensivos, antes de que lleguen a producción, el entorno real del que dependen los usuarios reales.

| Entorno | Quién lo usa | Datos | Propósito |
|---|---|---|---|
| **Desarrollo** | Un desarrollador, en una computadora portátil o un espacio personal en la nube | Pocos, inventados | Escribir y probar código rápidamente. |
| **Pruebas (o integración)** | El equipo y el pipeline automatizado | Inventados o anonimizados | Ejecutar pruebas automáticas y exploratorias sobre los cambios combinados. |
| **Staging (o preproducción)** | Testers, product owners y, a veces, clientes seleccionados | Parecidos a los de producción en forma y volumen, nunca datos personales sin tratar y sin salvaguardas | Ensayar el lanzamiento en una copia lo más parecida posible a producción. |
| **Producción** | Clientes reales | Reales | Hacer funcionar el negocio. Los cambios llegan solo a través del pipeline. |

Cuanto más se diferencian los entornos de producción, más problemas se cuelan: una base de datos de pruebas con cien filas no revelará el índice que falta y que hace que una consulta tarde cuarenta segundos contra diez millones. Mantener los entornos parecidos es más fácil cuando se construyen a partir de las mismas imágenes de contenedor y la misma configuración automatizada mediante scripts, un tema de la infraestructura como código.

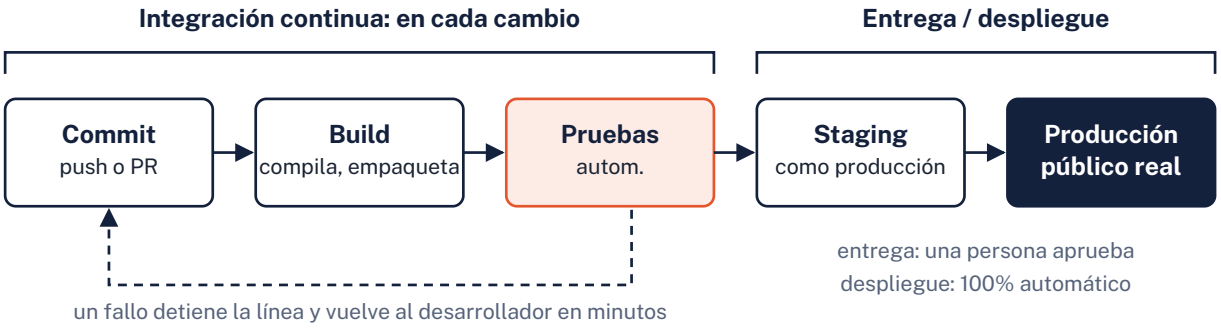
## Integración, entrega y despliegue continuos

La integración continua (CI) es la práctica de fusionar el trabajo de todos en la línea principal compartida con frecuencia, al menos a diario, con una compilación y una ejecución de pruebas automatizadas que comprueban cada cambio. Surgió de Extreme Programming en la década de 1990. Su sentido es que los problemas de integración se detectan en minutos, mientras el cambio sigue fresco en la mente de su autor, en lugar de en una dolorosa "fase de integración" semanas después. Un equipo que practica CI trata una compilación fallida como una emergencia: arreglarla va antes que cualquier trabajo nuevo.

La entrega continua (CD) amplía la CI para que el software esté siempre listo para lanzarse: cada cambio que supera el pipeline podría ponerse en producción pulsando un botón, y lanzar se convierte en una decisión de negocio rutinaria en lugar de un acontecimiento arriesgado. Jez Humble y David Farley expusieron este enfoque en su libro de 2010 *Continuous Delivery*. El

despliegue continuo va un paso más allá y lanza automáticamente cada cambio que supera las comprobaciones, sin aprobación humana; los equipos que lo hacen pueden desplegar muchas veces al día.

La automatización que lleva un cambio desde el commit hasta producción es un pipeline. Como cada paso está automatizado y es idéntico cada vez, los lanzamientos dejan de depender de que una persona recuerde una lista de comprobación a medianoche.




La integración continua comprueba automáticamente cada cambio; la entrega y el despliegue llevan la compilación aprobada hasta producción.

### Lista de comprobación: antes de que un cambio llegue a producción

Lo que comprueba el proceso de un equipo maduro, sobre todo de forma automática, en cada cambio.

- [ ] **El cambio está vinculado a una historia o incidencia**
  Cualquiera puede averiguar después por qué se hizo.
- [ ] **Es pequeño**
  Los cambios pequeños son más fáciles de revisar, probar, desplegar y, si hace falta, deshacer.
- [ ] **Un colega lo ha revisado**
  Mediante un pull request, con las preguntas respondidas y los comentarios resueltos.
- [ ] **Las pruebas automáticas pasan en todos los niveles**
  Pruebas unitarias y de integración en la CI, y las comprobaciones de extremo a extremo que cubren el recorrido afectado.
- [ ] **Se ejecutó en staging**
  La misma compilación, el mismo mecanismo de configuración y los mismos pasos de despliegue que en producción.
- [ ] **Hay una vía de vuelta atrás**
  Un rollback o una feature flag pueden desactivarlo rápidamente si algo sale mal.
- [ ] **Alguien se dará cuenta si se rompe**
  La monitorización y las alertas cubren lo que toca el cambio (Capítulo 9, observabilidad).

## Formas de trabajar: cascada, Scrum y Kanban

Los equipos también eligen cómo organizar el propio trabajo. En un enfoque secuencial, a menudo llamado *en cascada*, el proyecto avanza por fases en orden: requisitos, diseño, implementación, pruebas, despliegue. Cada fase termina antes de que empiece la siguiente. El nombre suele atribuirse a un artículo de Winston Royce de 1970, aunque Royce no usó esa palabra y de hecho advirtió que pasar una sola vez por las fases era arriesgado. Los planes secuenciales siguen siendo adecuados para trabajos en los que los requisitos son fijos y los cambios son muy costosos, como algunos proyectos regulados o que dependen del hardware. Su debilidad para la mayor parte del software es que los usuarios no ven nada hasta muy tarde, así que los malentendidos salen a la luz cuando son más caros de corregir.

Los enfoques ágiles, bautizados en el Manifiesto Ágil de 2001, entregan software que funciona en ciclos cortos y ajustan los planes a medida que aprenden. Scrum es el marco de trabajo más usado: un equipo pequeño trabaja en ciclos fijos llamados sprints, de un mes o menos, cada uno de los cuales empieza con una planificación y termina con una revisión del trabajo y una retrospectiva sobre cómo ha trabajado el equipo. La Guía de Scrum de 2020 define tres responsabilidades: el Product Owner (que decide qué es lo más valioso), el Scrum Master (que ayuda al equipo a usar Scrum con eficacia) y los Developers (que hacen el trabajo). Kanban, adaptado al software a partir del sistema de producción de Toyota, no tiene ciclos fijos: los elementos de trabajo fluyen por un tablero visible (por hacer, en curso, revisión, hecho), y el equipo limita cuántos elementos puede haber en curso a la vez para terminar el trabajo antes de empezar más.

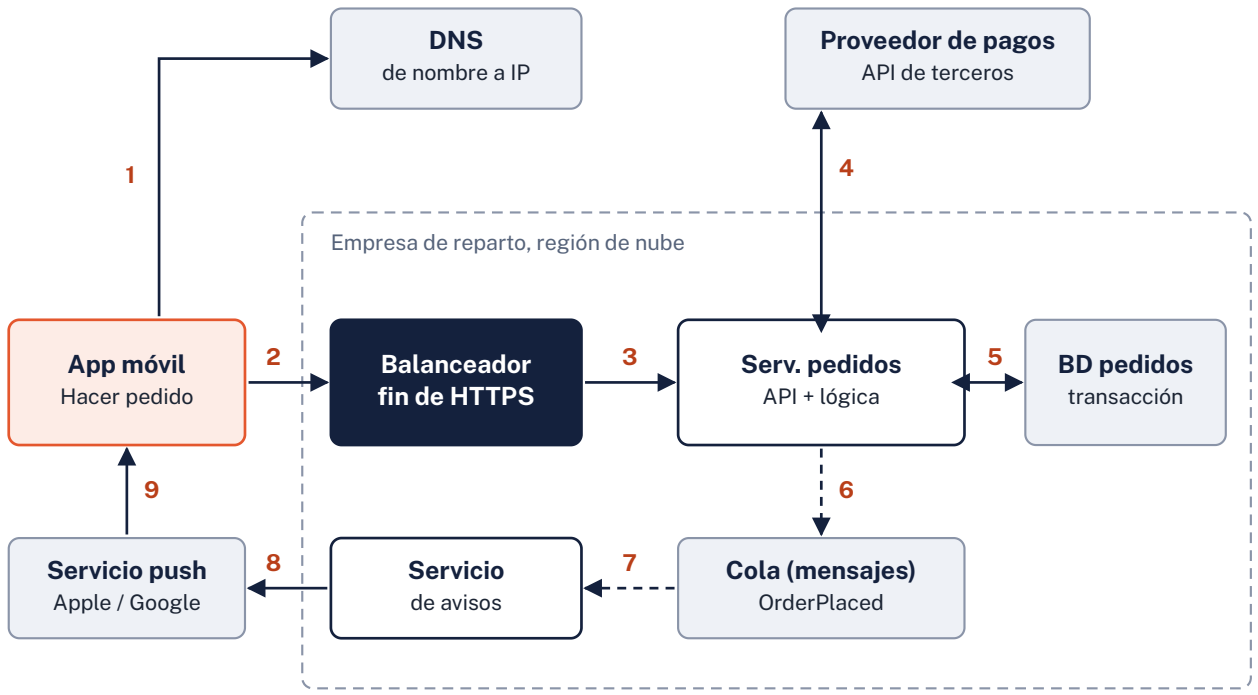
Ninguno de ellos es adecuado para todos los equipos, y muchas organizaciones los combinan: Scrum para los equipos de producto, Kanban para soporte y operaciones, un plan más secuencial en torno a un plazo regulatorio fijo. Llamar "ágil" a un proceso no lo convierte en ágil; lo que importa es si el software que funciona llega a usuarios reales con la frecuencia suficiente para aprender de ellos.

## DevOps: la responsabilidad de un solo equipo, del código al cliente

Tradicionalmente, los desarrolladores escribían el software y un equipo de operaciones separado lo hacía funcionar, con incentivos opuestos: a los desarrolladores se les recompensaba por el cambio y a operaciones por la estabilidad. DevOps es el movimiento, bautizado en la primera conferencia DevOpsDays de Gante en 2009, que une a ambos. Es ante todo una cultura de responsabilidad compartida ("you build it, you run it": quien lo construye, lo opera), apoyada en prácticas como CI/CD, la infraestructura como código, la monitorización y la revisión sin culpables de los incidentes. No es un puesto de trabajo ni una herramienta, aunque ambas cosas se venden con ese nombre. El programa de investigación DORA es conocido sobre todo por medir el rendimiento de entrega con cuatro métricas clave: con qué frecuencia despliega un equipo, cuánto tarda un cambio en llegar a producción, con qué frecuencia fallan los cambios y con qué rapidez se restablece el servicio tras un fallo.

## Ejemplo práctico: el recorrido de un toque

Todo lo visto en este capítulo se une en un solo momento. Un cliente ha llenado un carrito en una app de reparto de comida y toca **Hacer pedido**. En un segundo o dos la pantalla dice "Pedido confirmado", y veinte minutos después el teléfono vibra: "Tu repartidor está en camino". Esto es lo que ocurre, paso a paso, con el concepto que hay detrás de cada paso y dónde se presentó.




Un toque atraviesa el teléfono, internet, la región de nube de la empresa y dos proveedores externos; los números corresponden a los pasos de abajo.

**Paso a paso**

1. **0. La app construye la solicitud**
   La app es software en el teléfono (Sección 0.1). Reúne el carrito a partir de sus estructuras de datos en memoria (Sección 0.2) y lo convierte en un cuerpo JSON: restaurante, artículos, dirección de entrega y una clave de pedido única generada en el teléfono.

2. **1. DNS**
   La app necesita la dirección de `api.fooddelivery.example`. El teléfono pregunta a un resolutor DNS, que devuelve una dirección IP, a menudo desde una caché (Sección 0.4).

3. **2. HTTPS**
   La app abre una conexión TCP y una sesión TLS, comprobando el certificado del servidor, y luego envía `POST /v1/orders` con un token de acceso que demuestra qué cliente ha iniciado sesión (Secciones 0.4 y 0.6).

4. **3. Balanceador de carga**
   La solicitud llega a un balanceador de carga en la región de nube de la empresa. Este descifra el HTTPS y reenvía la solicitud a una de varias copias en buen estado del servicio de pedidos, que se ejecutan en contenedores repartidos entre zonas de disponibilidad (Sección 0.7).

5

### 4. API y pago

Al servicio de pedidos se accede a través de su API REST. Su código, organizado en módulos (Sección 0.3), valida el carrito y luego llama a la API web del proveedor de pagos para autorizar la tarjeta, esperando de forma síncrona con un timeout, porque al cliente no se le debe decir "confirmado" hasta que el pago esté aprobado (Sección 0.6). Envía la clave del pedido como clave de idempotencia, para que, si la llamada se reintenta, el cargo a la tarjeta se haga una sola vez.

6

### 5. Transacción de base de datos

En una sola transacción ACID, el servicio inserta el pedido y sus artículos y registra la referencia del pago. O se guarda todo o no se guarda nada; una vez confirmada, la transacción sobrevive a un fallo del sistema (Sección 0.5). Un índice sobre el ID del cliente hará que su historial de pedidos cargue rápido.

7

### 6. Cola

El servicio pone un mensaje `OrderPlaced` en una cola de mensajes y responde de inmediato `201 Created`. La app muestra "Pedido confirmado". Todo lo demás ocurre de forma asíncrona (Sección 0.6).

8

### 7. Otros servicios reaccionan

El servicio de tabletas del restaurante, el servicio de despacho de repartidores y el servicio de notificaciones leen el mensaje cuando están listos. Si uno de ellos se cae brevemente, sus mensajes esperan en la cola.

9

### 8. Notificación push

El servicio de notificaciones reacciona a `OrderPlaced`, y más tarde a un mensaje similar cuando se asigna un repartidor, llamando al servicio push de Apple o Google, otra API de terceros, que entrega la notificación en el teléfono del cliente.

10

### 9. Y detrás de todo

Cada fragmento de código de este recorrido se escribió a partir de una historia de usuario, se revisó en un pull request, se probó en varios niveles y se lanzó mediante un pipeline de CI/CD, pasando por staging, hasta producción (esta sección).

EJEMPLO PRÁCTICO

### Ejemplo práctico: lo que enseña el recorrido

Pregunta "¿y si…?" en cada paso y aparecerán las preguntas arquitectónicas.

| | |
|---|---|
| **¿Y si la llamada de pago supera el timeout?** | ¿Se hizo el cargo o no? La clave de idempotencia permite al servicio de pedidos volver a preguntar con seguridad, sin cobrar dos veces. |
| **¿Y si la confirmación (commit) en la base de datos falla después del pago?** | El pago y la base de datos son sistemas separados, así que ninguna transacción ACID única abarca ambos. El diseño necesita una forma de reembolsar o reintentar, un problema que el Capítulo 6 resuelve con sagas. |
| **¿Y si el servicio de notificaciones está caído?** | No se rompe nada visible: se siguen recibiendo pedidos y los mensajes esperan en la cola. Esa es la razón para hacer asíncrono este paso. |
| **¿Y si el tráfico de la hora de la cena se triplica?** | El balanceador de carga reparte las solicitudes entre más copias del servicio de pedidos, añadidas por el autoescalado; ahora el cuello de botella probable es la base de datos. |
| **¿Y si un lanzamiento rompe el proceso de pago?** | Las pruebas del pipeline deberían detectarlo; si no, la monitorización lanza una alerta y el equipo hace rollback a la versión anterior. |

Ninguna de estas preguntas trata de cómo escribir una función. Tratan de cómo se organizan las partes y de qué promete cada parte a las demás.

## Por qué esto importa para la arquitectura

El recorrido de un toque usa una docena de componentes, tres organizaciones y comunicación tanto síncrona como asíncrona, y aun así hubo que diseñarlo: alguien decidió que el pago es síncrono y las notificaciones no, que los pedidos residen en una base de datos relacional, que el servicio se ejecuta en contenedores repartidos en dos zonas, que el proveedor de pagos queda aislado detrás de un único módulo. Decidir cómo se organizan estas partes, qué cualidades logra cada organización y cuánto cuesta, y dejar constancia de por qué, es arquitectura de software. La forma en que se construye el software también importa a la arquitectura: una arquitectura que no se puede probar por partes, o que no se puede lanzar sin coordinar a diez equipos, ralentiza cada cambio. El Capítulo 1 comienza con qué es la arquitectura de software.

### COMPRUEBA LO QUE HAS ENTENDIDO

**▼ ¿Cuál es la diferencia entre la entrega continua y el despliegue continuo?**

Con la entrega continua, cada cambio que supera las comprobaciones puede lanzarse pulsando un botón, pero una persona decide cuándo. Con el despliegue continuo, cada cambio que supera las comprobaciones se lanza automáticamente.

**▼ ¿Por qué la pirámide de pruebas pone las pruebas unitarias en la base?**

Son rápidas, baratas y fiables, así que un equipo puede ejecutar miles con cada cambio y obtener respuesta en minutos. Las pruebas de extremo a extremo, más lentas y frágiles, se reservan para los pocos recorridos que las necesitan.

**▼ En el recorrido de un toque, ¿por qué la llamada de pago es síncrona y la notificación, asíncrona?**

La app no debe decir "confirmado" hasta que el pago esté aprobado, así que espera esa respuesta. La notificación puede enviarse un poco más tarde, así que pasa por una cola y su fallo no puede bloquear los pedidos.

**▼ ¿Cuáles son los dos principales beneficios de un pull request, además de detectar errores?**

Difunde el conocimiento, porque al menos otra persona entiende cada cambio, y mantiene la coherencia del diseño del código, porque los revisores pueden cuestionar las decisiones estructurales.

### FUENTES


- Martin Fowler: Continuous Integration: https://martinfowler.com/articles/continuousIntegration.html
- Martin Fowler: Continuous Delivery: https://martinfowler.com/bliki/ContinuousDelivery.html
- Ham Vocke: The Practical Test Pyramid (martinfowler.com): https://martinfowler.com/articles/practical-test-pyramid.html
- The 2020 Scrum Guide: https://scrumguides.org/scrum-guide.html
- Pro Git (Scott Chacon and Ben Straub), free online book: https://git-scm.com/book/en/v2

## Lecturas recomendadas

- *Code: The Hidden Language of Computer Hardware and Software (2nd edition)*, Charles Petzold (2022), Microsoft Press. Construye una computadora a partir de interruptores y códigos, en pasos fáciles de seguir; la mejor explicación no técnica de cómo el hardware ejecuta el software.
  https://www.microsoftpressstore.com/store/code-the-hidden-language-of-computer-hardware-and-software-9780137909100
- *Grokking Algorithms (2nd edition)*, Aditya Y. Bhargava (2024), Manning. Una introducción ilustrada y amable a las estructuras de datos, los algoritmos y la notación O grande (Big-O), ideal después de la sección de datos y algoritmos.
  https://www.manning.com/books/grokking-algorithms-second-edition
- *The Python Tutorial*, Python Software Foundation, docs.python.org. La introducción oficial y gratuita al lenguaje usado en los ejemplos de código de este capítulo, incluidas las funciones, los módulos y las clases.
  https://docs.python.org/3/tutorial/index.html
- *Learn web development*, MDN Web Docs contributors, Mozilla. Guías gratuitas y cuidadosamente mantenidas sobre cómo funciona la web, HTTP, HTML y JavaScript, escritas para principiantes.
  https://developer.mozilla.org/en-US/docs/Learn_web_development
- *Computer Networking: A Top-Down Approach (9th edition)*, James F. Kurose and Keith W. Ross (2025), Pearson. El libro de texto de referencia sobre redes, que empieza por la web y baja hasta los paquetes; léelo de forma selectiva cuando tengas claros los fundamentos.
- *Pro Git (2nd edition)*, Scott Chacon and Ben Straub (2014), Apress (free online at git-scm.com). El libro oficial y gratuito sobre Git: commits, ramas, fusiones y colaboración, tal como se presentan en la última sección.
  https://git-scm.com/book/en/v2
- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. Una definición breve y muy citada de la computación en la nube y de sus modelos de servicio y de implementación.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. El mejor paso siguiente cuando domines este capítulo: cómo se comportan realmente las bases de datos, las redes y los sistemas distribuidos, y por qué importa.

# Glosario

**ACID**
Atomicidad, consistencia, aislamiento y durabilidad: las garantías que ofrece una transacción de base de datos para que los datos sigan siendo correctos a pesar de caídas y usuarios simultáneos.

**Acoplamiento**
Cuánto depende una parte de un sistema de los detalles de otra. Un acoplamiento débil significa que un cambio en una parte rara vez obliga a hacer cambios en otras.

**Ágil**
Una familia de enfoques de desarrollo de software, formulada en el Manifiesto Ágil de 2001, que entregan software funcional en ciclos cortos y adaptan los planes según la retroalimentación.

**Algoritmo**
Un método preciso, paso a paso, para resolver un problema, como ordenar una lista o encontrar la ruta más corta, que siempre termina.

**Almacén clave-valor**
Base de datos que guarda valores bajo claves únicas y los recupera solo por clave, como un diccionario muy grande y muy rápido.

**Almacén de columnas anchas (wide-column)**
Base de datos NoSQL que agrupa los datos en filas identificadas por una clave, cada una con muchas columnas, repartidas entre muchas máquinas. Diseñada para volúmenes de escritura muy grandes.

**Ancho de banda**
Cuántos datos puede transportar una conexión por segundo, por ejemplo 100 megabits por segundo. Un ancho de banda alto no reduce la latencia.

**API (interfaz de programación de aplicaciones)**
Un conjunto definido de operaciones, entradas y salidas a través del cual un software usa otro, sin necesidad de saber cómo funciona por dentro.

**API web**
API ofrecida a través de una red, normalmente mediante HTTP, para que programas en máquinas distintas, escritos en lenguajes distintos, puedan trabajar juntos.

**Archivo**
Una secuencia de bytes con nombre almacenada en un disco y gestionada por el sistema operativo, como una foto, una hoja de cálculo o un registro.

**Balanceador de carga**
Componente que recibe las solicitudes entrantes y las reparte entre varios servidores, omitiendo los que no están en buen estado.

**Base de datos**
Una colección organizada de datos almacenada de modo que pueda consultarse, actualizarse y mantenerse coherente, normalmente gestionada por un sistema de gestión de bases de datos.

**Base de datos de grafos**
Una base de datos que almacena los elementos como nodos y las relaciones entre ellos como aristas, de modo que recorrer cadenas de conexiones es rápido.

**Base de datos documental**
Una base de datos NoSQL que almacena cada registro como un documento autónomo, normalmente de tipo JSON, que puede contener datos anidados y no necesita compartir un esquema fijo.

**Base de datos gestionada**
Servicio de base de datos operado por un proveedor de nube, que se encarga de la instalación, los parches, las copias de seguridad y la conmutación por error (failover), mientras el cliente gestiona el esquema, las consultas y el acceso.

**Base de datos relacional**
Base de datos que guarda los datos en tablas de filas y columnas y vincula las tablas mediante claves. Se consulta con SQL.

**Biblioteca**
Código reutilizable escrito por otra persona al que tu código llama cuando lo necesita, como una biblioteca para fechas o para enviar correo electrónico.

**Bit**
Un dígito binario, 0 o 1: la unidad de información más pequeña que almacena una computadora.

**Bug**
Un error en el software que hace que se comporte de forma distinta a la prevista, desde un bloqueo hasta una respuesta silenciosamente incorrecta.

**Byte**
Un grupo de ocho bits, capaz de contener 256 valores distintos. Los tamaños de los archivos y de la memoria se cuentan en bytes (kilobytes, megabytes, gigabytes).

**Bytecode**
Instrucciones compactas para una máquina virtual en lugar de para una CPU real. Java y Python compilan el código fuente a bytecode, que luego sus entornos de ejecución ejecutan en cualquier máquina compatible.

**Caché**
Una copia de datos guardada en un lugar al que se accede más rápido, para que las solicitudes repetidas eviten trabajo lento. Cambia memoria adicional, y el riesgo de datos desactualizados, por velocidad.

**Cambio incompatible (breaking change)**
Un cambio en una API que hace que el código cliente existente y correcto deje de funcionar, como renombrar o eliminar un campo.

**Centro de datos**
Un edificio diseñado para alojar muchos servidores con alimentación eléctrica fiable, refrigeración, seguridad física y conexiones de red.

**Clase y objeto**
Una clase es una plantilla que agrupa datos con las funciones (métodos) que trabajan con ellos; un objeto es una instancia creada a partir de esa plantilla, como la cuenta de un cliente concreto.

**Clave de API**
Una cadena secreta larga que identifica a la aplicación que llama ante una API. Cualquiera que tenga la clave puede actuar como esa aplicación, por lo que debe mantenerse en secreto.

**Clave foránea**
Una columna de una tabla que contiene la clave primaria de una fila de otra tabla, vinculando ambas. La base de datos puede rechazar valores que no apunten a nada.

**Clave primaria**
Columna, o conjunto de columnas, cuyo valor identifica de forma única cada fila de una tabla, como customer_id.

**Código de estado HTTP**
Un número de tres dígitos presente en cada respuesta HTTP que indica cómo ha ido la solicitud: 2xx éxito, 3xx redirección, 4xx un problema con la solicitud, 5xx un problema en el servidor.

**Código fuente**
El texto legible por personas de un programa, escrito en un lenguaje de programación como Python o Java, que luego se compila o se interpreta para que una computadora pueda ejecutarlo.

**Código máquina**
Instrucciones codificadas como números que una familia concreta de CPU puede ejecutar directamente. El código máquina de una familia (por ejemplo, x86-64) no se ejecuta en otra (por ejemplo, Arm).

**Cohesión**
Hasta qué punto el contenido de una parte de un sistema está relacionado entre sí. Una cohesión alta significa que un módulo hace un único trabajo claro.

**Cola de mensajes**
Software que almacena los mensajes de los emisores hasta que los receptores están listos para procesarlos, de modo que no es necesario que ambos estén en ejecución o sean rápidos al mismo tiempo.

**Commit**
Una instantánea guardada de los cambios en el control de versiones, con un autor, una marca de tiempo y un mensaje que explica el cambio.

**Compilador**
Un programa que traduce el código fuente a otra forma, como código máquina o bytecode, antes de que el programa se ejecute.

**Computación en la nube**
Alquilar recursos informáticos (servidores, almacenamiento, bases de datos y otros servicios) a través de internet, bajo demanda, y pagar por lo que usas.

**Contenedor**
Una aplicación empaquetada con sus bibliotecas y su configuración, que se ejecuta como un proceso aislado que comparte el kernel del sistema operativo del host. Es más ligero y arranca más rápido que una máquina virtual.

**Contrato de API**
La promesa documentada que hace una API: qué solicitudes acepta, qué devuelve cada una, qué errores pueden producirse y qué no cambiará sin previo aviso.

**Control de versiones**
Sistema que registra cada cambio en un conjunto de archivos, quién lo hizo y por qué, para que las personas puedan trabajar en paralelo, revisar los cambios y volver a cualquier versión anterior.

**Copia de seguridad**
Una copia de los datos guardada por separado del original para poder restaurarla tras una pérdida, corrupción o eliminación.

**CPU (procesador)**
La unidad central de procesamiento: el chip que ejecuta las instrucciones de un programa, un paso sencillo cada vez, miles de millones de veces por segundo. Las CPU modernas tienen varios núcleos que trabajan en paralelo.

**Dependencia**
Cualquier software que tu código necesita para funcionar, como una biblioteca, un framework u otro servicio. Si cambia o se rompe, tu código también puede romperse.

**Despliegue continuo**
Lanzar automáticamente a producción cada cambio que supera el pipeline, sin ningún paso de aprobación manual.

**DevOps**
Una cultura y un conjunto de prácticas en los que desarrollo y operaciones trabajan como uno solo, compartiendo la responsabilidad de construir, lanzar y operar el software, con el apoyo de la automatización y la medición.

**Diccionario (mapa)**
Una estructura de datos que almacena valores bajo claves únicas, de modo que un valor puede encontrarse directamente por su clave, normalmente en un tiempo aproximadamente constante. También se llama mapa o tabla hash.

**Dirección IP**
La dirección numérica que identifica un dispositivo en una red para que puedan entregársele paquetes, como 192.0.2.10 (IPv4) o 2001:db8::1 (IPv6).

**DNS**
Domain Name System (sistema de nombres de dominio): el servicio de directorio de internet, que convierte nombres como www.example.com en direcciones IP.

**Docker**
Un conjunto de herramientas muy utilizado, publicado en 2013, para construir imágenes de contenedor y ejecutar contenedores. Su formato de imagen se convirtió en un estándar abierto.

**Encapsulamiento**
Mantener privados los datos internos y los detalles de un objeto o un módulo, de modo que otro código solo pueda usarlo mediante las operaciones que decide ofrecer.

**Entorno**
Una copia separada y completa del sistema y su configuración que se usa con un fin, como desarrollo, pruebas, staging o producción.

**Entrega continua**
Mantener el software siempre en un estado apto para lanzarse, de modo que cualquier versión que supere el pipeline automatizado pueda ponerse en producción con solo pulsar un botón.

**Escalado horizontal (scaling out)**
Soportar más carga añadiendo más máquinas y repartiendo el trabajo entre ellas, normalmente detrás de un balanceador de carga.

**Escalado vertical (scaling up)**
Soportar más carga pasando a una máquina más grande con más procesadores o memoria.

**Esquema**
La estructura declarada de una base de datos: sus tablas, columnas, tipos de datos, claves y reglas.

**Estructura de datos**
Una forma de organizar los datos en memoria, como una lista, un diccionario, un conjunto, una cola, una pila, un árbol o un grafo, elegida para que las operaciones que necesitas sean rápidas.

**Framework**
Código reutilizable que proporciona la estructura general de una aplicación y llama a tu código en los momentos adecuados, como Django o Spring.

**Función**
Un fragmento de código reutilizable y con nombre que recibe entradas (parámetros), realiza un trabajo y normalmente devuelve una salida.

**Gestor de paquetes**
Herramienta, como pip (Python) o npm (JavaScript), que descarga bibliotecas y las bibliotecas de las que estas dependen, en las versiones que especifica un proyecto.

**Git**
El sistema de control de versiones más utilizado, creado por Linus Torvalds en 2005. Cada desarrollador tiene una copia completa del historial del proyecto.

**Hardware**
Las partes físicas de una computadora que podrías tocar: el procesador, la memoria, el almacenamiento, la tarjeta de red, la pantalla, etc.

**Hilo (thread)**
Una secuencia de instrucciones que se ejecuta dentro de un proceso. Los hilos de un mismo proceso comparten su memoria, de modo que pueden trabajar a la vez pero deben coordinarse.

**Hipervisor**
Software que crea y ejecuta máquinas virtuales, repartiendo entre ellas los procesadores, la memoria y los dispositivos de una computadora física y manteniéndolas aisladas.

**Historia de usuario**
Descripción breve de una funcionalidad desde el punto de vista de la persona que la quiere, a menudo escrita así: Como [rol], quiero [capacidad], para [beneficio].

**HTTP**
Hypertext Transfer Protocol: el lenguaje de solicitud y respuesta de la web, en el que un cliente envía un método como GET a una URL y un servidor responde con un código de estado y un contenido.

**IaaS (infraestructura como servicio)**
Un modelo de nube en el que alquilas máquinas virtuales, almacenamiento y redes, y gestionas tú mismo el sistema operativo y todo lo que está por encima.

**Imagen de contenedor**
Un paquete de solo lectura y versionado que contiene una aplicación y todo lo que necesita para ejecutarse, a partir del cual pueden iniciarse contenedores idénticos en cualquier lugar.

**Índice**
Una estructura de datos adicional, mantenida junto a los datos, que permite a un sistema encontrar elementos por un campo elegido sin recorrerlo todo, como el índice al final de un libro.

**Índice de base de datos**
Una estructura de datos adicional, normalmente un árbol B, que permite a la base de datos encontrar filas por el valor de una columna sin leer toda la tabla. Acelera las lecturas, pero ralentiza las escrituras y ocupa espacio.

**Integración continua (CI)**
La práctica de fusionar con frecuencia, al menos a diario, los cambios de cada desarrollador en la línea principal compartida, con una compilación y unas pruebas automatizadas que verifican cada cambio.

**Interfaz**
El conjunto acordado de operaciones que un software ofrece a otros, con sus entradas y salidas: un contrato que oculta cómo se hace el trabajo.

**Intérprete**
Un programa que lee código y ejecuta sus instrucciones sobre la marcha, en lugar de traducir todo el programa por adelantado.

**Inversión de control**
La organización en la que un framework, y no tu propio código, controla el flujo del programa y llama a tu código cuando hace falta.

**Join**
Operación SQL que combina filas de dos o más tablas haciendo coincidir columnas relacionadas, como los pedidos con los clientes que los hicieron.

**JSON (JavaScript Object Notation)**
Formato de texto para datos estructurados compuesto por objetos (pares nombre-valor entre llaves), arrays (listas entre corchetes), cadenas, números, true, false y null. Estandarizado en el RFC 8259.

**Kanban**
Forma de gestionar el trabajo como un flujo continuo en un tablero visual, limitando cuántos elementos pueden estar en curso a la vez.

**Latencia**
El retardo antes de que lleguen los datos o vuelva una respuesta, medido normalmente en milisegundos. La distancia, los saltos de red y el procesamiento lento se suman a ella.

**Límite de tasa**
Tope de cuántas solicitudes puede hacer un cliente en un periodo, como 100 por minuto. Las solicitudes que lo superan se rechazan, normalmente con el código de estado HTTP 429.

**Linux**
Núcleo de sistema operativo libre y de código abierto que, junto con otro software, se usa como sistema operativo en la mayoría de los servidores, máquinas en la nube y contenedores, así como en los teléfonos Android.

**Llamada síncrona**
Solicitud en la que quien llama espera la respuesta antes de continuar, como en una llamada normal a una API HTTP.

**Máquina virtual (VM)**
Imitación por software de una computadora completa, con su propio sistema operativo, que se ejecuta junto a otras VM en una máquina física bajo un hipervisor.

**Mensajería asíncrona**
Comunicación en la que el emisor deja un mensaje en algún lugar para que el receptor lo procese más tarde, y continúa sin esperar respuesta.

**Modelo cliente-servidor**
Una organización en la que los clientes (como los navegadores o las apps móviles) envían solicitudes y los servidores, que contienen los datos y la lógica compartidos, responden.

**Módulo**
Unidad de código, a menudo un solo archivo, que agrupa funciones y datos relacionados bajo un nombre y decide qué pone a disposición de otro código. Un paquete es un grupo de módulos.

**NoSQL**
Término general para las bases de datos que no usan el modelo relacional de tablas, como las bases de datos documentales, clave-valor, de columnas anchas y de grafos.

**Notación O grande (Big-O)**
Una forma de describir cómo crece el trabajo que hace un algoritmo a medida que crece la cantidad de datos, por ejemplo O(n) para un crecimiento lineal u O(1) para un tiempo constante.

**PaaS (plataforma como servicio)**
Modelo de nube en el que tú aportas el código o el contenedor de tu aplicación y el proveedor ejecuta, parchea y escala la plataforma que hay debajo.

**Pago por uso (pay-as-you-go)**
Modelo de precios en el que se te cobra por los recursos solo mientras los usas, por ejemplo por segundo de tiempo de servidor o por gigabyte almacenado.

**Paquete (de red)**
Pequeño fragmento de datos enviado por una red con información de direccionamiento adjunta. Los mensajes grandes se dividen en muchos paquetes y se vuelven a ensamblar en el otro extremo.

**Pipeline de CI/CD**
La secuencia automatizada de pasos (compilación, pruebas, despliegue en staging, despliegue en producción) por la que pasa cada cambio en su camino hacia los usuarios.

**Pirámide de pruebas**
Guía, de Mike Cohn, sobre las proporciones de las pruebas automatizadas: muchas pruebas unitarias rápidas, menos pruebas de integración y solo unas pocas pruebas lentas de extremo a extremo.

**Proceso**
Instancia en ejecución de un programa, con su propia memoria privada y su parte del tiempo de CPU, creada y gestionada por el sistema operativo.

**Producción**
El entorno real del que dependen los usuarios reales y los datos reales.

**Programa**
Lista de instrucciones precisas que una computadora sigue para hacer un trabajo. Un programa en ejecución se llama proceso.

**Prueba de extremo a extremo**
Una prueba automatizada que maneja todo el sistema como lo haría un usuario, por ejemplo, tocando las pantallas de una app, para comprobar un recorrido completo.

**Prueba de integración**
Una prueba automatizada que comprueba que varias partes funcionan correctamente juntas, como el código con una base de datos real o con la API de otro servicio.

**Prueba unitaria**
Prueba automatizada que comprueba una pequeña porción de código, como una función, de forma aislada, y que se ejecuta en milisegundos.

**Puerto**
Número de 0 a 65535 que identifica qué programa de un dispositivo debe recibir el tráfico de red, como el 443 para HTTPS.

**Pull request**
Solicitud para fusionar una rama en la línea principal, en un servicio como GitHub, donde los compañeros revisan el cambio y se ejecutan comprobaciones automáticas antes de aceptarlo.

**RAM (memoria)**
Memoria de acceso aleatorio (random-access memory): almacenamiento rápido y temporal que guarda los programas y datos en uso en ese momento. Su contenido se pierde cuando se corta la corriente o termina el programa.

**Rama**
Una línea de desarrollo separada en el control de versiones, que se usa para trabajar en un cambio sin afectar a la línea principal hasta que esté listo para fusionarse.

**Red de distribución de contenidos (CDN)**
Una red de servidores en muchas ubicaciones que guarda copias de contenido como imágenes, scripts y vídeo cerca de los usuarios, para que se cargue más rápido y el servidor de origen trabaje menos.

**Región**
Zona geográfica, como Europa Occidental o el este de EE. UU., en la que un proveedor de nube opera un grupo separado de centros de datos. Las regiones están aisladas entre sí.

**Requisito**
Enunciado de algo que un sistema debe hacer, o de una cualidad que debe tener, acordado con las personas que lo necesitan.

**REST (Representational State Transfer)**
Estilo arquitectónico para API web, descrito por Roy Fielding en 2000, en el que los clientes actúan sobre recursos identificados por URL mediante los métodos HTTP estándar.

**Revisión de código**
El examen por parte de colegas de un cambio de código propuesto antes de fusionarlo, para encontrar defectos, compartir conocimiento y mantener la coherencia del diseño.

**SaaS (software como servicio)**
Software completo que se usa por internet, normalmente desde un navegador o una app, con el proveedor operándolo todo.

**Scrum**
Marco ágil en el que un equipo pequeño trabaja en ciclos fijos llamados sprints, de un mes o menos, con eventos, responsabilidades y artefactos definidos.

**SDK (kit de desarrollo de software)**
Biblioteca, normalmente proporcionada por el propietario de una API, que envuelve las llamadas HTTP de la API en funciones normales para un lenguaje de programación concreto.

**Serverless y FaaS (función como servicio)**
Modelo de nube en el que subes funciones individuales que solo se ejecutan cuando un evento las desencadena; el proveedor se encarga de todos los servidores y pagas por ejecución.

**Servicio gestionado**
Servicio en la nube, como una base de datos, una cola o un servicio de almacenamiento, que el proveedor instala, opera, parchea y escala, de modo que los clientes lo usan sin tener que operarlo.

**Servidor**
Computadora, o programa que se ejecuta en ella, que espera solicitudes de otras computadoras (clientes) y las responde.

### Sistema de gestión de bases de datos (DBMS)

El software que almacena una base de datos y controla el acceso a ella: responde consultas, hace cumplir reglas, atiende a muchos usuarios a la vez y se recupera tras las caídas. PostgreSQL y MySQL son ejemplos.

### Sistema operativo

El software que gestiona el hardware de una computadora y lo reparte entre los programas: tiempo de CPU, memoria, archivos, red y seguridad. Algunos ejemplos son Linux, Windows, macOS, Android e iOS.

### Software

Los programas junto con los datos y la configuración que necesitan; las instrucciones que le dicen al hardware qué hacer.

### SQL (Structured Query Language)

El lenguaje estándar para definir, consultar y modificar datos en bases de datos relacionales. Describes el resultado que quieres y la base de datos averigua cómo obtenerlo.

### TCP

Transmission Control Protocol: entrega un flujo de datos fiable y ordenado entre dos programas, reenviando los paquetes perdidos. Tradicionalmente lo ha usado la mayor parte del tráfico web.

### Tipo de datos

El tipo de valor que es un dato, como un número entero, un número decimal, texto o verdadero/falso, que determina qué operaciones tienen sentido sobre él.

### TLS y HTTPS

Transport Layer Security cifra los datos entre dos programas y, mediante certificados, demuestra la identidad del servidor. HTTPS es HTTP transportado sobre TLS.

### Token de acceso

Una credencial de corta duración, emitida después de que un usuario o una aplicación inicia sesión, que se envía con cada solicitud a la API para demostrar lo que el llamante puede hacer.

### Transacción

Grupo de operaciones de base de datos tratadas como una unidad: o todas surten efecto (commit) o no lo hace ninguna (rollback).

### UDP

User Datagram Protocol: envía paquetes individuales sin garantía de entrega ni de orden. Es más rápido y sencillo que TCP; se usa en videollamadas, juegos y DNS.

### UTF-8

La forma dominante de codificar texto como bytes. Puede representar cualquier carácter Unicode, usando un byte para las letras básicas del inglés y hasta cuatro bytes para los demás.

**Versionado de API**
Publicar los cambios de una API bajo un nuevo identificador de versión, como /v2/, manteniendo la versión anterior en funcionamiento durante un tiempo para que los clientes puedan migrar a su ritmo.

**Webhook**
Solicitud HTTP que un proveedor envía a una URL registrada por su cliente cuando ocurre un evento, como un pago que se completa con éxito. Invierte la dirección habitual de una llamada a una API.

**Zona de disponibilidad**
Uno de varios grupos de centros de datos físicamente separados dentro de una región de la nube, cada uno con alimentación eléctrica, refrigeración y red independientes, lo bastante cerca entre sí para tener conexiones rápidas.