Desarrollo web completo

Aprende frontend, backend y nube — un concepto, tres lenguajes, y ningún prerrequisito

Stefa

2026-09-27

Tabla de contenidos

Prefacio

Este libro te lleva de **cero a desarrollo web completo**: primero aprendes frontend (lo que el navegador hace con tu código), luego backend y bases de datos (lo que tu servidor hace con los datos), y finalmente la nube (dónde vive todo eso en producción). Ningún prerrequisito asumido — si vienes "de la era de la AI" y nunca aprendiste esto en orden, el libro empieza donde estás.

Las tres secciones

Sección	Aprendes	Proyecto
I · Aprende Frontend	El navegador como runtime, JS/TS, componentes, llamadas a APIs, build	La UI de Taskflow
II · Aprende Backend y BD juntos	HTTP, APIs, validación, PostgreSQL, auth, testing — en 3 lenguajes	Taskflow + Nexus
III · Aprende Nube	Servidores reales, Docker, AWS/Azure/GCP, serverless	Taskflow en producción

Un concepto, tres lenguajes

En la Sección II, cada funcionalidad se explica como concepto universal y luego se implementa en tres stacks minimalistas:

0.0.1. TypeScript

Node.js + Express — lo más ubicuo del ecosistema que ya conoces.

0.0.2. Python

FastAPI — validación declarativa con type hints y Pydantic.

0.0.3. Go

`net/http` de la stdlib — el servidor *es* tu código, sin framework.

Después de cada ejemplo viene la pregunta importante: **¿por qué cada ecosistema lo resuelve así?** — y el bloque clicable **"Otras cimentaciones"**: las alternativas con su precio.

La filosofía

> Aprender el concepto JUSTO CUANDO se necesita para construir algo real — y reconocer el patrón, no memorizar la sintaxis.

- **En una frase** al abrir cada capítulo: qué vas a aprender antes de la teoría.
- **Prompts listos**: los ejemplos clave traen un prompt copiable para tu AI CLI.
- **No memorices**: usa la búsqueda del sidebar y lee con un AI al lado.

Etiquetas de concepto

- U Universal — vale en cualquier stack
- TS Específico de Node/TypeScript
- Py Específico de Python
- Go Específico de Go
- D Bases de datos y persistencia
- Ops Infraestructura, nube y operación

Cómo leer este libro

- **¿No sabes frontend?** Empieza en la Sección I — está hecha para ti.
- **Ya sabes frontend:** salta directo a la Sección II; la I queda como repaso rápido.
- Elige **un** lenguaje para implementar en la Sección II; los otros dos dan las soluciones de referencia.
- Cada capítulo cierra con un checklist: si no puedes marcarlo, vuelve atrás.

Parte I

Sección I · Aprende Frontend

1 F1. El navegador es un runtime

Qué pasa de verdad cuando escribes una URL y le das Enter

> **En una frase**
>
> Vas a entender **qué hace el navegador con tu código**: descarga archivos, construye el DOM, ejecuta JavaScript y pinta píxeles. Cuando entiendas que el navegador *es un runtime* (como Node, pero con pantalla), todo lo demás del frontend se acomoda solo.

1.1. El problema

Escribes `localhost:5173`, le das Enter, y aparece tu app. ¿Qué pasó en el medio? La mayoría de la gente que "sabe frontend" no puede responder esto completo — y es la base de todo lo que viene.

> **Explicación de: localhost / 127.0.0.1**
>
> **localhost** significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

```
Escribes URL → el navegador pide archivos → los interpreta → pinta
```

Ese flujo simple esconde cuatro actores: la red (HTTP), el parser (HTML→DOM), el motor JS (que ejecuta tu código) y el motor de renderizado (que pinta). Aquí los destapamos uno por uno — sin teoría vacía, cada uno con algo que puedes ver *ahora* en tu navegador.

> **Explicación de: DOM**
>
> El **DOM** es la página como árbol de objetos vivos: HTML es el papel, el DOM es lo que JavaScript puede tocar. `element.textContent = "hola"` cambia el DOM y el navegador repinta — el HTML original no se entera.

1.2. Cómo lo resuelve un equipo

Cuando algo "no se ve", un frontend senior no adivina: abre las DevTools y mira *qué etapa falló*:

- ¿Llegaron los archivos? → pestaña **Network** (¿algún request rojo 404/500?)
- ¿Se construyó el DOM? → pestaña **Elements** (¿el HTML tiene lo esperado?)
- ¿Corrió el JS? → pestaña **Console** (¿algún error rojo?)
- ¿Se pintó pero mal? → pestaña **Styles/Computed**

Esa división (red / estructura / lógica / pintura) *es* el mapa del navegador — y del debugging de frontend entero.

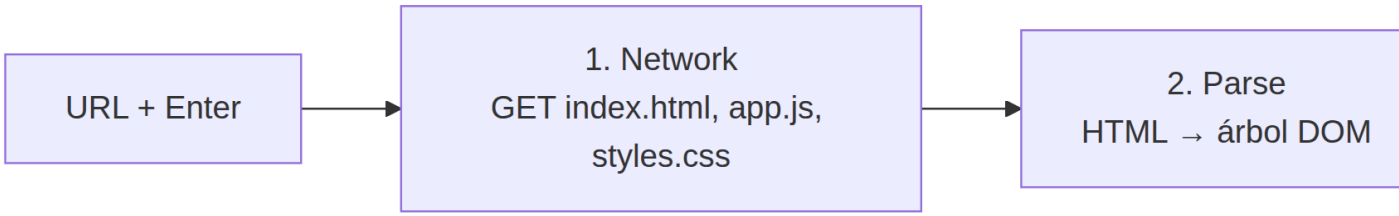
1.3. Conceptos nuevos

- U **Runtime**: el programa que ejecuta tu código. El navegador es el runtime del frontend; Node es el mismo motor (V8) sin pantalla.

> Explicación de: runtime
>
> El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

- U **DOM**: el árbol de objetos que el navegador construye al leer tu HTML — lo que JavaScript puede tocar.
- U **Render blocking**: el navegador pausa cuando encuentra `<script>` — por eso importa *dónde* pones los scripts.
- U **DevTools**: el debugger que ya llevas instalado — Network, Elements, Console, Sources.

1.4. La explicación visual



Míralo en vivo: abre cualquier página, presiona F12 → pestaña Network, recarga. Cada fila es un archivo que el navegador pidió — ese *es* el paso 1.

1.5. Implementación

La página mínima que demuestra las 4 etapas — crea `index.html` y ábrela:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    .card { border: 2px solid #2563eb; padding: 1rem; }
  </style>
</head>
<body>
  <div id="app" class="card">Cargando…</div>
  <script>
    // Paso 3: el motor JS toca el DOM construido en el paso 2
    document.getElementById("app").textContent =
      "El navegador construyó el DOM y JS lo modificó";
  </script>
</body>
</html>
```

Qué estás viendo: el navegador **parseó** el HTML a un árbol (paso 2), **ejecutó** el script que modificó ese árbol (paso 3) y **pintó** el resultado con el CSS (paso 4). El paso 1 fue traer `index.html` desde tu disco o servidor.

Prompt listo para tu AI

```
Explícame el ciclo de vida completo de una página web cuando escribo una
URL y presiono Enter: qué archivos pide el navegador, en qué orden, qué es
el DOM y cuándo se ejecuta el JavaScript. Usa este HTML de ejemplo:
[pegar tu index.html]. Quiero una explicación paso a paso, no una
definición de Wikipedia - y dime qué puedo ver en DevTools para
comprobar cada etapa.
```

1.6. ¿Por qué importa esto para backend?

Lo único que necesitas llevarte: el navegador es un *cliente HTTP con pantalla*. Cuando en la Sección II construyas el servidor, tu única misión será responder bien al paso 1 — el navegador hace el resto.

Detalle: navegador vs Node, el mismo motor

El motor JS de Chrome es **V8** — el mismo que corre Node.js. Por eso JavaScript "saltó" del navegador al servidor: Node es V8 + APIs de sistema (archivos, red, procesos) en vez de APIs de navegador (DOM, eventos, canvas). Mismo lenguaje, distintas herramientas alrededor — esa es toda la diferencia frontend/backend a nivel runtime.

1.7. Errores comunes

- **Confundir el archivo con lo que se ve**: el HTML que escribiste no es lo que se pinta — es lo que el DOM tiene *después* de que JS lo tocó. Mira Elements, no el archivo.
- **Script antes del DOM**: si tu script corre antes de que exista el elemento, `getElementById` devuelve `null`. Por eso los scripts van al final del `<body>` o con `defer`.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

- **Debuggear a ciegas**: editar HTML sin mirar Network/Console es apagar fuegos con los ojos cerrados.

1.8. Buenas prácticas

- **DevTools abiertas siempre** mientras desarrollas — son tu debugger, no un accesorio.
- **`type="module"` y `defer`** para scripts modernos — respetan el orden y no bloquean el parse.

Explicación de: serializar / parse

Serializar es convertir un objeto en memoria a texto viajero (`JSON.stringify`); **parsear/deserializar** es el viaje de vuelta (`JSON.parse`). Todo lo que cruza la red o la BD pasa por este par.

- **Pregunta "¿en qué etapa estoy?"** ante cualquier bug: ¿llegó el archivo? ¿está en el DOM? ¿corrió el JS? ¿se pintó?

1.9. Ejercicio

1. Crea el `index.html` de arriba, ábrelo y cambia el texto del div desde la **consola de DevTools** (`document.getElementById("app").textContent = "otro"`). ¿Qué pasó — el archivo cambió o el DOM?
2. En Network, recarga y encuentra el request de `index.html`: mira su status, su tamaño y su tiempo.

Explicación de: request / petición

Un **request** (petición) es el mensaje que un cliente le manda al servidor: "quiero X". Tiene una dirección (URL), un verbo (GET, POST…), headers (metadatos) y a veces un body (los datos). La respuesta es el **response**.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

1.10. Mini reto

Abre una web real (cualquiera) y responde solo con DevTools: ¿cuántos requests hizo al cargar? ¿cuál fue el más lento? ¿hay algún error en Console? Esas tres respuestas son el 80 % del diagnóstico frontend.

Soluciones

Ejercicio 1. Cambió el **DOM en memoria**, no el archivo — si recargas, vuelve el texto original. Esa es *la* lección del capítulo: el navegador pinta el DOM (árbol vivo), no tu archivo.
Ejercicio 2. En Network verás `index.html` con status `200`, tipo `document`, y su tiempo de descarga — el paso 1 del diagrama.
Mini reto. Respuesta modelo: una página típica hace 20–80 requests (HTML + JS + CSS + imágenes + APIs). El más lento suele ser un bundle JS grande o una API — ordena por columna "Time" en Network. Errores rojos en Console = JS que falló en el paso 3.

1.11. Vocabulario técnico del capítulo

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: tipos / tipado estático

Los **tipos** (`int`, `string`, `boolean`, `Task`) son el contrato de cada dato. *Tipado estático* (TypeScript, Go, Python con hints) = los tipos se revisan al escribir, no cuando explota en producción.

Explicación de: cliente

El **cliente** es quien *pide*: el navegador, una app móvil, otro servidor. El **servidor** es quien *atiende* y responde. La misma computadora puede ser cliente de una cosa y servidor de otra.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: evento

Un **evento** es "algo que pasó" convertido en dato: el click del usuario, el `payment.succeeded` del webhook, el mensaje del socket. La programación moderna es reaccionar a eventos más que seguir un guion.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: API

Una **API** (Application Programming Interface) es el menú de un programa: la lista de operaciones que otros programas pueden pedirle. Tu frontend no habla con la base de datos — le pide cosas a la API, y la API decide.

1.12. Lo que deberías saber hacer ahora

☐ Explicar las 4 etapas (network → DOM → JS → render) sin mirar

☐ Diagnosticar “no se ve nada” ubicando qué etapa falló con DevTools
☐ Explicar por qué el archivo HTML y lo que se ve no son lo mismo
☐ Decir qué comparten el navegador y Node.js (y qué no)

2 F2. JavaScript y TypeScript: lo esencial para este libro

Solo las piezas del lenguaje que el libro usa — nada de relleno

> **En una frase**
>
> El JavaScript/TypeScript mínimo que necesitas: variables, funciones, objetos, arrays, módulos y `async/await`. Si vienes "de la era de la AI" y nunca aprendiste esto en orden, este capítulo es tu base — todo lo demás del libro se apoya aquí.

2.1. El problema

Sabes pedirle a una AI "hazme un componente" — pero cuando el código que devuelve usa `.map()`, `...spread`, `async/await` o `import`, ¿entiendes qué hace cada pieza? Sin esa base, eres copiloto sin saber volar: la AI escribe y tú no puedes revisarla.

> **Explicación de: bucle (loop)**
>
> Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

> **Explicación de: return**
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

> **Explicación de: asíncrono**
>
> **Asíncrono** = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: módulo / import

Un **módulo** es un archivo de código que exporta cosas para que otros archivos las importen. Es la unidad de organización: en vez de un archivo gigante, el código vive en módulos con responsabilidad propia.

Este capítulo no es un curso de JavaScript — es **el inventario exacto de piezas que el libro usa**, cada una con un ejemplo que puedes correr en la consola del navegador *ahora mismo* (F12 → Console).

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

2.2. Cómo lo resuelve un equipo

En un equipo real nadie memoriza el lenguaje entero — se domina el 20 % que se usa el 95 % del tiempo, y el resto se busca. Ese 20 % en este libro es:

```
variables → objetos/arrays → funciones → módulos → async → tipos TS
```

Cada pieza abajo tiene: qué es, un ejemplo mínimo, y *por qué la vas a usar en backend.*

2.3. Conceptos nuevos

- U `const/let` — variables (usa `const` por defecto).
- U Objetos y arrays — las estructuras que viajan en JSON.
- U `.map()` / `.filter()` — transformar colecciones sin loops.
- U `import/export` — módulos ES.
- U `async/await` — esperar operaciones lentas sin bloquear.
- TS Type annotations — el "JS con etiquetas" que es TypeScript.

2.4. Las piezas, una por una

2.4.1. Variables y objetos

```
const title = "deploy";           // const: no se reasigna
let count = 0;                    // let: sí puede cambiar
count = count + 1;
```

```
const task = {                    // objeto = el JSON que ya viste
  id: 1,
  title: "deploy",
  done: false,
};
task.title;                       // "deploy" - acceso por punto
const { title: t } = task;        // destructuring: sacar campos
```

Por qué importa: un endpoint devuelve objetos; un body es un objeto. Si lees `{ id, title, done }` como "una tarea", ya tienes el 80 % del JSON del libro.

2.4.2. Arrays y transformaciones

```
const tasks = [
  { id: 1, title: "a", done: true },
  { id: 2, title: "b", done: false },
];

tasks.map((t) => t.title);          // ["a","b"] - transforma cada uno
tasks.filter((t) => !t.done);       // [{id:2…}] - deja solo los que cumplen
tasks.find((t) => t.id === 2);      // la tarea o undefined
```

Por qué importa: `GET /tasks` devuelve un array; tu backend hará `.find()`/`filter()` constantemente antes de que exista la base de datos.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

2.4.3. Funciones

```
// tres formas de la misma función
function add(a, b) { return a + b; }
const add2 = (a, b) => a + b;              // arrow function
const add3 = async (a, b) => a + b;        // async (devuelve Promise)
```

Por qué importa: un *handler* es literalmente una función que recibe el request y devuelve la respuesta. Ya sabes handlers — no lo llamabas así.

2.4.4. Módulos

```
// services/tasks.js
export function createTask(title) { … }

// index.js
import { createTask } from "./services/tasks.js";
```

Por qué importa: el Cap. 7 organiza el backend en archivos — `import` es cómo se conectan.

2.4.5. async/await

```
const res = await fetch("/tasks");   // pausa aquí hasta que responde
const tasks = await res.json();      // otra pausa: parsear el body
```

`async` marca que la función puede pausar; `await` espera el resultado de algo lento (red, disco). **Por qué importa:** todo lo que tarda (llamar APIs, leer la BD) es async — en los tres lenguajes del libro.

2.4.6. TypeScript: JS + etiquetas

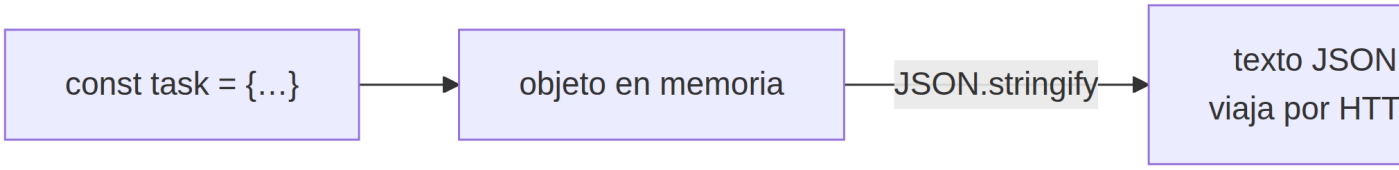
```
interface Task { id: number; title: string; done: boolean }

function createTask(title: string): Task {
  return { id: 1, title, done: false };
}
```

Por qué importa: TypeScript es JS con *etiquetas de tipo* que el editor chequea antes de correr. Las etiquetas desaparecen al ejecutar — son andamiaje, no runtime (el Cap. 4 vuelve a esto).

> Explicación de: runtime
>
> El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

> Explicación de: roles / RBAC
>
> Los **roles** agrupan permisos: *viewer* lee, *editor* escribe, *admin* gestiona. RBAC = el permiso depende del rol que tienes *en ese recurso* — puedes ser admin de un equipo y viewer de otro.

2.5. La explicación visual



Objeto JSON objeto: ese viaje es toda la comunicación frontend backend.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Prompt listo para tu AI

```
Quiero un repaso práctico de JavaScript moderno SOLO de estos temas:
const/let, objetos, arrays con map/filter/find, arrow functions,
import/export de ES modules, async/await, y qué agrega TypeScript encima.
Para cada tema: un ejemplo de 3-5 líneas que pueda pegar en la consola del
navegador, qué imprime, y un uso real en una API. No me des un curso
completo - solo estas piezas.
```

2.6. Errores comunes

- **Confundir `map` con `forEach`**: `.map()` *devuelve* un array nuevo (útil); `.forEach()` solo itera (no devuelve nada).
- **`await` fuera de `async`**: error de sintaxis — el `await` necesita vivir dentro de una función `async` (o top-level en módulos modernos).
- **Creer que los tipos TS existen en runtime**: `interface` se evapora al compilar — no valida nada que llegue por red (ese es el trabajo del Cap. 4).

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

- **`const` no es inmutable**: `const task = {}` no se reasigna, pero `task.done = true` sí muta el contenido.

2.7. Buenas prácticas

- **`const` por defecto**, `let` solo cuando reasignas — menos sorpresas.
- **Destructuring al recibir objetos**: `const { title } = req.body` es más legible que `req.body.title`.
- **Una exportación por concepto**: funciones pequeñas con nombre claro > un archivo gigante.
- **Aprende a leer antes que a escribir**: si entiendes `.map().filter()`, ya lees el 70 % del código frontend y backend real.

2.8. Ejercicio

En la consola del navegador (F12):

1. Crea un array de 3 tareas y usa `.filter()` para quedarte con las no hechas; `.map()` para sacar solo los títulos.
2. Escribe `JSON.stringify(tuArray)` y luego `JSON.parse()` del resultado — verifica que obtienes el mismo array.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: serializar / parse

Serializar es convertir un objeto en memoria a texto viajero (`JSON.stringify`); **parsear/deserializar** es el viaje de vuelta (`JSON.parse`). Todo lo que cruza la red o la BD pasa por este par.

2.9. Mini reto

Sin ejecutarlo: ¿qué devuelve esto y por qué?

```
const tasks = [{ id: 1, done: false }, { id: 2, done: true }];
tasks.filter(t => t.done).map(t => t.id);
```

Pista: encadena las dos operaciones — primero filtra, luego transforma.

Soluciones

Ejercicio 1.

```
const tasks = [
  { id: 1, title: "a", done: true },
  { id: 2, title: "b", done: false },
  { id: 3, title: "c", done: false },
];
tasks.filter(t => !t.done);        // [{id:2…},{id:3…}]
tasks.map(t => t.title);           // ["a","b","c"]
```

Ejercicio 2. `JSON.stringify` produce texto (`'[{"id":1,…}]'`); `JSON.parse` lo convierte de vuelta a objetos — el mismo array. Ese ida y vuelta es literalmente lo que hacen `fetch` y tu futuro servidor.
Mini reto. `[2]`: `filter` deja solo `{id:2, done:true}`, luego `map` extrae su `id`. Encadenar `filter().map()` es el patrón más común del mundo para transformar colecciones — lo verás cientos de veces.

2.10. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de “cadena de caracteres” — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: booleano

Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

2.11. Lo que deberías saber hacer ahora

- ☐ Usar `.map()`, `.filter()`, `.find()` sin dudar
- ☐ Explicar qué hacen `async`/`await` y por qué la red es async
- ☐ Leer un `import`/`export` y decir qué se mueve entre archivos
- ☐ Explicar qué es un `interface` de TS y por qué desaparece en runtime
- ☐ Convertir un objeto a JSON y de vuelta — y decir por qué importa

3 F3. DOM, eventos y estado: por qué existe React

El problema que los frameworks resuelven — mostrado a mano primero

> **En una frase**
>
> Vas a tocar el DOM a mano (sin frameworks): seleccionar elementos, escuchar clicks y **descubrir por tu cuenta el problema que React resolvió** — mantener la pantalla sincronizada con los datos. Es el capítulo que explica por qué los frameworks existen.

3.1. El problema

Tienes una lista de tareas en memoria (un array de JS) y quieres verla en pantalla. Con HTML puro ya sabes escribir `<ul><li>…</li></ul>` — pero la lista *cambia*: el usuario agrega tareas, las marca hechas, las borra. ¿Cómo haces que la pantalla refleje el array actual?

> **Explicación de: array / lista / slice**
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> **Explicación de: memoria (RAM)**
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

La respuesta ingenua: "cuando cambie el array, reescribo el HTML". Intenta hacerlo a mano y sentirás el dolor que React vino a curar.

3.2. Cómo lo resuelve un equipo

Antes de React (2013), los equipos sincronizaban datos pantalla a mano con jQuery — y las apps grandes se volvían spaghetti de `$(this).find(...)` imposibles de razonar. La idea que cambió todo:

La pantalla es una función del estado. `UI = f(state)`. No actualices el DOM elemento por elemento; declara cómo se ve la UI para cada estado y deja que el framework calcule el cambio.

3.3. Conceptos nuevos

- U **DOM**: el árbol vivo que JS puede modificar (`document.getElementById`, `createElement`, `append`).
- U **Eventos**: clicks, inputs — el usuario "emite" eventos y tú registras listeners.

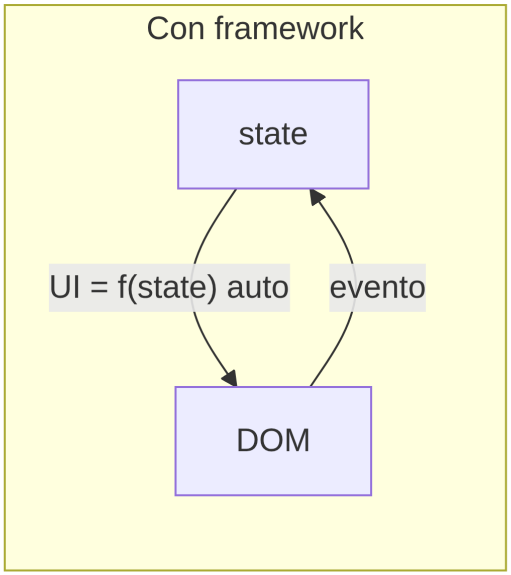
> Explicación de: evento
>
> Un **evento** es "algo que pasó" convertido en dato: el click del usuario, el `payment.succeeded` del webhook, el mensaje del socket. La programación moderna es reaccionar a eventos más que seguir un guion.

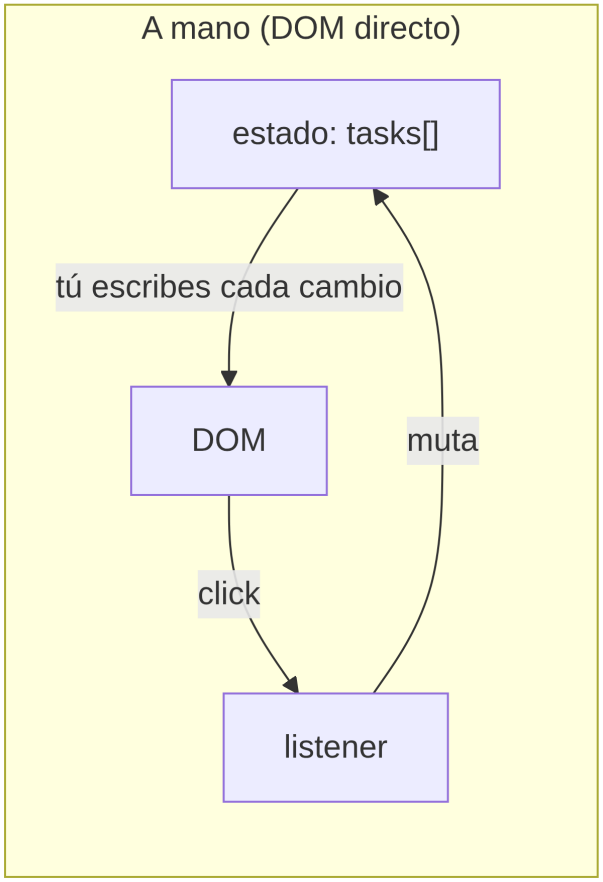
- U **Estado**: los datos que cambian en el tiempo (el array de tareas) — la fuente de verdad.
- U **Re-render**: volver a pintar cuando el estado cambia — a mano es tedioso; los frameworks lo automatizan.

> Explicación de: framework
>
> Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

> Explicación de: estado (state)
>
> El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

3.4. La explicación visual





En el modelo manual *tú* escribes cada actualización del DOM; en el de framework tú cambias el estado y la UI se recalcula sola.

> Explicación de: DOM
>
> El **DOM** es la página como árbol de objetos vivos: HTML es el papel, el DOM es lo que JavaScript puede tocar. `element.textContent = "hola"` cambia el DOM y el navegador repinta — el HTML original no se entera.

3.5. Implementación: el dolor a mano (y por qué importa)

Un contador + lista con DOM puro — fíjate en cuánto código es solo "sincronizar pantalla":

```
<ul id="list"></ul>
<input id="newTask" placeholder="nueva tarea" />
<button id="add">Agregar</button>

<script>
  const tasks = [];   // ← el ESTADO: la fuente de verdad

  function render() {            // ← reescribir el DOM entero cada vez
    const ul = document.getElementById("list");
    ul.innerHTML = "";
    tasks.forEach((t) => {
      const li = document.createElement("li");
      li.textContent = t;
      ul.append(li);
    });
  }

  document.getElementById("add").addEventListener("click", () => {
    tasks.push(document.getElementById("newTask").value);  // cambia estado
    render();                                              // sincroniza a mano
  });
</script>
```

Qué estás viendo: **toda la complejidad está en `render()`** — reescribir el DOM cada vez que el estado cambia. Con 3 elementos es manejable; con 50 componentes anidados es el infierno que React vino a resolver: tú declaras `UI = f(state)` y React hace el `render()` por ti.

Prompt listo para tu AI

```
Quiero entender por qué existen los frameworks frontend. Muéstrame el
mismo ejercicio (una lista donde agrego ítems con un input) en DOS
versiones: (1) DOM puro con addEventListener y re-render manual, (2)
React con useState. Para cada una explica qué línea sincroniza el estado
con la pantalla - quiero ver exactamente qué trabajo me ahorra el
framework.
```

Otras cimentaciones: ¿siempre un framework para el DOM?

Esta decisión es material: *sincronizar estado pantalla* es estructural; cómo hacerlo varía mucho.

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
DOM manual (este capítulo)	`render()` a mano	Crece mal con la complejidad	Páginas casi estáticas, aprender
React (canónico)	`UI = f(state)`, JSX	Curva de conceptos (hooks)	El default de la industria
Vue / Svelte	Reactividad más directa	Ecosistema menor que React	Equipos que la prefieren — igual de válidas
HTMX / Alpine	Reactividad mínima en HTML	No escala a apps complejas	Apps server-rendered con poca interactividad

3.6. Errores comunes

- `innerHTML +=`: borra y recrea todo el DOM — pierde el foco, el scroll y es un agujero de seguridad (inyección). Los frameworks lo evitan.
- **Estado duplicado**: si el dato vive en el array *y* en el DOM, se desincronizan. Una sola fuente de verdad: el estado.
- **Confundir evento con estado**: un click no "guarda" nada — dispara una función que cambia el estado, y *eso* repinta.

3.7. Buenas prácticas

- **El estado es la fuente de verdad**: el DOM solo lo *refleja*; nunca leas datos "desde el DOM".
- **Un evento → cambia estado → re-render**: ese ciclo es el latido de toda app frontend, con o sin framework.
- **Entiende el dolor manual antes del framework**: quien entiende qué resuelve React la usa mejor.

3.8. Ejercicio

1. Completa el ejemplo del contador agregando un botón "Borrar última" que quite el último item del array y re-renderice.
2. Agrega un `<span>` que muestre `tasks.length` — actualízalo en `render()`.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

3.9. Mini reto

En una frase: ¿qué es exactamente lo que React te ahorra escribir? Si tu respuesta menciona "el render manual", entendiste el capítulo.

Soluciones

Ejercicio 1. `tasks.pop(); render();` — el patrón es siempre: mutar estado → llamar render.
Ejercicio 2. `document.getElementById("count").textContent = tasks.length;` dentro de `render()` — otro dato derivado del estado.
Mini reto. React te ahorra escribir el `render()` y la sincronización manual del DOM: declaras cómo se ve la UI para cada estado (`UI = f(state)`) y el framework calcula el cambio mínimo. Todo lo demás (JSX, hooks, componentes) son detalles de esa idea central.

3.10. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

3.11. Lo que deberías saber hacer ahora

- ☐ Seleccionar y modificar elementos del DOM con JS puro
- ☐ Registrar un `addEventListener` y explicar qué pasa al click
- ☐ Explicar "UI = f(state)" y por qué fue la idea clave
- ☐ Decir qué es la fuente de verdad en una app (el estado, no el DOM)

4 F4. Frameworks: el mismo componente en cuatro sabores

React, Vue, Angular y Svelte resolviendo el mismo problema

> **En una frase**
>
> El mismo componente — un contador con un botón — escrito en **JS vanilla, React, Vue, Angular y Svelte**. Verás que el concepto (estado → UI) es idéntico y solo cambia la sintaxis: elegir framework es elegir dialecto, no religión.

4.1. El problema

"¿Qué framework aprendo?" es la pregunta que paraliza a todo el que empieza. La respuesta honesta: **el concepto es uno** — componentes con estado que repintan la UI. Los frameworks son cuatro dialectos de esa idea. Verlos lado a lado resolviendo *exactamente lo mismo* es la forma más rápida de entenderlos todos.

> **Explicación de: framework**
>
> Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

> **Explicación de: estado (state)**
>
> El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

4.2. Cómo lo resuelve un equipo

Un equipo no elige "el mejor" — elige por contexto:

- ¿Qué usa ya la empresa/equipo? (la razón #1 en la vida real)

- ¿Cuánta estructura quieres impuesta? (Angular la impone toda; React te deja decidir)
- ¿Tamaño del ecosistema y contratación? (React gana por goleada)

4.3. Conceptos nuevos

- U **Componente**: una pieza de UI con su estado y su vista — botón, tarjeta, página entera.
- U **Reactividad**: estado cambia → UI se actualiza sola. Cada framework la implementa distinto.
- U **Vite + dev server**: la herramienta que sirve tu app en desarrollo (`npm run dev`).

> Explicación de: paquete / package manager
>
> Un **paquete** es una librería empaquetada lista para instalar. El *package manager* (npm, pip, go mod) las descarga, resuelve versiones compatibles y las registra en `package.json` / `pyproject.toml` / `go.mod`.

- U **JSX / templates / SFC**: cómo cada framework escribe "HTML con lógica dentro".

4.4. Implementación: el contador, cinco formas

4.4.1. Fundamentos (JS vanilla)

```
<div id="app"></div>
<script>
  let count = 0;                    // estado
  function render() {               // UI = f(state), a mano
    document.getElementById("app").innerHTML =
      `<p>Count: ${count}</p><button id="b">+1</button>`;
    document.getElementById("b").onclick = () => { count++; render(); };
  }
  render();
</script>
```

Ya conoces el patrón del Cap. F3: mutas estado → llamas render. Todo lo demás es esto mismo con mejor ergonomía.

4.4.2. React (canónico)

```
import { useState } from "react";

export default function Counter() {
  const [count, setCount] = useState(0);   // estado declarativo
  return (
```

```
    <div>                                    {/* UI = f(state) */}
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

`useState` devuelve `[valor, setter]`: llamar `setCount` cambia el estado **y React re-renderiza solo** — no escribes `render()`.

4.4.3. Vue

```
<script setup>
import { ref } from "vue";
const count = ref(0);              // estado reactivo
</script>

<template>
  <p>Count: {{ count }}</p>
  <button @click="count++">+1</button>
</template>
```

`ref` hace el dato reactivo: `count++` actualiza el estado y el template se repinta solo — ni `setCount` ni `render`.

4.4.4. Angular

```
import { Component } from "@angular/core";

@Component({
  selector: "app-counter",
  template: `
    <p>Count: {{ count }}</p>
    <button (click)="count = count + 1">+1</button>
  `,
})
export class CounterComponent {
  count = 0;   // el estado es un campo de la clase
}
```

Angular organiza en clases + decoradores: el template reacciona a cambios del campo. Más estructura impuesta, más "framework completo".

4.4.5. Svelte

```
<script>
  let count = 0;    // variable normal - Svelte la hace reactiva al compilar
</script>

<p>Count: {count}</p>
<button on:click={() => count++}>+1</button>
```

Svelte es el más "mágico": escribes JS casi normal y el compilador genera el código reactivo — el que menos boilerplate tiene.

4.5. ¿Por qué cada framework lo hace así?

Lo único que necesitas llevarte: los cinco hacen `estado → UI`. La diferencia real es *cuánto decides tú* vs *cuánto decide el framework*: React y Svelte te dan un mecanismo; Vue un sistema; Angular una arquitectura completa.

Detalle: cómo reacciona cada uno

- **React**: re-ejecuta la función del componente y compara el resultado (virtual DOM) — explícito, predecible.
- **Vue**: `ref` envuelve el dato y rastrea qué lo usa — reactividad automática por proxys.
- **Angular**: detección de cambios sobre la clase + signals modernos — el más estructurado (inyección de dependencias nativa, como el backend de NestJS).
- **Svelte**: el compilador detecta qué variables usa el template y genera updates quirúrgicos del DOM — sin virtual DOM, más rápido y simple.

Prompt listo para tu AI

```
Muéstrame el MISMO componente (un contador con botón que incrementa) en
React, Vue, Angular y Svelte. Para cada uno: el código mínimo completo,
qué línea declara el estado, y qué línea hace que la UI se actualice.
Después compáralos en una tabla: sintaxis de estado, cómo se re-renderiza,
y qué tanto decide el framework por mí.
```

4.6. Herramientas: cómo arranca un proyecto frontend

Todos comparten el mismo ritual (idéntico al del backend del Cap. 2):

```
npm create vite@latest my-app -- --template react   # o vue / svelte / vanilla
cd my-app && npm install
npm run dev    # → http://localhost:5173 con hot reload
```

Vite es el dev server estándar hoy: sirve tus archivos, compila JSX/TS al vuelo y recarga al guardar. El `npm run dev` del frontend es el gemelo del `uvicorn --reload` del backend.

4.7. Errores comunes

- **"Aprender React" sin entender estado**: si no entiendes `estado → UI` del Cap. F3, los hooks se sienten mágicos y frágiles.
- **Mutar el estado directamente en React**: `count++` no repinta — hay que llamar `setCount`. En Vue/Svelte sí se muta.
- **Peleas de frameworks**: el mejor framework es el del equipo que te contrata — el concepto transferible es lo que importa.

4.8. Buenas prácticas

- **Aprende UNO bien, lee los demás**: con React sólido, Vue/Svelte se aprenden en días — el concepto es el mismo.
- **Componentes pequeños**: un componente = una pieza de UI; si crece, se divide.
- **Estado lo más cerca posible de quien lo usa** — el global es para lo que de verdad comparten muchos.

4.9. Ejercicio

1. `npm create vite@latest counter -- --template react`, corre `npm run dev` y cambia el texto del botón a "incrementar".
2. Agrega un segundo botón "-1" al contador.

4.10. Mini reto

Explica con tus palabras por qué en React escribes `setCount(count + 1)` pero en Vue/Svelte basta `count++`. ¿Qué está haciendo cada framework por debajo?

Soluciones

Ejercicio 2.

```
<button onClick={() => setCount(count - 1)}>-1</button>
```

Mini reto. React necesita el setter porque re-ejecuta el componente solo cuando el estado cambia *a través de su API* — `count++` mutaría la variable sin avisarle a React. Vue/Svelte interceptan la asignación (proxy/compilador) y *saben* que el dato cambió solos. Misma reactividad, distinto mecanismo de "aviso".

4.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: módulo / import

Un **módulo** es un archivo de código que exporta cosas para que otros archivos las importen. Es la unidad de organización: en vez de un archivo gigante, el código vive en módulos con responsabilidad propia.

Explicación de: API

Una **API** (Application Programming Interface) es el menú de un programa: la lista de operaciones que otros programas pueden pedirle. Tu frontend no habla con la base de datos — le pide cosas a la API, y la API decide.

Explicación de: DOM

El **DOM** es la página como árbol de objetos vivos: HTML es el papel, el DOM es lo que JavaScript puede tocar. `element.textContent = "hola"` cambia el DOM y el navegador repinta — el HTML original no se entera.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: dev server / hot reload

El **dev server** (Vite, `npm run dev`) sirve tu app mientras desarrollas y recarga el navegador al guardar — *hot reload*. No es el servidor de producción: es la mesa de trabajo.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

Explicación de: reverse proxy

Un **reverse proxy** (nginx, ALB, Cloudflare) es el portero: recibe todo el tráfico en el 443, termina TLS y reparte a tu app interna. La app nunca mira a internet directamente — el proxy la protege.

4.12. Lo que deberías saber hacer ahora

- ☐ Explicar qué es un componente y el ciclo estado→UI
- ☐ Leer un componente en React, Vue o Svelte e identificar su estado
- ☐ Arrancar un proyecto con Vite y explicar qué hace `npm run dev`
- ☐ Argumentar por qué elegir framework es contexto, no religión

5 F5. Hablar con una API desde el frontend

fetch, axios, React Query y HttpClient — el cliente del contrato

> **En una frase**
>
> Todas las formas de llamar una API desde el cliente: `fetch` nativo, `axios`, React Query/SWR y el `HttpClient` de Angular — y por qué en una app real no basta un `fetch` suelto. Este capítulo es el puente: el cliente que consume el contrato que tú construirás en la Sección II.

5.1. El problema

Tu componente necesita las tareas del servidor. La primera aproximación:

```
const res = await fetch("/api/tasks");
const tasks = await res.json();
```

Funciona — hasta que necesitas *loading* mientras carga, *error* si falla, *reintento* si la red se cae, *caché* para no pedir lo mismo, y *mutación* para el POST de crear. Ahí es donde un `fetch` suelto se queda corto y nace el ecosistema de clientes.

5.2. Conceptos nuevos

- U **fetch nativo**: la Promise del navegador para HTTP — la base de todo.
- U **Estados de la llamada**: loading / error / data — el ciclo completo de una petición real.
- U **Cliente HTTP con extras**: axios (interceptors, JSON auto), React Query (caché + estados).

> **Explicación de: estado (state)**
>
> El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

- U **Observable** (Angular): stream reactivo en vez de Promise — el modelo de RxJS.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

5.3. Implementación: de menos a más

5.3.1. fetch nativo

```
async function getTasks() {
  const res = await fetch("/api/tasks");
  if (!res.ok) throw new Error(`HTTP ${res.status}`);  // 404 no lanza solo
  return res.json();
}

// POST con body JSON:
await fetch("/api/tasks", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "ship v1" }),
});
```

Tres detalles: `fetch` **no rechaza en 4xx/5xx** (revisa `res.ok`), el body se parsea con `.json()`, y `credentials: "include"` envía cookies.

5.3.2. axios

```
import axios from "axios";
const { data } = await axios.get("/api/tasks");     // JSON automático
await axios.post("/api/tasks", { title: "ship v1" });

// interceptor: agrega auth a TODOS los requests
axios.interceptors.request.use((cfg) => {
  cfg.headers.Authorization = `Bearer ${token}`;
```

```
  return cfg;
});
```

Axios lanza error en 4xx/5xx (mejor que fetch) y los interceptors son middleware del lado cliente.

5.3.3. React Query (TanStack)

```
import { useQuery, useMutation } from "@tanstack/react-query";

function Tasks() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["tasks"],
    queryFn: () => fetch("/api/tasks").then((r) => r.json()),
  });

  const createTask = useMutation({
    mutationFn: (t) =>
      fetch("/api/tasks", { method: "POST", body: JSON.stringify(t) }),
  });

  if (isLoading) return <p>Cargando…</p>;
  if (error) return <p>Error: {error.message}</p>;
  return <ul>{data.map((t) => <li key={t.id}>{t.title}</li>)}</ul>;
}
```

`useQuery` resuelve el ciclo completo: loading/error/data + caché + revalidación + reintentos — lo que a mano son 40 líneas de `useState`/ `useEffect`.

5.3.4. Angular HttpClient

```
// Angular devuelve un Observable, no una Promise
this.http.get<Task[]>("/api/tasks").subscribe({
  next: (tasks) => (this.tasks = tasks),
  error: (err) => console.error(err.status),
});
```

Observable vs Promise: el Observable es *frío* — no hay request hasta `.subscribe()` — puede emitir varios valores y se cancela con `unsubscribe()`. Angular trae interceptors nativos para auth.

5.4. ¿Por qué tantas opciones?

Lo único que necesitas llevarte: todas hablan HTTP igual — la diferencia es *cuánto te ayudan con el ciclo completo* (loading, error, caché). `fetch` es la base; axios/React Query agregan la gestión que una app real necesita.

Detalle: Promise vs Observable

Promise (`fetch`, axios): un valor futuro, se dispara al crearla, no se cancela nativamente (`AbortController` aparte). Observable (`HttpClient`): un *stream* de valores posibles, no hace nada hasta suscribirse, se cancela con `unsubscribe()`. Angular eligió Observables porque las UIs son flujos de eventos continuos — modelo más potente, curva más empinada.

Prompt listo para tu AI

```
Quiero llamar una API REST desde React. Muéstrame la evolución completa:
(1) fetch suelto con useState+useEffect (loading, error, data), (2) axios
con un interceptor para Authorization, (3) TanStack Query con useQuery y
useMutation. Para cada versión explica qué problema de la anterior
resuelve. El endpoint es GET/POST /api/tasks que devuelve {id,title,done}.
```

Otras cimentaciones: ¿siempre una librería para llamar APIs?

Esta decisión es material: llamar la API debe pasar; con qué depende de cuánto ciclo de vida necesitas.

Alternativa	Qué te cuesta	Cuándo elegirla
`fetch` + useState a mano	Tú gestionas loading/error/caché	Una llamada puntual, demo, aprender
axios	Una dependencia, su API	Auth con interceptors, apps medianas
React Query / SWR	Conceptos de caché/invalidación	React serio — casi siempre la respuesta
HttpClient (Angular)	RxJS — la curva más alta	Angular: obligado, y es bueno
GraphQL client	Cambia el contrato a GraphQL	Si el backend ya es GraphQL

5.5. Errores comunes

- **`fetch` sin revisar `res.ok`**: un 404 llega como "éxito" y rompes al hacer `.json()` o al pintar.
- **useEffect con fetch sin cleanup**: doble llamada en StrictMode, race conditions si el componente se desmonta.
- **Guardar el token en el interceptor... leído una vez**: el token cambia al hacer login; léelo *dentro* del interceptor cada request.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o

no pasar.

- **Manejar loading/error a mano cuando ya usas React Query**: reinventar la rueda dentro de la librería que la resuelve.

5.6. Buenas prácticas

- **Un `api-client` centralizado**: todas las llamadas pasan por un archivo — cambiar base URL o auth se hace en un lugar.
- **Loading y error son parte del contrato UI**: diseña los tres estados (cargando / error / datos) — no es opcional en producción.

> Explicación de: producción
>
> **Producción** (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

- **Interceptors para auth**, no token pegado en cada llamada.
- **Type the response**: `fetch<Task[]>()` o validación — el JSON que llega no está garantizado (eso lo resuelve el backend con el contrato, Sección II).

5.7. Ejercicio

1. Escribe `getTasks()` con fetch que lance error si `!res.ok` — y pruébalo contra `http://localhost:8000/tasks` cuando corras la API del libro.
2. Haz el POST de crear tarea con fetch y verifica que aparece en la lista.

> Explicación de: array / lista / slice
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> Explicación de: localhost / 127.0.0.1
>
> **localhost** significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

5.8. Mini reto

Tu `useEffect` + fetch se ejecuta **dos veces** en desarrollo. ¿Por qué (React StrictMode) y cómo lo resuelves *bien* — con cleanup o con React Query?

Soluciones

Ejercicio 1.

```
async function getTasks() {
  const res = await fetch("http://localhost:8000/tasks");
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  return res.json();
}
```

Ejercicio 2.

```
await fetch("http://localhost:8000/tasks", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "nueva" }),
});
```

Mini reto. StrictMode monta→desmonta→remonta en dev para detectar efectos sin cleanup. Fix manual: `let cancelled = false` + cleanup que la pone `true`. Fix real: React Query maneja dedupe/cancelación por ti — otra razón por la que las apps serias no hacen fetch en useEffect a mano.

5.9. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: evento

Un **evento** es "algo que pasó" convertido en dato: el click del usuario, el `payment.succeeded` del webhook, el mensaje del socket. La programación moderna es reaccionar a eventos más que seguir un guion.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: cookie

Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: retry / backoff

Un **retry** es reintentar lo que falló — las redes fallan, es normal. **Backoff** = esperar más cada vez (1s, 2s, 4s…): evita martillar a un servicio que ya está sufriendo.

5.10. Lo que deberías saber hacer ahora

☐ Llamar GET y POST con fetch incluyendo `res.ok` y JSON
☐ Explicar qué agregan axios (interceptors) y React Query (caché/estados)
☐ Explicar la diferencia Promise vs Observable (Angular)
☐ Diseñar los tres estados de una llamada en la UI

6 F6. De dev server a build: qué produce tu frontend

npm run dev no es tu app — es la fábrica que la construye

> **En una frase**
>
> `npm run dev` levanta un servidor *de desarrollo* que compila tu código al vuelo; `npm run build` produce **archivos estáticos** (HTML+JS+CSS) que un servidor cualquiera puede servir. Entender esa diferencia es entender cómo el frontend llega a producción — y cómo se conecta con tu backend.

6.1. El problema

En desarrollo todo funciona en `localhost:5173` con Vite recargando solo. Luego alguien pregunta: "¿y en producción cómo corre?" — y la respuesta sorprende: **tu React no corre en el servidor**. Se *construye* a archivos estáticos y cualquier servidor (nginx, un CDN, tu propio backend) los sirve tal cual. El navegador del usuario hace todo el trabajo.

> **Explicación de: CDN**
>
> Una **CDN** (Content Delivery Network) es una red de copias: tu estático se cachea en servidores cercanos al usuario. El de Buenos Aires no viaja a Virginia por una imagen — la recibe del nodo local.

> **Explicación de: producción**
>
> **Producción** (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

> **Explicación de: localhost / 127.0.0.1**
>
> **localhost** significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

6.2. Conceptos nuevos

- U **Dev server**: compila al vuelo, hot reload, solo para desarrollar — nunca para usuarios.
- U **Build**: `npm run build` → `dist/` con archivos estáticos optimizados (minificados, con hash).

Explicación de: hash / bcrypt

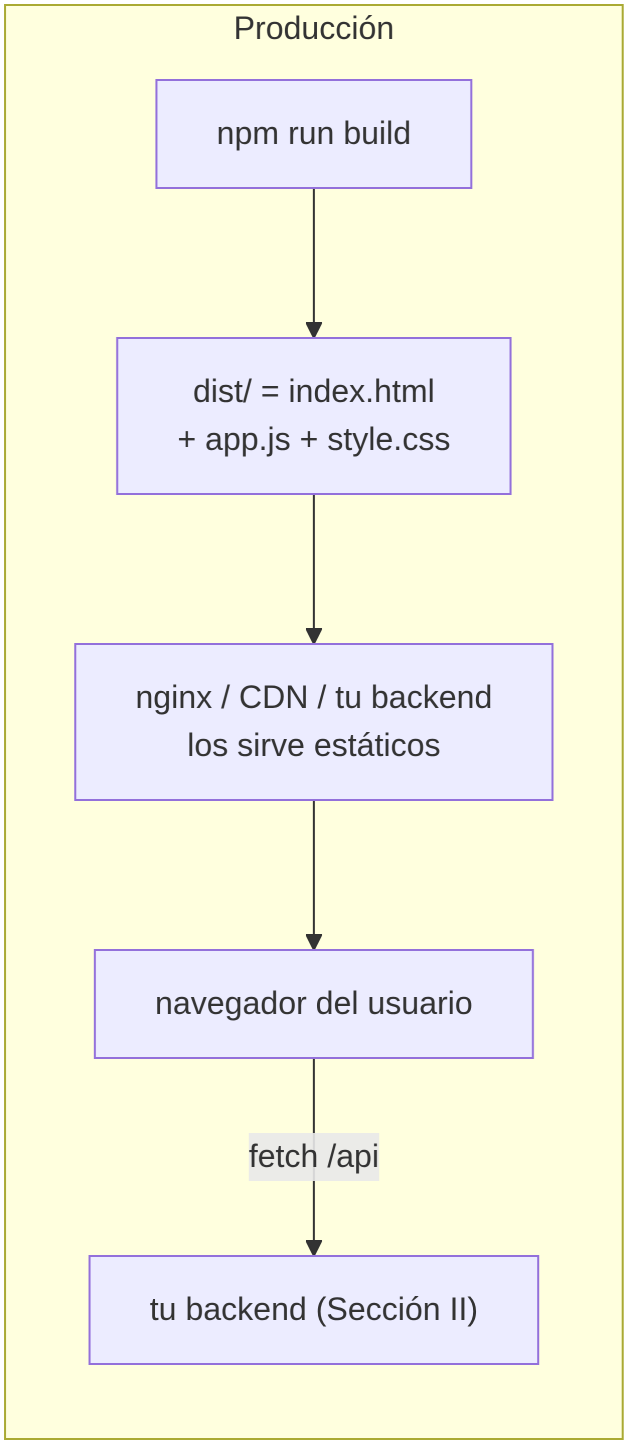
Un **hash** es una función de un solo sentido: `contraseña` →$2b10…‘. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

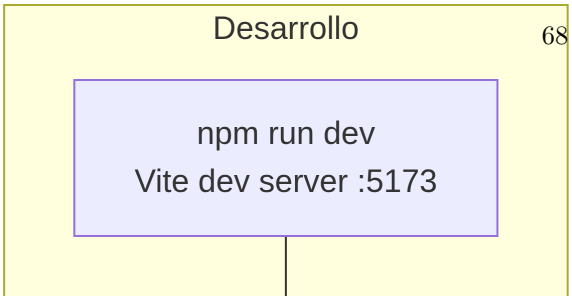
- U **SPA vs SSR**: la app corre en el navegador (SPA) vs el servidor prerenderiza HTML (SSR).
- U **API vs static hosting**: tu frontend son archivos; tu backend es un proceso — dos cosas distintas a desplegar.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que “se caiga el proceso” = el programa murió.

6.3. La explicación visual





La lección clave: en producción hay **dos servicios** — el que sirve los archivos estáticos (frontend) y el que responde `/api` (backend). Pueden ser el mismo proceso o dos distintos.

6.4. Implementación

```
npm run dev      # Vite dev server - solo desarrollo, hot reload
npm run build    # → dist/ : index.html + assets/*.js + *.css minificados
npm run preview  # sirve dist/ localmente para probar el build
```

Mira lo que genera `build`:

> Explicación de: build
>
> El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

```
dist/
   index.html              # el esqueleto que carga todo
   assets/
       index-a1b2c3.js     # TODO tu React, minificado, con hash
       index-d4e5f6.css    # tus estilos, minificados
```

El hash en el nombre (`a1b2c3`) es caché-busting: cambia cuando el código cambia, así el navegador sabe cuándo re-descargar.

> Explicación de: caché / Redis
>
> Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

> Prompt listo para tu AI
>
> ```
> Explícame la diferencia entre `npm run dev`, `npm run build` y
> `npm run preview` en un proyecto Vite + React. Quiero saber: qué genera
> el build físicamente, por qué el servidor de producción solo sirve
> archivos estáticos, y cómo se conecta el frontend desplegado con un
> backend en /api. Usa un ejemplo con nginx o sirviendo dist/ desde el
> propio backend.
> ```

Otras cimentaciones: ¿siempre SPA + build estático?

Esta decisión es material: entregar tu UI al navegador debe pasar; cómo llega ahí tiene variantes reales.

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
SPA estática (este capítulo)	Build → archivos servidos por cualquiera	SEO más difícil, primera carga vacía	Apps privadas, dashboards — el default
SSR (Next.js, Nuxt, Angular Universal)	El servidor prerenderiza HTML por request	Un proceso Node corriendo siempre	SEO, contenido público, e-commerce
SSG (static site generation)	HTML generado en build por página	Rebuild al cambiar contenido	Blogs, docs, marketing
Server components (React moderno)	Componentes que corren solo en servidor	Modelo mental nuevo, framework-specific	Apps grandes con Next.js

6.5. Errores comunes

- **Desplegar el dev server**: `npm run dev` en producción — está hecho para velocidad de desarrollo, no para usuarios.
- **Pensar que el frontend "corre" en el servidor**: solo se sirven archivos; el código corre en el navegador del usuario.
- **API hardcodeada a localhost**: `fetch("http://localhost:8000")` en el build de producción — la base URL debe venir del entorno.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

6.6. Buenas prácticas

- **`dist/` a un estático** (nginx, CDN, S3) — el frontend no necesita un proceso vivo, solo archivos servidos.
- **La API como variable de entorno**: `VITE_API_URL` — el build se adapta a cada entorno sin re-compilar lógica.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

- **Mismo dominio si puedes**: servir frontend y `/api` desde el mismo origen evita CORS (Sección II lo explica).

Explicación de: CORS

CORS es la política del navegador: por defecto, una página de `sitioA.com` no puede leer respuestas de `apiB.com`. El servidor declara qué orígenes acepta. Es protección del navegador, no del servidor (curl ni la ve).

6.7. Ejercicio

1. Corre `npm run build` en tu app Vite y mira `dist/` — encuentra el JS con hash y confirma que `index.html` lo referencia.
2. Sirve `dist/` con `npm run preview` y verifica que funciona sin el dev server.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

6.8. Mini reto

Si el frontend son solo archivos estáticos, ¿cómo "sabe" a qué backend llamar? Diseña la respuesta — pista: variable de entorno en build o mismo-origen con proxy.

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: reverse proxy

Un **reverse proxy** (nginx, ALB, Cloudflare) es el portero: recibe todo el tráfico en el 443, termina TLS y reparte a tu app interna. La app nunca mira a internet directamente — el proxy la protege.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Soluciones

Ejercicio 1. `dist/index.html` referencia `assets/index-<hash>.js` — un solo archivo con todo tu código minificado; el hash cambia al reconstruir.
Mini reto. Dos caminos: (a) `VITE_API_URL` inyectada en el build (`import.meta.env.VITE_API_URL`) apunta al backend por entorno; (b) mismo-origen — el servidor que sirve `dist/` también responde `/api/*` (nginx proxy o el propio backend sirviendo estáticos), y el frontend usa `fetch("/api/...")` sin host. El (b) es el más simple en producción y evita CORS por completo.

6.9. Vocabulario técnico del capítulo

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: bucket / object storage

Un **bucket** (S3, Azure Blob, GCS) es almacenamiento de archivos como servicio: subes un PDF/imagen, te devuelve una URL. No es un disco — no lo montas, lo consultas por HTTP. Barato, infinito, durable.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: dev server / hot reload

El **dev server** (Vite, `npm run dev`) sirve tu app mientras desarrollas y recarga el navegador al guardar — *hot reload*. No es el servidor de producción: es la mesa de trabajo.

6.10. Lo que deberías saber hacer ahora

- ☐ Explicar qué genera `npm run build` y por qué son archivos estáticos
- ☐ Decir la diferencia entre dev server, preview y producción
- ☐ Explicar SPA vs SSR y cuándo cada uno
- ☐ Diseñar cómo el frontend desplegado encuentra a su backend

7 F7. Mini-proyecto: la UI de Taskflow

Todo lo de la Sección I junto: componentes, estado y llamadas a tu API

> **En una frase**
>
> Vas a construir **el frontend de Taskflow completo**: una lista de tareas que se carga desde la API, un input para crear nuevas, y los estados loading/error — en React. Es el puente: cuando llegues a la Sección II ya tendrás un cliente real esperando tu backend.

7.1. El problema

La Sección I te dio las piezas por separado: runtime, lenguaje, DOM, frameworks, llamadas a APIs, build. Ahora las juntamos en algo real: una UI que consume `GET /tasks` y `POST /tasks` — los endpoints que el libro construye en la Parte de backend.

> **Explicación de: runtime**
>
> El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

> **Explicación de: DOM**
>
> El **DOM** es la página como árbol de objetos vivos: HTML es el papel, el DOM es lo que JavaScript puede tocar. `element.textContent = "hola"` cambia el DOM y el navegador repinta — el HTML original no se entera.

> **Explicación de: framework**
>
> Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

7.2. Cómo lo resuelve un equipo

Antes de escribir, el equipo define el *contrato* que la UI espera (el mismo que el backend promete):

```
GET  /api/tasks           → 200 [{ id, title, done }]
POST /api/tasks  {title}  → 201 { id, title, done }
```

Con el contrato claro, la UI se escribe sin backend todavía — contra un mock o contra la API cuando exista. Ese es el punto: **el contrato permite que frontend y backend se construyan en paralelo.**

Explicación de: concurrencia vs paralelismo

Concurrencia = manejar muchas cosas en progreso (atender 1000 requests intercalando). **Paralelismo** = ejecutar varias a la vez en varios núcleos. Un camarero con 10 mesas es concurrente; 10 camareros son paralelos.

Explicación de: mock / stub

Un **mock** es un doble de pruebas: finge ser el servicio externo (Stripe, SMTP) para que el test no cobre ni mande emails. Se mockea en la *frontera* (tu adaptador), nunca la BD — fingir la BD es mentir el test.

7.3. Implementación

La app completa en React — estado, loading, error, lista y formulario:

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: estado (state)

El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

```
import { useState, useEffect } from "react";

const API = "/api";   // mismo-origen en producción; o VITE_API_URL

export default function TasksPage() {
  const [tasks, setTasks] = useState([]);
  const [title, setTitle] = useState("");
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {                       // cargar al montar
    fetch(`${API}/tasks`)
      .then((r) => { if (!r.ok) throw new Error(`HTTP ${r.status}`); return r.json(); })
      .then(setTasks)
      .catch(setError)
      .finally(() => setLoading(false));
  }, []);

  async function addTask(e) {
    e.preventDefault();
    const res = await fetch(`${API}/tasks`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ title }),
    });
    const created = await res.json();
    setTasks((ts) => [...ts, created]);   // agregar al estado → re-render
    setTitle("");
  }

  if (loading) return <p>Cargando…</p>;
  if (error) return <p>Error: {error.message}</p>;
  return (
    <div>
      <form onSubmit={addTask}>
        <input value={title} onChange={(e) => setTitle(e.target.value)} />
        <button>Agregar</button>
      </form>
      <ul>{tasks.map((t) => <li key={t.id}>{t.title}</li>)}</ul>
    </div>
  );
}
```

Qué estás viendo: **todo lo de la sección junto** — `useState` (estado), `useEffect` (cargar al montar), `fetch` con `res.ok`, loading/error como parte de la UI, y `setTasks` que repinta solo.

Prompt listo para tu AI

```
Construye un componente React "TasksPage" que consuma mi API REST:
- GET /api/tasks devuelve [{id, title, done}]
- POST /api/tasks con body {title} devuelve la tarea creada (201)
Requisitos: useState para tasks/title/loading/error, useEffect para
cargar al montar, formulario controlado para crear, estados de
loading/error visibles en la UI, y base URL desde variable de entorno.
Usa fetch nativo - sin librerías extra - y explica cada hook que uses.
```

7.4. ¿Sin backend todavía?

Tres formas de desarrollar la UI sin que la API exista:

Alternativa	Cómo	Cuándo
Mock en el frontend	`const tasks = [...]` local	UI primero, API después
Mock server (json-server, MSW)	Un archivo JSON que simula la API	Desarrollo real en paralelo
La API real	La construyes en la Sección II	Cuando llegues ahí — encaja directo

MSW (Mock Service Worker) es la favorita profesional: intercepta `fetch` y responde mocks — tu código de producción no cambia nada.

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

7.5. Errores comunes

- **fetch en el body del componente** (sin useEffect): se dispara en cada render — bucle infinito.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

- **`setTasks([...tasks, created])` vs `tasks.push()`**: mutar no repinta en React — siempre crear array nuevo.

- **Olvidar `e.preventDefault()`** en el form: la página recarga entera.

7.6. Ejercicio

1. Levanta esta página contra un array mock local primero — que funcione sin API.
2. Cuando tengas la API (Sección II), cambia solo la base URL y verifica que todo sigue igual — eso es "contrato cumplido".

> Explicación de: función
>
> Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

7.7. Mini reto

Agrega el checkbox `done`: click marca la tarea hecha. ¿Qué endpoint necesitas? (Ya lo diseñaste en el Cap. 3: `PATCH /tasks/{id}`.)

> Soluciones
>
> **Ejercicio 1.** `useState([{id:1,title:"a",done:false}])` inicial + el `useEffect` comentado — la UI funciona sin red; al conectar la API, solo cambia `API`.
> **Mini reto.** `PATCH /api/tasks/{id}` con `{done: true}` → luego `setTasks(ts => ts.map(t => t.id===id ? {...t, done:true} : t))`. El patrón: mutación al servidor + actualización optimista del estado.

7.8. Vocabulario técnico del capítulo

> Explicación de: variable
>
> Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

> Explicación de: null / None / nil
>
> **null** (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: módulo / import

Un **módulo** es un archivo de código que exporta cosas para que otros archivos las importen. Es la unidad de organización: en vez de un archivo gigante, el código vive en módulos con responsabilidad propia.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

7.9. Lo que deberías saber hacer ahora

- ☐ Construir una UI completa (estado, loading, error, form) que consuma una API
- ☐ Explicar por qué el contrato permite desarrollar FE y BE en paralelo
- ☐ Desarrollar contra un mock sin cambiar el código de producción
- ☐ Conectar la UI a la API real cambiando solo la base URL

Parte II

Sección II · Backend — Fundamentos

8 0. Fundamentos: lo que ya casi sabes

El piso mínimo para leer todo este libro — y salir sabiendo backend

> **En una frase**
>
> Repaso exprés de lo que ya casi sabes: terminal, HTTP, JSON, clases y cómo se define un endpoint — traducido a TypeScript, Python y Go. Lee las secciones que necesites; el checklist del final te dice si estás listo.

Este capítulo es el único "teórico" del libro. Su trabajo es simple: que nada de lo que viene después te pille desprevenido. Si vienes de JavaScript, **ya sabes el 80 %** — aquí solo traducimos lo que sabes y rellenamos los huecos que un frontend nunca necesitó.

> **No memorices esto.** Es material de consulta: puedes saltarlo, seguir el libro, y volver aquí cuando algo no te suene. El libro está pensado para leerse con un AI CLI al lado — pregúntale "explícame X de este capítulo" sin culpa.

Al final hay un checklist: si puedes marcarlo todo, puedes leer el libro entero sin frenarte.

8.1. 0.1 La terminal: tu otro navegador

En backend, la terminal es donde vive tu aplicación — no hay DevTools que te salve. Cinco cosas que usarás todos los días:

> **Explicación de: terminal / consola**
>
> La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

```
cd mi-proyecto          # entrar a una carpeta
ls                      # ver qué hay (en Windows: dir)
curl http://localhost:8000/health      # llamar una API sin frontend
curl -X POST http://localhost:8000/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "learn backend"}'      # POST con JSON body
export DEBUG=true       # variable de entorno (Windows PowerShell: $env:DEBUG="true")
```

`curl` es tu `fetch` de terminal: verás en el libro que casi todo endpoint se prueba primero con curl — si funciona ahí, funcionará en tu React.

8.2. 0.2 HTTP en quince minutos

Ya usas HTTP desde `fetch`, pero ahora **tú eres quien responde**. Un request tiene cuatro piezas:

```
POST /tasks?urgent=true HTTP/1.1     ← método + ruta + query
Host: api.ejemplo.com
Content-Type: application/json         headers (metadatos)
Authorization: Bearer eyJhbGc…

{"title": "deploy", "priority": 2}   ← body (opcional)
```

Y una response, tres:

```
HTTP/1.1 201 Created                 ← status code
Content-Type: application/json       ← headers

{"id": 7, "title": "deploy", "done": false}   ← body
```

Los métodos son semántica, no decoración:

Método	Promesa	Ejemplo
`GET`	Solo leo, nunca modifico	`GET /tasks`
`POST`	Creo o ejecuto algo	`POST /tasks`
`PUT`	Reemplazo el recurso entero	`PUT /tasks/7`
`PATCH`	Modifico parte	`PATCH /tasks/7`
`DELETE`	Elimino	`DELETE /tasks/7`

Los status codes son familias: `2xx` salió bien, `4xx` culpa del cliente (mal body, sin permiso, no existe), `5xx` culpa del servidor (tu bug). Un backend profesional elige el código *a propósito* — es parte del contrato, no un detalle.

> Explicación de: autorización / permisos
>
> La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

8.3. 0.3 JSON: el idioma común

JSON es un formato de texto para datos — hijo de JavaScript, adoptado por todos los lenguajes:

```
{
  "id": 7,
  "title": "deploy",
  "done": false,
  "tags": ["ops", "release"],
  "owner": { "id": 3, "name": "Ana" },
  "due": null
}
```

Tipos: `string`, `number`, `boolean`, `null`, array `[]`, objeto `{}`. No hay funciones, ni `undefined`, ni fechas — una fecha viaja como string ISO `"2026-10-01"` y cada lado la interpreta.

> Explicación de: función
>
> Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

> Explicación de: array / lista / slice
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> Explicación de: objeto / dict / struct
>
> Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

> Explicación de: string
>
> Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

> Explicación de: booleano
>
> Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

En backend, *serializar* es convertir tu objeto interno a JSON para la respuesta; *parsear* es convertir el JSON del request a tu objeto. Harás ambas cosas cientos de veces en este libro.

8.4. 0.4 Recordatorio: así llamas a una API desde el frontend

> **Si esto no te suena del todo:** la **Sección I · Aprende Frontend** lo enseña completo (fetch, axios, React Query, HttpClient). Lee el resumen y vuelve aquí — o salta allá primero si lo prefieres.

El punto que importa para backend: del otro lado de cada una de estas llamadas hay un servidor — y a partir del Cap. 1 ese servidor eres tú.

```
// fetch - la base de todo
const res = await fetch("/api/tasks");
if (!res.ok) throw new Error(`HTTP ${res.status}`);  // 404 no lanza solo
const tasks = await res.json();

// axios - JSON auto, errores en 4xx, interceptors para auth
const { data } = await axios.get("/api/tasks");

// React Query - loading/error/caché resueltos
const { data, isLoading } = useQuery({
  queryKey: ["tasks"],
  queryFn: () => fetch("/api/tasks").then((r) => r.json()),
});

// Angular HttpClient - Observable (frío): sin subscribe no hay request
this.http.get<Task[]>("/api/tasks").subscribe((t) => (this.tasks = t));
```

El puente: tu `lib/api-client.ts` del frontend era **el consumidor del contrato** — ahora tú produces ese contrato. Y tu `services/` del frontend (hablador de APIs) es el espejo del `services/` del backend (lógica de negocio) del Cap. 7: mismo nombre, distinto rol.

8.5. 0.5 Clases y objetos: por qué importan aquí

Si usaste clases en JS/TS, esto es repaso con traducción. Si no, la idea es simple: una **clase** es un molde que agrupa *datos* (atributos) y

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

comportamiento (métodos); un **objeto** es una instancia creada desde ese molde.

¿Por qué le importa esto a un backend? Porque tres piezas centrales se expresan como clases:

- **Schemas/modelos**: `class TaskCreate` define la forma válida del body
- **Modelos de datos**: `class Task` mapea a una fila de tabla
- **Servicios/handlers**: agrupan la lógica por dominio

8.5.1. TypeScript

```typescript
// interface = solo la forma (desaparece al compilar)
interface Task {
  id: number;
  title: string;
  done: boolean;
}

// class = forma + comportamiento (existe en runtime)
class TaskService {
  private tasks: Task[] = [];            // atributo privado

  create(title: string): Task {          // método
    const task = { id: this.tasks.length + 1, title, done: false };
    this.tasks.push(task);               // this = esta instancia
    return task;
  }
}

const svc = new TaskService();           // instancia
svc.create("deploy");
```

8.5.2. Python

```python
# Python no tiene interfaces - las clases hacen ambos papeles
class Task:
    def __init__(self, id: int, title: str):
        self.id = id          # self = esta instancia (explícito, se escribe)
        self.title = title
        self.done = False

class TaskService:
```

```python
    def __init__(self):
        self.tasks: list[Task] = []

    def create(self, title: str) -> Task:   # self es el primer parámetro
        task = Task(len(self.tasks) + 1, title)
        self.tasks.append(task)
        return task

svc = TaskService()             # instancia - sin "new"
svc.create("deploy")
```

8.5.3. Go

```go
// Go no tiene clases: tiene structs (datos) + métodos (funciones pegadas)
type Task struct {
    ID    int
    Title string
    Done  bool
}

type TaskService struct {
    tasks []Task              // campo privado (minúscula = privado en Go)
}

// método: la "instancia" es el receiver (s *TaskService)
func (s *TaskService) Create(title string) Task {
    t := Task{ID: len(s.tasks) + 1, Title: title}
    s.tasks = append(s.tasks, t)
    return t
}

svc := &TaskService{}         // instancia
svc.Create("deploy")
```

Las diferencias que importan de verdad:

- **`self`/`this`/receiver**: la referencia a "esta instancia". En JS se escribe `this` (y a veces desaparece mágicamente); en Python se declara `self` como primer parámetro de cada método; en Go es el *receiver* `(s *TaskService)` — el método es una función que recibe el struct.

> Explicación de: parámetro / argumento
>
> El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

- **Go no tiene clases ni herencia**: struct = datos, método = función con receiver. Suena limitado y a propósito lo es — la composición explícita reemplaza la herencia (lo verás en servicios).
- **En Python todo es objeto** y `self` es obligatorio: `def create(self, title)` — olvidarlo es el error clásico del que viene de JS.

Otras cimentaciones: ¿siempre clases para modelar datos?

Esta decisión es material: necesitas *alguna* forma de darle nombre y estructura a tus datos — pero "clase" es solo una de las cimentaciones.

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Clases completas	Datos + comportamiento juntos	Más ceremonial; tentación de meter lógica de más	Servicios con estado (`TaskService`), patrones OO
Solo forma (interface/DTO)	`interface` TS, `TypedDict`, struct	Sin métodos — solo describe la forma	Schemas de entrada/salida (lo más común en APIs)
`dataclass` / `BaseModel`	Clase reducida a "datos con validación"	Dependencia (Pydantic) o menos flexibilidad	Python moderno — es lo que FastAPI usa
Structs + funciones	Datos y lógica separados (Go)	No hay herencia — composición explícita	Go: es *la* forma, no una opción
Funciones + objetos literales	Sin tipos nombrados	Nada te garantiza la forma	Scripts chicos — en backend se queda corto rápido

La regla práctica del libro: **los datos que cruzan la frontera (request, response, DB) llevan forma declarada; los servicios que orquestan llevan comportamiento.**

8.6. 0.6 Funciones, parámetros y módulos

La traducción rápida:

8.6.1. TypeScript

```
// función tipada
function createTask(title: string, priority: number = 3): Task {
  return { id: 1, title, done: false };
}

import { createTask } from "./services/tasks.js";  // ES module
```

8.6.2. Python

```python
# def + indentación - los bloques son por sangría, no por llaves
def create_task(title: str, priority: int = 3) -> Task:
    return Task(id=1, title=title)

from app.services.tasks import create_task   # módulo = archivo .py
```

8.6.3. Go

```go
// func + llaves; el tipo va DESPUÉS del nombre
func createTask(title string, priority int) Task {
    return Task{ID: 1, Title: title}
}

import "taskflow/internal/service"   // package = directorio
```

Dos detalles Python que confunden al principio:

- **snake_case**: `create_task`, no `createTask`. Convención del lenguaje.
- **La indentación ES la sintaxis**: `def`/`if`/`for` terminan en `:` y el cuerpo va indentado — no hay `{}` ni `end`.

8.7. 0.7 Estructuras de datos mínimas

Concepto	TypeScript	Python	Go
Mapa clave→valor	`object` / `Map`	`dict`	`map[K]V`
Lista ordenada	`Array`	`list`	`[]T` (slice)
Conjunto sin duplicados	`Set`	`set`	`map[K]struct{}`
Tupla inmutable	`readonly [A,B]`	`tuple`	— (usa struct)

```python
# Python - las que más verás
user = {"id": 3, "name": "Ana"}        # dict   objeto JS
user["name"]                            # acceso por clave
user.get("email", "sin email")          # con default - evita KeyError

tasks = [t1, t2, t3]                    # list   array
active = [t for t in tasks if not t.done]  # comprehension - el .filter() de Python
titles = [t.title for t in tasks]          # el .map() equivalente
```

Las **comprehensions** de Python (`[x for x in y if cond]`) merecen un momento: son `.filter()` y `.map()` fusionados en sintaxis — las verás en servicios y queries por todos lados.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

8.8. 0.8 Errores en una frase (el Cap. 6 los abre de verdad)

- **TS/Python**: `throw`/`raise` una excepción; alguien arriba hace `catch`/`except`. Si nadie la atrapa → 500.
- **Go**: la función *devuelve* el error como valor: `t, err := getTask(id)` — no salta nada, tú decides.

8.9. 0.9 Qué es un endpoint (la anatomía mínima)

Un endpoint = **(verbo + ruta) → handler**. El handler es solo una función que recibe el request y produce la response:

8.9.1. TypeScript

```
// verbo.get(ruta, handler)
app.get("/tasks/:id", (req, res) => {
  const task = findTask(Number(req.params.id));
  if (!task) return res.status(404).json({ error: "not found" });
  res.json(task);                    // 200 + body
});
```

8.9.2. Python

```
# decorator: "cuando llegue GET /tasks/{id}, ejecuta esta función"
@app.get("/tasks/{task_id}")
def get_task(task_id: int):
    task = find_task(task_id)
    if not task:
        raise HTTPException(404, "not found")
    return task                        # se serializa a JSON solo
```

8.9.3. Go

```go
// patrón "método espacio ruta" → handler(w, r)
mux.HandleFunc("GET /tasks/{id}", func(w http.ResponseWriter, r *http.Request) {
    id, _ := strconv.Atoi(r.PathValue("id"))
    if t, ok := findTask(id); ok {
        json.NewEncoder(w).Encode(t)
    } else {
        http.Error(w, "not found", 404)
    }
})
```

Leído en voz alta, los tres dicen lo mismo: *"cuando llegue un GET a /tasks/{id}, corre esta función que busca la tarea y la devuelve como JSON — o responde 404"*. El resto del libro solo agrega capas encima de esa idea.

8.10. Ejercicio

1. Escribe en tu lenguaje elegido una clase/struct `User` con `id`, `email`, `is_active` y un método `deactivate()` que la apague.
2. Haz `curl -i http://localhost:8000/tasks` contra el servidor del Cap. 3 y lee los headers de la response — encuentra el status y el `Content-Type`.
3. Traduce a tu lenguaje: `[u.name for u in users if u.is_active]` (la comprehension de arriba) a JS: ¿qué combinación de métodos es?

> Explicación de: localhost / 127.0.0.1
>
> **localhost** significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

8.11. Mini reto

Sin código: escribe con tus palabras la diferencia entre *la clase* `Task` y *un objeto* `task`, y da un ejemplo de cuándo en backend usas cada una. Si puedes explicar por qué un schema de validación (Cap. 4) se define como clase pero un endpoint es una función, entendiste la sección 0.5.

> Explicación de: esquema (schema)
>
> El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Soluciones

Ejercicio 1.

8.11.1. TypeScript

```typescript
class User {
  constructor(
    public id: number,
    public email: string,
    public is_active = true,
  ) {}
  deactivate() { this.is_active = false; }
}
```

8.11.2. Python

```python
class User:
    def __init__(self, id: int, email: str):
        self.id, self.email, self.is_active = id, email, True
    def deactivate(self):
        self.is_active = False
```

8.11.3. Go

```go
type User struct {
    ID       int
    Email    string
    IsActive bool
}
func (u *User) Deactivate() { u.IsActive = false }
```

Ejercicio 2. Deberías ver `HTTP/1.1 200 OK` y `Content-Type: application/json` en los headers impresos por `-i`.
Ejercicio 3. `users.filter(u => u.is_active).map(u => u.name)` — filter primero, map después. La comprehension hace ambos en una expresión.
Mini reto. La clase es el molde (la *definición* de qué es una tarea); el objeto es una instancia concreta (la tarea #7 que creó Ana). Un schema es clase porque define una *forma* que se instancia por cada request que entra; un endpoint es función porque describe una *acción* (recibir un request, devolver una response), no un estado que mantener.

8.12. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

8.13. Lo que deberías saber hacer ahora

- ☐ Reconocer todas las formas de llamar una API desde el cliente: fetch, axios, React Query/SWR, HttpClient de Angular
- ☐ Explicar la diferencia Promise vs Observable (hot vs cold)
- ☐ Probar un endpoint con `curl` sin abrir el navegador

- ☐ Leer un request HTTP completo: método, ruta, headers, body
- ☐ Definir una clase/struct con un método en tu lenguaje
- ☐ Traducir mentalmente: dict objeto, list array, `self` `this`
- ☐ Explicar qué es un endpoint: (verbo + ruta) → handler
- ☐ Decir qué es serializar y por qué el backend vive serializando

Si marcaste todo: **puedes leer el libro entero**. Lo que falte, aparecerá justo cuando el problema lo necesite.

9 0.5. La terminal, en serio

Donde vive el backend, las cosas que nunca debes hacer, y tus nuevos compañeros de AI

> **En una frase**
>
> La terminal es el escritorio del backend: este capítulo te da el tour esencial (navegar, pipes, procesos), las **12 cosas que jamás debes hacer** y un mapa de los agentes de AI que ahora viven ahí.

9.1. El problema real

Tu aplicación funciona — en tu máquina, con VS Code abierto y `npm run dev` corriendo en una terminal integrada que cerras cuando terminas. Entonces llega el día: "hay que desplegarlo". Y de repente todo es terminal: el servidor no tiene pantalla, Docker no tiene botones, los logs son un stream infinito, y la única forma de entrar a producción es `ssh`.

> **Explicación de: log / logging estructurado**
>
> Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

> **Explicación de: terminal / consola**
>
> La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

En frontend la terminal era un accesorio. En backend **es el escritorio donde trabajas**. Este capítulo corto te pone cómodo ahí — y te dice qué cosas jamás hacer, porque la terminal no tiene Ctrl+Z.

9.2. Cómo se resuelve en un equipo

- **Nadie memoriza comandos raros.** Los equipos escriben *runbooks*: archivos con los comandos exactos para desplegar, ver logs, reiniciar.
- **Los READMEs arrancan con comandos**, no con prosa: "clona, corre esto, abre este puerto".

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

- **Desde hace poco, la terminal tiene nuevos habitantes:** los agentes de AI (Devin CLI, Claude Code, Copilot, Gemini CLI) viven ahí — porque la terminal es la interfaz universal donde se leen archivos y se ejecutan cosas.

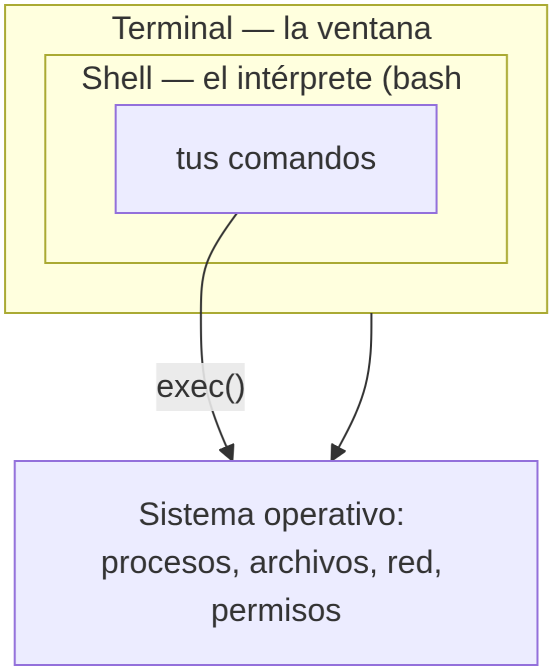
Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

9.3. Conceptos nuevos

- La diferencia entre **terminal, shell y consola** (son tres cosas).
- Qué es el **PATH** y por qué `command not found` existe.
- **Exit codes**: el contrato silencioso de todo comando.
- **Pipes y redirección**: componer programas como compones funciones.
- Las **reglas de oro** de lo que nunca se hace.
- Qué son los **agentes CLI de AI** y cómo trabajar con ellos sin que te rompan nada.

9.4. Explicación visual: tres capas que la gente confunde



- **Terminal**: solo la ventana que dibuja texto (Windows Terminal, iTerm, la pestaña de VS Code). No entiende nada.
- **Shell**: el programa que interpreta lo que escribes — `bash`, `zsh`, `fish`, `pwsh`. *El mismo comando puede significar cosas distintas en cada shell.*
- **SO**: quien realmente ejecuta: crea procesos, abre puertos, borra archivos.

Por eso `export PORT=8000` funciona en bash pero en PowerShell es `$env:PORT = "8000"`. Misma intención, distinto intérprete.

Otras cimentaciones: ¿qué shell y qué terminal elegir?

Esta decisión es material: necesitas *un* shell — pero cuál es como elegir el tipo de cimentación: todas sostienen, cambia el terreno donde mejor funcionan.

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
bash	El estándar universal	Sintaxis vieja, quirks históricos	Scripts portables, servidores Linux — default seguro
zsh	bash mejorado (default en macOS)	Casi nada — compatible con bash	Uso diario en Mac/Linux
PowerShell / pwsh	Shell de objetos (no texto)	Sintaxis distinta; menos portable a Linux	Windows nativo, automatización .NET

fish	Autocompletado y UX moderna	NO compatible con sintaxis bash	Terminal interactiva, no scripts
Nushell	Datos estructurados en pipes	Joven; ecosistema pequeño	Curiosos, pipelines de datos
Git Bash (Windows)	bash emulado en Windows	No es Linux real — casos borde	Windows sin WSL; lo que usa este libro

Y la terminal (la ventana): Windows Terminal, iTerm2, la integrada de VS Code — es decoración, elige la que te guste. **El shell es lo que importa para los comandos.**

9.5. El tour esencial

Nada de memorizar — esto es lo que usarás cada día, agrupado por intención:

9.5.1. Orientarse y archivos

```
pwd                    # dónde estoy
ls -la                 # qué hay aquí (con ocultos)
cd .. && cd -          # subir / volver al directorio anterior
mkdir -p src/services  # crear carpetas anidadas
cp .env .env.example   # copiar
mv old.ts new.ts       # mover/renombrar
cat app.log            # ver un archivo
tail -f app.log        # ver logs EN VIVO (el más usado en backend)
```

9.5.2. Componer con pipes (la idea más poderosa)

```
cat app.log | grep ERROR | wc -l    # cuántos errores hay
history | grep docker               # qué comandos docker usé
curl -s localhost:8000/health       # probar tu API sin navegador
echo "hello" > file.txt             # escribir (sobreescribe)
echo "more" >> file.txt             # añadir (append)
```

`|` conecta la salida de un programa con la entrada del siguiente — es `.map().filter()` pero entre procesos completos.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

9.5.3. Procesos, puertos y exit codes

```
Ctrl+C                  # SIGTERM amable: "termina por favor"
kill -9 1234            # SIGKILL: mata sin dejar limpiar (último recurso)
netstat -ano | grep 8000     # Windows: quién ocupa el puerto
lsof -i :8000                # Unix: lo mismo
echo $? / echo $LASTEXITCODE # exit code del comando anterior (0 = ok)
```

9.5.4. Variables de entorno

```
export PORT=8000        # bash/zsh
$env:PORT = "8000"      # PowerShell
env | grep PORT         # ver qué existe
```

Las variables de entorno son cómo la config llega a tu app sin hardcodearla — el Cap. 24 vive de esto. Y tu repo ya lo usa: el `.env` con tu API key.

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

9.6. Las cosas que NUNCA debes hacer

Reglas de oro — la terminal no tiene Ctrl+Z

1. **`rm -rf` sin mirar el path dos veces.** Jamás con variables que puedan estar vacías: `rm -rf "$DIR/"` con `DIR` vacío borra desde la raíz. Regla: `echo` el comando primero si hay variables.
2. **`curl ... | bash` sin leer el script.** Estás ejecutando código arbitrario de internet, a veces como root. Descarga, lee, *después* ejecuta. Los instaladores serios te dejan revisar.
3. **Commitear secretos.** `.env`, keys, tokens, PEMs. Borrar el archivo después NO lo quita del historial de git — si se fue, hay que **rotar la credencial** (esto ya te pasó con una API key en este proyecto).
4. **`sudo` por costumbre.** Si necesitas sudo para todo, algo está mal configurado (npm global, permisos de carpeta). sudo es el bisturí, no el martillo.
5. **`chmod 777` para "arreglar" permisos.** "Funcionó" = le diste todo a todos. Entiende qué permiso falta (`644` archivos, `755` ejecutables).
6. **`git push --force` a ramas compartidas** (`main`, `develop`). Reescribes la historia de otros. `--force-with-lease` en TU rama es el límite aceptable.
7. **Pegar comandos que no entiendes** — especialmente de foros o con `sudo`. Si no sabes qué hace, pregúntale a un AI CLI primero ("explícame este comando") — para eso están.
8. **`kill -9` como primera opción.** SIGKILL no deja al proceso cerrar conexiones ni escribir el último log. Primero Ctrl+C o `kill` (SIGTERM); `-9` solo si ignora lo demás.
9. **Poner secretos en la línea de comandos:** `mysql -pSuperSecret` queda en `history` y es visible en `ps`. Usa `.env`, flags interactivos o gestores de secretos.
10. **Rutas con espacios sin comillas:** `cd C:\Repositorios\Grupo umbral` falla (dos argumentos); `cd "C:\Repositorios\Grupo umbral"` funciona. Ironía: este repo lo tiene.
11. **"Solo voy a probar rápido" en producción.** No hay sandbox en prod: `DROP DATABASE` no pregunta "¿estás seguro?". Cualquier comando destructivo se ensaya en staging primero.

12. **`docker run --privileged` o montar `/` por curiosidad.** Le diste al contenedor tu máquina entera.

9.7. La nueva realidad: los CLI de AI

La terminal dejó de ser solo para comandos: desde hace unos años también es donde viven los **agentes de AI** — programas que leen tu repo, proponen cambios, editan archivos y ejecutan comandos por ti.

CLI	Comando	Qué es
Devin CLI	`devin`	Agente autónomo (el que está escribiendo este libro contigo)
Claude Code	`claude`	Agente de Anthropic en tu repo
Gemini CLI	`gemini`	Agente de Google, free tier generoso
Copilot CLI	`gh copilot` / `copilot`	Sugerencias y agente de GitHub
Aider	`aider`	Pair-programming open source por terminal

¿Por qué viven en la terminal y no en una app con botones? Porque la terminal es **la interfaz universal**: todo lo que un dev puede hacer — leer archivos, correr tests, git, ssh — es un comando. Un agente que sabe usar la terminal sabe usar todo lo que tú usas.

9.7.1. Las reglas de oro aplican al agente también

Un agente es un junior muy rápido que a veces se equivoca con total confianza:

- **Tú apruebas lo que ejecuta.** Si propone `rm -rf` o un `--force`, las reglas de arriba aplican igual — la responsabilidad es tuya.
- **Revisa el diff antes de commitear** lo que escribió, como harías con el PR de un compañero.
- **Nunca pegues secretos en el prompt.** `.env` y gestores de secretos, igual que con cualquier herramienta.
- **Dale contexto, no órdenes vagas:** "agrega validación al endpoint POST /tasks de `src/routes/tasks.ts`" funciona; "arregla el backend" no.

Prompt listo para tu AI

```
Explícame este comando antes de que lo ejecute: [pegar comando].
Quiero: qué hace cada flag y cada pipe, qué archivos/procesos toca, si
es destructivo o reversible, y la forma segura de lograr lo mismo si es
peligroso. No lo ejecutes - solo explícalo.
```

9.8. Errores comunes

- **`command not found`**: el programa no está en el PATH. Revisa si está instalado y si tu terminal es vieja (el PATH se carga al abrirla — ya nos pasó con `quarto`).

> Explicación de: PATH
>
> **PATH** es la lista de carpetas donde la terminal busca programas cuando escribes un comando. Si `node` no está en el PATH, la terminal responde "command not found" aunque esté instalado en algún lado.

- **"Cerré la terminal y sigue corriendo"** (o al revés): procesos hijos, `nohup`, servicios — Ctrl+C mata el foreground, no la familia.
- **Mezclar sintaxis de shells**: `export` no existe en PowerShell, `$env:` no existe en bash. Mira qué shell tienes antes de pegar comandos de un tutorial.

9.9. Buenas prácticas

- **El README empieza con los 3 comandos** para correr el proyecto. Si necesitas más, ponlos en `scripts/` o un `Makefile`.
- **`history` + `Ctrl+R`**: tu memoria está a una búsqueda de distancia.
- **Antes de un comando destructivo, la versión inofensiva:** `ls` donde iría el `rm`, `--dry-run` si existe, `echo` donde hay variables.
- **Scripts para lo que repites:** si lo escribiste tres veces, es un archivo `.sh`/`.ps1` commiteado.

> EXTRA · Control de versiones y dónde viven los repos
>
> EXTRA Los roadmaps piden dos cosas: *comandos básicos de Git* (clone, add, commit, push, pull, branch, merge) y *servicios de alojamiento de repositorios*:
>
> - **GitHub** — la comunidad más grande; donde vive el open source y tu portafolio
> - **GitLab** — ciclo completo en una plataforma (repos + CI/CD + issues)
> - **Bitbucket** — el clásico del ecosistema Atlassian
> - **AWS CodeCommit** — repos Git privados administrados por Amazon
>
> El cometario que vale: tu perfil de GitHub *es* tu portafolio — los equipos que contratan lo miran antes que el CV.

9.10. Ejercicio

Sin salir de la terminal: (1) navega a un directorio temporal, (2) crea `playground/logs/`, (3) escribe tres líneas en `app.log` con `>>`, (4) cuenta las líneas con un pipe, (5) levanta `python -m http.server 8000` (o `npx serve`), (6) pruébalo con `curl` desde **otra** terminal, (7) mátalo con Ctrl+C.

9.11. Mini reto

Un compañero te pega esto y te dice "córralo":

```
cd /tmp && curl -sL https://example.com/setup.sh | bash && rm -rf $OLD_BUILD
```

Antes de tocar nada: enumera las **tres** reglas de oro que viola.

9.12. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: parámetro / argumento

El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: Git

Git es el historial de tu proyecto: cada *commit* es una foto del código con mensaje. Permite volver atrás, trabajar en ramas paralelas y fusionar el trabajo de varias personas sin pisarse.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

9.13. Lo que deberías saber hacer ahora

- ☐ Explicar la diferencia terminal/shell/SO con tus palabras
- ☐ Componer comandos con `|` y `grep` sin buscar en Google
- ☐ Encontrar qué proceso ocupa un puerto y terminarlo *bien*
- ☐ Recitar de memoria por qué `curl | bash` y `rm -rf $VAR` son peligrosos
- ☐ Pedirle a un AI CLI que te explique un comando antes de correrlo

Soluciones

Ejercicio (bash):

```
cd /tmp && mkdir -p playground/logs && cd playground
echo "INFO start" >> logs/app.log
echo "ERROR oops" >> logs/app.log
echo "INFO done" >> logs/app.log
cat logs/app.log | wc -l          # 3
python -m http.server 8000 &      # en otra terminal: curl localhost:8000
# Ctrl+C para detener
```

Mini reto — las tres reglas violadas:

1. `curl | bash` ejecuta código no revisado de internet (regla 2).
2. `rm -rf $OLD_BUILD` usa una variable que puede estar **vacía** → `rm -rf /` (regla 1).
3. Y la meta-regla 7: correr algo que no entendiste. El flujo correcto: `curl -sL .../setup.sh -o setup.sh`, leerlo, *entonces* decidir.

Pregunta de reflexión: ¿por qué crees que los agentes de AI viven en la terminal y no en una app gráfica? Si tu respuesta es "porque la terminal puede hacer todo lo que un dev hace", entendiste el capítulo.

10 1. Dejas de consumir APIs. Ahora las construyes.

Todo lo que fetch oculta, alguien tuvo que construirlo

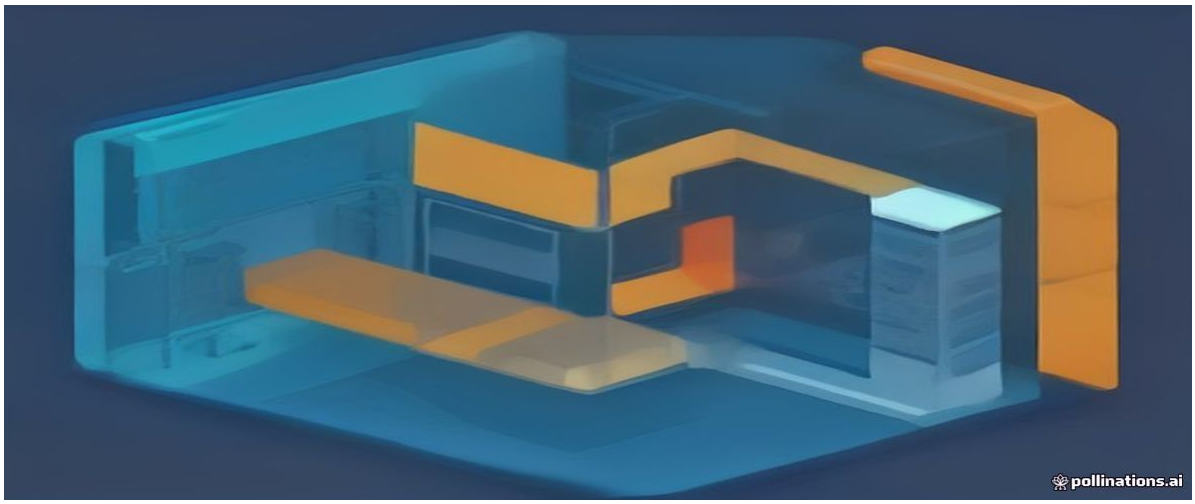


Figura 10.1: Parte 0 — El cambio de mentalidad

> **En una frase**
>
> Vas a aprender **qué pasa del otro lado de un `fetch`**: cómo un programa escucha un puerto, decide qué función atiende cada request y responde JSON. Al final del capítulo habrás levantado tu primer servidor en tu lenguaje.

> **No memorices nada de este libro.** El objetivo no es recitar sintaxis — es *reconocer el patrón* cuando lo veas. Todo lo que necesites está a una búsqueda de distancia (la lupa del sidebar) o a una pregunta de tu AI CLI.

10.1. El problema

Llevas tiempo escribiendo código así:

```
const res = await fetch("/api/tasks", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "Deploy to prod" }),
});
const task = await res.json();
```

Sabes usar `fetch`. Pero… ¿qué hay *del otro lado*? ¿Qué programa escuchó ese request, dónde corre, cómo decidió responder `201` y no `200`, quién convirtió tu JSON en una fila de base de datos?

Ese "otro lado" es lo que vas a construir durante todo este libro. Y la buena noticia: **ya sabes la mitad**. El protocolo es el mismo HTTP que conoces; lo que cambia es el rol — pasas de cliente a servidor.

10.2. Cómo lo resuelve un equipo

Cuando un equipo backend recibe un requerimiento ("hay que crear tareas"), la conversación nunca empieza en el código. Empieza en el **contrato**:

- ¿Qué ruta y qué verbo? `POST /api/tasks`
- ¿Qué shape tiene el body? `{ title: string, priority?: number }`
- ¿Qué responde en éxito? `201` con la tarea creada
- ¿Qué responde si el body está mal? `400`/`422` con un error estructurado
- ¿Quién puede llamarlo? ¿Cualquiera o solo usuarios autenticados?

Ejemplo de contrato terminado — esto es lo que el equipo acuerda *antes* de codear:

```
POST /api/tasks
Body:    { "title": "Deploy to prod", "priority": 2 }
Éxito:   201 { "id": 7, "title": "Deploy to prod", "done": false }
Error:   422 { "error": { "code": "VALIDATION",
                          "fields": { "title": "required" } } }
```

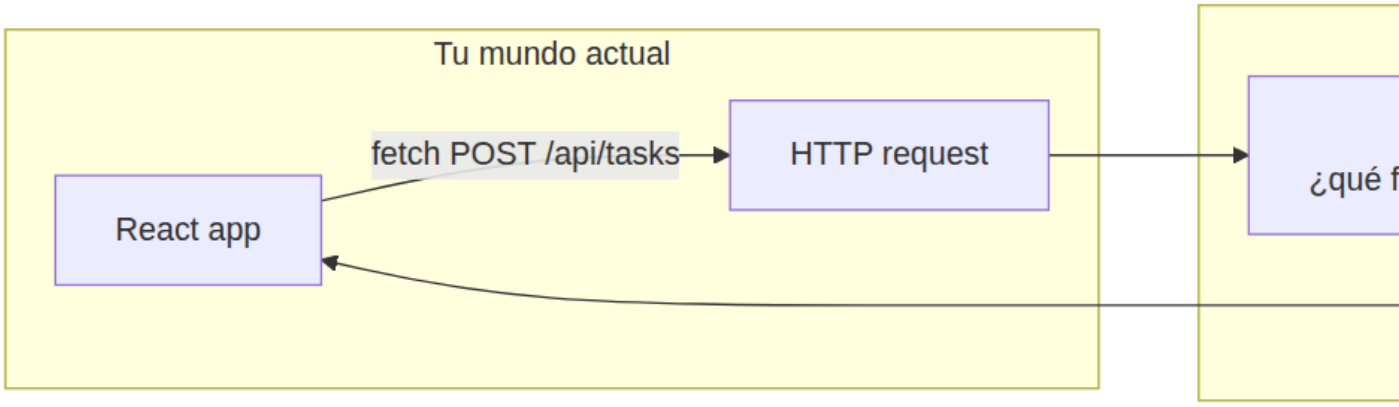
Ese contrato es lo único que el frontend conoce de ti. Es también lo que te permite reescribir tu backend en otro lenguaje sin que el frontend se entere — o que otro equipo escriba el frontend sin hablar contigo. **El contrato es el producto; la implementación es un detalle.**

10.3. Conceptos nuevos

- U **Servidor**: un proceso que corre *siempre*, escuchando un puerto, esperando requests. En frontend tu código lo ejecuta el navegador del usuario; en backend tú eres responsable de que el proceso esté vivo.
- U **El ciclo request/response**: `Request → Routing → Validación → Lógica → Datos → Response`. Cada capítulo del libro profundiza en una de esas etapas.
- U **Las cuatro preguntas**: ante cada request, un backend se pregunta *¿quién eres?* (autenticación), *¿puedes?* (autorización), *¿los datos son válidos?* (validación), *¿qué hago con ellos?* (lógica + persistencia).

- TS Py Go **App vs servidor**: en Node y Python el framework y el servidor son piezas separadas; en Go son la misma cosa. Detalle abajo.

10.4. La explicación visual



Cada etapa del diagrama es un capítulo (o varios). La gracia del diseño: si cada capa hace **un** trabajo, puedes testear la lógica sin HTTP y cambiar la base de datos sin tocar las rutas.

10.5. Implementación

El "hola mundo" honesto de un backend no es imprimir texto: es **responder un request**. Estos tres programas hacen exactamente lo mismo — exponen `GET /health` en el puerto 8000:

> Explicación de: puerto
>
> Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

10.5.1. TypeScript

10.5.1.1. Fundamentos (Node puro)

```
// src/index.ts - sin frameworks: node:http es lo que Express usa debajo
import { createServer } from "node:http";

const server = createServer((req, res) => {
  if (req.method === "GET" && req.url === "/health") {
    res.writeHead(200, { "Content-Type": "application/json" });
```

```
    res.end(JSON.stringify({ status: "ok" }));
    return;
  }
  res.writeHead(404).end(); // ← routing a mano: un if por ruta
});

server.listen(8000);
```

Esto **es** un servidor HTTP completo — Express solo te ahorra los `if`.

10.5.1.2. Express

```
// src/index.ts - el canónico del libro
import express from "express";

const app = express();

app.get("/health", (_req, res) => {
  res.json({ status: "ok" });
});

app.listen(8000, () => {
  console.log("listening on http://localhost:8000");
});
```

```
npm install express
npx tsx src/index.ts
```

10.5.1.3. Fastify / NestJS / Hono

```
// Fastify - misma idea, más rápido y con schemas integrados
import Fastify from "fastify";
const app = Fastify();
app.get("/health", async () => ({ status: "ok" }));
app.listen({ port: 8000 });
```

Qué te cuesta: Fastify es casi drop-in (más performance, plugins). **NestJS** te da arquitectura completa (decoradores, DI — estilo Angular) pero impone su estructura entera. **Hono** es ultraligero y corre en edge/bun/deno. Express sigue siendo el más común en equipos y docs.

10.5.2. Python

10.5.2.1. Fundamentos (stdlib)

```
# app/main.py - http.server: lo que hay debajo de todo
from http.server import BaseHTTPRequestHandler, HTTPServer

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == "/health":
            self.send_response(200)
            self.send_header("Content-Type", "application/json")
            self.end_headers()
            self.wfile.write(b'{"status": "ok"}')
        else:
            self.send_response(404)
            self.end_headers()

HTTPServer(("0.0.0.0", 8000), Handler).serve_forever()
```

Feo a propósito: así se siente HTTP sin framework. Uvicorn+FastAPI existe para no escribir esto.

10.5.2.2. FastAPI

```
# app/main.py - el canónico del libro
from fastapi import FastAPI

app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok"}
```

```
pip install fastapi uvicorn
uvicorn app.main:app --port 8000
```

10.5.2.3. Flask / Django

```
# Flask - el clásico minimalista: menos magia, más cables visibles
from flask import Flask
app = Flask(__name__)

@app.get("/health")
def health():
    return {"status": "ok"}
```

Qué te cuesta: Flask no trae validación ni async moderno (los agregas con plugins). **Django** es lo opuesto: baterías incluidas (ORM, admin, auth) pero impone "the Django way" entero. FastAPI queda en el medio: moderno, tipado, sin imponerte un monolito.

10.5.3. Go

10.5.3.1. net/http (stdlib — el canónico)

```go
// main.go - en Go la stdlib YA es la respuesta seria
package main

import (
    "encoding/json"
    "net/http"
)

func health(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(map[string]string{"status": "ok"})
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /health", health)
    http.ListenAndServe(":8000", mux)
}
```

```
go mod init taskflow
go run .
```

10.5.3.2. chi / Gin / Echo

```go
// chi - router ligero sobre net/http: mismos handlers, mejor routing
import "github.com/go-chi/chi/v5"

func main() {
    r := chi.NewRouter()
    r.Get("/health", health)
    http.ListenAndServe(":8000", r)
}
```

Qué te cuesta: **chi** es casi gratis (stdlib compatible, solo mejor mux). **Gin**/**Echo** son frameworks completos: más cómodos (JSON binding, middleware listo) pero otro modelo mental y una dependencia grande. En Go la stdlib es tan buena que la pregunta real es "¿necesito framework?" — muchas empresas dicen que no.

Prompt listo para tu AI

```
Crea un servidor HTTP mínimo que exponga GET /health respondiendo
{ "status": "ok" } en el puerto 8000, en [Express+TypeScript /
FastAPI+Python / net/http+Go]. Requisitos: código mínimo pero real,
comando para instalar dependencias y arrancar, y un curl para probarlo.
Después explícame línea por línea qué hace cada parte - quiero entender
qué es la "app" y qué es el "servidor" en este stack.
```

10.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres hacen lo mismo; lo que cambia es *cuánto te esconden*. Express y FastAPI separan "la app" de "el servidor"; en Go son la misma pieza. Si quieres el detalle, aquí está:

Detalle: app vs servidor en cada stack

- **Node/Express**: el *framework* (`app`) define rutas y responde; `app.listen` crea el servidor HTTP debajo. Dos piezas disfrazadas de una.
- **Python/FastAPI**: la separación es explícita — `app` es solo una *aplicación ASGI* (un objeto que sabe responder); **otro proceso** (`uvicorn`) es quien escucha el puerto y le pasa los requests. Por eso arrancas con `uvicorn app.main:app` y no `python main.py`.
- **Go**: no hay framework que separar — `net/http` *es* el servidor y el enrutador a la vez. El binario que compilas escucha el puerto directamente.

¿Por qué existe la separación app/servidor? Porque permite poner el servidor "tonto" (uvicorn, nginx) delante de varias aplicaciones, escalar workers, o testear la app sin abrir sockets. Go decide que esa indirección no vale la pena: simplicidad operativa sobre flexibilidad. Ninguna respuesta es "la correcta" — es una decisión de diseño del ecosistema, y reconocerlas así es pensar como backend.

Otras cimentaciones: ¿siempre un proceso escuchando un puerto?

Esta decisión es estructural: *algo* tiene que recibir el request y producir la respuesta — eso no se negocia. Pero la forma de la cimentación sí:

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Proceso + framework (este libro)	Tu proceso escucha un puerto 24/7	Tú mantienes el proceso vivo	El default: control total, cualquier host
Serverless (Lambda, Cloud Functions)	Tu función despierta por request	Cold starts, límites de tiempo, vendor lock-in	Tráfico irregular, sin ops
Edge (Workers, Deno Deploy)	Tu código corre cerca del usuario	Runtime limitado (no todo Node corre)	Latencia global, APIs ligeras

Contenedor + orquestador (K8s)	El proceso es el mismo; otros lo escalan	Complejidad operativa seria	Escala grande, muchos servicios

El punto: **el contrato HTTP y las capas internas son idénticos en las cuatro** — por eso el libro enseña el modelo `Request → … → Response` como estructura, y deja el runtime como material intercambiable.

10.7. Errores comunes

- **Creer que el framework es el servidor.** `app` no escucha nada; en Python lo verás explícito desde el día 1 (`uvicorn app:app`), en Express queda escondido en `app.listen`.
- **`print`/`console.log` como respuesta.** Responder un request es devolver bytes HTTP por la conexión, no escribir en tu terminal.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

- **Levantar el servidor y no probarlo.** Abre `http://localhost:8000/health` o haz `curl` — ver la respuesta JSON es el primer "funciona" real.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

10.8. Buenas prácticas

- **`/health` desde el primer commit.** Es el endpoint más aburrido del libro y el más importante: Docker, CI y cualquier balanceador lo usarán para saber si tu proceso está vivo.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

- **Puerto configurable**, no hardcodeado — en producción el puerto lo decide el entorno, no tu código (volveremos a esto en configuración).

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

- **Un endpoint, una responsabilidad.** El handler de `/health` no sabe de tareas ni de usuarios. Esa disciplina es la arquitectura por capas en miniatura.

10.9. Ejercicio

1. Levanta el servidor en **tu lenguaje elegido** y verifica `GET /health` desde el navegador.
2. Agrega `GET /version` que responda `{ "version": "0.1.0" }`.
3. Ahora hazlo en **uno de los otros dos lenguajes**. Cronometra: ¿qué fue distinto, qué fue idéntico?

10.10. Mini reto

Dibuja —a mano o en Mermaid— el ciclo de vida completo de un `POST /login` que ya conozcas desde el frontend: qué envía el navegador, qué verifica el servidor, qué devuelve (cookie, token), qué pasa con ese token en el siguiente request. Si hay etapas que no puedes dibujar, acabas de encontrar qué capítulos necesitas más.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Soluciones

Ejercicio. `GET /health` responde `{"status":"ok"}`; `GET /version` responde `{"version":"0.1.0"}`. Al repetirlo en otro lenguaje, lo idéntico es el contrato (ruta, verbo, status, JSON); lo distinto es solo la sintaxis para declarar el handler — esa es la tesis del libro.
Mini reto. Respuesta modelo: `POST /login` → el body lleva `{email, password}` → el servidor busca el usuario, verifica el hash del password (Cap. 14) → genera un JWT (Cap. 15) → lo devuelve en una cookie httpOnly → el navegador la adjunta sola en el siguiente request → un middleware/dependency la valida antes del handler (Cap. 16). Si te faltaron etapas 3–6, es exactamente la Parte 3 del libro.

10.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

10.12. Lo que deberías saber hacer ahora

- ☐ Explicar qué pasa entre `fetch()` y tu handler, etapa por etapa
- ☐ Levantar un proceso que responde HTTP en al menos un lenguaje

☐ Explicar la diferencia entre *aplicación* y *servidor* — y por qué en Go esa diferencia no existe
☐ Decir qué es un contrato de API y por qué es más importante que la implementación

Parte III

Sección II · Taskflow en memoria

11 2. Necesitamos que alguien escuche el puerto 8000

En backend, tú eres el navegador de alguien más

Figura 11.1: Parte 1 — Taskflow en memoria

> **En una frase**
>
> Vas a aprender **cómo se inicializa un proyecto backend** para que cualquiera (tu laptop, CI, producción) lo corra igual: manifiesto de dependencias, entorno aislado y un comando para arrancar. Es el `npm install` que ya conoces — ahora del lado del servidor.

11.1. El problema

El servidor del capítulo anterior era un juguete: un archivo, una librería, un comando. Pero ahora imagina que **otra persona clona tu repositorio**. ¿Qué instala? ¿Qué versión del runtime? ¿Qué comando arranca el proyecto?

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

En frontend esto lo resolviste mil veces sin pensar: `package.json`, `npm install`, `npm run dev`. Backend tiene exactamente el mismo problema — con una diferencia: ahora hay **tres** formas de resolverlo, y cada ecosistema decidió distinto.

11.2. Cómo lo resuelve un equipo

Un equipo real exige tres cosas antes de escribir la primera línea de lógica:

1. **Reproducibilidad**: cualquier máquina (tu laptop, CI, producción) instala *exactamente* las mismas dependencias.
2. **Aislamiento**: las librerías del proyecto A no contaminan al proyecto B.
3. **Un solo comando para arrancar**: si el README dice "paso 1, paso 2… paso 9", ya perdiste.

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Ejemplo de lo que un equipo espera encontrar en el README:

```
git clone … && cd taskflow
pip install -r requirements.txt   # o: npm install / go mod download
uvicorn app.main:app --reload     # o: npm run dev / go run .
# → GET http://localhost:8000/health responde {"status":"ok"}
```

Nuestro proyecto — **Taskflow** — nace aquí: la API de tareas que acompañará todo el libro. Empezamos por su esqueleto de entorno, no por sus endpoints.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

11.3. Conceptos nuevos

- U **Runtime**: el programa que ejecuta tu código (Node, el intérprete Python, el binario compilado de Go). En frontend era el navegador; aquí es un proceso del sistema operativo.

Explicación de: lenguaje compilado vs interpretado

Compilado (Go): el código se traduce a binario una vez y ese binario corre solo — rápido, deploy simple. **Interpretado** (Python, JS): un runtime lee el código en cada ejecución — flexible, pero necesita el runtime instalado.

Explicación de: proceso

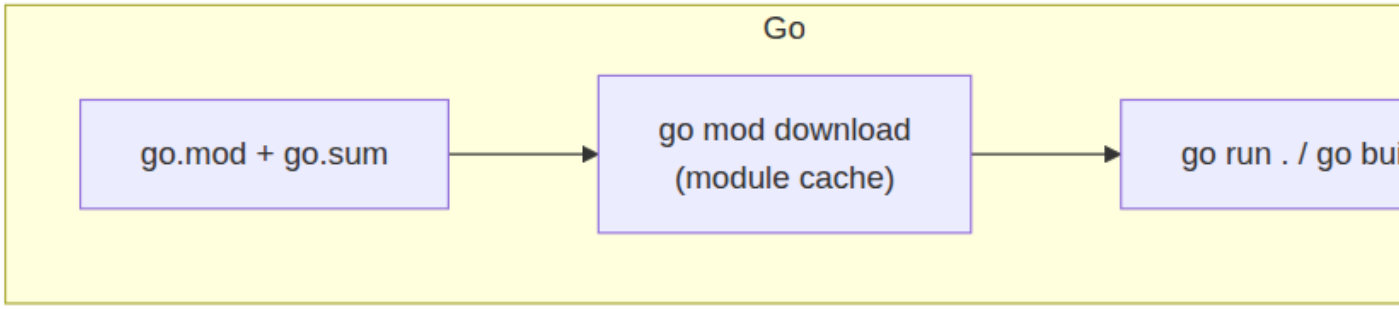
Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

- TS `package.json` + lockfile: manifiesto de dependencias + resolución exacta.
- Py `venv` + `requirements.txt`/`pyproject.toml`: el "node_modules por proyecto" de Python — pero como *entorno*, no carpeta.
- Go `go.mod`: dependencias versionadas por el propio toolchain; no hay "gestor" separado, `go` *es* el gestor.
- U **Watch mode**: reiniciar el servidor al guardar — `tsx --watch`, `uvicorn --reload`, `air` (o rebuild manual).

11.4. La explicación visual



Los tres responden a la misma pregunta — *"¿de qué depende este proyecto y cómo lo instalo igual en cualquier lado?"* — con mecánicas distintas.

11.5. Implementación

Inicializamos el proyecto Taskflow en los tres stacks. Fíjate en qué archivo es el "contrato de dependencias" de cada uno:

11.5.1. TypeScript

```
mkdir taskflow && cd taskflow
npm init -y
npm install express
npm install -D typescript tsx @types/express
npx tsc --init
```

`package.json` queda como manifiesto; `package-lock.json` fija las versiones exactas que se instalaron (ese archivo **sí** se commitea).

```
taskflow/
   package.json
   package-lock.json
   tsconfig.json
   src/
       index.ts
```

```
npx tsx watch src/index.ts   # arranca y reinicia al guardar
```

11.5.2. Python

```
mkdir taskflow && cd taskflow
python -m venv .venv          # el entorno aislado - tu "node_modules" conceptual
source .venv/bin/activate     # en Windows: .venv\Scripts\activate
pip install fastapi uvicorn
pip freeze > requirements.txt # el lockfile honesto: versiones exactas
```

```
taskflow/
   .venv/                # NO se commitea (va en .gitignore)
   requirements.txt
   app/
       __init__.py
       main.py
```

```
uvicorn app.main:app --reload --port 8000
```

`app.main:app` significa: "dentro del paquete `app`, módulo `main`, objeto `app`". Uvicorn importa tu código y le sirve HTTP — recuerda del Cap. 1: **uvicorn es el servidor, FastAPI es la aplicación**.

11.5.3. Go

```
mkdir taskflow && cd taskflow
go mod init taskflow          # crea go.mod - ya tienes gestor de dependencias
```

```
taskflow/
   go.mod
   main.go
```

```
go run .                      # compila y ejecuta
# o mejor, con rebuild automático:
go install github.com/air-verse/air@latest && air
```

Go no tiene carpeta de dependencias local: los módulos van a un caché global (`~/go/pkg/mod`) y `go.sum` registra los hashes exactos — el lockfile criptográfico viene incluido.

Prompt listo para tu AI

```
Inicializa un proyecto backend nuevo llamado "taskflow" en
[Express+TypeScript / FastAPI+Python / Go]. Requisitos: manifiesto de
dependencias (package.json / requirements.txt / go.mod), entorno aislado
(si aplica: venv), .gitignore correcto para el stack, estructura de
carpetas mínima, y un README con los 3 comandos: instalar, arrancar en
watch mode, y probar GET /health con curl. Dame los comandos de terminal
exactos paso a paso.
```

11.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: npm y pip instalan dependencias *en el proyecto*; Go las funde en un solo binario. Por eso el deploy de Go es copiar un archivo. El detalle:

Detalle: tres filosofías de dependencias

	npm (TS)	pip+venv (Py)	go mod (Go)
Dependencias viven en	`node_modules/` del proyecto	`.venv/` del proyecto	caché global con hashes
Manifiesto	`package.json`	`requirements.txt` / `pyproject.toml`	`go.mod`
Lockfile	`package-lock.json`	`pip freeze` (manual)	`go.sum` (automático)
Aislamiento	por carpeta	por entorno activado	por hashes en caché

La diferencia conceptual que importa: **npm y pip instalan dependencias "locales"** (el proyecto carga lo que está en su carpeta/entorno), mientras **Go compila un único binario** — las dependencias terminan fundidas en el ejecutable y desaparecen como concepto en producción. Por eso el deployment de Go (Cap. 30) será ridículamente simple: copias un archivo y listo.
Y una nota honesta: si en Express escribir `package.json` te pareció natural, el `venv` de Python te parecerá un paso extra torpe (hay que *activarlo*) — es el precio de que Python no tuviera `node_modules` por diseño durante años. Hoy existen herramientas modernas (`uv`, `poetry`) que lo hacen más cómodo; el libro usa `venv`+`pip` porque es lo que entenderás en cualquier equipo.

Otras cimentaciones: ¿siempre estos gestores de paquetes?

Esta decisión es material: *declarar y fijar dependencias* es estructural — pero el gestor concreto es intercambiable dentro de cada ecosistema.

Ecosistema	El del libro	Alternativas	Qué te cuesta / qué ganas
Node	`npm`	**pnpm** (más rápido, disco duro feliz), **yarn**, **bun** (runtime+gestor)	pnpm cambia `node_modules` a links — casi gratis y mejor; bun es joven pero rapidísimo
Python	`venv`+`pip`	**uv** (Rust, 10-100× más rápido), **poetry**/**pdm** (resolución+lockfile integrados), **conda**	uv es drop-in hoy; poetry/pdm gestionan el proyecto entero pero imponen su flujo
Go	`go mod`	— (el toolchain *es* el gestor)	No hay elección real — ventaja Go: una herramienta, una forma

La lección: cuando leas "instala con X", tradúcelo a "declara dependencia en el manifiesto del ecosistema". La sintaxis cambia; la obligación (manifiesto + lockfile + aislamiento) no.

11.7. Errores comunes

- **Instalar librerías globalmente** (`pip install` sin venv): funciona hoy, rompe mañana cuando otro proyecto necesita otra versión. En Python, el `venv` no es opcional — es la norma.
- **Commitear `node_modules/` o `.venv/`**: son artefactos locales; lo que se versiona es el *manifiesto* (`package.json`, `requirements.txt`, `go.mod`).
- **No fijar versiones**: "funciona con la última versión" es una bomba de tiempo. Lockfile siempre.
- **Olvidar activar el venv** en una terminal nueva: el síntoma es `ModuleNotFoundError` aunque "lo instalaste ayer" — lo instalaste *dentro* del entorno, no en el Python global.

11.8. Buenas prácticas

- **README con un comando de arranque** desde el día 1 (`npm run dev`, `uvicorn … --reload`, `go run .`).
- **`.gitignore` desde el primer commit**: `node_modules/`, `.venv/`, `.env`, `__pycache__/`, binarios compilados.
- **Versiones explícitas** en `requirements.txt` / `package.json` — nada de `install` a ciegas en producción.
- **El proyecto arranca limpio en una máquina virgen** como definición de "setup terminado": clona → instala → corre.

EXTRA · El panorama de lenguajes backend

EXTRA Curado de roadmaps externos (roadmap.sh, guías de bootcamp): los lenguajes que verás en el mercado, con su reputación honesta —

- **Go** — moderno y simple; rendimiento general excelente
- **Python** — elegante, ideal para apps backend sencillas
- **JavaScript / Node.js** — serial para aplicaciones web; desarrolladores de sobra
- **Java** — código más complejo, pero increíblemente rápido
- **C#** — excelente si trabajas en el ecosistema Microsoft
- **Ruby** — bueno si quieres Rails y te gusta ese estilo
- **PHP** — excelente para sitios web simples; menos para APIs

El libro elige TypeScript + Python + Go porque entre los tres cubren casi todos los modelos (event loop, async/GIL, goroutines). No hay elección "correcta" — hay el que tu equipo y tu mercado local usan. → *cap-40 compara a fondo.*

11.9. Ejercicio

1. Inicializa Taskflow en tu lenguaje elegido con la estructura de arriba.
2. Añade a `.gitignore` lo que corresponda a tu stack.
3. Escribe en el README los tres comandos: instalar, arrancar, probar `/health`.

11.10. Mini reto

Sin mirar documentación: explica qué información lleva un lockfile que el manifiesto no tiene, y por qué un equipo *exige* commitearlo. Pista: ¿qué pasa si Express publica mañana una versión con un bug y tu compañero instala después que tú?

Soluciones

Ejercicio. La estructura queda como los árboles de la sección Implementación; el `.gitignore` mínimo por stack: `node_modules/` (TS), `.venv/` + `__pycache__/` (Py), binario compilado (Go). Los tres comandos del README: instalar deps → arrancar en watch mode → `curl localhost:8000/health`.
Mini reto. El manifiesto dice "quiero Express ^4" — un *rango*; el lockfile registra la *versión exacta resuelta* (4.19.2) y la de cada dependencia transitiva, con hashes. Sin lockfile, tu compañero instala mañana la 4.19.3 con el bug y "en mi máquina funcionaba" — con lockfile, `npm ci`/`pip install -r`/`go mod download` instalan bit a bit lo mismo.

11.11. Vocabulario técnico del capítulo

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña →$2b10…`. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432,

HTTP el 80, HTTPS el 443.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

11.12. Lo que deberías saber hacer ahora

- ☐ Crear un entorno aislado y reproducible en tu stack
- ☐ Explicar la diferencia entre manifiesto de dependencias y lockfile
- ☐ Arrancar el servidor en watch mode y saber qué proceso lo reinicia
- ☐ Decir por qué `.venv`/`node_modules` no se commitean pero `requirements.txt`/`package-lock.json` sí

12 3. Necesitamos responder a URLs y verbos

Qué es exactamente un endpoint y qué promete

> **En una frase**
>
> Vas a construir tu **primer CRUD real**: cinco endpoints (`GET`, `POST`, `DELETE`…) que listan, crean y borran tareas en memoria — y a diseñar la tabla verbo+ruta que *es* el contrato de la API.

12.1. El problema

Taskflow necesita su primera funcionalidad real: **gestionar tareas**. El frontend (que ya sabes escribir) va a hacer esto:

```
await fetch("/api/tasks");                          // listar
await fetch("/api/tasks", { method: "POST", … });  // crear
await fetch("/api/tasks/42");                       // ver una
await fetch("/api/tasks/42", { method: "DELETE" });// borrar
```

Cada una de esas llamadas es una promesa que tu servidor debe cumplir: qué ruta atiende, qué verbo significa qué, qué código de estado devuelve. Diseñar eso **es** diseñar la API — y es una conversación de equipo antes que de código.

> **Explicación de: return**
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

12.2. Cómo lo resuelve un equipo

Antes de escribir handlers, un equipo se sienta sobre una tabla así:

Verbo + ruta	Significado	Éxito	Error esperado
`GET /tasks`	listar todas	200 + array	—

Verbo + ruta	Significado	Éxito	Error esperado
`POST /tasks`	crear	`201` + tarea creada	`400` body inválido
`GET /tasks/{id}`	una tarea	`200` + tarea	`404` no existe
`PATCH /tasks/{id}`	modificar parcial	`200` + tarea	`404`, `400`
`DELETE /tasks/{id}`	borrar	`204` sin body	`404`

Dos decisiones de diseño ya están tomadas en esa tabla — fíjate:

- **Sustantivos plurales, no verbos en la URL**: `/tasks`, no `/getTasks` ni `/createTask`. El verbo ya lo pone el método HTTP.
- **Los códigos de estado son parte del contrato**: `201` dice "se creó", `204` dice "salió bien y no hay nada que devolver", `404` dice "eso no existe". Devolver `200` con `{"error": "not found"}` es mentirle al cliente.

12.3. Conceptos nuevos

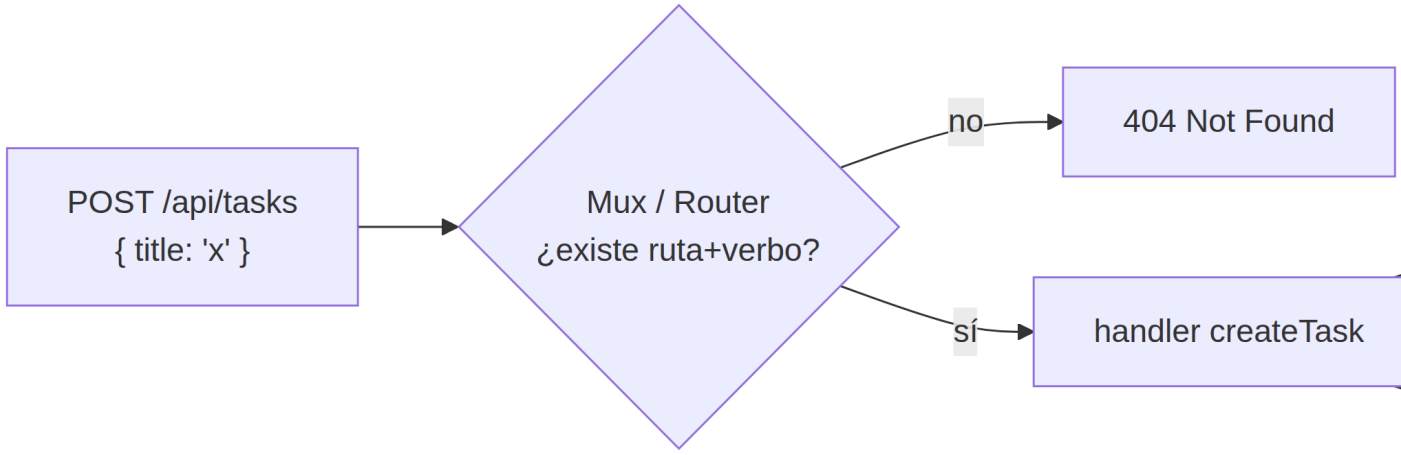
- U **Endpoint**: la pareja (verbo + ruta) que atiende un handler. `GET /tasks` y `POST /tasks` son dos endpoints distintos aunque compartan ruta.
- U **Path params** (`/tasks/{id}`) vs **query params** (`/tasks?status=done`): el path identifica *qué recurso*; el query modifica *cómo* se devuelve (filtros, orden, paginación).

> Explicación de: query / consulta
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

- U **Idempotencia** (primera aparición): `GET`, `PUT`, `DELETE` repetidos dan el mismo resultado; `POST` no — cada llamada crea otra tarea. Esto importa cuando el frontend reintenta.
- TS Py Go Tres sintaxis de routing: cadena con `:id` (Express), decorator + firma (FastAPI), patrón `"GET /tasks/{id}"` en `ServeMux` (Go 1.22+).

> Explicación de: firma / HMAC
>
> Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

12.4. La explicación visual



El router es un **despachador**: compara `(método, path)` contra la tabla de rutas registradas y entrega el request al handler correspondiente — o responde 404 él mismo si nadie reclamó la ruta.

12.5. Implementación

CRUD completo en memoria. El "modelo" es intencionalmente una lista — la base de datos llega en la Parte 2; hoy nos importa el *contrato HTTP*.

> Explicación de: array / lista / slice
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> Explicación de: memoria (RAM)
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

12.5.1. TypeScript

12.5.1.1. Fundamentos (Node puro)

```
// Routing a mano: así se siente sin Express
import { createServer } from "node:http";

createServer((req, res) => {
  const parts = req.url!.split("/");          // "/tasks/42" → ["","tasks","42"]
  if (req.method === "GET" && parts[1] === "tasks" && !parts[2]) {
    res.end(JSON.stringify(tasks));
  } else if (req.method === "GET" && parts[1] === "tasks") {
    const id = Number(parts[2]);              // path param manual
    // …find, 404, res.end…
  }
  // …un if anidado por cada (método, ruta)
}).listen(8000);
```

Esto funciona — y es exactamente lo que Express abstrae: parsing de path params, dispatch por método, helpers de respuesta.

12.5.1.2. Express

```
// src/index.ts - el canónico del libro
import express from "express";

const app = express();
app.use(express.json());

interface Task {
  id: number;
  title: string;
  done: boolean;
}

const tasks: Task[] = [];
let nextId = 1;

app.get("/tasks", (_req, res) => {
  res.json(tasks);
});

app.post("/tasks", (req, res) => {
  const { title } = req.body ?? {};
  if (typeof title !== "string" || !title.trim()) {
    return res.status(400).json({ error: "title is required" });
```

```
  }
  const task: Task = { id: nextId++, title, done: false };
  tasks.push(task);
  res.status(201).json(task);
});

app.get("/tasks/:id", (req, res) => {
  const task = tasks.find((t) => t.id === Number(req.params.id));
  if (!task) return res.status(404).json({ error: "task not found" });
  res.json(task);
});

app.delete("/tasks/:id", (req, res) => {
  const i = tasks.findIndex((t) => t.id === Number(req.params.id));
  if (i === -1) return res.status(404).json({ error: "task not found" });
  tasks.splice(i, 1);
  res.status(204).send();
});

app.listen(8000);
```

12.5.1.3. Hono / Fastify

```
// Hono - las mismas rutas, API casi idéntica, corre en edge/bun/deno
import { Hono } from "hono";
const app = new Hono();

app.get("/tasks", (c) => c.json(tasks));
app.get("/tasks/:id", (c) => {
  const task = tasks.find((t) => t.id === Number(c.req.param("id")));
  return task ? c.json(task) : c.json({ error: "not found" }, 404);
});
```

Qué te cuesta: Hono/Fastify son swaps casi directos — mismo modelo, APIs más rápidas/modernas. **NestJS** es otro planeta: decoradores `@Get("/tasks")` sobre clases con DI — más estructura, más framework.

12.5.2. Python

12.5.2.1. Fundamentos (stdlib)

```
# Routing manual: lo que FastAPI hace por ti
from http.server import BaseHTTPRequestHandler

class Handler(BaseHTTPRequestHandler):
```

```
    def do_GET(self):
        if self.path == "/tasks":
            ...  # listar
        elif self.path.startswith("/tasks/"):
            task_id = int(self.path.split("/")[2])  # path param a mano
            ...
```

12.5.2.2. FastAPI

```
# app/main.py - el canónico del libro
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class Task(BaseModel):
    id: int
    title: str
    done: bool = False

class TaskCreate(BaseModel):
    title: str

tasks: list[Task] = []
next_id = 1

@app.get("/tasks")
def list_tasks() -> list[Task]:
    return tasks

@app.post("/tasks", status_code=201)
def create_task(payload: TaskCreate) -> Task:
    global next_id
    task = Task(id=next_id, title=payload.title)
    next_id += 1
    tasks.append(task)
    return task

@app.get("/tasks/{task_id}")
def get_task(task_id: int) -> Task:
    for t in tasks:
        if t.id == task_id:
            return t
    raise HTTPException(status_code=404, detail="task not found")

@app.delete("/tasks/{task_id}", status_code=204)
def delete_task(task_id: int):
```

```
    for i, t in enumerate(tasks):
        if t.id == task_id:
            tasks.pop(i)
            return
    raise HTTPException(status_code=404, detail="task not found")
```

12.5.2.3. Flask / Django

```
# Flask - mismo mapa de rutas, validación manual
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.get("/tasks/<int:task_id>")
def get_task(task_id):
    task = next((t for t in tasks if t["id"] == task_id), None)
    return (jsonify(task), 200) if task else (jsonify({"error": "not found"}), 404)
```

Qué te cuesta: Flask te da routing y JSON, pero la validación la escribes tú (no hay `task_id: int` que convierta+valide en la firma). **Django REST Framework** va al otro extremo: ViewSets + serializers — potencia total, framework entero.

12.5.3. Go

12.5.3.1. net/http (stdlib — el canónico)

```
// main.go
package main

import (
    "encoding/json"
    "net/http"
    "strconv"
)

type Task struct {
    ID    int    `json:"id"`
    Title string `json:"title"`
    Done  bool   `json:"done"`
}

var tasks []Task
var nextID = 1

func listTasks(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(tasks)
```

```
}

func createTask(w http.ResponseWriter, r *http.Request) {
    var in struct{ Title string `json:"title"` }
    if err := json.NewDecoder(r.Body).Decode(&in); err != nil || in.Title == "" {
        http.Error(w, `{"error":"title is required"}`, http.StatusBadRequest)
        return
    }
    t := Task{ID: nextID, Title: in.Title}
    nextID++
    tasks = append(tasks, t)
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(t)
}

func getTask(w http.ResponseWriter, r *http.Request) {
    id, _ := strconv.Atoi(r.PathValue("id"))
    for _, t := range tasks {
        if t.ID == id {
            json.NewEncoder(w).Encode(t)
            return
        }
    }
    http.Error(w, `{"error":"task not found"}`, http.StatusNotFound)
}

func deleteTask(w http.ResponseWriter, r *http.Request) {
    id, _ := strconv.Atoi(r.PathValue("id"))
    for i, t := range tasks {
        if t.ID == id {
            tasks = append(tasks[:i], tasks[i+1:]...)
            w.WriteHeader(http.StatusNoContent)
            return
        }
    }
    http.Error(w, `{"error":"task not found"}`, http.StatusNotFound)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /tasks", listTasks)
    mux.HandleFunc("POST /tasks", createTask)
    mux.HandleFunc("GET /tasks/{id}", getTask)
    mux.HandleFunc("DELETE /tasks/{id}", deleteTask)
    http.ListenAndServe(":8000", mux)
}
```

12.5.3.2. chi / Gin

```
// chi - mismos handlers http.HandlerFunc, router con más conveniencias
r := chi.NewRouter()
r.Get("/tasks", listTasks)
r.Get("/tasks/{id}", getTask)   // chi.URLParam(r, "id")

// Gin - framework completo: binding, JSON helpers, middleware built-in
router := gin.Default()
router.GET("/tasks/:id", func(c *gin.Context) {
    id, _ := strconv.Atoi(c.Param("id"))
    // c.JSON(200, task) / c.AbortWithStatusJSON(404, …)
})
```

Qué te cuesta: chi es stdlib-friendly y casi gratis. Gin/Echo dan contextos propios (`*gin.Context` en vez de `(w, r)`) — cómodos pero te atan a su API. En Go muchas APIs serias viven solo con `ServeMux`.

Así se prueba cualquiera de los tres — misma llamada, mismo resultado:

```
curl -i -X POST http://localhost:8000/tasks \
  -H "Content-Type: application/json" -d '{"title":"ship v1"}'

# HTTP/1.1 201 Created
# {"id":1,"title":"ship v1","done":false}

curl -i http://localhost:8000/tasks/99
# HTTP/1.1 404 Not Found
# {"error":"task not found"}
```

Prompt listo para tu AI

```
Implementa un CRUD en memoria para una API de tareas en
[Express+TypeScript / FastAPI+Python / net/http+Go]. Endpoints:
GET /tasks (lista), POST /tasks (crear, body {title}), GET /tasks/{id}
(una, 404 si no existe), DELETE /tasks/{id} (204 o 404). Requisitos:
status codes correctos (201, 204, 404), sin base de datos (array en
memoria), y los comandos curl para probar cada endpoint. Después
explícame qué convención REST aplica cada ruta.
```

12.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: las rutas, verbos y status codes son *idénticos* en los tres — eso es HTTP, no el lenguaje. Lo que cambia es cuánto trabajo te quita el framework encima. El detalle:

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Detalle: tres filosofías de routing

- **Express** es "Imperativo minimalista": tú parseas el body (`express.json()`), tú conviertes `req.params.id` de string a número, tú eliges `res.status(201)`. Libertad total = responsabilidad total.
- **FastAPI** hace lo contrario: declaras `task_id: int` en la firma y el framework *convierte y valida* — si llega `/tasks/abc`, responde `422` antes de tocar tu función. Menos código, más "magia" (que el Cap. 16 desmontará).
- **Go stdlib** es explícito al máximo: `r.PathValue("id")` devuelve string y tú la conviertes; `http.Error` escribe la respuesta de error a mano. Cero magia, cero sorpresas, más líneas.

Nótese lo que es **idéntico** en los tres: la forma de las rutas, los verbos, los status codes. Eso es el protocolo HTTP — el lenguaje solo cambia *cómo* lo expresas.

Otras cimentaciones: ¿siempre REST con sustantivos?

Esta decisión es material: *algún* mapa verbo+ruta→handler debe existir (estructural), pero las **convenciones de diseño** son elegibles — REST es una cimentación popular, no la única.

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
REST de recursos (este libro)	Sustantivos + verbos HTTP + status	Diseñar recursos bien; no todo encaja en CRUD	APIs públicas, equipos diversos — el default
RPC-style (`/getTask`, `/doAction`)	Funciones remotas por nombre	Pierdes convenciones (caché, idempotencia) que HTTP regala	APIs internas, operaciones que no son recursos
GraphQL	Un endpoint, el cliente pide campos	Otra capa entera: schema, resolvers, N+1	Clientes con necesidades de datos muy distintas
tRPC / OpenAPI codegen	Contrato tipado end-to-end	Acopla cliente y servidor al mismo lenguaje/schema	Monorepos TS, equipos full-stack

El libro usa REST porque sus convenciones (status codes, idempotencia) son *lecciones gratis de diseño* — y porque es lo que el 90 % de los equipos espera ver.

12.7. Errores comunes

- **Responder siempre 200.** `200 {"error":"not found"}` obliga al cliente a parsear el body para saber si falló. El status code existe para eso.
- **Verbos en la URL** (`/getTask/42`, `/deleteTask/42`): duplica lo que el método ya dice y rompe las convenciones que caches y herramientas asumen.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

- **Olvidar `express.json()`** (o equivalente): `req.body` llega `undefined` y pasas 20 minutos creyendo que el cliente manda mal.
- **Asumir que `{id}` es número**: en Express y Go llega string — `Number("abc")` es `NaN`, `Atoi("abc")` es error. FastAPI lo resuelve por ti; en los otros es tu trabajo (hasta el capítulo de validación).

12.8. Buenas prácticas

- **Tabla de rutas antes que código**: la tabla de arriba *es* el diseño.
- **`201` + el recurso creado** en el body — el frontend no necesita un segundo `GET` para pintar la tarea.
- **`204` para DELETE exitoso**: "salió bien, no hay contenido" es más honesto que devolver `{}`.
- **Query params para filtros** (`/tasks?done=false`), path params para identidad (`/tasks/42`). Mezclarlos confunde el contrato.

EXTRA · Frameworks que te encontrarás fuera

EXTRA El libro usa Express, FastAPI y Go estándar, pero el ecosistema real es más amplio. Roadmaps externos recomiendan conocer —

- **Ruby on Rails** — "convención sobre configuración", testing integrado, localización; open source escrito en Ruby
- **Django** (Python) — "baterías incluidas": auth, admin, ORM; hoy también async
- **Flask** (Python) — microframework minimalista, servidor de desarrollo integrado, rápido para prototipar
- **Express** (Node) — ligero y adaptable; depuración y desarrollo de servidores rápidos

Mismo patrón en todos: rutas → handlers → middleware. Si entiendes el de este libro, lees cualquiera de estos en una tarde.

12.9. Ejercicio

1. Implementa el CRUD en tu stack y pruébalo con `curl` o Thunder Client: crear dos tareas, listar, obtener una, borrar otra, verificar el 404.
2. Agrega `PATCH /tasks/{id}` que solo permita cambiar `done`.
3. Agrega `GET /tasks?done=true` que filtre por query param.

12.10. Mini reto

Diseña las rutas de una API de "likes" sobre tareas: un usuario puede dar like a una tarea, quitarlo, y ver quiénes dieron like. Sin verbos en la URL y sin ambigüedad — escribe la tabla antes de pensar en código.

Soluciones

Ejercicio 2 (PATCH). Solo se acepta `done`:

```python
@app.patch("/tasks/{task_id}")
def update_task(task_id: int, payload: TaskUpdate):
    task = find_or_404(task_id)
    if payload.done is not None:
        task.done = payload.done
    return task
# TaskUpdate = BaseModel con done: bool | None = None
```

Ejercicio 3 (filtro): `GET /tasks?done=true` → `[t for t in tasks if t.done == done]` cuando el query param está presente.
Mini reto. Tabla modelo:

Verbo + ruta	Significado	Éxito
`PUT /tasks/{id}/like`	dar like (idempotente: repetir no duplica)	`204`
`DELETE /tasks/{id}/like`	quitar like	`204`
`GET /tasks/{id}/likes`	quiénes dieron like	`200` + lista

La decisión clave: `PUT` en vez de `POST` para dar like — como un usuario solo puede dar like una vez, la operación es idempotente y `PUT` lo expresa.

12.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: booleano

Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

12.12. Lo que deberías saber hacer ahora

- ☐ Diseñar una tabla verbo+ruta+status antes de escribir handlers
- ☐ Explicar la diferencia entre path param y query param y cuándo usar cada uno
- ☐ Implementar un CRUD en memoria con status codes correctos en tu stack
- ☐ Decir qué hace el router cuando nadie reclama `(método, path)`

13 4. Necesitamos confiar en lo que entra (o no confiar)

El cliente manda basura; la validación es la muralla

> **En una frase**
>
> Vas a aprender a **rechazar basura antes de que toque tu lógica**: un schema declarativo (Zod / Pydantic / validator) describe la forma válida una vez y los requests malos mueren con 422 antes de llegar al handler.

13.1. El problema

El Cap. 3 terminó con una mentira piadosa: dijimos que `400` era "body inválido", pero lo único que validamos fue `title` a mano. Mira lo que el cliente puede mandar de verdad:

```
{
  "title": 42,
  "priority": "mucho",
  "due_date": "ayer",
  "evil": "<script>alert(1)</script>",
  "unexpected_field": "¿y esto qué?"
}
```

Tu handler recibe bytes. `JSON.parse` los convierte en un objeto — **pero nadie garantiza que ese objeto tenga la forma que tu lógica espera**. En frontend, un dato malo rompe la UI del usuario que lo mandó. En backend, un dato malo entra a la base de datos y corrompe el sistema para todos.

> **Explicación de: objeto / dict / struct**
>
> Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

13.2. Cómo lo resuelve un equipo

La regla profesional es simple y brutal:

> **Nada que venga del cliente se usa sin pasar por validación.**

Ejemplo del objetivo: que este request

```
curl -X POST /tasks -d '{"title": "", "priority": 9}'
```

muera *antes* del handler con algo así:

```
422 {
  "error": {
    "code": "VALIDATION",
    "fields": {
      "title": "String should have at least 1 character",
      "priority": "Input should be less than or equal to 5"
    }
  }
}
```

Y la forma moderna de hacerlo es *declarativa*: en vez de veinte `if` sueltos en el handler, describes **la forma válida** una sola vez — un schema — y dejas que una capa anterior rechace lo que no cumple. El handler solo se ejecuta si el dato ya es válido. En el diagrama del Cap. 1, la validación es el muro que está *antes* de la lógica de negocio:

> Explicación de: esquema (schema)
>
> El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

```
Request → Router → [VALIDACIÓN] → Handler → …
                     ↑ si falla: 400/422 y el handler ni se entera
```

13.3. Conceptos nuevos

- U **Schema de validación**: la declaración de la forma válida — campos, tipos, reglas (`min`, `max`, formato email).
- U **Parse, don't validate**: no "revisar y seguir", sino *transformar* el JSON crudo en un objeto de tu dominio — si no puede transformarse, se rechaza ahí.

> Explicación de: dominio
>
> Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

- TS **Zod**: el schema define y *genera* el tipo TypeScript — `z.infer` conecta validación de runtime con tipos de compilación.

> Explicación de: runtime
>
> El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

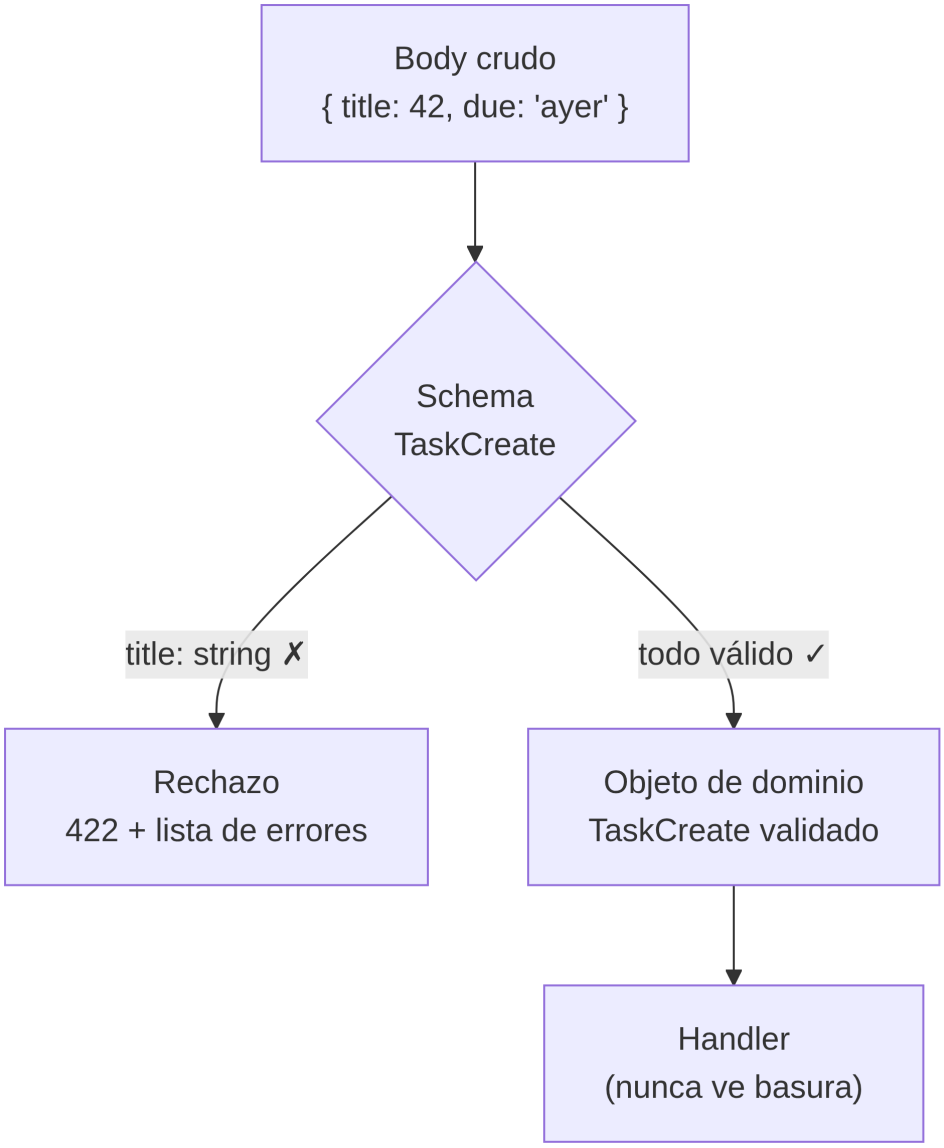
> Explicación de: compilador / transpilador
>
> Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

- Py **Pydantic + type hints**: `class TaskCreate(BaseModel)` — el type hint *es* la validación; FastAPI la aplica antes del handler y devuelve `422` solo.

> Explicación de: return
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

- Go **Validación manual o struct tags**: Go no tiene runtime reflexivo mágico cómodo — se valida explícito o con `go-playground/validator` sobre struct tags.
- U **400 vs 422**: `400` = JSON roto o semántica general inválida; `422` = JSON bien formado que no cumple el schema (convención popularizada por FastAPI). Elige una y sé consistente.

13.4. La explicación visual



El punto clave: **el handler vive aguas abajo de la validación**. Si tu código de negocio está chequeando tipos, el muro está en el lugar equivocado.

13.5. Implementación

Agregamos el schema `TaskCreate` real: `title` requerido (1–200 chars), `priority` opcional 1–5, `due_date` opcional en formato ISO.

13.5.1. TypeScript

13.5.1.1. Fundamentos (a mano)

```
// Validación sin librería: veinte ifs - el problema antes de la solución
app.post("/tasks", (req, res) => {
  const { title, priority, due_date } = req.body ?? {};
  if (typeof title !== "string" || title.length < 1 || title.length > 200)
    return res.status(422).json({ error: "title invalid" });
  if (priority !== undefined &&
      (typeof priority !== "number" || priority < 1 || priority > 5))
    return res.status(422).json({ error: "priority invalid" });
  // …y esto solo cubre 3 campos de un endpoint
});
```

Nota lo que falta: el `if` *rechaza* pero no produce un objeto tipado — `title` sigue siendo `any` para TypeScript.

13.5.1.2. Zod (canónico)

```
import { z } from "zod";

const createTaskSchema = z.object({
  title: z.string().trim().min(1).max(200),
  priority: z.number().int().min(1).max(5).optional(),
  due_date: z.string().date().optional(),  // "2026-10-01"
});

// el tipo TS sale del schema - no se escribe dos veces
type TaskCreate = z.infer<typeof createTaskSchema>;

app.post("/tasks", (req, res) => {
  const parsed = createTaskSchema.safeParse(req.body);
  if (!parsed.success) {
    return res.status(422).json({
      error: "validation failed",
      details: parsed.error.issues.map((i) => ({
        field: i.path.join("."),
        message: i.message,
      })),
    });
  }
  const data: TaskCreate = parsed.data;   // ya es del tipo correcto
  // … crear tarea con data
});
```

13.5.1.3. Valibot / Joi / ArkType

```
// Valibot - API casi idéntica a Zod, pero modular (bundle más chico)
import * as v from "valibot";
const createTaskSchema = v.object({
  title: v.pipe(v.string(), v.minLength(1), v.maxLength(200)),
  priority: v.optional(v.pipe(v.number(), v.minValue(1), v.maxValue(5))),
});
```

Qué te cuesta: Valibot/ArkType = mismo modelo que Zod, mejor bundle o más velocidad — son swaps directos. **Joi** es el veterano de Express (mismo concepto, API anterior a TS-first). El patrón es uno: *schema → parse → objeto de dominio.*

13.5.2. Python

13.5.2.1. Fundamentos (dict manual)

```
# Validación sobre el dict crudo - lo que Pydantic te ahorra
@app.post("/tasks")
def create_task(request: Request):
    # body = await request.json() → dict sin garantías
    ...
    if not isinstance(body.get("title"), str) or not (1 <= len(body["title"]) <= 200):
        raise HTTPException(422, "title invalid")
    # …y body sigue siendo un dict - sin tipos, sin autocompletar
```

13.5.2.2. FastAPI + Pydantic (canónico)

```
from datetime import date
from fastapi import FastAPI
from pydantic import BaseModel, Field

class TaskCreate(BaseModel):
    title: str = Field(min_length=1, max_length=200)
    priority: int | None = Field(default=None, ge=1, le=5)
    due_date: date | None = None   # Pydantic parsea "2026-10-01" a date

@app.post("/tasks", status_code=201)
def create_task(payload: TaskCreate):
    # si llegas aquí, payload YA es válido - el 422 ocurrió antes
    ...
```

Si el cliente manda `{"title": 42}`, FastAPI responde automáticamente:

```
{
  "detail": [{
    "loc": ["body", "title"],
    "msg": "Input should be a valid string",
    "type": "string_type"
  }]
}
```

13.5.2.3. msgspec / Marshmallow / DRF serializers

```
# msgspec - el mismo modelo que Pydantic, compilado: más rápido
import msgspec

class TaskCreate(msgspec.Struct):
    title: str
    priority: int | None = None
```

Qué te cuesta: msgspec valida igual pero con menos features de conversión. **Marshmallow/DRF serializers** son la generación anterior: mismo patrón de schema declarativo, más verboso. La idea (declarar la forma, parsear, rechazar antes) es idéntica en todas.

13.5.3. Go

13.5.3.1. net/http + validación manual

```
// Sin librería: Decode + ifs explícitos - Go te obliga a ver cada check
var in struct {
    Title    string `json:"title"`
    Priority *int   `json:"priority"`
}
if err := json.NewDecoder(r.Body).Decode(&in); err != nil {
    http.Error(w, `{"error":"invalid json"}`, 400)
    return
}
if in.Title == "" || len(in.Title) > 200 {
    http.Error(w, `{"error":"title invalid"}`, 422)
    return
}
```

13.5.3.2. validator (canónico del capítulo)

```go
type CreateTaskInput struct {
    Title    string  `json:"title"    validate:"required,min=1,max=200"`
    Priority *int    `json:"priority" validate:"omitempty,min=1,max=5"`
    DueDate  *string `json:"due_date" validate:"omitempty,datetime=2006-01-02"`
}

func createTask(w http.ResponseWriter, r *http.Request) {
    var in CreateTaskInput
    if err := json.NewDecoder(r.Body).Decode(&in); err != nil {
        http.Error(w, `{"error":"invalid json"}`, http.StatusBadRequest)
        return
    }
    if err := validate.Struct(in); err != nil { // go-playground/validator
        w.WriteHeader(http.StatusUnprocessableEntity)
        json.NewEncoder(w).Encode(map[string]any{
            "error":   "validation failed",
            "details": err.Error(),
        })
        return
    }
    // in es válido - el handler sigue
}
```

13.5.3.3. ozzo-validation / manual

```go
// ozzo - validación por reglas de función en vez de struct tags
func (i CreateTaskInput) Validate() error {
    return validation.ValidateStruct(&i,
        validation.Field(&i.Title, validation.Required, validation.Length(1, 200)),
        validation.Field(&i.Priority, validation.Min(1), validation.Max(5)),
    )
}
```

Qué te cuesta: `validator` usa struct tags (declarativo, pero errores crípticos); ozzo usa código normal (más explícito, IDE-friendly). En Go la validación manual también es legítima — no hay magia que esconder.

Prompt listo para tu AI

```
Agrega validación declarativa al endpoint POST /tasks de mi API en
[Zod+Express / Pydantic+FastAPI / validator+Go]. El body válido es:
{ title: string (1-200 chars, requerido), priority: number (1-5,
opcional), due_date: string ISO date (opcional) }. Requisitos: schema
definido UNA sola vez fuera del handler, rechazo con 422 y errores
estructurados por campo ({field, message}), y que el handler solo se
ejecute si el dato ya es válido. Incluye curl de ejemplo con un body
inválido y la respuesta 422 esperada.
```

13.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: declaras la forma válida una vez y el muro rechaza lo demás — en los tres stacks. La diferencia profunda es *qué existe en runtime*: los tipos TS se evaporan al compilar, los hints de Python solo son notas… hasta que Pydantic los lee, y los tipos de Go sí viven en el binario. El detalle:

Detalle: quién valida en runtime

- **TypeScript**: los tipos (`interface Task`) *no existen en runtime* — se evaporan al compilar. Por eso necesitas Zod: es el puente entre "lo que TS cree" y "lo que realmente llegó". Si declaras el tipo sin schema, tienes una ilusión de seguridad.
- **Python**: los type hints normalmente también son solo para el editor — pero Pydantic los lee *en runtime* y los convierte en validación. Es el truco que hace a FastAPI tan cómodo: escribes el tipo una vez y sirve para autocompletar, validar y documentar.
- **Go**: el tipado estático *sí* existe en runtime (es compilado), así que un `int` nunca será un string — pero Go no puede validar "min 1, max 5" en el tipo; por eso las struct tags o el chequeo manual. Menos mágico, más honesto.

La lección universal: **el contrato de entrada debe existir fuera del handler**, expresado una sola vez, y rechazar antes de ejecutar lógica. Zod, Pydantic y validator son tres implementaciones del mismo muro.

Otras cimentaciones: ¿siempre un schema declarativo?

Esta decisión es estructural: validar la entrada *antes* del handler es obligatorio — es la zapata sin la cual el edificio se cae. Lo intercambiable es *cómo* declaras la forma válida:

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Schema en código (Zod/Pydantic/validator)	La forma válida vive junto al handler	Una dependencia por ecosistema	El default moderno — lo del capítulo

JSON Schema / OpenAPI primero	El contrato es un documento; el código se genera	Toolchain extra (codegen), curva	APIs públicas con muchos consumidores
Validación manual (`if`s)	Reglas en el handler	Se desordena, se olvida, sin tipos	Solo un campo trivial — se degrada rápido
Validación solo en frontend	El formulario rechaza	**Nada protege al servidor** — `curl` lo salta	Nunca como única defensa (sí como UX)
Validación en DB (constraints)	Postgres rechaza el INSERT	El error llega tarde y feo	Como *segunda* muralla, no la primera

La cimentación correcta: schema declarativo en el borde + constraints en la DB como respaldo. Lo demás son materiales para el mismo muro.

13.7. Errores comunes

- **Validar dentro del handler con `if` sueltos**: se desordena, se olvida, y cada endpoint termina con reglas distintas.
- **Creer que el tipo TypeScript valida**: `interface` no existe en runtime — `req.body` es `any` disfrazado.
- **Devolver errores crudos del validador**: el frontend necesita `{ field, message }` estructurado, no el volcado interno de la librería.
- **Validar solo en el frontend**: el formulario valida para UX; el servidor valida porque es *la última línea de defensa* — cualquiera puede hacer `curl` saltándose tu React.

13.8. Buenas prácticas

- **Schema por operación**: `TaskCreate` (sin `id`), `TaskUpdate` (todo opcional), `TaskPublic` (salida). Un solo schema para todo es deuda.
- **Rechazar campos desconocidos** cuando el contrato es estricto (`z.object().strict()`, `model_config = ConfigDict(extra="forbid")`).
- **Errores de validación con forma estable**: el frontend mostrará “priority: debe ser 5” junto al campo — dale la estructura para hacerlo.
- **La validación también es seguridad**: strings con límite de longitud, enums cerrados y formatos estrictos cortan inyecciones antes de nacer.

13.9. Ejercicio

1. Agrega a Taskflow el schema `TaskCreate` de arriba en tu stack y prueba mandar: sin `title`, `priority: 9`, `due_date: "mañana"`. Verifica que ninguna llegue al handler.
2. Diseña `TaskUpdate` (todo opcional, pero al menos un campo presente) — ¿cómo expresas “al menos uno” en tu validador?

13.10. Mini reto

Haz que un email malformado **nunca** llegue al handler de un futuro `POST /users` — en tu lenguaje. Pregunta extra: ¿aceptarías `"a@b"` como email válido? Investiga qué decide cada librería y por qué la validación de emails es más tramposa de lo que parece.

Soluciones

Ejercicio 1. El schema `TaskCreate` con `Field(min_length=1, max_length=200)` / `z.string().min(1).max(200)` / `validate:"required,max=200"` rechaza los tres casos con 422 — el handler no se ejecuta.
Ejercicio 2 (`TaskUpdate`, "al menos un campo"): en Pydantic, `model_validator(mode="after")` que falle si todos los campos son `None`; en Zod, `.refine(obj => Object.values(obj).some(v => v !== undefined))`; en Go, chequeo manual tras `Decode`. Mismo concepto: la regla "al menos uno" no cabe en un tipo — es un validador cruzado.
Mini reto. Email como campo validado: `z.string().email()` / `EmailStr` (Pydantic, requiere `email-validator`) / `validate:"email"`. Sobre `"a@b"`: Zod y Pydantic lo rechazan (exigen dominio con punto), validator de Go lo acepta por defecto — la RFC 5322 permite dominios sin punto. Lección: "email válido" es una decisión de negocio, no solo de formato — muchos equipos validan formato básico y confirman con un email de verificación (Cap. 14).

13.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: OpenAPI

OpenAPI es el formato estándar para describir una API: cada endpoint, sus parámetros, sus respuestas, sus errores. De ese archivo salen la documentación, los clientes generados y los tests de contrato.

Explicación de: roles / RBAC

Los **roles** agrupan permisos: *viewer* lee, *editor* escribe, *admin* gestiona. RBAC = el permiso depende del rol que tienes *en ese recurso* — puedes ser admin de un equipo y viewer de otro.

Explicación de: constraint / restricción

Una **constraint** es una regla que la BD hace cumplir: `NOT NULL`, `UNIQUE`, `CHECK (price > 0)`, `FOREIGN KEY`. Es la última línea de defensa — el código puede tener bugs; la constraint no perdona.

Explicación de: backup / respaldo

Un **backup** es una copia de los datos en otro lugar, hecha seguido y *probada* (un backup que nunca se restauró no es backup, es esperanza). Regla 3-2-1: 3 copias, 2 medios, 1 fuera del sitio.

13.12. Lo que deberías saber hacer ahora

- ☐ Explicar por qué validar en el servidor aunque el frontend ya valide
- ☐ Definir un schema de entrada declarativo en tu stack
- ☐ Elegir entre 400 y 422 y devolver errores estructurados por campo
- ☐ Explicar por qué los tipos TypeScript no protegen en runtime — y qué sí lo hace en cada lenguaje

14 5. Necesitamos controlar lo que sale

Si devuelves el objeto tal cual, filtras datos

> **En una frase**
>
> Vas a aprender a **definir qué campos salen** de tu API: un contrato de salida (`TaskPublic`) actúa como aduana — lo no declarado no sale, así tu `password_hash` futuro nunca se filtre por accidente.

14.1. El problema

La tarea en memoria ahora tiene más campos de los que el cliente debe ver:

> **Explicación de: memoria (RAM)**
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

```
Task interno = {
  id, title, done,
  owner_id,          // quién la creó - el cliente no necesita saberlo
  internal_notes,    // notas del equipo
  created_at, updated_at,
  deleted_at,        // soft-delete interno
}
```

La tentación universal del que viene de frontend: `res.json(task)` y ya — total, "es mi objeto". El problema: **todo lo que devuelves queda grabado en el contrato**. Si hoy sale `owner_id`, mañana alguien escribe frontend contra ese campo, y cuando quieras quitarlo rompes clientes. Peor: si el objeto interno contiene `password_hash` (Cap. 14), acabas de publicar tu base de datos.

> **Explicación de: objeto / dict / struct**
>
> Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con obje-

tos/`interface`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

14.2. Cómo lo resuelve un equipo

La regla es el espejo del capítulo anterior:

> **Nada sale del servidor sin pasar por el contrato de salida.**

El equipo define explícitamente **qué campos son públicos** — el `TaskPublic` — y el serializador filtra todo lo demás. Así, aunque el servicio devuelva el objeto interno completo (con campos privados), la capa de salida actúa como aduana: solo sale lo declarado.

Ejemplo concreto de la frontera — mismo objeto interno, dos salidas:

```
// Objeto interno (nunca sale)
{ "id": 7, "title": "deploy", "done": false,
  "owner_id": 3, "internal_notes": "rev 2 del plan",
  "deleted_at": null, "created_at": "2026-09-22T10:00:00Z" }

// GET /tasks/7  → TaskPublic (detalle)
{ "id": 7, "title": "deploy", "done": false,
  "created_at": "2026-09-22T10:00:00Z" }

// GET /tasks    → TaskSummary (lista)
{ "id": 7, "title": "deploy", "done": false }
```

Hay una consecuencia arquitectónica importante: llegamos a **tres formas distintas del mismo concepto**:

Forma	Rol	Vive en
`TaskCreate`	lo que entra	schema de validación (Cap. 4)
`Task` (interno)	lo que se guarda	modelo de datos (Cap. 10)
`TaskPublic`	lo que sale	contrato de respuesta

Un junior mezcla las tres en una sola clase "Task" — y paga el precio cada vez que el contrato debe cambiar sin tocar la base de datos (o viceversa).

> Explicación de: clase / struct
>
> Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

14.3. Conceptos nuevos

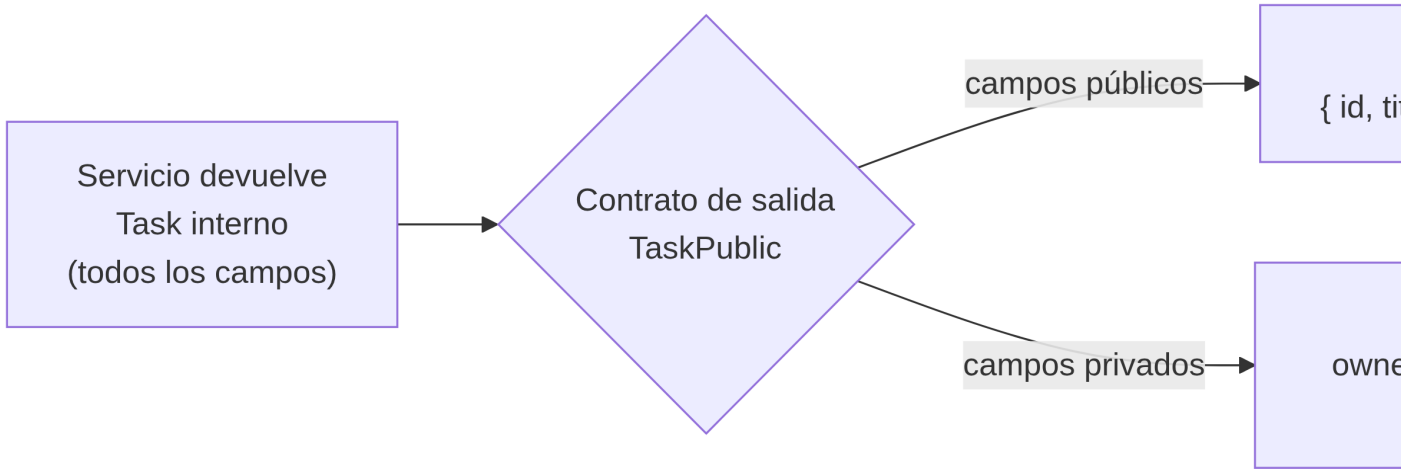
- U **Serialización**: convertir el objeto interno a JSON de respuesta — y decidir qué campos cruzan la frontera.
- U **DTO/contrato de salida**: `TaskPublic` declara la forma pública; el filtrado no es un `delete obj.secret` manual sino una *whitelist* declarada.
- Py `response_model`: FastAPI filtra el objeto ORM/dominio contra el schema declarado — `password_hash` nunca sale aunque el servicio lo devuelva.

> Explicación de: esquema (schema)
>
> El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

> Explicación de: ORM
>
> Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

- TS Mapeo explícito o schema de salida con Zod: la forma pública se construye a mano (`pick`, función `toPublic`).
- Go Structs públicos con tags `json:` — el filtro *es* el tipo: una struct `TaskPublic` distinta, convertida explícitamente.
- U **Serialización no trivial**: fechas, UUIDs, enums, `Decimal` — qué se convierte solo a JSON y qué necesita ayuda en cada lenguaje.

14.4. La explicación visual



La dirección importa: es **whitelist** (“sale solo lo declarado”), no blacklist (“quito lo que sé que es secreto”). La blacklist falla el día que alguien agrega un campo nuevo y olvida filtrarlo; la whitelist no filtra nada que no se haya declarado público.

14.5. Implementación

14.5.1. TypeScript

14.5.1.1. Fundamentos (a mano)

```
// La tentación: devolver el objeto tal cual
app.get("/tasks/:id", (req, res) => {
  const task = findTask(Number(req.params.id));
  res.json(task);  // ← owner_id e internal_notes salen gratis
});

// El "arreglo" ingenuo - blacklist manual:
const { owner_id, internal_notes, deleted_at, ...pub } = task;
res.json(pub);     // funciona… hasta que llega el próximo campo privado
```

14.5.1.2. Explícito (canónico del libro)

```
// La forma pública se construye explícitamente - función o pick de zod
interface Task {
  id: number;
  title: string;
```

```
  done: boolean;
  owner_id: number;        // interno
  internal_notes: string;  // interno
  created_at: string;
}

interface TaskPublic {
  id: number;
  title: string;
  done: boolean;
  created_at: string;
}

function toPublic(t: Task): TaskPublic {
  return {
    id: t.id,
    title: t.title,
    done: t.done,
    created_at: t.created_at,
  };
}

app.get("/tasks/:id", (req, res) => {
  const task = findTask(Number(req.params.id));
  if (!task) return res.status(404).json({ error: "task not found" });
  res.json(toPublic(task));   // la aduana, antes de salir
});
```

14.5.1.3. class-transformer / NestJS serializers

```
// class-transformer - la whitelist son decoradores sobre la clase
class TaskPublic {
  @Expose() id: number;
  @Expose() title: string;
  @Exclude() owner_id: number;   // o simplemente: no declararlo
}
res.json(plainToInstance(TaskPublic, task));
```

Qué te cuesta: NestJS lo activa global con `ClassSerializerInterceptor` — la aduana automática estilo FastAPI, pero pagas el framework entero. Con Express puro, la función `toPublic` honesta es lo que verás en la mayoría de codebases.

14.5.2. Python

14.5.2.1. Fundamentos (dict a mano)

```python
@app.get("/tasks/{task_id}")
def get_task(task_id: int):
    task = find_task(task_id)
    # whitelist manual - funciona pero se desincroniza del contrato
    return {"id": task.id, "title": task.title, "done": task.done}
```

14.5.2.2. response_model (canónico)

```python
class TaskPublic(BaseModel):
    id: int
    title: str
    done: bool
    created_at: datetime

@app.get("/tasks/{task_id}", response_model=TaskPublic)
def get_task(task_id: int):
    task = find_task(task_id)   # devuelve el objeto interno completo
    if not task:
        raise HTTPException(404, "task not found")
    return task   # FastAPI lo filtra contra TaskPublic antes de responder
```

`response_model` es la aduana: aunque `task` tenga `owner_id` y `internal_notes`, el JSON de salida solo lleva los campos declarados.

14.5.2.3. DRF serializers / Marshmallow

```python
# Django REST Framework - el serializer ES la clase que filtra
class TaskPublicSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField()
    done = serializers.BooleanField()

# return Response(TaskPublicSerializer(task).data)
```

Qué te cuesta: DRF/Marshmallow son la generación anterior — mismo concepto de whitelist declarada, más verboso y sin type hints gratis. El patrón es idéntico: declarar lo público, filtrar lo demás.

14.5.3. Go

14.5.3.1. Structs + tags (canónico)

```
// Dos structs: la interna y la pública. El filtro ES el tipo.
type Task struct {
    ID            int    `json:"id"`
    Title         string `json:"title"`
    Done          bool   `json:"done"`
    OwnerID       int    `json:"-"`          // nunca se serializa
    InternalNotes string `json:"-"`
    CreatedAt     string `json:"created_at"`
}

// Alternativa explícita cuando las formas divergen de verdad:
type TaskPublic struct {
    ID        int    `json:"id"`
    Title     string `json:"title"`
    Done      bool   `json:"done"`
    CreatedAt string `json:"created_at"`
}

func toPublic(t Task) TaskPublic {
    return TaskPublic{ID: t.ID, Title: t.Title, Done: t.Done, CreatedAt: t.CreatedAt}
}
```

`json:"-"` es el caso simple ("este campo jamás sale"); una struct separada es el caso general ("la forma pública difiere de la interna").

14.5.3.2. Alternativas de encoder

```
// jsoniter / goccy - drop-in más rápido, mismos tags:
// import jsoniter "github.com/json-iterator/go"
// json.NewEncoder(w) → jsoniter.ConfigFastest.NewEncoder(w)
```

Qué te cuesta: los encoders alternativos solo cambian *velocidad*, no el modelo — la frontera sigue siendo la struct con tags. En Go no hay "serializers mágicos" populares: el tipo *es* el contrato, y eso es una decisión de diseño del lenguaje.

Prompt listo para tu AI

```
Crea un contrato de salida para mi API de tareas en
[Express+TypeScript / FastAPI+Python / Go]. El objeto interno Task tiene
campos privados (owner_id, internal_notes, deleted_at) que NUNCA deben
salir en el JSON. Requisitos: forma pública TaskPublic con solo
{id, title, done, created_at}, whitelist declarativa (no blacklist),
aplicada a GET /tasks/{id} y a la lista GET /tasks con una forma
resumida TaskSummary. Muéstrame el JSON que sale vs el objeto interno
para comprobar el filtrado.
```

14.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: "qué sale" es una whitelist declarada en los tres — FastAPI la pone en la firma (`response_model`), Express te deja construirla a mano (`toPublic`), Go la graba en el tipo (`json:"-"`). El detalle:

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Detalle: tres formas de aduana

- **FastAPI** hace la frontera *declarativa y automática*: `response_model` es tan visible en la firma del endpoint que es difícil olvidarlo — y de regalo genera el schema en OpenAPI (la documentación).
- **Express/Node** no tiene esa capa: la disciplina es 100 % tuya. La función `toPublic` es el patrón estándar; algunos equipos usan `class- transformer` o Zod `.pick()` para lo mismo.
- **Go** hace la frontera *estructural*: como los tags `json` son parte del tipo, "qué sale" lo decide el compilador — pero por eso mismo necesitas una struct distinta por forma pública. Verborrea a cambio de que sea imposible filtrar por accidente.

La lección universal: **entrada, almacenamiento y salida son tres contratos distintos**. Cualquier atajo que los fusione ("devuelvo el objeto de BD directo") es un agujero de seguridad esperando un campo nuevo.

Otras cimentaciones: ¿siempre un contrato de salida separado?

Esta decisión es estructural: la frontera "qué sale" debe existir — es carga del edificio, no decoración. Lo intercambiable es *dónde* vive:

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Schema/DTO en código (este capítulo)	`TaskPublic`/`response_model` declarado	Una declaración extra por forma	El default — explícito y testeable
Selección en la query (SQL `SELECT id, title`)	La BD nunca devuelve lo privado	No protege si otro código usa la fila completa	Excelente *complemento*, insuficiente solo
Codegen desde OpenAPI	El contrato genera los tipos	Toolchain + codegen en CI	Contratos publicados a terceros
GraphQL (cliente pide campos)	El filtro lo hace la query del cliente	Todo el stack GraphQL; auth por campo	Cuando los clientes necesitan formas muy variables

14.7. Errores comunes

- **`res.json(row)` directo del ORM/driver**: publicas el esquema interno — incluyendo campos que aún no existen pero existirán (el futuro `password_hash`).
- **Blacklist en vez de whitelist**: `delete task.internal_notes` funciona hasta que llega `internal_secret_2`.
- **Un solo modelo para todo**: `Task` usado como entrada, fila de BD y respuesta — imposible de evolucionar sin romper clientes.
- **Olvidar los endpoints de lista**: `GET /tasks` también debe pasar por la aduana — filtrar mil objetos es el mismo contrato, solo en plural.

14.8. Buenas prácticas

- **Nombra la frontera**: `TaskPublic`, `toPublic()`, `response_model` — que sea visible en el código que hay una frontera.
- **El contrato de salida define la documentación**: si usas FastAPI, tu `response_model` *es* el OpenAPI; si no, mantén la docu alineada a mano.
- **Campos internos prefijados o segregados** (`internal_*`) — que el nombre mismo avise.
- **Versionar la forma, no el campo**: si la salida cambia de forma, es una decisión de contrato (¿`v2`? ¿campo nuevo opcional?), no un refactor invisible.

14.9. Ejercicio

1. Agrega a tu `Task` interno los campos `owner_id`, `internal_notes`, `deleted_at`; crea `TaskPublic` y verifica que `GET /tasks/{id}` no los filtra.
2. Haz que `GET /tasks` (lista) devuelva una forma *resumida* (`TaskSummary`: solo `id`, `title`, `done`) y `GET /tasks/{id}` la completa. Dos contratos de salida, un mismo recurso.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

14.10. Mini reto

Devuelve la misma tarea en dos formas según *quién pregunta*: el owner ve `internal_notes`, los demás no. ¿Dónde debería vivir esa decisión — en el serializador o antes? Diseña la solución; la implementamos en el capítulo de autorización.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Soluciones

Ejercicio 1. `TaskPublic` con `id, title, done, created_at` filtra los tres campos internos — `GET /tasks/7` responde solo lo público.
Ejercicio 2. Dos contratos: `TaskSummary {id, title, done}` para `GET /tasks` y `TaskPublic` completo para `GET /tasks/{id}`. La razón de diseño: la lista viaja miles de veces y se pinta en tarjetas — traer campos de detalle es peso muerto y exposición innecesaria.
Mini reto. La decisión "quién pregunta" es *autorización*, no serialización: vive en el servicio/handler (con el usuario del request ya resuelto — Cap. 16–17), que elige el contrato `TaskOwnerView` vs `TaskPublic`. El serializador solo filtra; **decidir qué forma corresponde a cada quién es lógica de negocio**. Si la decisión viviera en el serializador, éste tendría que conocer auth — capas invertidas.

14.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: booleano

Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

14.12. Lo que deberías saber hacer ahora

- ☐ Explicar por qué entrada, almacenamiento y salida son tres contratos
- ☐ Implementar una frontera de salida (whitelist) en tu stack
- ☐ Decir qué riesgo concreto evita `response_model`/equivalente
- ☐ Diseñar formas de salida distintas para lista vs detalle

15 6. Necesitamos fallar correctamente

Tres lenguajes, tres filosofías de error

> **En una frase**
>
> Vas a aprender a **fallar bien**: un formato de error único para el cliente y una frontera que traduce errores de negocio a HTTP — y de paso verás la diferencia más real entre los tres lenguajes: excepciones vs errores como valores.

15.1. El problema

`GET /tasks/99` cuando solo existen 5 tareas. ¿Qué devuelves?

> **Explicación de: return**
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

- ¿`null`? — el frontend hace `task.title` y explota.
- ¿`200 {"error": "not found"}`? — el `res.ok` de `fetch` dice *true* y tu React pinta una tarea fantasma.
- ¿Dejar que `tasks[99]` reviente con un stack trace? — el cliente ve las tripas de tu servidor (y un atacante también).

Fallar es inevitable. **Fallar bien es diseño**: el error es parte del contrato, igual que la respuesta exitosa. Y aquí nos encontramos con la primera divergencia filosófica real entre los tres lenguajes.

15.2. Cómo lo resuelve un equipo

Un equipo maduro decide tres cosas sobre los errores:

1. **La forma**: un formato único para todos los errores — el frontend escribe *una* función `handleError(res)` y funciona para siempre:

   ```
   { "error": { "code": "TASK_NOT_FOUND", "message": "Task 99 does not exist" } }
   ```

   Y esto es lo que **nunca** debe ver el cliente — pero sí tu log:

> Explicación de: log / logging estructurado
>
> Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

```
Cliente ve:    500 { "error": { "code": "INTERNAL", "message": "unexpected error" } }
Log guarda:    Traceback … line 42, in get_task … sqlite3.OperationalError:
               database is locked  (request_id=req_8f3a)
```

2. **La frontera**: los errores de negocio ("la tarea no existe") viajan como valores normales hasta la capa de transporte, que los convierte en HTTP. El servicio no sabe de status codes; el router no sabe de negocio.
3. **Los códigos estables**: `TASK_NOT_FOUND` es contrato — el mensaje puede cambiar, el código no. El frontend programa contra el código.

15.3. Conceptos nuevos

- U **Error de dominio vs error de sistema**: "tarea no existe" (esperado, 404) vs "se cayó la BD" (inesperado, 500). Se manejan distinto y se reportan distinto.

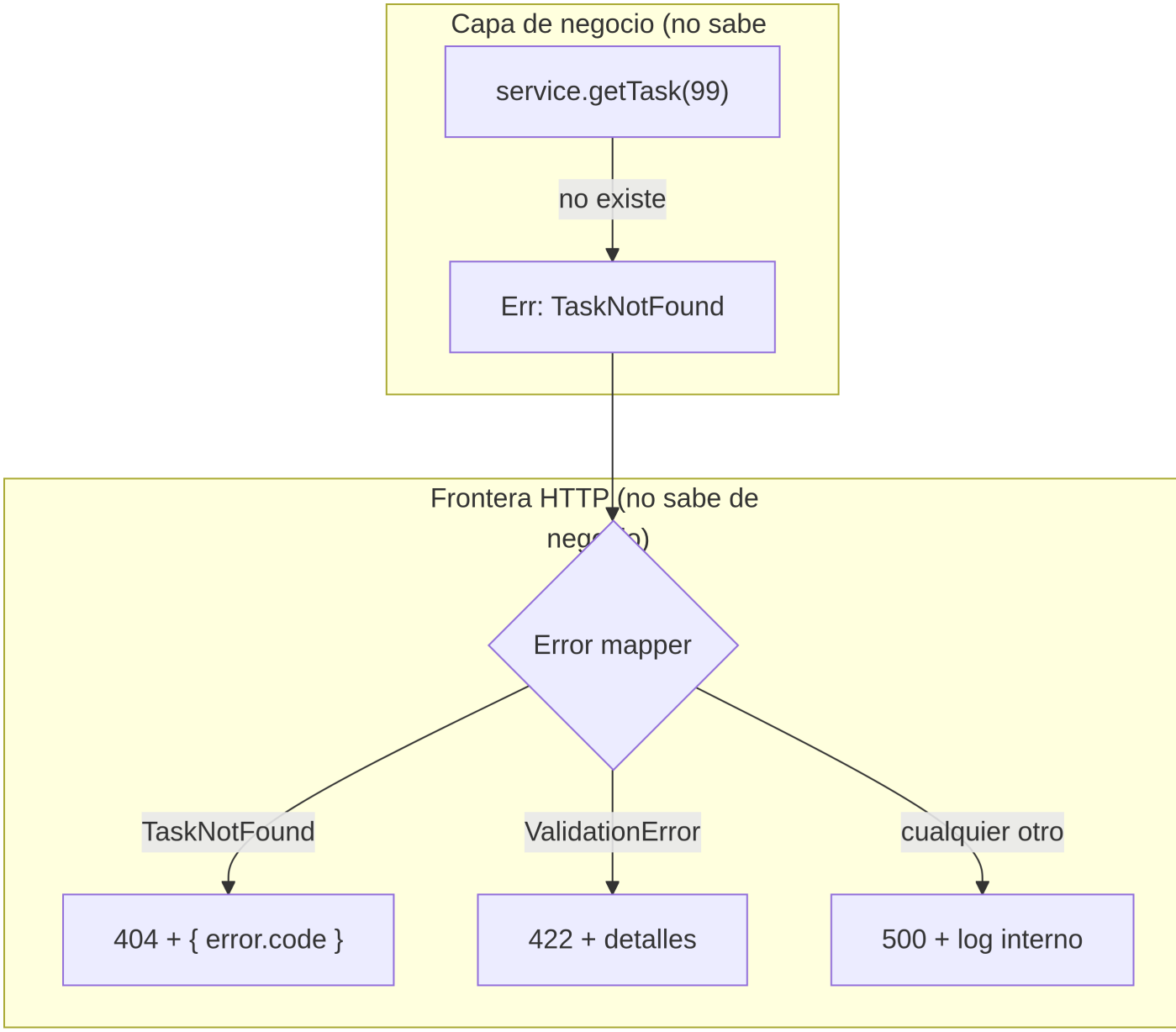
> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

- TS **Excepciones**: `throw`/`catch`, más el *error middleware* de Express como frontera HTTP.
- Py **Excepciones**: `raise`/`except`, `HTTPException` y los exception handlers globales.
- Go **Error como valor**: no hay excepciones — las funciones devuelven `(result, error)` y tú decides. El modelo más explícito, y el que mejor enseña el concepto.

> Explicación de: función
>
> Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

- U **Jerarquía de errores**: `DomainError → NotFoundError → TaskNotFoundError` — clasificar para responder.

15.4. La explicación visual



La línea divisoria es la regla de oro: **los errores nacen en el dominio como datos del dominio, y se traducen a HTTP en el borde**. Si tu servicio devuelve `HTTPException` o `res.status()`, acabas de acoplar la lógica al transporte — y mañana no podrás reusarla desde un WebSocket o un test.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: WebSocket

WebSocket es una conexión que queda *viva*: en vez de preguntar- responder-cerrar como HTTP, ambos pueden hablar cuando quieran. Es como el servidor puede *empujar* — por eso sirve para chat y colaboración en vivo.

15.5. Implementación

15.5.1. TypeScript

15.5.1.1. Excepciones + middleware (canónico)

```
// errores de dominio
class DomainError extends Error {}
class NotFoundError extends DomainError {
  constructor(public code: string, msg: string) { super(msg); }
}

// servicio: lanza errores de dominio, no sabe de HTTP
function getTask(id: number): Task {
  const t = tasks.find((t) => t.id === id);
  if (!t) throw new NotFoundError("TASK_NOT_FOUND", `Task ${id} does not exist`);
  return t;
}

// handler: delgado - la traducción a HTTP vive en el middleware de error
app.get("/tasks/:id", (req, res, next) => {
  try {
    res.json(toPublic(getTask(Number(req.params.id))));
  } catch (e) { next(e); }
});

// la frontera: un solo lugar que traduce errores → HTTP
app.use((err: Error, _req, res, _next) => {
  if (err instanceof NotFoundError) {
    return res.status(404).json({ error: { code: err.code, message: err.message } });
  }
  console.error(err); // el 500 se loguea completo, se devuelve poco
  res.status(500).json({ error: { code: "INTERNAL", message: "unexpected error" } });
});
```

15.5.1.2. Result types (neverthrow)

```ts
// neverthrow - errores como VALORES (estilo Go/Rust) dentro de TS
import { Result, ok, err } from "neverthrow";

function getTask(id: number): Result<Task, NotFoundError> {
  const t = tasks.find((t) => t.id === id);
  return t ? ok(t) : err(new NotFoundError("TASK_NOT_FOUND", "…"));
}

// en el handler: result.match(t => res.json(t), e => next(e))
```

Qué te cuesta: `Result` hace el error visible en la firma (como Go), pero es una convención de librería — tu equipo entero debe adoptarla. Interesante saber que existe: es el modelo de Go empaquetado para TS.

15.5.2. Python

15.5.2.1. Excepciones + handlers (canónico)

```py
# errores de dominio
class DomainError(Exception): ...
class NotFoundError(DomainError):
    def __init__(self, code: str, message: str):
        self.code, self.message = code, message

# servicio: lanza errores de dominio
def get_task(task_id: int) -> Task:
    for t in tasks:
        if t.id == task_id:
            return t
    raise NotFoundError("TASK_NOT_FOUND", f"Task {task_id} does not exist")

# la frontera: exception handler global - el handler ni try/except necesita
@app.exception_handler(NotFoundError)
def not_found_handler(_req, exc: NotFoundError):
    return JSONResponse(
        status_code=404,
        content={"error": {"code": exc.code, "message": exc.message}},
    )

@app.get("/tasks/{task_id}", response_model=TaskPublic)
def get_task_endpoint(task_id: int):
    return get_task(task_id)   # si falla, el handler global traduce
```

15.5.2.2. Result pattern / RFC 9457

```
# La alternativa "errores como valores" en Python (librería result/returns):
from result import Result, Ok, Err

def get_task(task_id: int) -> Result[Task, NotFoundError]:
    ...
    return Err(NotFoundError("TASK_NOT_FOUND", "…"))
```

Qué te cuesta: `result/returns` traen el modelo explícito de Go a Python — pero pelean contra la idiomática del lenguaje (todo el mundo espera excepciones). Y en el *formato* del error existe un estándar: **RFC 9457** `application/problem+json` — lo verás en APIs grandes.

15.5.3. Go

15.5.3.1. error como valor (stdlib — canónico)

```
// errores de dominio como valores - no hay excepciones
var ErrTaskNotFound = errors.New("task not found")

// servicio: devuelve (resultado, error) - explícito siempre
func getTask(id int) (Task, error) {
    for _, t := range tasks {
        if t.ID == id {
            return t, nil
        }
    }
    return Task{}, fmt.Errorf("%w: %d", ErrTaskNotFound, id)
}

// handler: la traducción es visible en cada endpoint (o un helper writeError)
func getTaskHandler(w http.ResponseWriter, r *http.Request) {
    id, _ := strconv.Atoi(r.PathValue("id"))
    t, err := getTask(id)
    if err != nil {
        if errors.Is(err, ErrTaskNotFound) {
            writeError(w, 404, "TASK_NOT_FOUND", err.Error())
            return
        }
        log.Println("internal:", err)
        writeError(w, 500, "INTERNAL", "unexpected error")
        return
    }
    json.NewEncoder(w).Encode(toPublic(t))
}
```

15.5.3.2. panic / recover (y por qué no)

```
// Go SÍ tiene "excepciones": panic. Pero es para bugs irrecuperables,
// no para errores de negocio - un panic en un handler derriba el request.
// net/http hace recover por request, pero usarlo como flujo de control
// es el equivalente a "throw para todo": se considera malpractice.
```

Qué te cuesta: `panic` existe pero su rol es distinto — "esto no debería pasar jamás" (índice fuera de rango, invariante roto). Los errores *esperados* viajan como valores. El modelo de Go te obliga a esta distinción que otros lenguajes dejan difusa.

Prompt listo para tu AI

```
Implementa el manejo de errores de mi API en
[Express+TypeScript / FastAPI+Python / Go]. Requisitos: jerarquía de
errores de dominio (DomainError → NotFoundError con code estable tipo
TASK_NOT_FOUND), el servicio lanza errores de dominio sin conocer HTTP,
UN solo lugar traduce errores a HTTP (middleware / exception handler /
helper), formato de respuesta { error: { code, message } } consistente,
y el 500 loguea el stack completo pero devuelve solo {code: "INTERNAL"}
- nunca el trace al cliente.
```

15.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: TS y Python usan excepciones (el error *salta* a la frontera); Go los devuelve como valores (la firma *grita* que puede fallar). Mismo objetivo, filosofías opuestas. El detalle:

Detalle: excepciones vs errores como valor

- **TypeScript y Python** comparten el modelo de excepciones: el error *salta* de la capa profunda a la frontera sin que los niveles intermedios lo mencionen. Cómodo — pero la firma de `get_task()` **no dice** que puede fallar; tienes que saberlo.
- **Go** rechaza las excepciones a propósito: `getTask(id)` devuelve `(Task, error)` y la firma *grita* que puede fallar. Cada llamada exige `if err != nil` — verborrea real, pero también **cero errores invisibles viajando por capas**. Es la diferencia filosófica más honesta del libro: ¿prefieres errores que emergen solos o errores que declaras a mano?

`errors.Is`/`errors.As` merece una mirada: envolver con `%w` crea una cadena (`"task not found: 99"`) que el borde puede interrogar sin parsear strings — el equivalente Go a la jerarquía de clases de error.

Otras cimentaciones: ¿siempre formato de error propio?

Esta decisión es material: devolver *un* formato consistente es estructural (el frontend programa contra él); cuál formato es intercambiable.

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Envelope propio (este libro)	`{ error: { code, message, details } }`	Diseñarlo y mantenerlo tú	APIs internas/propias — el default pragmático
RFC 9457 `application/problem+json`	Estándar: `type`, `title`, `status`, `detail`	Media type + campos fijos	APIs públicas, interoperabilidad
Errores por campo estilo formulario	`{ fields: { title: ["required"] } }`	Menos expresivo fuera de forms	APIs que solo sirven formularios
GraphQL errors array	`{ errors: [{ message, extensions }] }`	Acoplado a GraphQL	Si ya eres GraphQL

15.7. Errores comunes

- **El 200 mentiroso**: devolver `{"error": ...}` con status 200 (ya sabes por qué `res.ok` lo empeora).
- **`HTTPException` dentro del servicio** (Py) o `res.status()` en la lógica (TS): acoplas dominio a transporte — el mismo error no sirve para un CLI, un test o un WebSocket.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

- **`try/except` que traga todo**: `except Exception: pass` convierte un bug en un misterio. Captura lo que esperas; deja escapar lo demás.
- **Devolver el stack trace al cliente**: el 500 se loguea entero en el servidor y se devuelve como `{ "code": "INTERNAL" }` — nunca las tripas.
- **En Go, ignorar el error**: `t, _ := getTask(id)` compila y es una bomba. `errcheck` en CI lo caza.

15.8. Buenas prácticas

- **Un formato de error, para siempre**: `{ error: { code, message, details? } }` desde el Cap. 6 — y nunca cambiarlo.
- **Códigos estables en SCREAMING_SNAKE**: el frontend los switchea.
- **Los errores esperados no se loguean como ERROR**: un 404 por id inexistente es INFO/WARN — el log de ERROR es para lo que requiere despertar a alguien.

- **El middleware/handler de errores se registra una vez** y cubre todos los endpoints — consistente por construcción.

15.9. Ejercicio

1. Implementa la jerarquía `DomainError → NotFoundError` en tu stack y el mapper de la frontera.
2. Agrega `ConflictError` (409) para "no se puede borrar una tarea con subtareas pendientes" — y su mapeo.
3. Provoca un 500 a propósito y verifica: el cliente ve solo `{ code: "INTERNAL" }` y el log tiene el stack completo.

15.10. Mini reto

Simula un error *inesperado* en plena creación de tarea (por ejemplo, la "BD" en memoria lanzando una excepción rara). Verifica que el cliente nunca ve el stack trace y que el log del servidor sí lo tiene. Bonus: ¿cómo harías que el error devuelva un `request_id` que el usuario pueda reportar? (El Cap. 23 lo implementa de verdad.)

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Soluciones

Ejercicio 1. `NotFoundError` con `code` y `message` + mapper: TS en el error middleware (`err instanceof NotFoundError → 404`), Py en `@app.exception_handler`, Go con `errors.Is(err, ErrTaskNotFound) → writeError(w, 404, …)`.
Ejercicio 2. `ConflictError` mapea a 409 igual que NotFound a 404 — misma frontera, otra entrada del mapper. La regla de negocio ("no borrar con subtareas") vive en el servicio, no en el handler.
Ejercicio 3. Provocar el 500 (p. ej. `raise RuntimeError("boom")` en el servicio): el cliente recibe solo `{"error":{"code":"INTERNAL",…}}` y el log muestra el stack completo — eso es lo correcto.
Mini reto. El `request_id` se genera en un middleware al inicio del request (UUID corto), se guarda en el contexto (`res.locals` / `context.Context` / request state), se incluye en TODOS los logs del request y se devuelve en el body del error. El usuario reporta "me dio INTERNAL req_8f3a" y tú buscas ese id en los logs — Cap. 23 lo implementa.

15.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: estado (state)

El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

15.12. Lo que deberías saber hacer ahora

- ☐ Separar errores de dominio de errores de transporte — y explicar por qué
- ☐ Implementar un mapper de errores→HTTP en un solo lugar
- ☐ Explicar la diferencia filosófica entre excepciones y errores como valor — y dar un argumento a favor de cada modelo

Explicación de: parámetro / argumento

El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

- ☐ Garantizar que un 500 nunca filtra el stack trace al cliente

16 7. Necesitamos que esto quepa en la cabeza de otra persona

Un archivo de 800 líneas con 30 endpoints no es un proyecto

> **En una frase**
>
> Vas a **organizar el código en capas** — transporte, negocio, datos, contratos — para que cada archivo haga una sola cosa y la llegada de la base de datos (Parte 2) no rompa nada. Es el mismo `components/services/ hooks` que ya usas en frontend, pero del lado servidor.

16.1. El problema

Taskflow ya tiene 12 endpoints, schemas, errores de dominio, la lista en memoria… todo en un archivo que empieza a dar miedo. Cuando llegue la base de datos (Parte 2), la autenticación (Parte 3) y los tests (Parte 6), ese archivo será inmantenible.

> **Explicación de: array / lista / slice**
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> **Explicación de: memoria (RAM)**
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

> **Explicación de: autenticación**
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Pero el problema real no es el tamaño — es la **mezcla**: en la misma función convive "leer el path param" (HTTP), "la tarea no existe" (negocio) y "meterla en la lista" (datos). Si mañana el storage cambia de

lista a Postgres, tocas *cada* handler. Si quieres testear la lógica sin levantar el servidor, no puedes — la lógica está pegada a HTTP.

16.2. Cómo lo resuelve un equipo

La decisión arquitectónica más común y más útil del backend: **capas**. Cada capa tiene *un* trabajo y solo habla con su vecina:

```
routes/handlers   →  HTTP  lenguaje: verbos,
                     params, status codes

services          →  negocio: reglas, errores
                     de dominio

storage/models    →  datos: queries, persistencia

schemas/contracts →  contratos: lo que entra
                     y lo que sale (Cap. 4 y 5)
```

La regla de oro que hace funcionar todo: **los handlers son tontos**. `createTask` en la ruta solo traduce HTTP → llamada al servicio → HTTP. La regla "no borres una tarea con subtareas" vive en el servicio — así se puede llamar desde otro endpoint, un test o un WebSocket sin duplicar.

El mismo endpoint, antes y después:

```python
# ANTES - handler gordo: transporte + negocio + datos mezclados
@app.delete("/tasks/{task_id}")
def delete(task_id: int):
    task = next((t for t in tasks if t.id == task_id), None)
    if not task: raise HTTPException(404)
    if any(s for s in subtasks if s.task_id == task_id):
        raise HTTPException(409, "has subtasks")      # regla en la ruta
    tasks.remove(task)

# DESPUÉS - handler tonto: la regla vive en el servicio
@router.delete("/{task_id}", status_code=204)
def delete(task_id: int):
    svc.delete_task(task_id)    # TaskNotFound/Conflict salen del servicio
```

La prueba de que lo hiciste bien llega en la Parte 6: los tests de servicio no necesitan ni levantar HTTP.

16.3. Conceptos nuevos

- U **Arquitectura por capas**: transporte / negocio / datos / contratos — y por qué cada capa ignora a las demás.

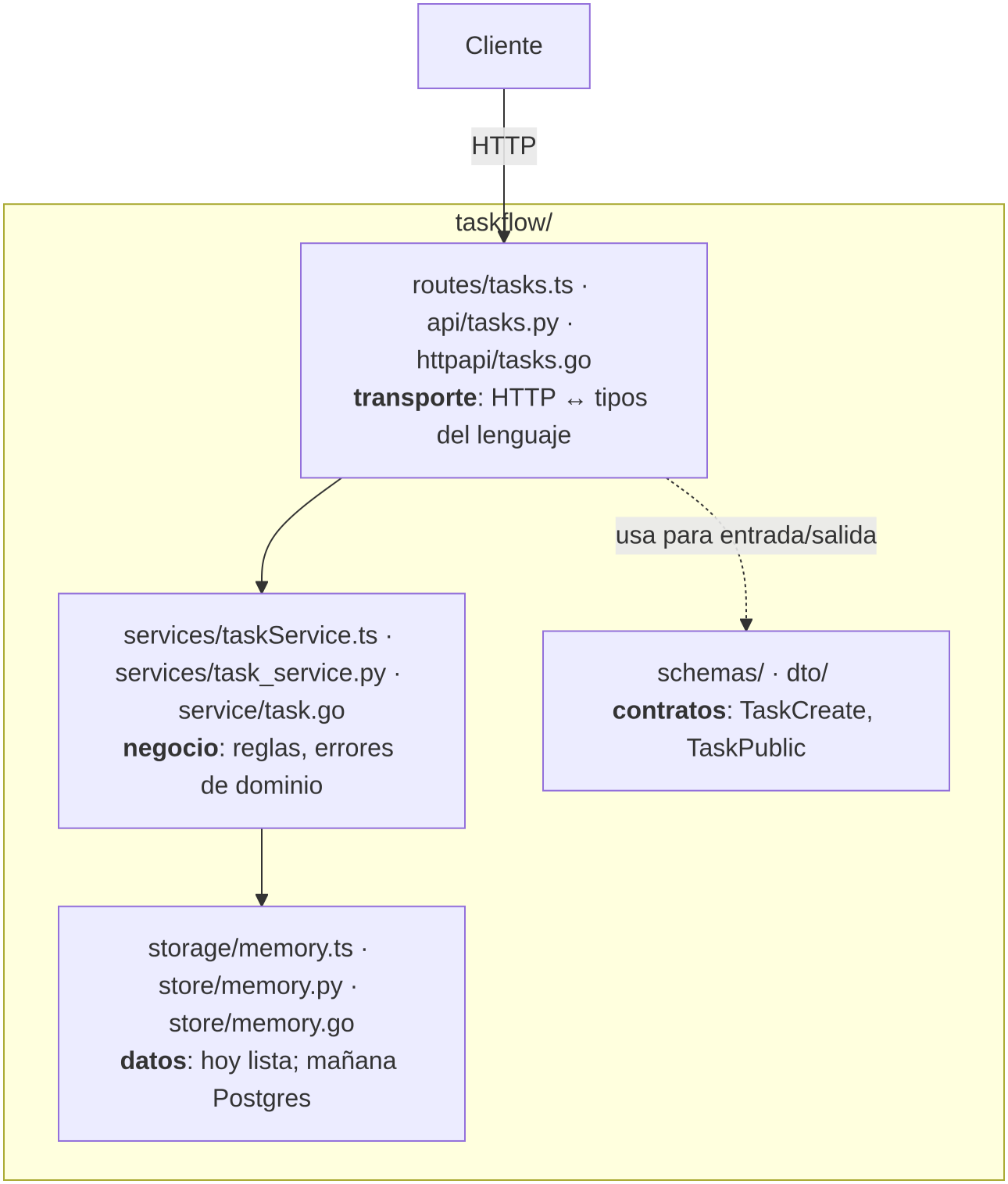
- TS **ES modules + routers de Express**: `import/export`, `express.Router()` para separar rutas por dominio.
- Py **Módulos y paquetes**: `import`, `from x import y`, `__init__.py` — el equivalente a ES modules con sus trampas; `APIRouter` para agrupar endpoints.
- Go **Packages**: un directorio = un package; nombres cortos; `internal/` como frontera que el compilador

> Explicación de: compilador / transpilador
>
> Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

hace cumplir. - U **Lógica reusable pre-handler**: middleware (Express) vs `Depends` (FastAPI) vs `func(http.Handler) http.Handler` (Go) — el mismo patrón, tres cantidades de magia. Primer contacto; el Cap. 16 lo exprime.

> Explicación de: función
>
> Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

16.4. La explicación visual



Cuando llegue Postgres (Cap. 10), solo `storage/` cambia. Cuando llegue auth (Cap. 16), se cuelga una dependencia antes del handler. **Las features nuevas se agregan sin reescribir** — esa es la definición operativa de "buena arquitectura".

> Explicación de: dependencia
>
> Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

16.5. Implementación

El refactor del Cap. 3–6 a capas, en los tres stacks:

16.5.1. TypeScript

16.5.1.1. Capas manuales (canónico)

```
src/
  index.ts              # ensambla: app + json + routers + error middleware
  routes/tasks.ts       # transporte: verbos, params, status
  services/tasks.ts     # negocio: reglas, DomainErrors
  storage/memory.ts     # datos: la lista (mañana: Postgres)
  schemas/tasks.ts      # contratos: zod + toPublic
```

```
// routes/tasks.ts - el handler tonto
import { Router } from "express";
import * as svc from "../services/tasks.js";
import { createTaskSchema } from "../schemas/tasks.js";

export const tasksRouter = Router();

tasksRouter.post("/", (req, res, next) => {
  const parsed = createTaskSchema.safeParse(req.body);
  if (!parsed.success) return res.status(422).json(/* … */);
  try {
    res.status(201).json(svc.createTask(parsed.data));
  } catch (e) { next(e); }
});
```

```
// index.ts - el ensamblador
import express from "express";
import { tasksRouter } from "./routes/tasks.js";
import { errorMiddleware } from "./errors.js";
```

```
const app = express();
app.use(express.json());
app.use("/tasks", tasksRouter);
app.use(errorMiddleware);
app.listen(8000);
```

16.5.1.2. NestJS (framework con estructura impuesta)

```
// NestJS hace las capas por ti: module → controller → service con DI
@Module({
  controllers: [TasksController],
  providers: [TasksService],   // el servicio se inyecta solo
})
export class TasksModule {}

@Controller("tasks")
export class TasksController {
  constructor(private svc: TasksService) {}  // DI declarativa
  @Post() create(@Body() dto: CreateTaskDto) { return this.svc.create(dto); }
}
```

Qué te cuesta: la arquitectura viene decidida (módulos, decoradores, guards, interceptors — estilo Angular en el backend). Ganas convenciones fuertes y DI gratis; pagas con magia, build más pesado y "the NestJS way". En Express tú eres el arquitecto; en NestJS lo es el framework.

16.5.2. Python

16.5.2.1. Capas manuales (canónico)

```
app/
   main.py            # ensambla: FastAPI + routers + exception handlers
   api/tasks.py       # transporte
   services/tasks.py  # negocio
   store/memory.py    # datos
   schemas/tasks.py   # contratos: TaskCreate, TaskPublic
   errors.py          # DomainError + handlers
```

```
# api/tasks.py - el endpoint delgado
from fastapi import APIRouter
from app.schemas.tasks import TaskCreate, TaskPublic
from app.services import tasks as svc

router = APIRouter(prefix="/tasks")
```

```python
@router.post("", status_code=201, response_model=TaskPublic)
def create_task(payload: TaskCreate):
    return svc.create_task(payload)
```

```python
# main.py - el ensamblador
from fastapi import FastAPI
from app.api.tasks import router as tasks_router
from app.errors import register_handlers

app = FastAPI()
app.include_router(tasks_router)
register_handlers(app)
```

16.5.2.2. Django (apps: la convención impuesta)

```
# Django divide el proyecto en "apps" - capas predefinidas por dominio
taskflow/
   tasks/                  # app por dominio
      models.py           # datos (ORM integrado)
      views.py            # transporte
      serializers.py      # contratos (DRF)
      services.py         # negocio (convención, no impuesta)
```

Qué te cuesta: Django decide la estructura por ti (models/views/serializers por app) — aprendes un sistema completo a cambio de que todo esté resuelto (ORM, admin, auth). FastAPI te deja arquitecto: `api/services/store/schemas` fue *tu* decisión — y por eso este libro la enseña.

16.5.3. Go

16.5.3.1. Packages (canónico)

```
taskflow/
   main.go                  # ensambla: mux + handlers
   internal/
      httpapi/tasks.go     # transporte
      service/tasks.go     # negocio
      store/memory.go      # datos
      domain/task.go       # contratos + errores de dominio
```

```go
// internal/httpapi/tasks.go - el handler delgado
func (h *TaskHandler) create(w http.ResponseWriter, r *http.Request) {
    var in domain.CreateTaskInput
    if err := json.NewDecoder(r.Body).Decode(&in); err != nil {
        writeError(w, 400, "INVALID_JSON", "invalid json")
```

```
        return
    }
    t, err := h.svc.Create(in)
    if err != nil { writeDomainError(w, err); return }
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(domain.ToPublic(t))
}
```

```
// main.go - el ensamblador
func main() {
    store := store.NewMemory()
    svc := service.New(store)      // inyección explícita de dependencias
    h := httpapi.NewTaskHandler(svc)
    mux := http.NewServeMux()
    mux.HandleFunc("POST /tasks", h.create)
    http.ListenAndServe(":8000", mux)
}
```

16.5.3.2. chi / convenciones de la comunidad

```
# La estructura de packages no cambia con el router - chi solo es mejor mux
taskflow/
   cmd/api/main.go          # entrypoint separado (convención cmd/)
   internal/
       httpapi/ · service/ · store/ · domain/
```

Qué te cuesta: en Go los frameworks (Gin, Echo) casi nunca imponen estructura — la comunidad convergió en `cmd/` + `internal/` + packages planos. El "framework de arquitectura" de Go es la convención misma, y `internal/` la hace cumplir el compilador.

Prompt listo para tu AI

```
Refactoriza mi API de tareas (ahora en un solo archivo) a arquitectura
por capas en [Express+TypeScript / FastAPI+Python / Go]. Estructura:
routes/handlers (solo traducen HTTP), services (reglas de negocio,
errores de dominio), storage (persistencia), schemas (contratos de
entrada/salida), y un ensamblador (index/main) que solo conecta piezas.
Requisito: los handlers deben quedar "tontos" - la regla "no borrar
tarea con subtareas" vive en el servicio. Muéstrame el árbol de carpetas
y el código del handler delgado.
```

16.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: la estructura de capas es *idéntica* en los tres; lo que cambia es cómo se conectan las piezas (routers montados, `Depends` declarativo, o wiring explícito en `main.go`). El detalle:

Detalle: cómo se conectan las capas

- **Express**: los routers se montan con `app.use("/tasks", router)` — composición por middleware.
- **FastAPI**: `include_router` + `Depends` — la inyección de dependencias es *declarativa* (la pides en la firma y el framework la resuelve).
- **Go**: la inyección es **manual y explícita**: `service.New(store)`, `NewTaskHandler(svc)` — el wiring se ve en `main.go`. Más verboso, imposible de no entender.

Y un detalle Go que vale oro: `internal/` no es una convención — el **compilador prohíbe** que otros módulos importen paquetes bajo `internal/`. La frontera de capas la hace cumplir la herramienta, no la disciplina.

Otras cimentaciones: ¿siempre arquitectura por capas?

Esta decisión es estructural en espíritu (separar transporte de negocio es innegociable) pero **material** en forma — hay varias cimentaciones legítimas:

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Capas (este libro)	routes → services → storage	Disciplina manual de fronteras	El default pragmático — 80 % de los proyectos
MVC	Model-View-Controller clásico	"View" no aplica bien a APIs JSON	Frameworks que lo traen (Django, Rails)
Clean / Hexagonal	Puertos y adaptadores, dominio al centro	Muchas interfaces y carpetas para un CRUD	Dominios complejos, larga vida, equipos grandes
Feature folders	Todo de "tasks" junto (ruta+servicio+storage)	Acopla capas dentro del feature	Proyectos chicos, microservicios por dominio
Framework impone estructura	NestJS modules, Django apps	Aprendes "la forma del framework"	Cuando el equipo ya usa ese framework

Regla de experiencia: **empieza en capas; migra a hexagonal cuando el dominio lo grite** — no antes. La sobre-ingeniería temprana también es una mala cimentación: pilotes para una casa de un piso.

16.7. Errores comunes

- **El handler gordo**: lógica de negocio dentro de la ruta → imposible de testear sin HTTP, imposible de reusar.
- **Imports cruzados**: `storage` importando `schemas` de HTTP, servicios importando `express`/`fastapi` — la flecha de dependencias siempre apunta *hacia adentro* (datos no conocen transporte).
- **Carpetas por tipo técnico en vez de por capa**: `controllers/`, `utils/`, `misc/` — donde todo acaba siendo cajón desastre.
- **Sobre-ingeniería temprana**: interfaces, factories y repositorios abstractos "por si acaso" en un CRUD de memoria. Las capas que usamos bastan; más abstracción cuando duela de verdad.

16.8. Buenas prácticas

- **La regla del test**: si no puedes testear la regla de negocio sin levantar el servidor, está en la capa equivocada.
- **El ensamblador es plano**: `main`/`index` solo conecta piezas — leerlo debe contar la arquitectura entera.
- **Errores de dominio viven en el dominio** (Cap. 6): los handlers los traducen, no los inventan.
- **Estructura que un extraño navega en 2 minutos**: si alguien clona el repo y no encuentra "dónde está la regla X", la estructura falló.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

EXTRA · Patrones arquitectónicos del roadmap

EXTRA Las taxonomías de arquitectura que listan los roadmaps backend —

- **Monolítico** — todo en un proceso; el default correcto al empezar
- **Microservicios** — procesos separados por dominio; paga cuando el equipo ya no cabe en un monolito
- **Serverless** — funciones por evento, sin servidor visible
- **Escalado vertical** (máquina más grande) vs **horizontal** (más máquinas — exige app sin estado local)
- **Balanceadores de carga** — reparten el tráfico entre réplicas

Todos aparecen como capítulos o decisiones en este libro; la lista sirve de mapa de vocabulario para reconocerlos en una entrevista. → *cap-30..35, nu-05, Ap. K.*

16.9. Ejercicio

1. Refactoriza Taskflow a la estructura por capas de tu stack — mueve reglas, no solo archivos.
2. Agrega una segunda entidad (`lists` — listas de tareas) como router paralelo, sin duplicar wiring.
3. Sube la apuesta: implementa `storage` detrás de una interfaz/contrato mínimo para que "lista en memoria" sea intercambiable.

16.10. Mini reto

Mueve una regla de negocio del handler al servicio (ej: "no puede haber dos tareas con el mismo título en la misma lista") y escribe un párrafo respondiendo: ¿qué se ganó? ¿Qué se volvió posible que antes no lo era? (Pista: piensa en el Cap. 28 — tests sin HTTP.)

Soluciones

Ejercicio 1. Tras el refactor, `main`/`index` solo ensambla (router + middleware + handlers de error), `services/` tiene las reglas, `storage/` la lista, `schemas/` los contratos — el árbol de la sección Implementación.
Ejercicio 2. `lists` es otro router con su propio par servicio/storage: `include_router(lists_router)` / `app.use("/lists", …)` / `mux.HandleFunc("GET /lists", …)` — la estructura escala por dominio sin tocar lo existente.
Ejercicio 3. El contrato mínimo de storage: métodos `list/create/get/ update/delete` (o `Store` interface en Go). Cambiar lista→Postgres en el Cap. 10 = cambiar solo la implementación, cero handlers.
Mini reto. Lo ganado: la regla se puede (a) testear sin HTTP (`svc.create_task` duplicado → `ConflictError`), (b) reutilizar desde otro endpoint o un futuro WebSocket/importador, (c) leer en un solo lugar — la fuente de verdad del negocio. Lo que se volvió posible: tests de servicio rápidos y un handler que solo traduce.

16.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: método

Un **método** es una función que vive dentro de un objeto/clase: `task.save()` — `save` es un método de `task`. La diferencia con una función suelta: el método conoce al objeto que lo contiene (`this`/ `self`).

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega

es el resultado del build, no tu código crudo.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: ORM

Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

16.12. Lo que deberías saber hacer ahora

- ☐ Explicar qué hace cada capa y qué le está prohibido saber
- ☐ Estructurar un proyecto en capas en tu stack
- ☐ Implementar un handler deliberadamente "tonto"
- ☐ Describir cómo se inyectan dependencias en cada stack — y qué hace `internal/` en Go que ningún otro hace

Parte IV

Sección II · PostgreSQL

17 8. Necesitamos que los datos sobrevivan al reinicio

La memoria no es un plan

Figura 17.1: Parte 2 — PostgreSQL

> **En una frase**
>
> El array en memoria de la Parte 1 muere cada vez que reinicias el servidor. Ahora Taskflow necesita una **base de datos de verdad** — y antes de escribir una línea de código, hay que *diseñarla*: qué tablas, qué columnas, qué tipos, cómo se relacionan. La mitad de este capítulo no es código: es pensar.

17.1. El problema

`let tasks = []` funcionó perfecto… hasta que el proceso reinició y las 200 tareas del cliente desaparecieron. La memoria RAM es volátil *por diseño* — es el escritorio donde trabajas, no el archivero donde guardas. Necesitamos algo que sobreviva: a reinicios, a crashes, a deploys.

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

¿Por qué no un archivo JSON en disco? Porque la siguiente pregunta llega inmediata: ¿cómo busco "las tareas pendientes del usuario 42" sin leer todo el archivo? ¿Qué pasa si dos requests escriben a la vez? Eso es exactamente lo que una **base de datos** resuelve desde 1970: guardar datos estructurados, consultarlos rápido, y sobrevivir a la concurrencia.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

17.2. Cómo lo resuelve un equipo

Un equipo serio **diseña antes de crear**. La conversación real suena así: "¿qué *cosas* existen en Taskflow?" → usuarios, tareas, etiquetas. "¿Cómo se relacionan?" → un usuario tiene muchas tareas; una tarea puede tener varias etiquetas. "¿Qué no puede ser nulo?" → toda tarea tiene título. Esa conversación se dibuja en un **diagrama ER** (entidad- relación: cajas y líneas, nada más) y *después* se traduce a SQL.

Elegimos **PostgreSQL** por razones concretas, no por moda: es relacional (los datos de Taskflow *son* relaciones — tareas de usuarios), open source, el estándar de facto, y sus `JSONB` te dan lo flexible cuando lo necesites. "MongoDB porque mi dato ya es JSON" es el atajo que revisamos en *Otras cimentaciones*.

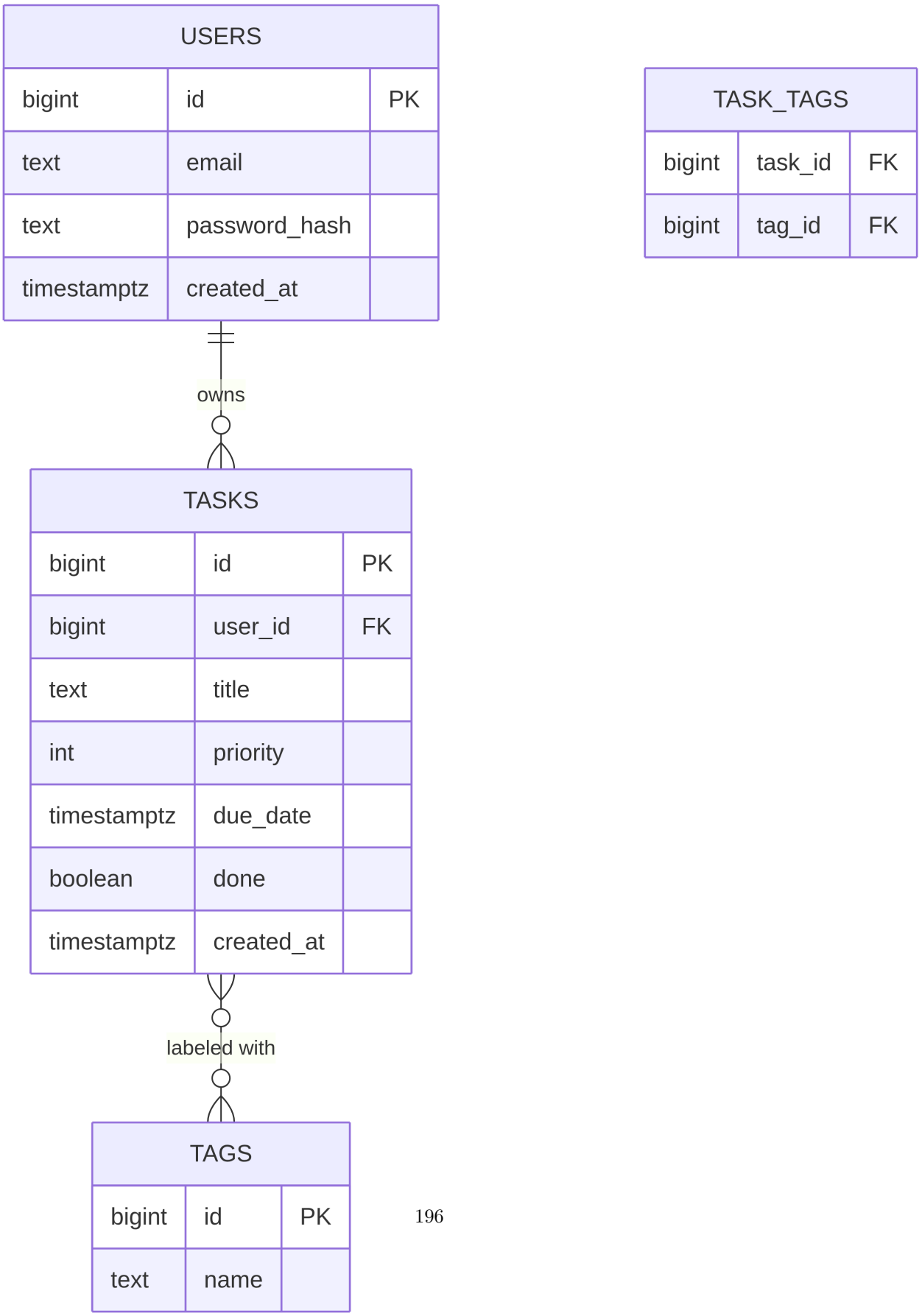
17.3. Conceptos nuevos

- D Base de datos relacional: tablas (entidades), filas (registros), columnas (atributos) — y *relaciones* entre ellas

- D Tipos de columna, `NOT NULL`, valores por defecto — la BD como *guardiana* de tus invariantes
- D Clave primaria: `SERIAL` vs `UUID` — y por qué exponer el id interno en la URL es una decisión, no un default
- D Clave foránea + `ON DELETE CASCADE` vs `SET NULL` — qué le pasa a los hijos cuando muere el padre
- D Diagrama ER: dibujar el modelo antes de crearlo
- D Relacional vs las 4 familias NoSQL — y la regla "empieza relacional"
- D Local vs producción: mismo motor, distinta operación
- U Capítulo agnóstico: SQL es el mismo en los tres stacks

17.4. La explicación visual

El modelo de Taskflow dibujado — esto es el diagrama ER:


USERS
bigint
id
PK
text
email
text
password_hash
timestamptz
created_at
owns
TASKS
bigint
id
PK
bigint
user_id
FK
text
title
int
priority
timestamptz
due_date
boolean
done
timestamptz
created_at
labeled with
TAGS
bigint
id
PK
text
name
TASK_TAGS
bigint
task_id
FK
bigint
tag_id
FK

Cómo se lee: `USERS ||--o{ TASKS` = *un usuario tiene cero o muchas tareas*. `TASKS }o--o{ TAGS` = *muchos a muchos* — y como una tabla no puede tener "listas" dentro, esa relación vive en `task_tags`, la tabla puente. **Toda decisión del dibujo se convierte en una línea de SQL.**

> Explicación de: array / lista / slice
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

17.5. Implementación

El diagrama traducido a `CREATE TABLE` — SQL puro, idéntico para los tres stacks (los drivers del cap. 10 ejecutarán esto mismo):

```
-- 001_initial.sql - the ER diagram as DDL
CREATE TABLE users (
  id            BIGSERIAL PRIMARY KEY,          -- auto-increment internal id
  public_id     UUID NOT NULL DEFAULT gen_random_uuid() UNIQUE,
  email         TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  created_at    TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE tasks (
  id         BIGSERIAL PRIMARY KEY,
  public_id  UUID NOT NULL DEFAULT gen_random_uuid() UNIQUE,
  user_id    BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  title      TEXT NOT NULL CHECK (char_length(title) BETWEEN 1 AND 200),
  priority   INT NOT NULL DEFAULT 3 CHECK (priority BETWEEN 1 AND 5),
  due_date   TIMESTAMPTZ,
  done       BOOLEAN NOT NULL DEFAULT FALSE,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE tags (
  id   BIGSERIAL PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
);

-- the many-to-many bridge: no data of its own, only pairs
CREATE TABLE task_tags (
  task_id BIGINT NOT NULL REFERENCES tasks(id) ON DELETE CASCADE,
  tag_id  BIGINT NOT NULL REFERENCES tags(id)  ON DELETE CASCADE,
  PRIMARY KEY (task_id, tag_id)               -- the pair IS the key
);
```

Lee esto como **invariantes escritos en piedra**: la validación del cap. 4 protege la API; estos `NOT NULL`/`UNIQUE`/`CHECK`/`REFERENCES` protegen el *dato* — la última frontera, la que no puedes saltarte ni con un bug en el handler ni con un script manual.

Tres decisiones que merecen una pausa:

1. **`id` interno (`BIGSERIAL`) vs `public_id` (`UUID`).** El id interno es para los JOINs (rápido, ordenado); el UUID es el que viaja en la URL `/tasks/7f3a...`. Exponer `id=42` le dice al mundo "hay ~42 tareas" e invita a probar `id=41` — el UUID no revela ni permite adivinar.
2. **`ON DELETE CASCADE`:** borrar un usuario borra sus tareas — *elegido, no default*. La alternativa `SET NULL` dejaría tareas huérfanas.
3. **`task_tags` con PK compuesta:** el par `(task_id, tag_id)` *es* la clave — no puede repetirse, y no necesita id propio.

Explicación de: JOIN

Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

Prompt listo para tu AI

```
Diseña el esquema PostgreSQL para mi app de tareas. Entidades: users,
tasks (title, priority 1-5, due_date, done), tags (many-to-many con
tasks). Requisitos: id interno BIGSERIAL + public_id UUID (el UUID es
el que se expone en la API), constraints NOT NULL/UNIQUE/CHECK donde
correspondan, FKs con ON DELETE pensado y justificado, y una tabla
puente para la relación N:M. Dame el SQL completo más un diagrama ER
en texto, y explícame cada decisión de diseño.
```

17.6. ¿Relacional o no relacional? — y local producción

Lo único que necesitas llevarte: existen otras familias de bases de datos — documento, clave-valor, columna ancha, grafo — y cada una existe porque alguna *forma de dato* le queda mejor que las tablas. Pero el 90 % de los productos (Taskflow y Nexus incluidos) son relaciones disfrazadas: **empieza relacional, agrega la especializada cuando una necesidad real lo pida.** Y la BD de tu laptop y la de producción son el mismo motor en ligas distintas: mismo SQL, distinta operación.

Otras cimentaciones: las familias NoSQL y cuándo cada una

Familia	Ejemplo	El dato se guarda como	Brilla cuando	Qué te cuesta
Documento	MongoDB, Firestore	Un JSON por registro; cada uno puede tener campos distintos	Registros con forma variable de verdad: catálogos, contenido, perfiles raros	Las relaciones las enforzas *tú* en código; no hay `REFERENCES` que proteja
Clave-valor	Redis, DynamoDB	Una llave → un valor, sin queries ricas	Leer/escribir por llave a velocidad extrema: caché, sesiones, rate limits	No puedes preguntar "los pendientes del usuario 42" — solo `GET` por llave
Columna ancha	Cassandra, Bigtable	Filas con columnas que pueden variar por fila	Escritura masiva continua: métricas, logs, IoT, series de tiempo	Diseñas por *query*, no por entidad — curva alta, nicho real
Grafo	Neo4j	Nodos + aristas: la relación *es* el dato	"Amigos de amigos", rutas, fraude, recomendación	Si tu dato no es una red, es overkill puro

Las señales de decisión — la pregunta que responde "¿cuál uso?":

- ¿Los datos se *relacionan* y esas relaciones deben cumplirse **siempre**? → **relacional**. `REFERENCES` lo garantiza; en Mongo lo garantizas tú (o no, y ahí nacen los datos huérfanos).
- ¿Cada registro puede tener campos distintos e impredecibles? → **documento**… o `JSONB` dentro de Postgres, que es la respuesta que termina ganando el 80 % de las veces.
- ¿Solo lees/escribes por una llave, a escala brutal? → **clave-valor**, casi siempre como *complemento* (Redis delante de Postgres, no en vez).
- ¿Millones de escrituras continuas tipo log/métricas? → **columna ancha**.
- ¿La pregunta del negocio es sobre la *red* (quién conoce a quién)? → **grafo**.

La trampa clásica: "MongoDB porque mi dato ya es JSON". Eso confunde el *formato* (JSON es el sobre) con la *estructura* (tus datos son relaciones: tareas **de** usuarios). Taskflow responde JSON en cada endpoint y aun así su base es relacional pura.

Alternativa relacional	Qué es	Cuándo elegirla
PostgreSQL	El estándar open source + JSONB	El default para casi todo producto
SQLite	Relacional en un archivo, sin servidor	Apps locales, embebidas, prototipos, tests
MySQL	Relacional igual de capaz	Equipo que ya lo usa; hosting legacy

17.6.1. Tu laptop vs producción: mismo motor, dos ligas

El Postgres que levantas con `docker compose up` y el RDS de AWS (o Cloud SQL de GCP, o Azure SQL) hablan **exactamente el mismo SQL** — tus queries, migraciones y drivers no cambian ni una línea. Lo que cambia es

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

todo lo que rodea al motor:

	Local (Docker)	Producción (administrada)
Datos	Descartables — borrar y re-sembrar es lo normal	Irremplazables — backups automáticos, point-in-time recovery
Si muere	Reinicias el contenedor	Réplica en otra zona, failover automático
Credenciales	`postgres:postgres` en el compose	Secret manager, usuarios con permisos mínimos, TLS obligatorio
Red	`localhost:5432`, todo abierto	Puerto visible solo desde tu app (VPC / security group)
Conexiones	10 bastan	Límite real → pooler (PgBouncer) cuando escala
El admin	Tú	El proveedor parchea y hace backup; tú configuras

La regla de oro: mismo motor en ambos entornos. Usar SQLite en local "y ya Postgres en prod" es cómo se fabrican los bugs que solo existen en producción — tipos distintos, SQL distinto, `NULL` y constraints que se comportan distinto. Por eso el compose del cap. 31 levanta Postgres real aunque pese más: dev debe parecerse a prod *en lo que importa* (mismo motor, mismas migraciones); puede diferir en lo demás (sin réplicas, sin backups — esos son problemas de operación, no de desarrollo).

17.7. Errores comunes

Error	Por qué pasa	Fix
Diseñar "sobre la marcha"	Crear tablas cuando ya hace falta el endpoint	Dibujo ER primero; SQL después
Exponer `id` secuencial en URLs	Es lo que el ORM da por defecto	`public_id` UUID para lo externo
`ON DELETE` por accidente	El framework eligió CASCADE sin que lo decidieras	Decidirlo por relación, escribirlo explícito
Validar solo en la API	"Ya lo valida el schema de Zod/Pydantic"	Constraints en la BD — la API es *una* entrada de muchas
Tablas sin `created_at`	"No lo necesito ahora"	Barato de añadir hoy, imposible de reconstruir mañana

17.8. Buenas prácticas

- **Dibuja el ER antes del primer `CREATE TABLE`** — cambiar una línea en el dibujo cuesta segundos; cambiar una tabla con datos cuesta una migración.
- **La BD es la última guardiana:** todo lo que *siempre* debe ser cierto (email único, prioridad 1–5, tarea con dueño) merece un constraint, no solo validación en el handler.

- **`id` interno para JOINs, `public_id` para el mundo** — separa la identidad interna de la expuesta.
- **Nombres consistentes:** tablas en plural y minúscula (`tasks`), snake_case en columnas (`due_date`), PK llamada `id`, FK llamada `tabla_id` (`user_id`). La convención evita mil decisiones pequeñas.

Explicación de: foreign key / clave foránea

Una **foreign key** es un puntero verificado: `tasks.project_id` apunta a `projects.id` y la BD *garantiza* que el proyecto existe. Es la diferencia entre "el dato dice 5" y "el dato apunta a algo real".

EXTRA · El paisaje completo de bases de datos

EXTRA Lo que los roadmaps listan como bases que debes reconocer —
Relacionales: MySQL (la más popular históricamente), PostgreSQL, MariaDB, MS SQL Server, Oracle.
NoSQL: MongoDB, CouchDB, DynamoDB (RethinkDB quedó en el camino). Muchas startups optan por NoSQL — pero la decisión es por *forma del dato*, no por moda (ver la sección anterior de este capítulo).
El libro elige PostgreSQL: relacional completo, JSONB cuando el dato varía, y el mismo motor en local y producción.

17.9. Ejercicio

1. Dibuja (en papel o texto) el ER si Taskflow agrega **comentarios** en las tareas: un usuario comenta en una tarea.
2. Escribe el `CREATE TABLE comments` correspondiente con sus FKs y decide el `ON DELETE`.
3. ¿Por qué `task_tags` no necesita una columna `id` propia?
4. Un compañero propone: "pasemos a MongoDB, total la API ya responde JSON". ¿Qué le respondes?
5. Tu app corre contra Postgres en Docker en tu laptop y contra RDS en producción. Nombra dos cosas que son **idénticas** entre ambos y dos que **cambian por completo**.

Soluciones

1. `USERS ||--o{ COMMENTS` y `TASKS ||--o{ COMMENTS` — un comentario pertenece a un usuario y a una tarea (dos FKs).

2. ```
   CREATE TABLE comments (
     id         BIGSERIAL PRIMARY KEY,
     public_id  UUID NOT NULL DEFAULT gen_random_uuid() UNIQUE,
     task_id    BIGINT NOT NULL REFERENCES tasks(id) ON DELETE CASCADE,
     user_id    BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
     body       TEXT NOT NULL CHECK (char_length(body) BETWEEN 1 AND 2000),
     created_at TIMESTAMPTZ NOT NULL DEFAULT now()
   );
   ```

   CASCADE en ambos: un comentario sin tarea o sin autor no tiene sentido. (Alternativa defendible: `ON DELETE SET NULL` en `user_id` si quisieras conservar comentarios de usuarios borrados — entonces la columna sería nullable. Lo importante es que sea *decisión*.)

3. Porque su PK compuesta `(task_id, tag_id)` ya es única y es toda la identidad que necesita: la fila *es* el vínculo. Un `id` extra sería una columna sin trabajo.

4. "JSON es el *formato* de la respuesta, no la *estructura* de los datos. Nuestras tareas son `de` usuarios y `con` etiquetas — esas relaciones las enforza `REFERENCES` gratis en Postgres; en Mongo las enforzaríamos nosotros en código (o no, y ahí nacen los huérfanos). Si algún día necesitamos forma variable, `JSONB` ya está incluido." — confundir formato con estructura es la trampa clásica.

5. Idénticas: el motor (mismo SQL, mismos tipos, mismas migraciones) y tu código (driver, pool, queries — ni una línea cambia). Cambian: la operación (tú admin vs backups/réplicas/failover del proveedor) y el entorno (credenciales `postgres:postgres` vs secret manager + TLS + puerto cerrado salvo a tu app).

17.10. Mini reto

Decidir qué pasa con las tareas cuando se borra su usuario — y defenderlo.

Soluciones

Con `ON DELETE CASCADE` (lo elegido): el usuario se va con sus tareas — correcto para datos personales (y GDPR-friendly: "borrar mi cuenta" borra *mis* datos). Alternativas: `SET NULL` solo si las tareas tienen valor histórico sin dueño (requeriría `user_id` nullable), o soft-delete (`deleted_at`) si la regulación exige retener. Para Taskflow, CASCADE es la defensiva: dato personal huérfano es peor que dato borrado.
   ```

17.11. Vocabulario técnico del capítulo

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: booleano

Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: ORM

Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: sesión

Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada llamada.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

17.12. Lo que deberías saber hacer ahora

- ☐ Dibujar el ER de un dominio pequeño antes de escribir SQL
- ☐ Escribir `CREATE TABLE` con PK, FKs, `NOT NULL`, `CHECK`, `UNIQUE`
- ☐ Justificar `id` interno vs `public_id` expuesto
- ☐ Elegir `ON DELETE CASCADE/SET NULL` como decisión, no default
- ☐ Modelar una relación muchos-a-muchos con tabla puente
- ☐ Nombres las 4 familias NoSQL y una necesidad real de cada una
- ☐ Defender "empieza relacional" con un criterio, no con un dogma
- ☐ Explicar qué es igual y qué cambia entre tu Docker local y la BD de prod

18 9. Necesitamos hablar SQL de verdad

El ORM viene después; primero hay que saber qué genera

> **En una frase**
>
> Antes de que cualquier ORM traduzca por ti, escribes las queries a mano en `psql`. SELECT para leer, INSERT/UPDATE/DELETE para escribir, JOIN para cruzar tablas, GROUP BY para reportes — y la lección de seguridad más importante del libro: **nunca concatenes datos del usuario en una query**.

18.1. El problema

En el cap. 8 creamos las tablas — están vacías. Ahora hay que *usarlas*: meter filas, leerlas, cruzarlas. Podríamos saltar directo al ORM ("el framework escribe el SQL por ti"), pero entonces cada query sería magia: no sabrías qué se ejecuta, por qué va lento, ni qué revisar cuando algo falla. Primero se aprende el idioma; después el traductor.

> **Explicación de: framework**
>
> Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

> **Explicación de: query / consulta**
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

> **Explicación de: ORM**
>
> Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

18.2. Cómo lo resuelve un equipo

`psql` es la terminal de PostgreSQL — el equivalente a tu shell, pero el idioma es SQL. Todo backend dev tiene una abierta siempre: inspeccionar datos reales, probar una query antes de meterla en código, arreglar una fila a mano. Los comandos de orientación primero:

> Explicación de: terminal / consola
>
> La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

```
psql "postgresql://app:dev@localhost:5432/taskflow"
```

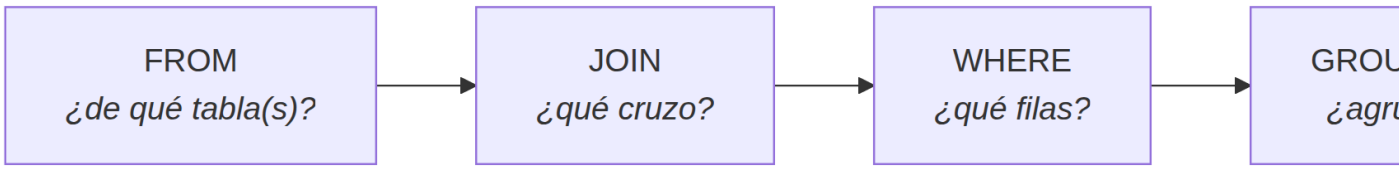
```
\dt                -- list tables
\d tasks           -- describe the tasks table (columns, constraints)
\q                 -- quit
```

18.3. Conceptos nuevos

- D `SELECT` / `WHERE` / `ORDER BY` / `LIMIT` — leer con filtros
- D `INSERT` / `UPDATE` / `DELETE` — y por qué `UPDATE` sin `WHERE` es un incidente con nombre
- D JOINs: `INNER` vs `LEFT` — cruzar tablas por sus FKs
- D `COUNT`, `GROUP BY`, `HAVING` — reportes sin traer toda la tabla
- D Inyección SQL: el error clásico y por qué los parámetros lo matan

18.4. La explicación visual

La anatomía de una query — siempre en este orden mental:



18.5. Implementación

Las 6 queries que Taskflow necesita, escritas a mano en `psql`. Lee cada una preguntándote: *¿de dónde, qué filas, qué columnas?*

1. Insertar — `INSERT ... RETURNING`:

```
INSERT INTO users (email, password_hash)
VALUES ('ana@taskflow.dev', '$2b$10$...')
RETURNING id, public_id;          -- give me back the generated ids
```

`RETURNING` es el truco de Postgres: te devuelve lo que acabas de crear sin una segunda query.

> Explicación de: return
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

2. Leer con filtro — `SELECT ... WHERE`:

```
SELECT public_id, title, due_date
FROM tasks
WHERE user_id = 1 AND done = FALSE
ORDER BY priority ASC, due_date ASC NULLS LAST
LIMIT 20;
```

3. Actualizar — el `WHERE` es tu cinturón:

```
UPDATE tasks
SET done = TRUE
WHERE public_id = '7f3a9c2e-...' AND user_id = 1;
--                     the user_id check: you can only touch YOUR tasks
```

4. Cruzar tablas — `JOIN`:

```
SELECT t.title, tg.name AS tag
FROM tasks t
JOIN task_tags tt ON tt.task_id = t.id
JOIN tags tg      ON tg.id = tt.tag_id
WHERE t.user_id = 1;
```

`INNER JOIN` trae solo filas con pareja en ambos lados; `LEFT JOIN` trae todas las de la izquierda aunque no tengan pareja (`NULL` en las columnas de la derecha) — útil para "tareas *con o sin* etiquetas".

> Explicación de: JOIN
>
> Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

5. Reportes — `GROUP BY`:

```sql
SELECT u.email, COUNT(t.id) AS open_tasks
FROM users u
LEFT JOIN tasks t ON t.user_id = u.id AND t.done = FALSE
GROUP BY u.id, u.email
HAVING COUNT(t.id) > 0
ORDER BY open_tasks DESC;
```

"Tareas abiertas por usuario" sin traer las tareas — la BD *cuenta*, tú recibes el resumen.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

6. La lección de seguridad — inyección SQL:

```python
# NEVER - the user types: ' OR '1'='1
query = f"SELECT * FROM users WHERE email = '{email}'"
# becomes: WHERE email = '' OR '1'='1'  → returns EVERY user

# ALWAYS - the driver sends data SEPARATE from code
cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
```

Con parámetros, el dato del usuario viaja *apartado* de la query — la BD lo trata como texto, jamás como código. Toda la sección de seguridad del libro cuelga de esta regla.

> Explicación de: parámetro / argumento
>
> El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

> Prompt listo para tu AI

```
Enséñame SQL practicando sobre este esquema: users(id, email),
tasks(id, user_id→users, title, priority 1-5, due_date, done),
tags(id, name), task_tags(task_id, tag_id). Para cada una de estas
necesidades dame la query y explícala línea por línea: (1) tareas
abiertas de un usuario ordenadas por prioridad, (2) tareas con sus
tags, (3) usuarios con conteo de tareas abiertas incluyendo los que
tienen cero, (4) marcar una tarea como done solo si pertenece al
usuario. Luego muéstrame cómo se vería cada query parametrizada y por
qué concatenar strings es inyección SQL.
```

18.6. ¿Por qué SQL primero y ORM después?

Lo único que necesitas llevarte: el ORM *genera* SQL — si no sabes leerlo, cada ORM es una caja negra que no puedes depurar. Con SQL en la cabeza, el ORM es un atajo que entiendes; sin él, es una dependencia que temes.

> Explicación de: dependencia
>
> Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

> Detalle: por qué los JOINs confunden
>
> El diagrama de Venn clásico miente un poco — un JOIN no es "intersección de conjuntos", es **multiplicación con filtro**: cada fila de `tasks` se empareja con cada fila de `tags` que cumpla el `ON`. Si una tarea tiene 3 tags, aparece 3 veces en el resultado (una por tag). Por eso "la lista de tareas" con JOIN a tags devuelve filas repetidas — y por eso el cap. 12 necesita hablar de cómo traer relaciones sin duplicar.

18.7. Errores comunes

Error	Por qué pasa	Fix
`UPDATE`/`DELETE` sin `WHERE`	Enter, pánico	Escribe el `WHERE` *primero*, luego el UPDATE; haz `SELECT` con ese WHERE antes
Concatenar strings en queries	"Es un string más"	Parámetros siempre — el dato nunca toca el SQL
`SELECT *` por costumbre	Traes columnas que no usas (y `password_hash` de regalo)	Nombra las columnas — es tu contrato de salida del cap. 5 aplicado a la BD
Probar la query solo en código	El error sale en el handler, lejos de la query	Pruébala en `psql` primero — ahí el error es inmediato
`LEFT JOIN` cuando era `INNER`	"Salen filas de más con NULLs"	Pregúntate: ¿quiero filas sin pareja?

18.8. Buenas prácticas

- **El `WHERE` se escribe antes que el `UPDATE`** — literal: redacta `WHERE public_id = '...'` y úsalo en un `SELECT` que devuelva exactamente la fila esperada; recién entonces conviértelo en UPDATE.
- **`RETURNING` para saber qué creaste** — una sola ida a la BD.
- **Alias cortos pero legibles** (`tasks t`, `users u`) — las queries de JOIN se leen como nombres completos.

- **Formato consistente:** keywords en mayúscula, una cláusula por línea. La query que lees en un incidente a las 3am agradece el aire.

18.9. Ejercicio

En `psql` (o en papel):

1. Lista las tareas del usuario 2 que vencen esta semana (`due_date < now() + interval '7 days'`), abiertas, por fecha.
2. Cuenta cuántas tareas tiene cada tag (`GROUP BY`).
3. Escribe el `INSERT` de una tarea con `RETURNING` y explica qué devuelve.

> **i** Soluciones

1. ```sql
   SELECT title, due_date
   FROM tasks
   WHERE user_id = 2 AND done = FALSE
     AND due_date < now() + interval '7 days'
   ORDER BY due_date ASC;
   ```

2. ```sql
   SELECT tg.name, COUNT(tt.task_id) AS task_count
   FROM tags tg
   JOIN task_tags tt ON tt.tag_id = tg.id
   GROUP BY tg.id, tg.name
   ORDER BY task_count DESC;
   ```

3. ```sql
   INSERT INTO tasks (user_id, title, priority)
   VALUES (2, 'Write chapter 9', 1)
   RETURNING id, public_id, created_at;
   ```

   Devuelve una fila con los valores *generados por la BD* — sin un segundo SELECT para conocerlos.

18.10. Mini reto

Escribir la query "tareas pendientes por usuario, ordenadas por prioridad" — y luego la versión peligrosa que un junior escribiría concatenando el `user_id`.
   ```

Soluciones

```sql
SELECT title, priority
FROM tasks
WHERE user_id = $1 AND done = FALSE      -- $1: parameter, never concatenated
ORDER BY priority ASC;
```

La versión peligrosa: `"WHERE user_id = " + userInput`. Si `userInput` es `0 OR 1=1`, la query devuelve las tareas de *todos* los usuarios. El parámetro `$1`/`%s` envía el dato por un canal separado — la BD no puede confundirlo con SQL.

18.11. Vocabulario técnico del capítulo

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: paginación

Paginar es servir la lista en páginas (20 por 20) en vez de las 10 millones. `LIMIT 21` + cursor = la página siguiente se pide con la última fila vista, no con "salta 40000" (offset, que se degrada).

Explicación de: constraint / restricción

Una **constraint** es una regla que la BD hace cumplir: `NOT NULL`, `UNIQUE`, `CHECK (price > 0)`, `FOREIGN KEY`. Es la última línea de defensa — el código puede tener bugs; la constraint no perdona.

Explicación de: inyección SQL

La **inyección SQL** es el clásico de romper la query: si concatenas input del usuario en el SQL, el usuario *escribe SQL*. `' OR 1=1--` borra tablas. La vacuna: queries parametrizadas (`$1`), siempre.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: tipos / tipado estático

Los **tipos** (`int`, `string`, `boolean`, `Task`) son el contrato de cada dato. *Tipado estático* (TypeScript, Go, Python con hints) = los tipos se revisan al escribir, no cuando explota en producción.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

18.12. Lo que deberías saber hacer ahora

- ☐ Escribir SELECT con WHERE, ORDER BY, LIMIT a mano
- ☐ Hacer INSERT/UPDATE/DELETE con `WHERE` consciente y `RETURNING`
- ☐ Cruzar tablas con INNER/LEFT JOIN y entender las filas duplicadas
- ☐ Armar un reporte con GROUP BY + HAVING
- ☐ Explicar la inyección SQL y por qué los parámetros la previenen

19 10. Necesitamos conectar el lenguaje con Postgres

Drivers, pools y por qué no abres una conexión por request

> **En una frase**
>
> Tu código y Postgres hablan por cable de red — el **driver** es el traductor, el **pool** es el grupo de conexiones reutilizables, y los **parámetros** son cómo envías datos sin inyección. Al final del capítulo Taskflow guarda tareas de verdad: reinicias el servidor y siguen ahí.

19.1. El problema

Las queries del cap. 9 existen en `psql`, escritas a mano. Pero el que necesita ejecutarlas es tu *código* — cuando llega `GET /tasks`, alguien tiene que traducir `listTasks()` a `SELECT ... WHERE user_id = $1`, enviarla por la red, y convertir las filas que vuelven en objetos de tu lenguaje. Ese "alguien" es el driver — y tiene trampas que no se ven en el tutorial de 5 líneas.

> **Explicación de: objeto / dict / struct**
>
> Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

> **Explicación de: query / consulta**
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

19.2. Cómo lo resuelve un equipo

La pieza conceptual nueva es el **pool de conexiones**. Abrir una conexión a Postgres cuesta: handshake TCP, autenticación, proceso nuevo en el servidor (~50ms y memoria). Si cada request abriera y cerrara una, la mitad de tu latencia sería *conectar*. El pool mantiene N conexiones abiertas y las presta:

`pool.query()` toma una libre, la usa, la devuelve. Con 10 conexiones atiendes cientos de requests — porque un request usa la conexión unos milisegundos.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

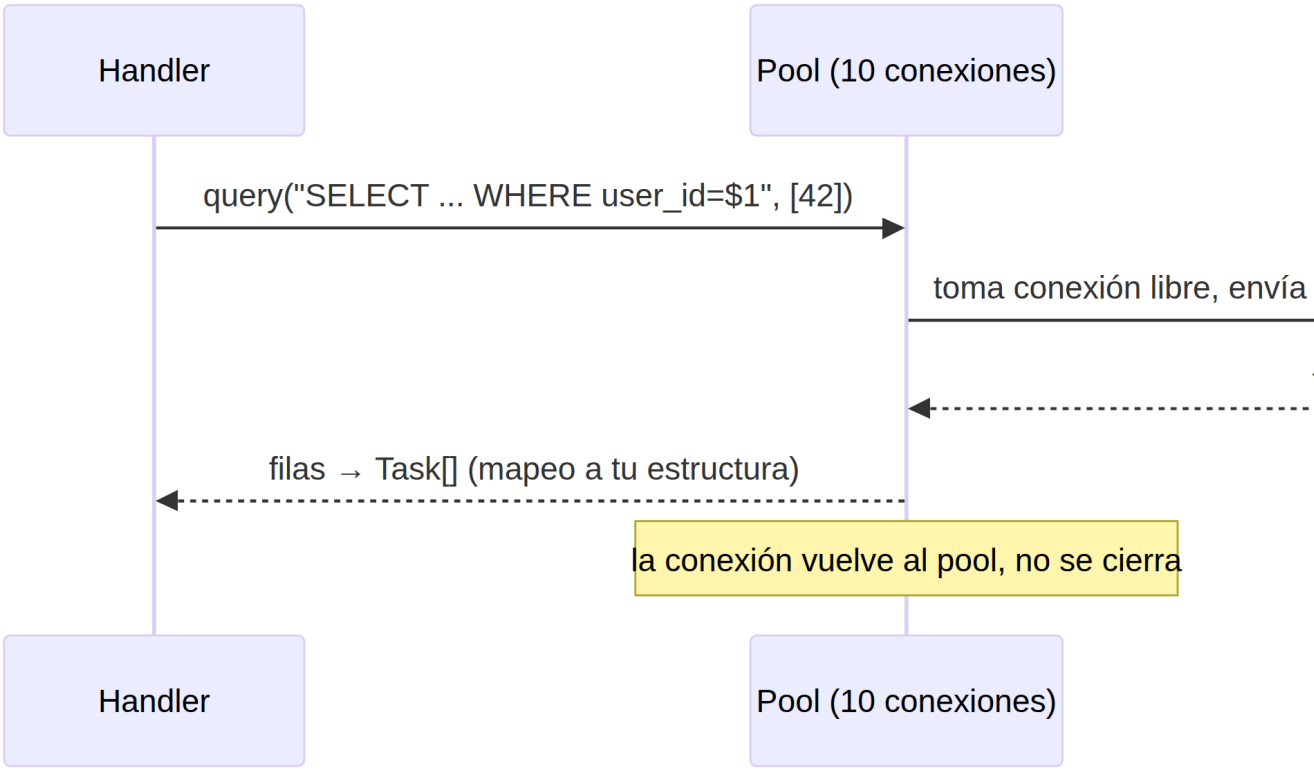
Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

El flujo completo del capítulo:



19.3. Conceptos nuevos

- D Driver: `pg` / `psycopg` / `pgx` — todos hablan el mismo protocolo de Postgres
- D Pool de conexiones: N conexiones abiertas prestadas por milisegundos — nunca una por request
- D Parámetros `$1`/`%s`: el dato viaja separado del SQL (la regla del cap. 9, ahora en código)
- U Mapear filas a estructuras: objeto vs clase vs `Scan()` a struct
- U ORM como *opción*: Prisma / SQLAlchemy / sqlc — y por qué el libro usa driver crudo primero

19.4. La explicación visual

Capa	Pieza	Análogo
Protocolo	Postgres wire protocol	El idioma del cable (igual para todos)
Driver	`pg`, `psycopg`, `pgx`	El traductor: objetos protocolo
Pool	`pg.Pool`, `psycopg_pool`, `pgxpool`	La flota de taxis: los prestas, no los compras
Tu código	`query(sql, params)` → structs	El que pide y recibe

19.5. Implementación

Taskflow persiste de verdad — `listTasks` con driver crudo en los tres stacks. Fíjate en las tres constantes: **pool creado una vez**, **params separados**, **filas mapeadas a tu estructura**.

19.5.1. TypeScript (pg)

```
// db.ts - the pool is created ONCE, at startup
import { Pool } from "pg";

export const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  max: 10,                       // at most 10 open connections
});

// tasks.ts - service layer uses the pool
export async function listTasks(userId: number): Promise<Task[]> {
  const { rows } = await pool.query(
    `SELECT public_id, title, priority, due_date, done
     FROM tasks WHERE user_id = $1
     ORDER BY priority ASC`,        // $1 - parameter, never `${userId}`
    [userId],
  );
  return rows;                      // pg already maps rows → objects
}
```

pg convierte cada fila en un objeto plano — el mapeo es automático, pero *los nombres de columna son tu contrato* (`public_id`, no `id`).

19.5.2. Python (psycopg)

```
# db.py - pool created once at startup
from psycopg_pool import ConnectionPool

pool = ConnectionPool(
    conninfo=os.environ["DATABASE_URL"],
    max_size=10,
)

# tasks.py - service layer
def list_tasks(user_id: int) -> list[Task]:
    with pool.connection() as conn:           # borrow one
        rows = conn.execute(
            """SELECT public_id, title, priority, due_date, done
               FROM tasks WHERE user_id = %s   -- %s - parameter
```

```
                ORDER BY priority ASC""",
            (user_id,),
        ).fetchall()
    return [Task(**dict(r)) for r in rows]    # rows → Task objects
```

`with pool.connection()` presta la conexión y la devuelve sola — el context manager de Python haciendo lo suyo (como `with open()`).

19.5.3. Go (pgx)

```
// db.go - pool created once at startup
pool, err := pgxpool.New(ctx, os.Getenv("DATABASE_URL"))
// pgxpool defaults to sensible max connections

// tasks.go - service layer
func listTasks(ctx context.Context, userID int64) ([]Task, error) {
    rows, err := pool.Query(ctx,
        `SELECT public_id, title, priority, due_date, done
         FROM tasks WHERE user_id = $1
         ORDER BY priority ASC`, userID)      // $1 - parameter
    if err != nil {
        return nil, err
    }
    defer rows.Close()                        // return the connection
    // pgx maps each row into the struct via Scan / CollectRows
    return pgx.CollectRows(rows, pgx.RowToStructByName[Task])
}
```

Go explicita todo: `ctx` para cancelación, `defer rows.Close()` para devolver la conexión, y el mapeo a `Task` hecho por `CollectRows`.

Lo que cambió en Taskflow: `storage/tasks.ts` ya no es un array — es el archivo que llama al pool. **El handler no se enteró** (cap. 7 pagando dividendos): `listTasks(userId)` devuelve `Task[]` igual que antes; solo cambia *dónde* viven los datos.

> Explicación de: array / lista / slice
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Prompt listo para tu AI

```
Conecta mi API de tareas [Express/FastAPI/Go] a PostgreSQL con el driver
crudo [pg / psycopg+pool / pgx]. Requisitos: pool de conexiones creado
UNA vez al arrancar con max ~10, DATABASE_URL desde variable de entorno,
todas las queries parametrizadas ($1/%s - nunca concatenar), la capa de
storage con las 4 funciones del CRUD (list/create/get/delete filtrando
por user_id), y las filas mapeadas a mis structs existentes. El handler
debe quedar idéntico - solo cambia el storage. Explícame por qué el pool
va en un archivo separado y qué pasa si lo creo por request.
```

19.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres drivers hacen exactamente lo mismo — pool, query parametrizada, filas→estructura. Cambia la *ceremonia* (context managers, `defer`, async/await) no el modelo. Aprende uno y los otros dos son un fin de semana.

Otras cimentaciones: driver crudo vs ORM vs query-builder

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
Driver crudo	Escribes SQL tú	Más líneas, pero control total	Este libro; queries que importan; aprendizaje
Query builder (Knex, Kysely, sqlc)	SQL con autocompletado	Otra capa que aprender	Equipo que quiere SQL con tipos sin magia
ORM (Prisma, SQLAlchemy, GORM)	Objetos que generan SQL	La query real se esconde; N+1 silencioso (cap. 12)	CRUDs simples, equipos que ya lo dominan
sqlc (Go)	Escribes SQL, genera código Go tipado	Solo en Go	El punto dulce del ecosistema Go

El orden del libro no es casual: **SQL → driver → (opcional) ORM**. Un ORM sobre entendimiento es atajo; sobre ignorancia, es deuda.

19.7. Errores comunes

Error	Por qué pasa	Fix
Pool creado por request	`new Pool()` dentro del handler → 10 conexiones nuevas por request	Pool global, creado en el arranque

Error	Por qué pasa	Fix
`f"WHERE id = {id}"`	El f-string se ve inocente	`$1`/`%s` + array de params — siempre
No devolver la conexión	`rows` sin cerrar / conn sin liberar → pool agotado y app colgada	`with`/`defer`/el driver devuelve tras `await`
El `Task` del código la fila	Mapeaste `id` en vez de `public_id` y expusiste el interno	SELECT nombra columnas; el struct usa `public_id`
Errores de BD llegando al cliente	El stack de Postgres en el JSON de error	El error mapper del cap. 6 los traduce a 500 genérico

19.8. Buenas prácticas

- **Un pool, un archivo, una instancia** — `db.ts` exporta el pool; nadie más crea conexiones.
- **`max` del pool < `max_connections` de Postgres** (default 100) — deja margen para `psql`, migraciones y la réplica futura.

> Explicación de: migración
>
> Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

- **Toda query parametrizada, sin excepciones** — incluso "valores que yo controlo": el día que deje de controlarlos no habrá alerta.
- **Cierra el pool al apagar** (`pool.end()` en el shutdown del servidor) — las conexiones zombie son fugas reales.

19.9. Ejercicio

1. Escribe la función `createTask(userId, title)` de tu stack con `INSERT ... RETURNING` parametrizada.
2. Explica qué pasa, paso a paso, desde `pool.query(...)` hasta que vuelven las filas.
3. Tu pool tiene `max: 10` y llegan 50 requests simultáneos que hacen una query cada uno. ¿Qué ocurre?

> Soluciones
>
> 1. TypeScript:

```ts
export async function createTask(userId: number, title: string) {
  const { rows } = await pool.query(
    `INSERT INTO tasks (user_id, title)
     VALUES ($1, $2)
     RETURNING public_id, title, priority, done, created_at`,
    [userId, title],
  );
  return rows[0];
}
```

2. `pool.query` toma una conexión libre del pool (o espera si todas están ocupadas) → el driver serializa query+params en el protocolo de Postgres → la BD ejecuta → devuelve filas → el driver las convierte a objetos → la conexión vuelve al pool → tu función retorna.
3. Los primeros 10 toman las conexiones; los otros 40 **esperan en cola** dentro del pool (no fallan) y van entrando conforme se liberan — cada query dura milisegundos, así que el drenaje es rápido. Por eso 10 conexiones atienden cientos de usuarios.

19.10. Mini reto

Detectar la diferencia entre "objeto modificado en memoria" y "fila actualizada en BD" — este código compila y *parece* funcionar:

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

```python
task = get_task(42)          # reads the row into an object
task.done = True             # updates... what exactly?
return task                  # the API says "done", but...
```

Soluciones

`task.done = True` solo cambió **el objeto en memoria** — la fila en Postgres sigue intacta. El siguiente request lee `done = FALSE` otra vez. Este es el bug fantasma de quien viene de ORMs que "sincronizan": con driver crudo *tú* eres la sincronización — falta el `UPDATE tasks SET done=TRUE`

`WHERE id=$1`. Regla mental: **el objeto es una foto de la fila, no la fila** — editar la foto no edita el archivo.

19.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: parámetro / argumento

El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: ORM

Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

19.12. Lo que deberías saber hacer ahora

- ☐ Explicar qué hacen el driver y el pool, y por qué el pool es uno solo
- ☐ Escribir CRUD con queries parametrizadas en tu stack
- ☐ Mapear filas a estructuras sin exponer el `id` interno
- ☐ Decidir cuándo un ORM suma y cuándo estorba

20 11. Necesitamos cambiar el esquema sin rezar

La BD de producción ya tiene datos; DROP TABLE no es una opción

> **En una frase**
>
> Hay que agregar `due_date` a `tasks` — pero la BD de producción ya tiene 50.000 filas y no puedes recrearla. La respuesta de la industria: **migraciones** — el esquema como archivos de código versionados que se aplican en orden, uno por uno, y nunca se editan una vez aplicados.

20.1. El problema

En dev borras la tabla y la recreas — total, los datos eran de prueba. En producción eso es un despido: la tabla tiene datos *reales* que no puedes perder. Necesitas **cambiar la forma sin destruir el contenido**: `ALTER TABLE` en vez de `DROP TABLE`. Y necesitas que ese cambio sea repetible — que la BD de Ana, la de CI y la de producción evolucionen

> **Explicación de: producción**
>
> **Producción** (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

> **Explicación de: base de datos**
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

exactamente igual.

20.2. Cómo lo resuelve un equipo

El patrón universal: cada cambio de esquema es **un archivo con timestamp**, guardado en el repo junto al código, que se aplica una sola vez y en orden:

> Explicación de: repositorio
>
> Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

> Explicación de: esquema (schema)
>
> El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

```
migrations/
  20240101000000_initial.up.sql      -- create users, tasks, tags, task_tags
  20240101000000_initial.down.sql    -- how to undo it
  20240115000000_add_due_date.up.sql -- ALTER TABLE tasks ADD due_date
  20240115000000_add_due_date.down.sql
```

La herramienta (`golang-migrate`, `alembic`, `prisma migrate`, `goose`) lleva una tabla especial en la propia BD — `schema_migrations` — que registra *qué archivos ya se aplicaron*. Correr `migrate up` aplica solo los pendientes, en orden. Así las tres BDs (dev, CI, prod) convergen al mismo esquema sin que nadie recuerde qué corrió a mano.

El flujo profesional: generar migración → revisarla como código (PR) → CI la aplica a la BD de pruebas → deploy la aplica a producción. La migración viaja con el código que la necesita — mismo commit.

> Explicación de: deploy / despliegue
>
> Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

> Explicación de: commit
>
> Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

> Explicación de: migración
>
> Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

20.3. Conceptos nuevos

- D Migración: `up`/`down`, diffs del esquema como código versionado
- D `golang-migrate` con `.sql` plano — las migraciones son *de la BD*, no del framework (los tres stacks las comparten)
- U `schema_migrations`: la tabla que registra qué se aplicó
- U Regla de oro: una migración aplicada **no se edita jamás** — se crea otra que la corrige
- D Migraciones destructivas: `ALTER ... TYPE`, `DROP COLUMN` — cómo se hacen en prod sin perder datos

20.4. La explicación visual

20.5. Implementación

Con `golang-migrate` (la misma herramienta sirve para los tres stacks — lee `.sql` plano, no es de Go aunque el nombre confunda):

```
# create a migration pair
migrate create -ext sql -dir migrations -seq add_tasks_due_date
# → migrations/000002_add_tasks_due_date.up.sql
# → migrations/000002_add_tasks_due_date.down.sql
```

```
-- 000002_add_tasks_due_date.up.sql
ALTER TABLE tasks ADD COLUMN due_date TIMESTAMPTZ;
-- additive and safe: existing rows get NULL, nothing breaks
```

```
-- 000002_add_tasks_due_date.down.sql
ALTER TABLE tasks DROP COLUMN due_date;
```

```
# apply everything pending - to any environment
migrate -path migrations -database "$DATABASE_URL" up
migrate -path migrations -database "$DATABASE_URL" version   # where am I?
```

¿Y el cambio *peligroso*? Renombrar una columna o cambiar su tipo en una tabla con millones de filas no cabe en un `ALTER` directo (bloquea la tabla). La técnica profesional es **expand-contract**: agregar la columna nueva → escribir en ambas → migrar datos en lotes → borrar la vieja. Varias migraciones pequeñas y seguras en vez de una grande y bloqueante.

Prompt listo para tu AI

```
Quiero migraciones versionadas para mi esquema PostgreSQL con
golang-migrate (SQL plano, independiente del lenguaje de mi app). Dame:
la migración inicial con mis tablas (users, tasks, tags, task_tags con
sus constraints), una segunda migración que agregue tasks.due_date, los
archivos .down correspondientes, y los comandos para crear/aplicar/
revertir/ver versión. Además explícame cómo haría un cambio destructivo
(renombrar tasks.title → tasks.name) en producción sin downtime con la
técnica expand-contract, en varias migraciones.
```

20.6. ¿Por qué el esquema va en el repo?

Lo único que necesitas llevarte: la BD es estado que el código *asume* — si el código dice `tasks.due_date` y la BD no la tiene, todo rompe. Versionar el esquema con el código significa que **commit X + migraciones hasta X = estado reproducible** en cualquier máquina. La BD deja de ser un artefacto misterioso y pasa a ser parte del proyecto.

Explicación de: estado (state)

El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

Otras cimentaciones: herramientas de migración

Alternativa	Cómo trabaja	Qué te cuesta	Cuándo elegirla
golang-migrate (`.sql` plano)	Tú escribes SQL up/down	Escribes el SQL tú (¿y eso es malo?)	Stacks mixtos, control total — el del libro
Alembic (Python)	Autogenera diff comparando modelos vs BD	El autogenerate a veces miente — hay que revisar	Proyectos Python/SQLAlchemy
`prisma migrate` (TS)	Migra desde el schema.prisma	Casada con Prisma	Si ya elegiste Prisma como ORM
`goose` / `dbmate`	`.sql` con pragmas en comentarios	Similar a migrate	Alternativas igual de sanas

Patrón: las herramientas *framework* migran "desde el modelo" (la BD adivina el diff); las de `.sql` plano te hacen escribirlo — y escribirlo es saber qué le pasa a tus datos.

20.7. Errores comunes

Error	Por qué pasa	Fix
Editar una migración ya aplicada	"Era solo un typo" → dev y prod divergen	Nueva migración que corrige — el historial es inmutable
`ALTER` destructivo directo en prod	Bloquea la tabla minutos/horas	Expand-contract en pasos
Migración sin `.down`	"Nunca se revierte" — hasta el día que sí	Toda migración tiene su undo (aunque sea `SELECT 1` documentado)
Correr migraciones a mano en prod	"Yo lo hice" no es un proceso	Las aplica el deploy/CI, mismo comando en todos lados
Código nuevo antes de migración	El handler usa `due_date` que aún no existe	Orden: migrar → deployar código

20.8. Buenas prácticas

- **Migración y código en el mismo PR** — la columna viaja con el handler que la usa.
- **Solo aditivas mientras puedas:** `ADD COLUMN` y `CREATE TABLE` son seguras; `DROP`/`ALTER TYPE` son las que piden expand-contract.
- **Revisa el SQL del autogenerate** — Alembic/Prisma generan SQL que *casi* siempre está bien; "casi" en producción es un incidente.
- **Practica el `down`** — aplicar y revertir en dev antes de prod.

20.9. Ejercicio

1. Escribe el par up/down de la migración "agregar tabla `comments`" del ejercicio del cap. 8.
2. Ordena este deploy: (a) correr migración, (b) deployar código que usa la columna nueva, (c) deployar código que deja de usar la vieja.
3. ¿Por qué `migrate` guarda `schema_migrations` en la propia BD en vez de un archivo?

Soluciones

1.
```sql
-- up
CREATE TABLE comments (
  id BIGSERIAL PRIMARY KEY,
  public_id UUID NOT NULL DEFAULT gen_random_uuid() UNIQUE,
  task_id BIGINT NOT NULL REFERENCES tasks(id) ON DELETE CASCADE,
  user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  body TEXT NOT NULL CHECK (char_length(body) BETWEEN 1 AND 2000),
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
-- down
DROP TABLE comments;
```

2. (a) → (b) → (c): primero la migración aditiva (la columna existe, nada la usa aún — seguro), luego el código que la usa, y *en otro deploy posterior* el código/migración que retira la vieja. El orden evita la ventana donde el código pide columnas inexistentes.

3. Porque el registro debe vivir *con la BD a la que describe*: si fuera un archivo, cada entorno necesitaría el suyo sincronizado y cualquiera podría editarlo. En la tabla, la propia BD dice "estoy en la versión 7" — inambiguo y viajero (un dump la incluye).

20.10. Mini reto

Agregar una columna con `DEFAULT` a una tabla "con millones de filas" — ¿qué consideraciones tiene en Postgres moderno?

Soluciones

Desde Postgres 11, `ADD COLUMN ... DEFAULT x` es **instantáneo** — la BD guarda el default como metadato y no reescribe las filas. La trampa histórica (versiones viejas reescribían toda la tabla: minutos de bloqueo) sobrevive en la memoria colectiva. Lo que *sí* sigue siendo caro: una columna con default *volátil* (`DEFAULT now()` evaluado por fila), un `NOT NULL` sin default (obliga a reescritura/validación), o un `ALTER TYPE` que cambia el formato de almacenamiento. Pregunta de entrevista disfrazada de mini reto: "¿cuánto tarda?" → depende de la versión y del tipo de default — por eso se prueba en staging con datos reales.

20.11. Vocabulario técnico del capítulo

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: registry (Docker)

Un **registry** es el almacén de imágenes (Docker Hub, ECR, Artifact Registry): el CI empuja `nexus:v42`, el servidor la jala. Es el intermediario entre "se construyó" y "está corriendo".

Explicación de: staging

Staging es el ensayo general: una copia de producción (misma config, datos falsos o anonimizados) donde pruebas el deploy antes de hacerlo de verdad. Si falla ahí, falló gratis.

Explicación de: lock / bloqueo (FOR UPDATE)

Un **lock** de fila (`SELECT ... FOR UPDATE`) dice "esta fila es mía hasta que mi transacción termine": otros que la pidan esperan. Es como el candado del probador de ropa — evita que dos editen lo mismo.

Explicación de: diff

Un **diff** es la lista de diferencias entre dos versiones: "línea 3 cambió de A a B". Git lo usa para mostrar cambios; los docs colaborativos para transmitir ediciones sin mandar todo el documento.

Explicación de: constraint / restricción

Una **constraint** es una regla que la BD hace cumplir: `NOT NULL`, `UNIQUE`, `CHECK (price > 0)`, `FOREIGN KEY`. Es la última línea de defensa — el código puede tener bugs; la constraint no perdona.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: ORM

Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

20.12. Lo que deberías saber hacer ahora

- ☐ Explicar qué es una migración y por qué el esquema vive en el repo
- ☐ Crear pares up/down con una herramienta de `.sql` plano
- ☐ Aplicar y verificar versiones con `migrate`
- ☐ Distinguir migraciones seguras (aditivas) de destructivas y planear expand-contract
- ☐ Respetar la regla: migración aplicada = inmutable

21 12. Necesitamos que la API no se ponga lenta

Con 5 filas todo es instantáneo; con 10.000, tu query «obvia» escanea la tabla

> **En una frase**
>
> En dev tu lista de tareas vuela con 5 filas. En prod hay 500.000 y tu query "obvia" las lee *todas* para encontrar 20. Este capítulo enseña las tres herramientas del backend que no se pone lento: **índices** (el índice del libro que estás leyendo), `EXPLAIN` (preguntarle a la BD cómo piensa ejecutar tu query) y **paginación** (nunca traer todo).

21.1. El problema

`SELECT * FROM tasks WHERE user_id = 42` es correcto… y lento. Sin índice, Postgres hace un *sequential scan*: lee la tabla entera, fila a fila, buscando las de `user_id = 42`. Con 10.000 filas no lo notas; con 10 millones tu endpoint tarda segundos y cada request hace un scan — la API muere de éxito. Mientras tanto, en otra parte del código hay un loop que ejecuta una query *por cada tarea* (el famoso N+1) y nadie lo ve porque en dev son 5.

> **Explicación de: bucle (loop)**
>
> Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

> **Explicación de: N+1**
>
> El **N+1** es el bug de rendimiento más común: 1 query para la lista + N queries (una por ítem) para los detalles = 101 viajes para 100 tareas. Se cura con JOIN o agregación: una sola query que lo trae todo.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

21.2. Cómo lo resuelve un equipo

El método profesional es medir, no adivinar:

Explicación de: método

Un **método** es una función que vive dentro de un objeto/clase: `task.save()` — `save` es un método de `task`. La diferencia con una función suelta: el método conoce al objeto que lo contiene (`this`/ `self`).

1. **`EXPLAIN` la query lenta** — Postgres te dice su plan: "voy a hacer Seq Scan" (malo, lee todo) o "voy a hacer Index Scan" (bueno).
2. **Crear el índice** que convierte el scan en búsqueda directa.
3. **Verificar con `EXPLAIN` otra vez** — el plan cambió o no hiciste nada.

Un índice es literalmente el índice de este libro: en vez de leer todas las páginas buscando "índice", vas a la lista ordenada del final que apunta directo. Postgres mantiene esa estructura (un árbol B-tree) actualizada en cada INSERT/UPDATE — **por eso los índices no son gratis**: aceleran lecturas y encarecen escrituras.

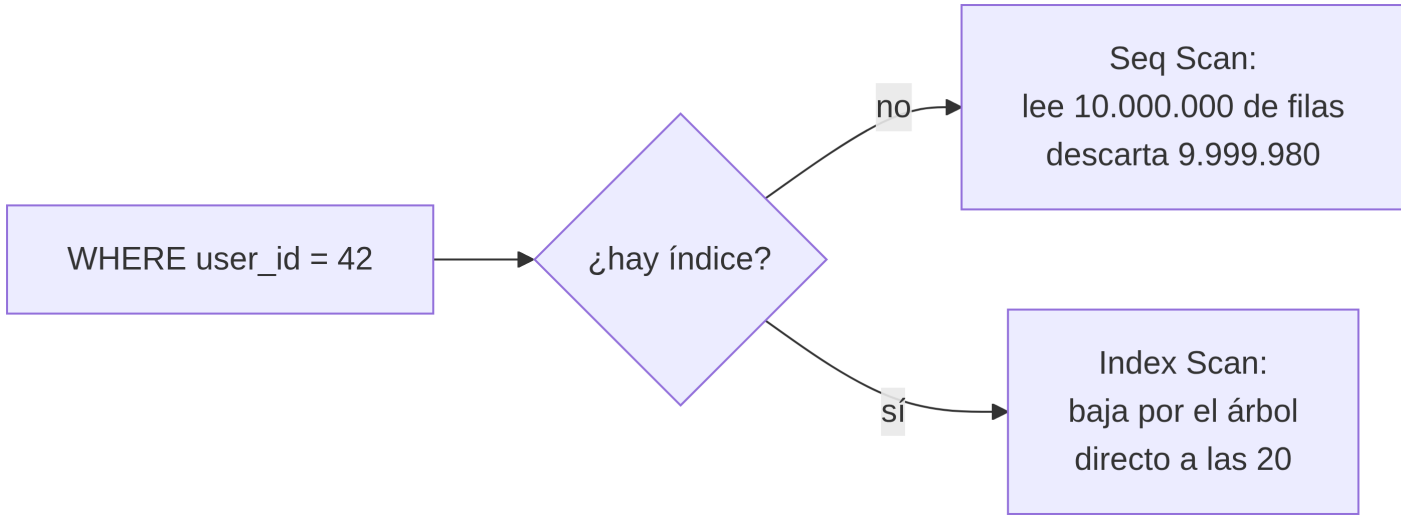
Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

21.3. Conceptos nuevos

- D Índices: qué son, qué aceleran, qué cuestan (cada write actualiza el árbol)
- D `EXPLAIN` / `EXPLAIN ANALYZE`: el plan de ejecución — Seq Scan vs Index Scan
- D El problema N+1: 1 query para la lista + N queries para los detalles
- D Paginación: `LIMIT/OFFSET` vs cursor — por qué `?page=500` degrada
- U `SELECT *`: traer 15 columnas para usar 4 también es performance

21.4. La explicación visual



21.5. Implementación

Paso 1 — ver el crimen. `EXPLAIN ANALYZE` ejecuta la query de verdad y muestra el plan con tiempos:

```
EXPLAIN ANALYZE
SELECT * FROM tasks WHERE user_id = 42 ORDER BY created_at DESC;

-- Seq Scan on tasks (cost=0.00..18334.00 rows=20)   ← reads EVERYTHING
--   Filter: (user_id = 42)
--   Rows Removed by Filter: 999980
-- Execution Time: 312.411 ms
```

`Rows Removed by Filter: 999980` — leyó un millón para entregar 20.

Paso 2 — el índice que falta:

```
CREATE INDEX idx_tasks_user_done_created
  ON tasks (user_id, done, created_at DESC);
```

Índice *compuesto*: columnas en el orden del query (igualdad primero, orden después). Ahora:

```
-- Index Scan using idx_tasks_user_done_created (rows=20)
-- Execution Time: 0.041 ms        ← 312ms → 0.04ms
```

Paso 3 — el N+1 que nadie ve. Este código parece inocente:

```
tasks = list_tasks(user_id)          # 1 query - returns 20 tasks
for t in tasks:
    t.tags = get_tags_for_task(t.id) # 20 MORE queries - one PER task
# total: 21 round trips to the DB for one HTTP request
```

La cura: traerlo todo en **una** query (JOIN o `IN`):

> Explicación de: JOIN
>
> Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

```
SELECT t.id, t.title, tg.name
FROM tasks t
LEFT JOIN task_tags tt ON tt.task_id = t.id
LEFT JOIN tags tg ON tg.id = tt.tag_id
WHERE t.user_id = $1 AND t.done = FALSE;
-- 1 query → group rows in code (3 rows per task if 3 tags)
```

Paso 4 — paginar. `?page=1&size=20` se traduce a `LIMIT 20 OFFSET 0` — bien hasta que `OFFSET 10000` obliga a la BD a contar y descartar 10.000 filas. La alternativa que escala es el

> Explicación de: paginación
>
> **Paginar** es servir la lista en páginas (20 por 20) en vez de las 10 millones. `LIMIT 21` + cursor = la página siguiente se pide con la última fila vista, no con "salta 40000" (offset, que se degrada).

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

> Explicación de: escalado horizontal / vertical
>
> **Vertical**: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

cursor: recuerdas *dónde* quedaste y sigues desde ahí:

```
-- page 1
SELECT public_id, title, created_at FROM tasks
WHERE user_id = $1 ORDER BY created_at DESC, id DESC LIMIT 20;
```

```
-- page 2: continue AFTER the last row you saw (the cursor)
SELECT public_id, title, created_at FROM tasks
WHERE user_id = $1 AND (created_at, id) < ($2, $3)
ORDER BY created_at DESC, id DESC LIMIT 20;
-- the index walks DIRECTLY to row 21 - no counting, no discarding
```

La API devuelve `next_cursor` (el `created_at,id` de la última fila) y el cliente lo devuelve en la siguiente llamada. Así paginan las APIs reales (Twitter, Stripe) — nunca `?page=`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Prompt listo para tu AI

```
Mi endpoint GET /tasks está lento en producción. Contexto: tabla tasks
con 500k filas, query actual SELECT * FROM tasks WHERE user_id=$1 AND
done=FALSE, y además hago una query por tarea para traer sus tags (N+1).
Dame: el CREATE INDEX correcto y por qué ese orden de columnas, la
query única con JOIN que elimina el N+1, cómo leer el EXPLAIN ANALYZE
antes y después, y la paginación por cursor (keyset) en vez de
OFFSET con el campo next_cursor en la respuesta. Explícame también
qué no debo indexar.
```

21.6. ¿Por qué no indexar todo entonces?

Lo único que necesitas llevarte: un índice es una *apuesta* — pagas costo de escritura y espacio a cambio de lecturas rápidas. Se indexa lo que los WHERE/JOIN/ORDER **realmente consultan**, se mide con `EXPLAIN`, y se borra lo que nadie usa. Performance no es adivinar: es preguntarle a la BD qué hizo.

Detalle: cómo decide Postgres

El planificador estima costos con estadísticas de tus datos — por eso la *misma* query puede elegir Seq Scan con 50 filas (correcto: el índice cuesta más que leer todo) e Index Scan con 5 millones. Un Seq Scan en `EXPLAIN` no siempre es un bug — lo es cuando la tabla crece y el plan no cambia. Y sí: a veces creas el índice perfecto y Postgres lo ignora porque estima que devuelves "casi toda la tabla" — entonces el scan es honestamente más barato.

21.7. Errores comunes

Error	Por qué pasa	Fix
N+1 invisible	En dev son 5 filas; el ORM lo disfraza de "código limpio"	Log de queries en dev: 1 request = N queries es la alarma
Índice en orden equivocado	`(created_at, user_id)` no sirve para `WHERE user_id=`	Igualdad primero, orden después — lee tu WHERE
`OFFSET 100000` en prod	"La paginación funcionaba"	Cursor/keyset para listas que crecen
`SELECT *` en endpoints de lista	Traes `password_hash` y 10 columnas que no usas	Las columnas del contrato, como en cap. 5
Índices "por si acaso"	Cada INSERT/UPDATE paga todos tus índices	`pg_stat_user_indexes` muestra los que nadie usa

21.8. Buenas prácticas

- **Log de queries en desarrollo** — ver "1 request → 21 queries" cura el N+1 antes de que nazca.
- **`EXPLAIN ANALYZE` con datos parecidos a prod** — explicar contra 5 filas miente: cualquier plan se ve rápido.
- **Crea índices en migraciones** (cap. 11) — son parte del esquema, versionados como todo.
- **`CREATE INDEX CONCURRENTLY`** en tablas grandes de prod — no bloquea escrituras mientras se construye.

> **EXTRA · Las estructuras de datos detrás de tu BD**
>
> EXTRA Los roadmaps piden "estructuras de datos y algoritmos" — aquí está por qué les importa a un backend, conectado con lo que acabas de ver:
>
> - **Tabla hash** → así funciona un índice `HASH`
> - **Árbol de búsqueda (BST)** → el B-tree de los índices normales es su primo balanceado; por eso la búsqueda es O(log n)
> - **Búsqueda binaria** → la misma idea: descartar la mitad en cada paso
> - **Arreglos / listas enlazadas / pilas / colas / grafos** → el lenguaje de la entrevista; las colas son literalmente los jobs del cap-21
> - **Recursión y ordenamientos** (burbuja, selección, inserción) → ejercicio clásico; en la vida real ordena la BD con `ORDER BY` + índice
>
> No necesitas implementar un árbol rojo-negro; necesitas *reconocer* qué estructura usa cada herramienta — eso es lo que hace predecible el rendimiento.

21.9. Ejercicio

1. Escribe el `EXPLAIN` para `WHERE user_id = 7 AND done = FALSE ORDER BY due_date` y diseña el índice compuesto ideal.
2. Reescribe el N+1 de "listar usuarios con su conteo de tareas abiertas" en una sola query.
3. ¿Por qué el cursor `(created_at, id)` necesita *dos* columnas y no solo `created_at`?

Soluciones

1. ```
   EXPLAIN ANALYZE SELECT * FROM tasks
   WHERE user_id = 7 AND done = FALSE ORDER BY due_date;

   CREATE INDEX idx_tasks_user_done_due
     ON tasks (user_id, done, due_date);
   ```

   Igualdad (`user_id`, `done`) primero, la de orden (`due_date`) al final — el índice ya sale ordenado.

2. ```
   SELECT u.id, u.email, COUNT(t.id) AS open_tasks
   FROM users u
   LEFT JOIN tasks t ON t.user_id = u.id AND t.done = FALSE
   GROUP BY u.id, u.email;
   ```

   Una query con `LEFT JOIN`+`COUNT` en vez de "usuarios" + una query de conteo por cada uno.

3. Porque `created_at` puede repetirse (dos tareas al mismo instante) — el cursor debe ser *determinista*: `(created_at, id)` es único y la paginación no se traga ni repite filas.

21.10. Mini reto

Encontrar el N+1 en este fragmento que "parece inocente":

```
const projects = await listProjects();            // SELECT * FROM projects
for (const p of projects) {
  p.owner = await getUser(p.owner_id);            // one query per project
  p.taskCount = await countTasks(p.id);           // and another one
  p.tags = await getTags(p.id);                   // and ANOTHER one
}
// 20 projects → 1 + 60 queries. In dev you never noticed.
```

Soluciones

Son **tres** N+1 anidados: por cada proyecto hay 3 queries (owner, count, tags) → $1 + 20 \times 3 = 61$ round trips. La cura es una query con JOINs y agregación:

```sql
SELECT p.*, u.email AS owner_email,
       COUNT(DISTINCT t.id) AS task_count,
       array_agg(DISTINCT tg.name) AS tags
FROM projects p
JOIN users u ON u.id = p.owner_id
LEFT JOIN tasks t ON t.project_id = p.id
LEFT JOIN project_tags pt ON pt.project_id = p.id
LEFT JOIN tags tg ON tg.id = pt.tag_id
GROUP BY p.id, u.email;
```

Una query, un viaje — y el log de queries en dev te lo habría mostrado el primer día.

21.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: ORM

Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

21.12. Lo que deberías saber hacer ahora

☐ Leer un `EXPLAIN` y distinguir Seq Scan de Index Scan
☐ Crear índices compuestos en el orden correcto de columnas
☐ Detectar y curar un N+1 con JOIN o `IN`
☐ Implementar paginación por cursor y explicar por qué OFFSET degrada
☐ Justificar qué indexar y qué no

22 13. Necesitamos que dos cosas pasen juntas o ninguna

Si falla el paso 2, la BD queda mentirosa

> **En una frase**
>
> "Crear tarea + registrar evento + actualizar contador": tres writes que deben pasar **juntos o ninguno**. Si el paso 2 falla, la BD queda inconsistente — la tarea existe pero el evento no, el contador dice 5 y hay 4. La herramienta es la **transacción**: `BEGIN` abre, `COMMIT` confirma todo, `ROLLBACK` deshace todo. La BD promete: nunca un estado a medias.

22.1. El problema

Cada `INSERT/UPDATE` que envías al pool se confirma *al instante* — es el modo "autocommit". Perfecto para una query; peligroso para tres. Imagina "crear la tarea y descontar un crédito del usuario":

> **Explicación de: query / consulta**
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

> **Explicación de: pool de conexiones**
>
> Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

```
insert_task(...)              #   committed - the task EXISTS now
charge_credit(user_id)        #   crashes - but the task is already saved
# the DB now lies: task created, credit never charged
```

No hay código que resuelva esto *después* — la tarea fantasma ya está confirmada. La única solución es que la BD entienda "estas operaciones son UNA".

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

22.2. Cómo lo resuelve un equipo

ACID en una frase por letra, con dinero (la metáfora original — mover $100 entre cuentas):

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

- **Atomicidad:** todo o nada — la transacción es indivisible.
- **Consistencia:** la BD nunca queda violando sus reglas (constraints).
- **Aislamiento:** dos transacciones simultáneas no se ven a medias.
- **Durabilidad:** lo confirmado sobrevive al crash — está en disco.

El patrón de código es idéntico en todos los stacks: **tomar UNA conexión del pool** (no el pool entero — la transacción vive en una conexión), `BEGIN`, ejecutar todas las queries ahí, `COMMIT` si todo bien, `ROLLBACK` si algo falla. Y la pregunta de arquitectura que lo hace interesante: ¿en qué capa vive la frontera de la transacción?

Respuesta: en el **servicio** (cap. 7). El handler no sabe de transacciones; el repositorio no decide cuándo empieza una — el servicio agrupa las operaciones que son *un hecho de negocio*.

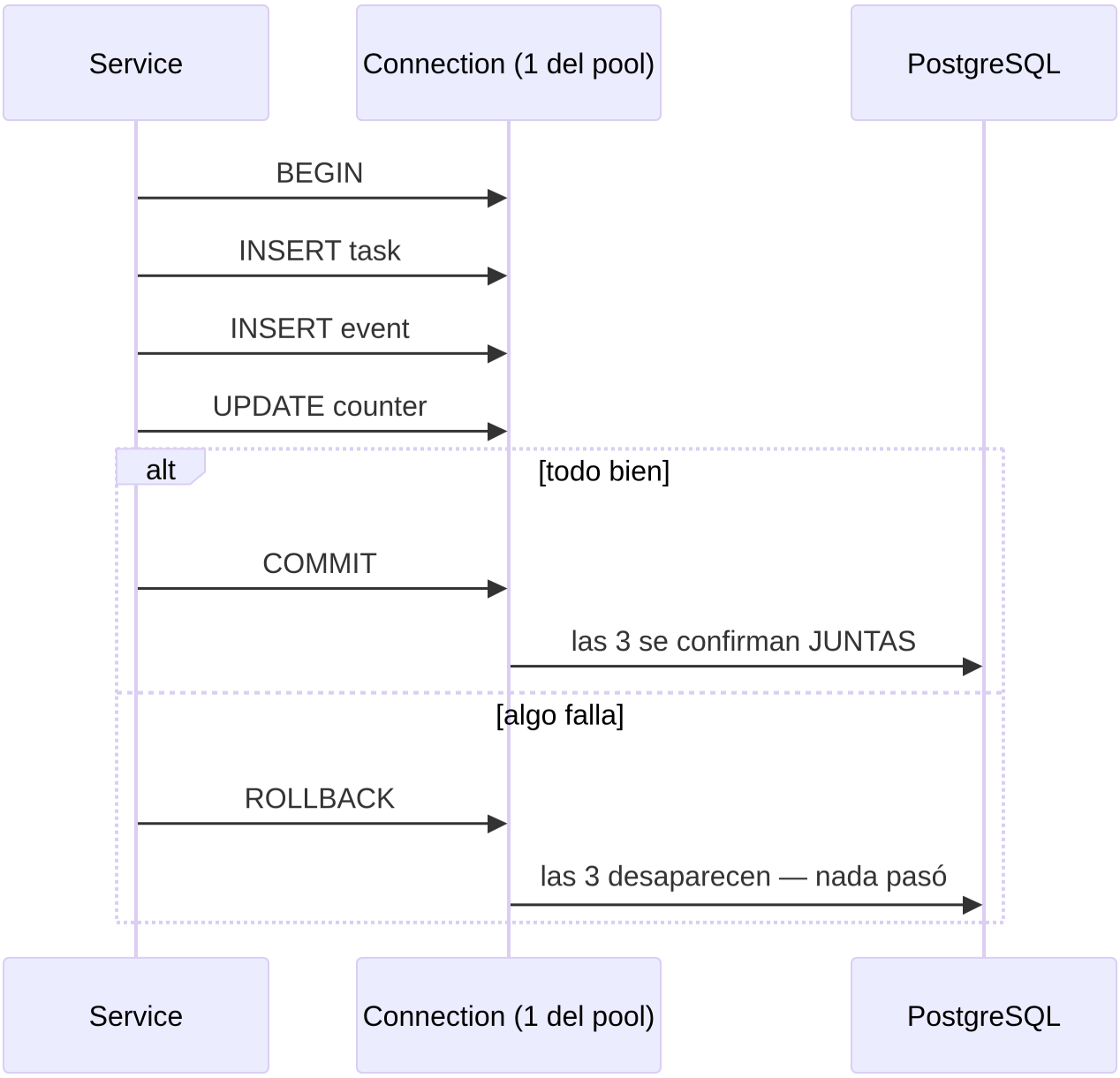
Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

22.3. Conceptos nuevos

- D Transacción y ACID, explicado con dinero
- D `BEGIN`/`COMMIT`/`ROLLBACK` en cada driver — la transacción vive en *una* conexión, no en el pool
- U `try/finally` vs `defer` vs context manager: tres formas de garantizar el ROLLBACK
- D `SELECT ... FOR UPDATE`: bloquear la fila que vas a modificar — contra el doble-submit
- U Dónde vive la frontera transaccional en una arquitectura por capas

22.4. La explicación visual



22.5. Implementación

"Crear tarea + registrar evento + descontar crédito" atómico en los tres stacks. Observa la constante: **conexión individual**, `BEGIN…COMMIT`, y el `ROLLBACK` garantizado por el mecanismo de limpieza del

lenguaje.

> Explicación de: evento
>
> Un **evento** es "algo que pasó" convertido en dato: el click del usuario, el `payment.succeeded` del webhook, el mensaje del socket. La programación moderna es reaccionar a eventos más que seguir un guion.

22.5.1. TypeScript (pg)

```
export async function createTaskWithEvent(userId: number, title: string) {
  const client = await pool.connect();       // borrow ONE connection
  try {
    await client.query("BEGIN");
    const { rows } = await client.query(
      `INSERT INTO tasks (user_id, title) VALUES ($1, $2)
       RETURNING public_id, title`,
      [userId, title],
    );
    await client.query(
      `INSERT INTO events (user_id, type, ref) VALUES ($1, 'task.created', $2)`,
      [userId, rows[0].public_id],
    );
    await client.query(
      `UPDATE users SET credits = credits - 1 WHERE id = $1`,
      [userId],
    );
    await client.query("COMMIT");            // all three land together
    return rows[0];
  } catch (err) {
    await client.query("ROLLBACK");          // all three vanish together
    throw err;                               // the error mapper (cap. 6) handles it
  } finally {
    client.release();                        // ALWAYS return the connection
  }
}
```

22.5.2. Python (psycopg)

```
def create_task_with_event(user_id: int, title: str) -> dict:
    with pool.connection() as conn:          # borrow ONE connection
        with conn.transaction():             # BEGIN...COMMIT or auto-ROLLBACK
            row = conn.execute(
                """INSERT INTO tasks (user_id, title) VALUES (%s, %s)
                   RETURNING public_id, title""",
```

```
                (user_id, title),
            ).fetchone()
            conn.execute(
                """INSERT INTO events (user_id, type, ref)
                   VALUES (%s, 'task.created', %s)""",
                (user_id, row["public_id"]),
            )
            conn.execute(
                "UPDATE users SET credits = credits - 1 WHERE id = %s",
                (user_id,),
            )
        return row                          # exiting = COMMIT; exception = ROLLBACK
```

El `with conn.transaction()` de psycopg *es* el try/finally — si algo lanza excepción adentro, el ROLLBACK se ejecuta solo.

22.5.3. Go (pgx)

```
func createTaskWithEvent(ctx context.Context, userID int64, title string) (Task, error) {
    tx, err := pool.Begin(ctx)              // BEGIN on one pooled connection
    if err != nil {
        return Task{}, err
    }
    defer tx.Rollback(ctx)                  // no-op if COMMIT succeeded

    var t Task
    err = tx.QueryRow(ctx,
        `INSERT INTO tasks (user_id, title) VALUES ($1, $2)
         RETURNING public_id, title`, userID, title).Scan(&t.PublicID, &t.Title)
    if err != nil {
        return Task{}, err
    }
    if _, err = tx.Exec(ctx,
        `INSERT INTO events (user_id, type, ref) VALUES ($1, 'task.created', $2)`,
        userID, t.PublicID); err != nil {
        return Task{}, err
    }
    if _, err = tx.Exec(ctx,
        `UPDATE users SET credits = credits - 1 WHERE id = $1`, userID); err != nil {
        return Task{}, err
    }
    return t, tx.Commit(ctx)                // all three land together
}
```

El `defer tx.Rollback()` de Go es elegante: si `Commit` ya corrió, el rollback es no-op; si saliste por error, el rollback se ejecuta al salir — imposible olvidarlo.

El doble-submit: dos clicks rápidos en "crear tarea" = dos requests casi simultáneos. Si ambos leen `credits = 5` antes de escribir, ambos descuentan → cobraste dos veces la misma operación. La cura es bloquear la fila mientras la transacción vive:

```
SELECT credits FROM users WHERE id = $1 FOR UPDATE;
-- any other transaction touching this row WAITS until you COMMIT
```

Prompt listo para tu AI

```
Implementa una operación atómica en mi API de tareas
[pg / psycopg / pgx]: crear tarea + insertar evento 'task.created' +
descontar 1 crédito del usuario. Requisitos: UNA conexión del pool para
toda la transacción (no el pool), BEGIN/COMMIT explícitos con ROLLBACK
garantizado por el mecanismo del lenguaje (try/finally, context manager
o defer), la frontera de la transacción en la capa de servicio, la
conexión devuelta siempre al pool, y el bloqueo FOR UPDATE en la lectura
de créditos para evitar el doble-submit. Explícame qué pasa si uso el
pool directo en vez de una conexión.
```

22.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: la semántica es idéntica — BEGIN, queries en *una* conexión, COMMIT/ROLLBACK. Lo que cambia es **quién te obliga a limpiar**: en TS tú escribes `finally`; en Python el `with` lo hace por contrato; en Go el `defer` lo hace imposible de olvidar. El lenguaje decide cuánta disciplina te regala.

Detalle: por qué una conexión y no el pool

Una transacción es un estado *de la conexión*: `BEGIN` la abre en ese cable concreto, y todas las queries deben ir por ese mismo cable para estar adentro. Si usaras `pool.query()` para cada una, cada query podría viajar por una conexión *distinta* — y la transacción no existiría: tres autocommits sueltos. Por eso el patrón es siempre "pedir prestada una conexión, hacer todo ahí, devolverla".

22.7. Errores comunes

Error	Por qué pasa	Fix
Queries sueltas en vez de transacción	Autocommit invisible	Cualquier multi-write de negocio = `BEGIN`
`pool.query` dentro de la transacción	Cada query puede tomar otra conexión	`client`/`conn`/`tx` — una sola

Error	Por qué pasa	Fix
Olvidar `client.release()`	Conexión prestada para siempre → pool vacío → app colgada	`finally`/`defer`/`with` la devuelve
Transacciones largas	Una tx que espera input del usuario bloquea filas	La tx dura milisegundos: leer → decidir → tx corta
ROLLBACK que traga el error	`catch` hace rollback y `return null`	Rollback y *relanza* — el mapper decide el HTTP

22.8. Buenas prácticas

- **La transacción es un hecho de negocio:** si dos writes deben ser verdad a la vez (o falso a la vez), van juntos — y eso lo decide el servicio, no el handler.
- **Corta y caliente:** la transacción contiene solo los writes — validaciones y lecturas lentas fuera.
- **`FOR UPDATE` solo donde haga falta:** bloquear la fila que vas a modificar (créditos, stock, contadores); no todas las lecturas.
- **El error siempre sube:** ROLLBACK silencioso + respuesta "ok" es la peor mentira del backend.

22.9. Ejercicio

1. Escribe en tu stack la operación "completar tarea + sumar 1 punto al usuario" como transacción.
2. Explica por qué `FOR UPDATE` en la lectura de `credits` evita el doble-submit.
3. ¿Qué le pasa a la conexión si tu código lanza excepción a mitad de la transacción y no usas `finally`/`with`/`defer`?

Explicación de: lock / bloqueo (FOR UPDATE)

Un **lock** de fila (`SELECT ... FOR UPDATE`) dice "esta fila es mía hasta que mi transacción termine": otros que la pidan esperan. Es como el candado del probador de ropa — evita que dos editen lo mismo.

Soluciones

1. Python:

```python
def complete_task(user_id: int, task_id: int) -> None:
    with pool.connection() as conn:
        with conn.transaction():
            conn.execute(
                "UPDATE tasks SET done = TRUE WHERE public_id = %s AND user_id = %s",
                (task_id, user_id),
            )
            conn.execute(
                "UPDATE users SET points = points + 1 WHERE id = %s",
                (user_id,),
            )
```

2. Sin `FOR UPDATE`: dos transacciones leen `credits = 5` a la vez, ambas calculan `5-1`, ambas escriben `4` — un descuento se pierde. Con `FOR UPDATE`, la segunda *espera* a que la primera haga COMMIT: lee el `4` ya actualizado y escribe `3`. El bloqueo convierte la lectura+escritura en algo atómico.
3. La conexión queda **prestada para siempre** con una transacción abierta — el pool se vacía conexión a conexión hasta que la app se cuelga esperando una libre. Por eso la devolución vive en el mecanismo de limpieza garantizada del lenguaje.

22.10. Mini reto

Diseñar qué debería pasar si dos usuarios completan la misma tarea "a la vez" — hay requisito: solo el primero debe ganar puntos.

Soluciones

Sin protección, ambos leen `done = FALSE`, ambos marcan `TRUE`, ambos ganan el punto. La solución no es `FOR UPDATE` sino un **UPDATE condicional** — la BD misma decide:

```sql
UPDATE tasks SET done = TRUE
WHERE public_id = $1 AND done = FALSE
RETURNING id;
-- returns 1 row ONLY to the first transaction;
-- the second gets 0 rows → "already completed" → no points
```

Lección: el `WHERE done = FALSE` convierte el "check then act" (que tiene ventana de carrera) en un solo acto atómico — el patrón se llama *compare-and-set* y resuelve la mitad de los problemas de concurrencia sin bloqueos.

22.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: constraint / restricción

Una **constraint** es una regla que la BD hace cumplir: `NOT NULL`, `UNIQUE`, `CHECK (price > 0)`, `FOREIGN KEY`. Es la última línea de defensa — el código puede tener bugs; la constraint no perdona.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: concurrencia vs paralelismo

Concurrencia = manejar muchas cosas en progreso (atender 1000 requests intercalando). **Paralelismo** = ejecutar varias a la vez en varios núcleos. Un camarero con 10 mesas es concurrente; 10 camareros son paralelos.

Explicación de: rollback

Un **rollback** es volver atrás: la migración se revierte, el deploy regresa a la versión anterior. Todo cambio serio tiene plan de rollback *antes* de ejecutarse — si no, el plan es rezar.

Explicación de: estado (state)

El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

22.12. Lo que deberías saber hacer ahora

- ☐ Explicar ACID con el ejemplo del dinero
- ☐ Escribir una transacción multi-write en tu stack (una conexión, ROLLBACK garantizado)
- ☐ Ubicar la frontera transaccional en la capa de servicio
- ☐ Usar `FOR UPDATE` y el UPDATE condicional contra el doble-submit
- ☐ Decir qué pasa con la conexión si olvidas devolverla

Parte V

Sección II · Usuarios, autenticación y permisos

23 14. Necesitamos guardar contraseñas sin traicionar a nadie

Texto plano nunca; cifrado tampoco. ¿Entonces qué?

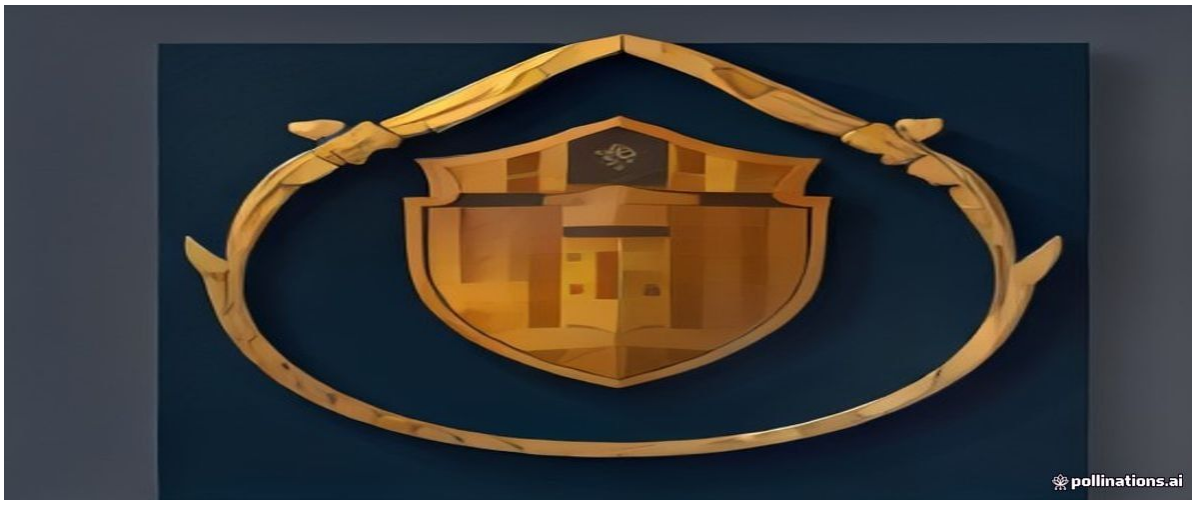


Figura 23.1: Parte 3 — Usuarios, autenticación y permisos

> **En una frase**
>
> Taskflow va a tener usuarios — y la primera decisión de seguridad del libro: **las contraseñas no se guardan, se hashean**. Ni texto plano (una filtración = todas las claves expuestas) ni cifradas (la clave que las descifra también puede filtrarse). Se guarda un hash bcrypt — *lento a propósito* — y al hacer login se hashea lo que escribió el usuario y se comparan. La BD jamás conoce la contraseña real.

23.1. El problema

`users.password TEXT` con la contraseña dentro es una bomba de tiempo: las bases de datos se filtran (backups perdidos, inyección SQL, un empleado descontento). Cuando pase — *cuando*, no *si* — todo

usuario queda desnudo, y como la gente **reutiliza contraseñas**, tu filtración abre sus cuentas en otros sitios. Tu falla se convierte en la de todos.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

¿Cifrarla entonces? Peor trampa: el cifrado es *reversible por diseño* — existe una clave que lo deshace, y esa clave vive en tu servidor. Quien robe la BD probablemente robe también la clave. Necesitas algo

> Explicación de: cifrado / encriptación
>
> El **cifrado** convierte datos en basura legible solo con la llave — y a diferencia del hash, *se puede revertir*. HTTPS cifra el cable; las contraseñas NO se cifran, se hashean (no hay por qué revertirlas).

irreversible: una función que transforma `micontraseña123` en `$2b$10$N9qo8uLO...` de tal forma que no hay camino de regreso. Eso es un **hash**.

> Explicación de: hash / bcrypt
>
> Un **hash** es una función de un solo sentido: `contraseña →$2b10…`. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

23.2. Cómo lo resuelve un equipo

El flujo universal (idéntico en los tres stacks):

1. **Registro:** el usuario manda su contraseña → el servidor la hashea → guarda el hash. La contraseña en claro existe solo durante ese request.
2. **Login:** el usuario manda su contraseña → el servidor hashea lo recibido → compara con el hash guardado → coinciden = es él.
3. La BD almacena `password_hash` — y el contrato de salida del cap. 5 se asegura de que nunca salga en un JSON.

¿Y qué impide que el atacante *también* hashee un diccionario de contraseñas comunes y compare? **El salt y el costo.** bcrypt agrega un salt aleatorio por usuario (dos usuarios con la misma clave tienen hashes distintos) y es deliberadamente *lento* (~100ms por intento): para tu login es imperceptible; para quien prueba mil millones de combinaciones es una eternidad. MD5 y SHA256 son hashes rápidos — rápidos para ti *y para el atacante*. Por eso no sirven para contraseñas.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

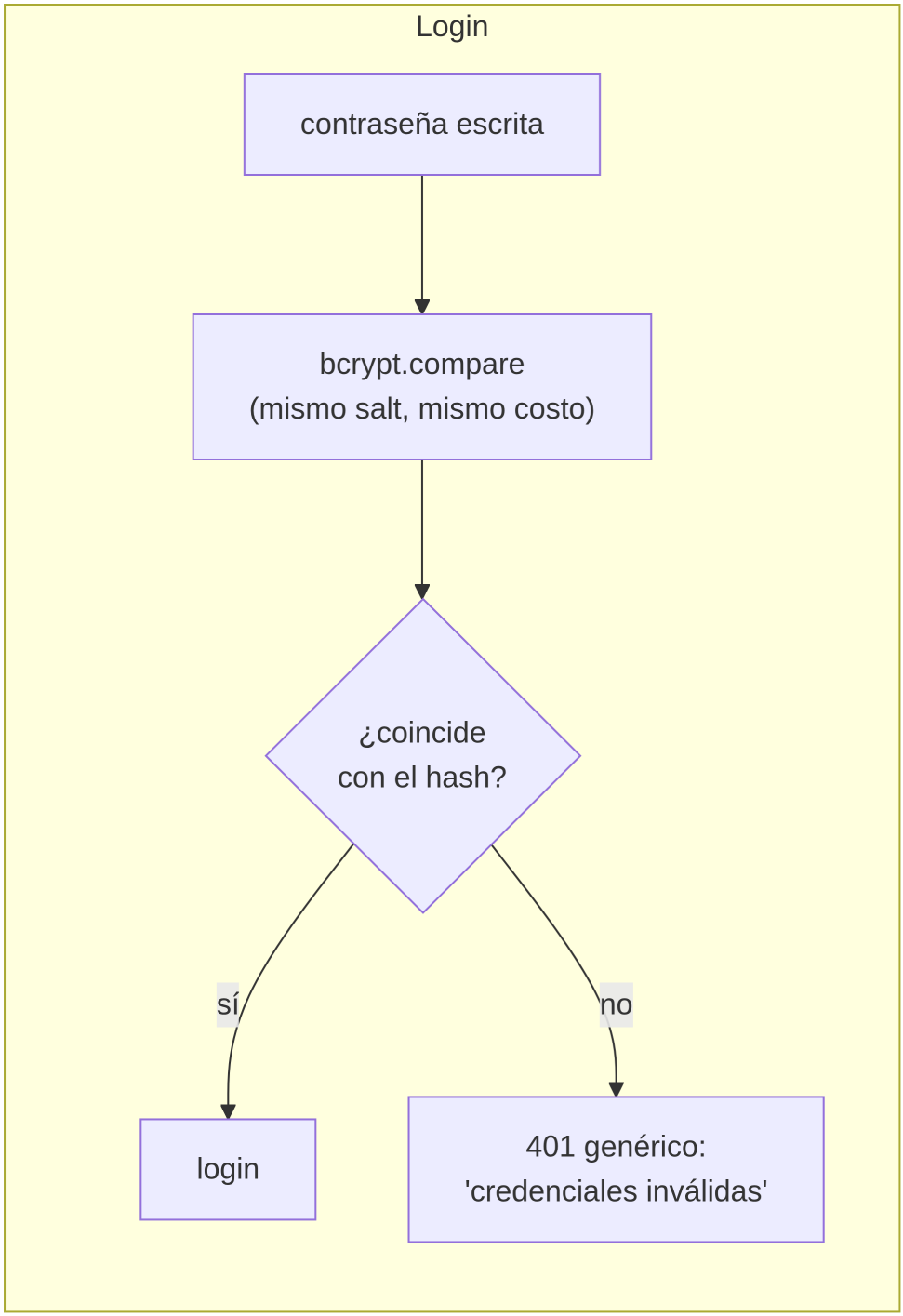
Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

23.3. Conceptos nuevos

- U Hash cifrado: irreversible vs reversible — *la* pregunta de entrevista
- U bcrypt: salt incorporado, factor de costo, "lento" como característica de seguridad
- TS `bcrypt` npm · Py `bcrypt` pip · Go `golang.org/x/crypto/bcrypt` — librerías distintas, algoritmo idéntico
- U Qué se guarda (`password_hash`) y qué jamás aparece en el contrato de salida

23.4. La explicación visual



El hash contiene *todo* empaquetado: `$2b$10$` dice "bcrypt, costo 10" y el salt va embebido — una sola columna basta.

23.5. Implementación

`POST /auth/register` real: validar (cap. 4) → hashear → persistir (cap. 10). La línea importante en cada stack es **una sola**.

23.5.1. TypeScript (`bcrypt`)

```ts
import bcrypt from "bcrypt";

// register - hash before persisting, the plaintext dies here
export async function registerUser(email: string, password: string) {
  const passwordHash = await bcrypt.hash(password, 10);  // cost factor 10
  const { rows } = await pool.query(
    `INSERT INTO users (email, password_hash) VALUES ($1, $2)
     RETURNING public_id, email`,          // never RETURNING password_hash
    [email, passwordHash],
  );
  return rows[0];
}

// login - compare what they typed against the stored hash
export async function verifyUser(email: string, password: string) {
  const { rows } = await pool.query(
    "SELECT id, password_hash FROM users WHERE email = $1", [email],
  );
  const user = rows[0];
  if (!user) return null;                                  // no user → null
  const ok = await bcrypt.compare(password, user.password_hash);
  return ok ? user : null;                                 // same answer either way
}
```

23.5.2. Python (`bcrypt`)

```py
import bcrypt

def register_user(email: str, password: str) -> dict:
    password_hash = bcrypt.hashpw(
        password.encode(), bcrypt.gensalt(rounds=10)   # salt + cost built in
    ).decode()
    row = conn.execute(
        """INSERT INTO users (email, password_hash) VALUES (%s, %s)
```

```
            RETURNING public_id, email""",              # never the hash
        (email, password_hash),
    ).fetchone()
    return row

def verify_user(email: str, password: str):
    user = conn.execute(
        "SELECT id, password_hash FROM users WHERE email = %s", (email,),
    ).fetchone()
    if not user:
        return None
    ok = bcrypt.checkpw(password.encode(), user["password_hash"].encode())
    return user if ok else None                          # same answer either way
```

23.5.3. Go (`x/crypto/bcrypt`)

```
func registerUser(ctx context.Context, email, password string) (User, error) {
    hash, err := bcrypt.GenerateFromPassword([]byte(password), 10) // cost 10
    if err != nil {
        return User{}, err
    }
    var u User
    err = pool.QueryRow(ctx,
        `INSERT INTO users (email, password_hash) VALUES ($1, $2)
         RETURNING public_id, email`, email, string(hash),  // never the hash
    ).Scan(&u.PublicID, &u.Email)
    return u, err
}

func verifyUser(ctx context.Context, email, password string) (*User, error) {
    var u User
    err := pool.QueryRow(ctx,
        "SELECT id, password_hash FROM users WHERE email = $1", email,
    ).Scan(&u.ID, &u.PasswordHash)
    if err != nil {
        return nil, nil                                  // no user → nil
    }
    if bcrypt.CompareHashAndPassword([]byte(u.PasswordHash), []byte(password)) != nil {
        return nil, nil                                  // same answer either way
    }
    return &u, nil
}
```

Tres detalles que valen oro:

1. **El handler solo ve éxito/fracaso.** `verifyUser` devuelve el usuario o `null` — *nunca* "usuario no existe" vs "contraseña mala" por separado. Esa distinción le dice al atacante qué emails están

registrados (enumeración de usuarios, cap. 17).
2. **`RETURNING/SELECT` jamás incluye `password_hash` hacia afuera.** Se lee para comparar y muere en el servicio — el `UserPublic` del contrato no tiene esa columna.
3. **Costo 10 hoy.** El factor es configurable: súbelo cuando el hardware lo tolere (cada +1 duplica el trabajo). Se cronometra en el mini reto.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Prompt listo para tu AI

```
Implementa registro y verificación de contraseña en mi API
[Express / FastAPI / Go]. Requisitos: bcrypt con costo 10 (hashpw/
GenerateFromPassword, NO sha256 ni MD5), la contraseña en claro nunca
se guarda ni se loguea, password_hash nunca sale en ninguna respuesta
JSON, el login devuelve el mismo error genérico para "usuario no existe"
y "contraseña incorrecta" (sin enumeración de usuarios), y las queries
van parametrizadas. Explícame por qué bcrypt lento es mejor que un hash
rápido para este caso.
```

23.6. ¿Por qué bcrypt y no otra cosa?

Lo único que necesitas llevarte: guardar contraseñas = guardar *verificadores*, no secretos. El hash bcrypt es un verificador: sirve para comprobar, no para recuperar. Tu base de datos puede publicarse entera mañana y las contraseñas siguen protegidas por el costo de calcularlas una a una.

Otras cimentaciones: algoritmos de hashing

Algoritmo	Qué es	Estado
bcrypt	El clásico probado (1999), salt+costo	Default sano — el del libro
argon2	El moderno: también resiste GPU (memoria dura)	Recomendado OWASP; librerías menos universales

scrypt	Similar a argon2, memoria dura	Sólido, menos común
PBKDF2	Iteraciones de hash estándar NIST	Aceptable; para compliance
MD5/SHA-256 solo	Hash rápido sin salt	**Prohibido para contraseñas** — GPUs los muelen

Si tu stack tiene argon2 maduro, es la elección 2024+. bcrypt sigue siendo defendible y omnipresente — lo que NO se debate es "hash lento con salt" vs cualquier otra cosa.

23.7. Errores comunes

Error	Por qué pasa	Fix
Contraseña en claro en la tabla	"Es la forma obvia"	`password_hash` bcrypt — la clave nunca persiste
`sha256(password)`	"Es un hash" — sí, rápido: 10^9 intentos/segundo en GPU	bcrypt/argon2: lento *es* el punto
Mismo salt para todos / sin salt	Rainbow tables lo resuelven en una query	bcrypt genera salt por usuario automáticamente
Error distinto para email vs password	"Email no registrado" confirma qué emails existen	Mismo "credenciales inválidas" para ambos casos
`password_hash` en el JSON del usuario	El SELECT * del cap. 9	Columnas explícitas + contrato de salida (cap. 5)
Loguear la contraseña "para debug"	El log vive para siempre y lo lee medio equipo	La contraseña no toca `console.log` jamás

23.8. Buenas prácticas

- **El hash se calcula en el servicio, no en el handler** — el handler traduce HTTP; la regla "nunca persistir en claro" es de negocio.
- **Compara, no decidas:** `bcrypt.compare` maneja timing seguro — no compares hashes con `===` (timing attacks son reales en teoría; usar `compare` cuesta lo mismo).
- **Costo ajustable por config**, no quemado en código — subirlo en prod no debería requerir refactor.
- **Límite de longitud razonable** (ej. 72 bytes es el máximo real de bcrypt): valida `password.length <= 72` o trunca conscientemente.

i EXTRA · El mapa de hashing y seguridad web

EXTRA Los roadmaps piden "entender la seguridad web" — el mapa completo, ubicado:

- **MD5** — roto; nunca para contraseñas (solo checksums)
- **Familia SHA** — integridad y firmas; demasiado rápida para passwords
- **scrypt** — alternativa seria a bcrypt, también lenta a propósito
- **bcrypt** — la de este capítulo: lenta + salt incorporado
- **HTTPS / SSL / TLS** — el canal cifrado (cap-33 lo sirve)
- **CORS** — política de orígenes del navegador (cap-18)

Regla que no caduca: contraseñas → bcrypt/scrypt/argon2; integridad → SHA/HMAC; MD5 → solo para verificar que un archivo no se corrompió.

23.9. Ejercicio

1. Explica en una frase por qué cifrar contraseñas es *peor* que hashearlas, aunque el cifrado suene más seguro.
2. Dos usuarios eligen la misma contraseña `hunter2`. ¿Qué ven en la columna `password_hash`? ¿Por qué?
3. Escribe el `verify` de tu stack que devuelva la *misma* respuesta para usuario inexistente y contraseña mala.

Soluciones

1. El cifrado es reversible: existe una clave que deshace todo — y quien roba tu BD probablemente roba también esa clave del servidor. El hash no tiene clave que robar: la única forma de "romperlo" es adivinar la contraseña original, intento a intento.

2. Dos hashes **completamente distintos** — bcrypt genera un salt aleatorio por cada `hash`/`hashpw`/`GenerateFromPassword`. Dos claves iguales producen hashes irreconocibles entre sí, así que la filtración no revela quiénes comparten contraseña.

3. TypeScript:

```ts
export async function verifyUser(email: string, password: string) {
  const { rows } = await pool.query(
    "SELECT id, password_hash FROM users WHERE email = $1", [email]);
  const user = rows[0];
  if (!user) return null;
  const ok = await bcrypt.compare(password, user.password_hash);
  return ok ? user : null;   // both failures return null
}
```

   El handler responde `401 {error: "invalid credentials"}` en ambos casos — el atacante no puede distinguirlos.

23.10. Mini reto

Cronometrar bcrypt con distintos costos y justificar la elección.

Soluciones

```
// time each cost factor - run once, pick deliberately
for (const cost of [8, 10, 12, 14]) {
  const t = Date.now();
  await bcrypt.hash("benchmark", cost);
  console.log(cost, Date.now() - t, "ms");
}
// typical: 8 25ms  10 80ms  12 320ms  14 1.3s
```

La justificación: busca el costo donde **tu login siga siendo instantáneo para un humano (~100-300ms) pero cada intento del atacante cueste máximo**. Costo 10-12 es el punto dulce actual. Detalle importante: el costo va *embebido en el hash* (`$2b$10$...`), así que hashes viejos siguen verificando aunque subas el factor — y puedes rehashear en el próximo login exitoso del usuario.

23.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

23.12. Lo que deberías saber hacer ahora

- ☐ Explicar hash vs cifrado y por qué "lento" es la característica
- ☐ Implementar register + verify con bcrypt en tu stack
- ☐ Mantener `password_hash` fuera de todo contrato de salida
- ☐ Dar el mismo error para usuario inexistente y clave mala

24 15. Necesitamos que el servidor recuerde quién eres

HTTP no tiene memoria: cada request es un extraño

> **En una frase**
>
> El login funcionó — y el siguiente request llega como de un extraño: **HTTP no tiene memoria**. La solución es un *token*: tras el login el servidor emite una credencial que el cliente presenta en cada request. Este capítulo construye el login de Taskflow: JWT firmado viajando en cookie httpOnly, con access corto + refresh largo.

24.1. El problema

El usuario hizo `POST /auth/login` con su email y clave — respondiste 200. Ahora pide `GET /tasks`. ¿Quién es? El request no trae memoria del anterior: HTTP es *stateless* por diseño (cada petición es independiente — es lo que hace que escale). Algo tiene que decir "soy yo, el del login de hace un segundo". Dos modelos clásicos:

> **Explicación de: memoria (RAM)**
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

> **Explicación de: autenticación**
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

- **Sesión con estado:** el servidor genera un id (`sess_abc123`), lo guarda en una tabla `sessions`, y el cliente lo presenta como una *pulsera de discoteca*. El servidor consulta la tabla cada vez.
- **Token sin estado (JWT):** el servidor firma una credencial que dice *quién eres* dentro — el cliente la lleva, el servidor solo verifica la firma. No hay tabla que consultar.

El trade-off real: el JWT no necesita consulta (escala gratis) pero **no se puede revocar** — quien lo roba lo usa hasta que expire. La sesión en BD se revoca con `DELETE`, pero cuesta una query por request. El cap. 18 combina ambos; aquí entendemos el JWT primero.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

> Explicación de: query / consulta
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

24.2. Cómo lo resuelve un equipo

El flujo completo que construye este capítulo:

24.3. Conceptos nuevos

- U JWT por dentro: `header.payload.signature` — **firmado, NO cifrado** (cualquiera lo lee, nadie lo falsifica)
- U Claims: `sub` (quién), `exp` (cuándo muere), `iat` (cuándo nació) — el payload es público
- U Cookie `httpOnly` vs `localStorage`: XSS, CSRF, `SameSite`, `Secure`
- U Access token corto (15min) + refresh token largo (30d): expiración corta sin re-login constante
- TS `jose` · Py `python-jose` · Go `golang-jwt` — claims idénticos en los tres

24.4. La explicación visual

Un JWT real son tres segmentos base64 separados por puntos:

```
eyJhbGciOiJIUzI1NiJ9 . eyJzdWIiOiI0MiIsImV4cCI6MTcxNzAwMDAwMH0 . SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJVad
  header             payload (claims)                signature
 {"alg":"HS256"}         {"sub":"42","exp":...,"iat":...}        HMAC(secret, header.payload)
```

Pégalo en jwt.io y lo lees entero — **sin la clave**. La firma no oculta el contenido; garantiza que *nadie lo modificó*. Por eso la regla: el payload nunca lleva contraseñas ni datos sensibles — es una tarjeta de identidad legible, no un sobre sellado.

24.5. Implementación

Login que emite el JWT en cookie `httpOnly` + `GET /me` que lo verifica:

> Explicación de: cookie
>
> Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

24.5.1. TypeScript (`jose`)

```ts
import { SignJWT, jwtVerify } from "jose";
const secret = new TextEncoder().encode(process.env.JWT_SECRET);

// login → set the cookie
app.post("/auth/login", async (req, res) => {
  const user = await verifyUser(req.body.email, req.body.password); // cap. 14
  if (!user) throw new UnauthorizedError("invalid credentials");

  const token = await new SignJWT({})                 // payload = claims
    .setSubject(String(user.id))                      // sub: who
    .setIssuedAt()                                    // iat: when born
    .setExpirationTime("15m")                         // exp: when it dies
    .sign(secret);

  res.cookie("token", token, {
    httpOnly: true,     // JS can't read it → XSS can't steal it
    secure: true,       // HTTPS only
    sameSite: "lax",    // not sent cross-site → CSRF barrier
    maxAge: 15 * 60 * 1000,
  });
  res.json({ user: { id: user.public_id, email: user.email } });
});

// GET /me → verify the token from the cookie
```

```
app.get("/me", async (req, res) => {
  const { payload } = await jwtVerify(req.cookies.token, secret);
  res.json({ userId: payload.sub });                // sub = your identity
});
```

24.5.2. Python (python-jose + FastAPI)

```
from jose import jwt
SECRET = os.environ["JWT_SECRET"]

@app.post("/auth/login")
def login(body: LoginIn, response: Response):
    user = verify_user(body.email, body.password)        # cap. 14
    if not user:
        raise UnauthorizedError("invalid credentials")

    token = jwt.encode(
        {"sub": str(user["id"]),
         "iat": datetime.now(timezone.utc),
         "exp": datetime.now(timezone.utc) + timedelta(minutes=15)},
        SECRET, algorithm="HS256",
    )
    response.set_cookie(
        "token", token,
        httponly=True, secure=True, samesite="lax",
        max_age=15 * 60,
    )
    return {"user": {"id": user["public_id"], "email": user["email"]}}

@app.get("/me")
def me(request: Request):
    payload = jwt.decode(request.cookies["token"], SECRET, algorithms=["HS256"])
    return {"user_id": payload["sub"]}
```

24.5.3. Go (golang-jwt)

```
func login(w http.ResponseWriter, r *http.Request) {
    var in struct{ Email, Password string }
    json.NewDecoder(r.Body).Decode(&in)
    user, err := verifyUser(r.Context(), in.Email, in.Password) // cap. 14
    if err != nil || user == nil {
        writeError(w, 401, "invalid credentials")
        return
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
```

```
        "sub": strconv.FormatInt(user.ID, 10),
        "iat": time.Now().Unix(),
        "exp": time.Now().Add(15 * time.Minute).Unix(),
    })
    signed, _ := token.SignedString([]byte(os.Getenv("JWT_SECRET")))

    http.SetCookie(w, &http.Cookie{
        Name: "token", Value: signed,
        HttpOnly: true, Secure: true, SameSite: http.SameSiteLaxMode,
        MaxAge: 15 * 60, Path: "/",
    })
    writeJSON(w, 200, map[string]any{"user": map[string]any{
        "id": user.PublicID, "email": user.Email,
    }})
}
```

El detalle clave del diseño: **la cookie viaja sola.** Con `httpOnly`, el navegador la adjunta automáticamente a cada request al dominio y *ningún JavaScript puede leerla* — un script malicioso inyectado (XSS) no puede robar el token. El precio: como viaja automática, un sitio maligno podría hacer que tu navegador la envíe sin saberlo (CSRF) — de ahí `sameSite: "lax"` y el cap. 18 profundiza.

¿Y el access + refresh? El access de 15 min muere rápido (si lo roban, dura poco); el refresh de 30 días solo sirve para *pedir access nuevos* y vive en otra cookie. El usuario no re-loguea cada 15 min, pero el token robado caduca rápido. El cap. 18 le suma revocación real.

Prompt listo para tu AI

```
Implementa login con JWT en mi API [Express / FastAPI / Go]. Requisitos:
token con claims sub (user id), iat, exp=15min, firmado HS256 con
JWT_SECRET de variable de entorno, enviado en cookie httpOnly + Secure
+ SameSite=lax (NO en el body ni en localStorage), endpoint GET /me que
verifique la cookie y devuelva el usuario. Explica: por qué httpOnly y
no localStorage, qué puede leer un atacante del JWT, y cómo añadirías
un refresh token de 30 días en cookie separada.
```

24.6. ¿Por qué cookie y no localStorage?

Lo único que necesitas llevarte: `localStorage` es legible por cualquier JS que corra en tu página — incluido el script que un XSS inyecte en un comentario mal sanitizado. La cookie `httpOnly` es invisible para JS: el XSS no puede robarla. A cambio, la cookie viaja sola (riesgo CSRF, mitigado con `SameSite`). Ninguna opción es gratis — `httpOnly` + `SameSite` + `Secure` es el balance profesional.

Otras cimentaciones: dónde vive la sesión

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
JWT en cookie httpOnly	Stateless, firma, expira	No revocable solo; CSRF a vigilar	APIs + SPA, el default moderno
Sesión en BD/Redis	id opaco, consulta por request	Una query/lookup por request	Revocación instantánea prioritaria
JWT en localStorage	Simple de implementar	XSS lo roba todo	Solo para aprender — no prod
Access + refresh	Dos tokens, dos vidas	Más complejidad (rotation, cap. 18)	Todo lo serio

24.7. Errores comunes

Error	Por qué pasa	Fix
JWT en localStorage	Es lo que enseña el tutorial viejo	Cookie httpOnly
Datos sensibles en el payload	"Está cifrado" — NO, está firmado	El payload es público: solo `sub`/claims inocuos
`exp` de días en el access token	"Para que no re-loguee"	Access corto + refresh largo
Token sin verificar firma real	Decodificar sin `verify` — te puedes inventar el payload	Siempre `jwtVerify`/`decode` con la clave
`JWT_SECRET` débil en el repo	HS256 se rompe por fuerza bruta si el secreto es "1234"	Secreto aleatorio largo en env/secret manager

24.8. Buenas prácticas

- **Claims mínimos:** `sub`, `exp`, `iat` bastan — roles y permisos se consultan frescos (cap. 17) o van en claims que cambian poco.
- **`Secure` siempre en prod** — sin HTTPS la cookie viaja en claro.
- **El secreto vive en el secret manager** (cap. N4), no en el repo.
- **Logout = borrar la cookie** (`maxAge: 0`) — con JWT stateless el token sigue "válido" hasta `exp`, pero sin cookie no viaja. La revocación real llega en cap. 18.

24.9. Ejercicio

1. Decodifica a mano (jwt.io o `atob`) un JWT y nombra sus tres partes y qué garantiza cada una.
2. ¿Por qué `sub` guarda el `id` interno y no el `public_id`? (pista: ¿quién lo lee?)
3. Diseña las dos cookies (access + refresh): tiempos y para qué sirve cada una.

Soluciones

1. `header` (algoritmo), `payload` (claims: sub/exp/iat — legible por cualquiera), `signature` (HMAC del contenido: garantiza integridad — quien lo modifique sin el secreto produce firma inválida).
2. El `sub` lo lee *el servidor* para buscar al usuario en la BD — el id interno es la FK natural, rápida y estable. El `public_id` es para URLs de la API (qué expone al mundo); dentro del token, lo interno es correcto porque el token es un contrato servidor servidor.
3. Access: cookie httpOnly, 15 min — la credencial de cada request. Refresh: cookie httpOnly, 30 días, path `/auth/refresh` (solo viaja ahí) — renueva el access sin re-login. Si el access se roba, dura minutos; el refresh se rota en cada uso (cap. 18).

24.10. Mini reto

"Robar" tu propio token desde la consola del navegador y explicar por qué `httpOnly` lo impide (o no).

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Soluciones

Abre DevTools → Console → `document.cookie`: la cookie `token` **no aparece** — `httpOnly` la excluye de la API de cookies de JS. Eso es el punto: un XSS que inyecte `<script>` no puede exfiltrarla por `document.cookie`. Pero honestidad: *sí* aparece en DevTools → Application → Cookies (tú eres el usuario legítimo mirando tu propia cookie) y viaja en cada request. `httpOnly` protege contra *robo por script*, no contra acceso físico a tu navegador. La defensa completa es el combo: httpOnly (anti-XSS) + SameSite (anti-CSRF) + Secure (anti-red) + exp corto (anti-robo).

24.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera

es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: foreign key / clave foránea

Una **foreign key** es un puntero verificado: `tasks.project_id` apunta a `projects.id` y la BD *garantiza* que el proyecto existe. Es la diferencia entre "el dato dice 5" y "el dato apunta a algo real".

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

24.12. Lo que deberías saber hacer ahora

- ☐ Explicar por qué HTTP necesita tokens (stateless)
- ☐ Nombrar las 3 partes del JWT y qué garantiza la firma vs el cifrado
- ☐ Emitir JWT en cookie httpOnly y verificarlo en un endpoint
- ☐ Justificar httpOnly vs localStorage y el par access/refresh

25 16. Necesitamos saber quién eres en cada request

¿Copiar la verificación del JWT en 20 endpoints? No.

> **En una frase**
>
> Veinte endpoints protegidos no pueden llevar veinte copias de "verifica el token". El patrón que lo resuelve es **middleware**: una función que se ejecuta *antes* de tu handler, resuelve quién eres, y te entrega el usuario listo — o corta el request con un 401. En este capítulo `get_current_user` nace y protege Taskflow sin duplicar una línea.

25.1. El problema

Cada endpoint privado necesita lo mismo: leer la cookie, verificar la firma, extraer `sub`, buscar el usuario, responder 401 si algo falla. Copiarlo es más que feo — es **inconsistente**: el endpoint donde olvides pegarlo queda público, y el día que cambie la verificación hay que tocar 20 archivos. La verificación de auth es un *cross-cutting concern*: una capa que atraviesa todo, no parte de ningún handler.

> **Explicación de: firma / HMAC**
>
> Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

> **Explicación de: autenticación**
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

> **Explicación de: cookie**
>
> Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

25.2. Cómo lo resuelve un equipo

El patrón universal: la lógica compartida se **envuelve alrededor** del handler, no dentro. Cada stack tiene su dialecto:

- **Express:** `app.use(auth)` — función que recibe `next()` y decide si la cadena continúa.
- **FastAPI:** `Depends(get_current_user)` — declaras la dependencia en la firma; el framework la resuelve y *inyecta* el resultado.

> Explicación de: framework
>
> Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

- **Go:** `auth(handler)` — una función que devuelve un handler envuelto: composición explícita, sin magia.

Y la pregunta gemela: una vez verificado, **¿dónde vive el usuario** para que el handler lo use sin otra query? Cada stack tiene su canal: `res.locals` (Express), el retorno del `Depends` (FastAPI), el `context.Context` del request (Go).

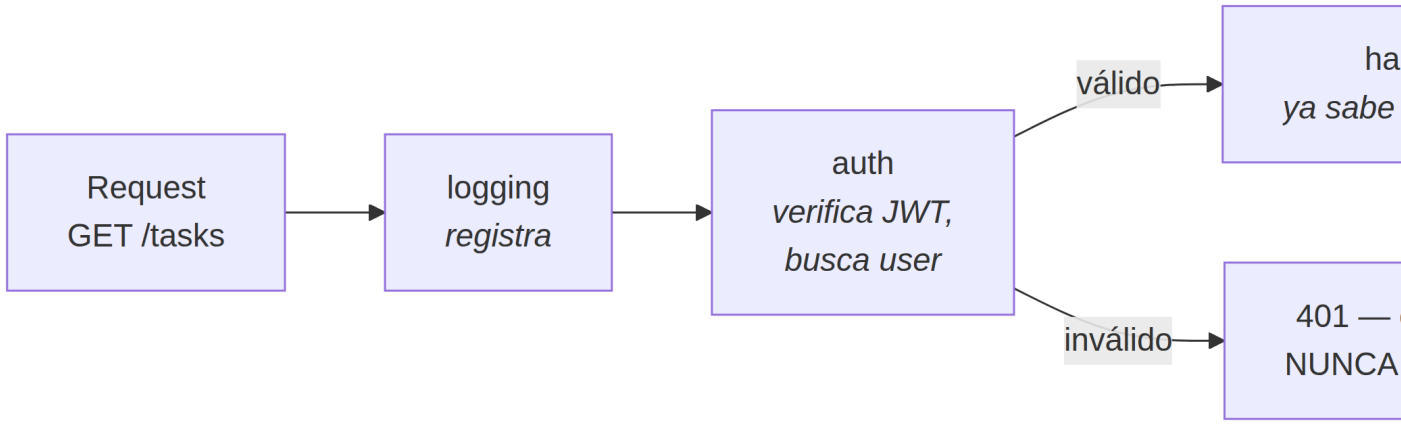
> Explicación de: query / consulta
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

25.3. Conceptos nuevos

- U El patrón middleware/pipeline: código que corre *antes* del handler y decide si este existe
- TS `app.use(auth)` — la función que envuelve la siguiente
- Py `Depends(...)` — inyección declarativa: pides el usuario en la firma
- Go `middleware(handler)` — composición: handler que devuelve handler
- U Propagar el usuario: `res.locals` vs retorno de DI vs `context.Context`
- U El orden del pipeline: auth antes del handler, logging antes de todo

25.4. La explicación visual

El pipeline completo de un request protegido:



El middleware es una *aduana*: el handler detrás puede asumir "quien llegó aquí ya mostró su identificación".

25.5. Implementación

El mismo patrón en tres dialectos — verifica el token, carga el usuario, lo entrega al handler:

> Explicación de: token
>
> Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

25.5.1. TypeScript (`app.use` + `res.locals`)

```
// auth.middleware.ts - the customs officer
export async function auth(req: Request, res: Response, next: NextFunction) {
  try {
    const { payload } = await jwtVerify(req.cookies.token, secret);
    const user = await getUserById(Number(payload.sub));
    if (!user) throw new Error("no user");
    res.locals.user = user;          // where the user lives for this request
    next();                          // proceed to the handler
  } catch {
    throw new UnauthorizedError("invalid or expired token");  // 401, handler skipped
  }
}

// routes.ts - one word protects the whole group
app.use("/tasks", auth);
app.get("/tasks", async (req, res) => {
```

```
  const user = res.locals.user;      // already verified, already loaded
  res.json(await listTasks(user.id));
});
```

`res.locals` es el bolsillo del request: vive lo que el middleware dejó. El handler nunca toca el JWT — solo pregunta "¿quién soy?".

25.5.2. Python (`Depends`)

```
# deps.py - the dependency FastAPI resolves for you
def get_current_user(request: Request) -> User:
    token = request.cookies.get("token")
    try:
        payload = jwt.decode(token, SECRET, algorithms=["HS256"])
        user = get_user_by_id(int(payload["sub"]))
    except Exception:
        raise UnauthorizedError("invalid or expired token")
    if not user:
        raise UnauthorizedError("invalid or expired token")
    return user                          # whatever you return gets INJECTED

# routes.py - asking for it in the signature IS the protection
@app.get("/tasks")
def list_tasks(user: User = Depends(get_current_user)):
    return list_tasks_service(user.id)   # user already verified + loaded
```

Esto es **inyección de dependencias**: `Depends(get_current_user)` dice "para ejecutar este handler, primero corre esa función y dame su resultado". FastAPI la llama; si lanza `UnauthorizedError`, el handler ni se ejecuta. La protección está en la *firma* — imposible olvidarla.

25.5.3. Go (`middleware(handler)` — composición)

```
// auth.go - a function that wraps a handler
func auth(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        cookie, err := r.Cookie("token")
        if err != nil {
            writeError(w, 401, "invalid or expired token")
            return                       // handler never runs
        }
        claims := jwt.MapClaims{}
        _, err = jwt.ParseWithClaims(cookie.Value, claims,
            func(t *jwt.Token) (any, error) { return []byte(os.Getenv("JWT_SECRET")), nil })
        if err != nil {
```

```
            writeError(w, 401, "invalid or expired token")
            return
        }
        sub, _ := strconv.ParseInt(claims["sub"].(string), 10, 64)
        user, err := getUserByID(r.Context(), sub)
        if err != nil || user == nil {
            writeError(w, 401, "invalid or expired token")
            return
        }
        ctx := context.WithValue(r.Context(), userKey, user)  // user travels in ctx
        next(w, r.WithContext(ctx))      // proceed to the handler
    }
}

// routes.go - wrap it, one word
mux.HandleFunc("GET /tasks", auth(listTasks))

func listTasks(w http.ResponseWriter, r *http.Request) {
    user := r.Context().Value(userKey).(*User)   // already verified
    writeJSON(w, 200, mustListTasks(r.Context(), user.ID))
}
```

Go no tiene `app.use` ni `Depends` — tiene funciones que devuelven funciones. `auth(listTasks)` *es* el handler: uno que hace aduana y delega. Todo explícito — nada se esconde.

El patrón en una línea: **la verificación vive en un solo lugar, y proteger un endpoint nuevo cuesta una palabra** (`Depends`, `app.use`, `auth(...)`). Cap. 17 le da argumentos — middlewares con parámetros.

Explicación de: parámetro / argumento

El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

Prompt listo para tu AI

```
Implementa el middleware de autenticación para mi API
[Express / FastAPI / Go]. Requisitos: verifica el JWT de la cookie
'token' (firma HS256 con JWT_SECRET de env), carga el usuario completo
desde la BD por su sub, lo propaga al handler por el mecanismo idiomático
(res.locals / Depends / context.Context), y responde 401 genérico sin
ejecutar el handler si falla. Debe proteger un grupo de rutas con UNA
línea (app.use / Depends / wrapper). Explícame el orden del pipeline y
dónde pondrías logging y rate-limit relativos a auth.
```

25.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres resuelven el mismo problema — "código antes del handler con usuario resuelto" — con el mecanismo que su lenguaje hace natural: Express encadena funciones, FastAPI declara dependencias, Go compone handlers. Si entiendes el patrón, cada dialecto se aprende en una tarde; si solo memorizas el dialecto, cada framework parece magia nueva.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Detalle: el orden del pipeline importa

```
request → logging → CORS → rate-limit → auth → handler
```

- `logging` primero: quieres registrar *todo*, incluso los 401.
- `rate-limit` antes de `auth`: no gastes `bcrypt.compare`/JWT verify en requests que vas a rechazar de todas formas (cap. 18).
- `auth` justo antes del handler protegido: lo público (`/health`, `/auth/login`) queda fuera.
- Los errores de cualquier capa caen al mapper del cap. 6.

En Express el orden es literal (el orden de `app.use`); en FastAPI es el orden de los `Depends`/middlewares; en Go es cómo anidas los wrappers: `logging(cors(rateLimit(auth(handler))))` — se lee de afuera hacia adentro.

25.7. Errores comunes

Error	Por qué pasa	Fix
Verificación copiada por endpoint	"Solo era un endpoint más"	Un middleware; el olvido cuesta un endpoint público
Auth que consulta la BD… por cada request… en endpoints que no la necesitan	El middleware carga el user aunque el handler no lo use	OK si es barato; si no, resuelve `sub` en middleware y carga el user en el servicio que lo necesita
Middleware registrado DESPUÉS de las rutas	Orden del pipeline al revés	`app.use(auth)` antes de las rutas protegidas
El user viaja en variable global	`let currentUser = ...` → requests concurrentes se mezclan	`res.locals`/`Depends`/`ctx` — por request, nunca global

Error	Por qué pasa	Fix
Rutas públicas detrás del auth	`/health` pidiendo JWT	Agrupa: público primero, `use(auth)` solo en el grupo privado

25.8. Buenas prácticas

- **El middleware resuelve identidad, el handler decide permisos** — separa autenticación (quién eres, cap. 16) de autorización (qué puedes, cap. 17).

> Explicación de: autorización / permisos
>
> La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

- **El usuario propagado es el *mínimo* necesario:** id + lo que el handler consulta siempre — no la fila entera si no hace falta.
- **401 temprano:** falla rápido, antes de tocar la BD si la firma ya es inválida.
- **Nombra la dependencia por lo que devuelve** (`get_current_user`), no por lo que hace (`check_token`) — el handler que la pide se lee solo.

25.9. Ejercicio

1. Dibuja el pipeline de `POST /tasks` en tu stack, marcando dónde se puede cortar con 401.
2. Escribe la línea exacta que protege `GET /projects` en tu stack.
3. ¿Por qué `currentUser` en variable global es un bug de concurrencia?

> Explicación de: variable
>
> Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

> Soluciones
>
> 1. `POST /tasks` → logging → rate-limit → **auth** (aquí muere con 401 si el token falta/expira) → validación del body (422) → handler → servicio → BD.
> 2. TS: `app.get("/projects", auth, listProjects)` (o `app.use("/projects", auth)` para todo el grupo). Py: `def list_projects(user: User = Depends(get_current_user))`. Go: `mux.HandleFunc("GET /projects", auth(listProjects))`.
> 3. Porque el servidor atiende muchos requests *a la vez*: mientras la request de Ana está en el handler, la de Bruno sobreescribe `currentUser` — y Ana ejecuta acciones como Bruno. El usuario es estado *del request*: vive en `res.locals`, el retorno del `Depends` o el

`context.Context`, que son por-request por diseño.

25.10. Mini reto

Proteger un endpoint nuevo en una sola línea — y entender por qué funciona.

Soluciones

Go:

```
mux.HandleFunc("DELETE /tasks/{id}", auth(deleteTask))
```

Funciona porque `auth` es una función que *devuelve un handler*: el handler resultante hace la aduana primero y, solo si pasa, llama a `deleteTask`. La línea parece declarativa ("este endpoint requiere auth") pero es composición real de funciones — `auth(deleteTask)` *es* un `http.HandlerFunc` válido. La misma línea en FastAPI es el `Depends(get_current_user)` en la firma; en Express, poner `auth` entre la ruta y el handler. Una palabra, toda la protección.

25.11. Vocabulario técnico del capítulo

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña` →$2b10…‘. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

25.12. Lo que deberías saber hacer ahora

- ☐ Explicar el patrón middleware y por qué auth no se copia
- ☐ Implementar la verificación como middleware/dependency/wrapper en tu stack
- ☐ Propagar el usuario por `res.locals` / `Depends` / `context.Context`
- ☐ Ordenar el pipeline: logging, CORS, rate-limit, auth, handler
- ☐ Proteger rutas públicas y privadas sin que se contaminen

26 17. Necesitamos que autenticado no signifique puede todo

Autenticación autorización

> En una frase
>
> El cap. 16 respondió *quién eres*; falta *qué puedes tocar*. Ahora mismo cualquier usuario logueado puede `DELETE /tasks/{id}` de otro — autenticado sí, autorizado no. Este capítulo implementa **ownership** ("¿esta tarea es tuya?") y **roles** (viewer/editor/admin), y la decisión fina: cuándo responder 403 y cuándo responder 404 *a propósito*.

26.1. El problema

Autenticación y autorización se confunden porque suenan parecido — son dos preguntas distintas:

> Explicación de: autorización / permisos
>
> La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

- **Autenticación (cap. 15–16):** ¿quién eres? → 401 si no sabemos.
- **Autorización (este cap.):** ya sé quién eres — ¿esto es tuyo? → 403 si sabemos y no puedes; **404 si ni siquiera debes saber que existe**.

El bug clásico se llama **IDOR** (Insecure Direct Object Reference): la URL dice `/tasks/41`, cambias a `42`, y ves la tarea de otro. La API verificó *quién eres* pero nunca *de quién es eso*. En Taskflow: Ana abre DevTools, cambia el id, borra las tareas de Bruno. La autenticación funcionó perfecto — eso es lo que la hace peligrosa.

> Explicación de: API
>
> Una **API** (Application Programming Interface) es el menú de un programa: la lista de operaciones que otros programas pueden pedirle. Tu frontend no habla con la base de datos — le pide cosas a la API, y la API decide.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

26.2. Cómo lo resuelve un equipo

Dos niveles, en orden de lo que Taskflow necesita:

1. **Ownership:** la regla `task.user_id == current_user.id`. El 90 % de la autorización real es esto — "solo tocas lo tuyo". Y la forma más robusta no es `if task.user_id != user.id: raise 404` *después* de leerla: es **que la query misma filtre por dueño** — `WHERE public_id = $1 AND user_id = $2`. Si no es tuya, la query devuelve cero filas: para ti, esa tarea *no existe* → 404.
2. **Roles:** cuando recursos se *comparten* (un proyecto con colaboradores), ownership no basta — se modela en la **relación**: `project_members(user_id, project_id, role)` con `viewer/editor/admin`. El rol no vive en el usuario (sería global: "eres admin de TODO"), vive en la membresía: eres editor *de este* proyecto.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: roles / RBAC

Los **roles** agrupan permisos: *viewer* lee, *editor* escribe, *admin* gestiona. RBAC = el permiso depende del rol que tienes *en ese recurso* — puedes ser admin de un equipo y viewer de otro.

Explicación de: recurso (REST)

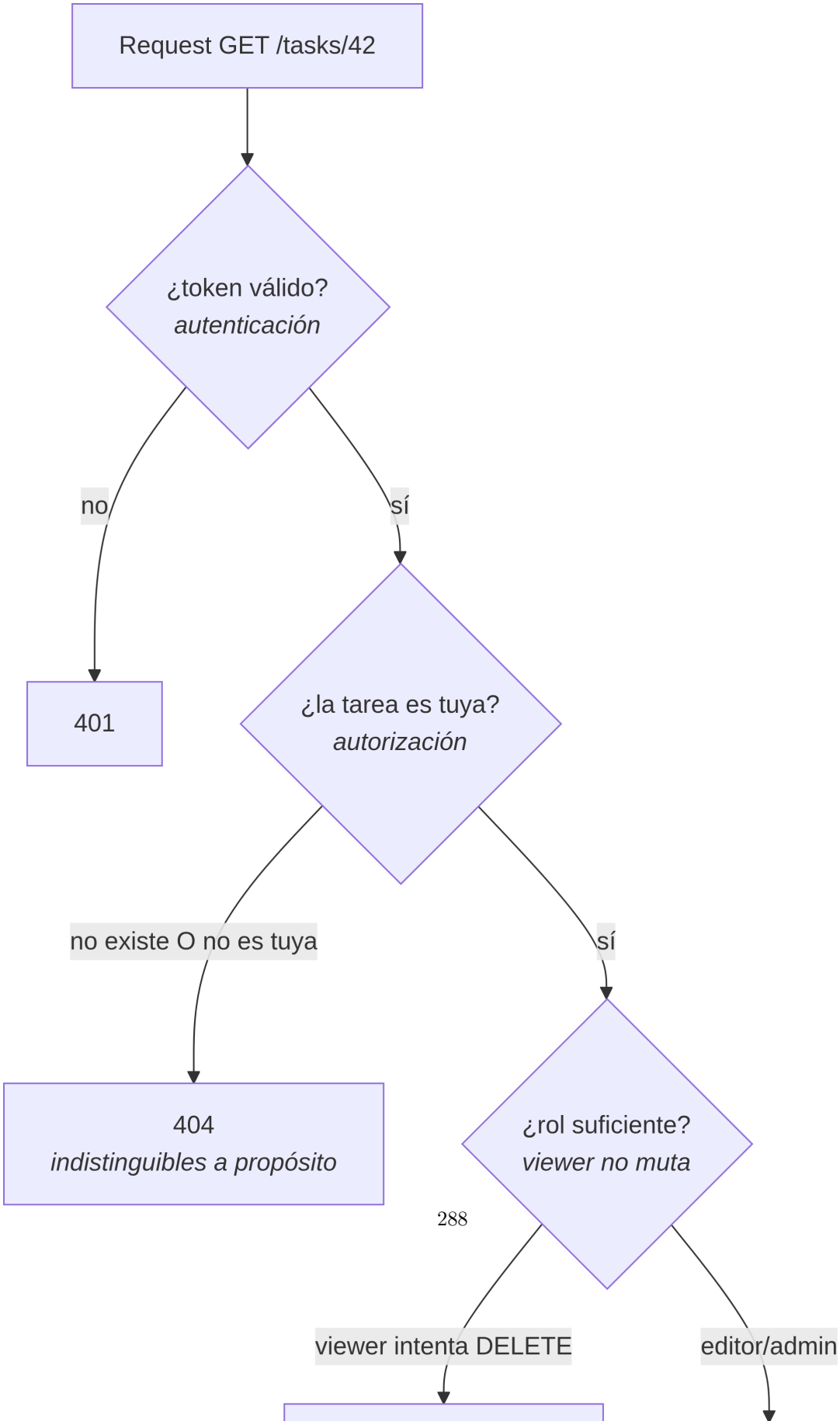
Un **recurso** es la "cosa" que tu API maneja: `tasks`, `users`, `projects`. Las URLs los nombran (`/tasks/5`), los verbos actúan sobre ellos. Diseñar REST = nombrar bien tus recursos.

Y la regla de seguridad que cuesta entender: **recursos ajenos responden 404, no 403.** Si devuelves 403 confirmas "existe pero no es tuyo" — el atacante acaba de aprender que `task/42` existe y puede enumerar cuántas hay. Con 404, la respuesta es idéntica a "no existe": no hay nada que enumerar.

26.3. Conceptos nuevos

- U Ownership: "¿esta tarea es tuya?" — filtrado en la query, no en el if
- U Roles en la relación: `project_members.role` — editor *de esto*, no admin *de todo*
- U 401 vs 403 vs 404: quién eres / lo sé y no puedes / no debes saber que existe
- U Enumeración: qué le regala un 403 al atacante
- TS Py Go Factories: `requireRole("editor")` — funciones que devuelven middleware

26.4. La explicación visual



Fíjate: 404 y 403 *no* son intercambiables — comunican cosas distintas a propósito.

26.5. Implementación

Taskflow agrega proyectos compartidos: `project_members` define tu rol *en ese proyecto*. La protección es un **factory**: `requireRole("editor")` devuelve el middleware/dependency que comprueba ese rol — así la misma lógica sirve para viewer, editor y admin.

> Explicación de: middleware
>
> Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

26.5.1. TypeScript

```
// ownership - the QUERY enforces it, not an if
export async function getTask(userId: number, publicId: string) {
  const { rows } = await pool.query(
    `SELECT public_id, title, done FROM tasks
     WHERE public_id = $1 AND user_id = $2`,   // not yours → zero rows
    [publicId, userId],
  );
  if (!rows[0]) throw new NotFoundError("task not found");  // 404, even if it exists
  return rows[0];
}

// roles - a factory that returns middleware
export function requireRole(minRole: "viewer" | "editor" | "admin") {
  const rank = { viewer: 1, editor: 2, admin: 3 };
  return async (req: Request, res: Response, next: NextFunction) => {
    const { rows } = await pool.query(
      `SELECT role FROM project_members
       WHERE project_id = $1 AND user_id = $2`,
      [req.params.projectId, res.locals.user.id],
    );
    if (!rows[0]) throw new NotFoundError("project not found");      // not a member → 404
    if (rank[rows[0].role] < rank[minRole])
      throw new ForbiddenError("insufficient role");                  // member but weak → 403
    next();
  };
}

app.delete("/projects/:projectId/tasks/:id",
```

```
  auth, requireRole("editor"), deleteTask);
// viewer sees (GET needs only "viewer"), only editor+ mutates
```

26.5.2. Python

```
# ownership - the query enforces it
def get_task(user_id: int, public_id: str) -> Task:
    row = conn.execute(
        """SELECT public_id, title, done FROM tasks
           WHERE public_id = %s AND user_id = %s""",
        (public_id, user_id),
    ).fetchone()
    if not row:
        raise NotFoundError("task not found")     # 404, even if it exists
    return row

# roles - a dependency factory
RANK = {"viewer": 1, "editor": 2, "admin": 3}

def require_role(min_role: str):
    def dep(project_id: str, user: User = Depends(get_current_user)) -> User:
        row = conn.execute(
            """SELECT role FROM project_members
               WHERE project_id = %s AND user_id = %s""",
            (project_id, user.id),
        ).fetchone()
        if not row:
            raise NotFoundError("project not found")          # not a member → 404
        if RANK[row["role"]] < RANK[min_role]:
            raise ForbiddenError("insufficient role")          # member but weak → 403
        return user
    return dep

@app.delete("/projects/{project_id}/tasks/{task_id}")
def delete_task(project_id: str, task_id: str,
                user: User = Depends(require_role("editor"))):
    ...
```

26.5.3. Go

```
// ownership - the query enforces it
func getTask(ctx context.Context, userID int64, publicID string) (Task, error) {
    var t Task
    err := pool.QueryRow(ctx,
        `SELECT public_id, title, done FROM tasks
```

```
         WHERE public_id = $1 AND user_id = $2`, publicID, userID,
    ).Scan(&t.PublicID, &t.Title, &t.Done)
    if errors.Is(err, pgx.ErrNoRows) {
        return Task{}, NotFoundError{"task not found"}    // 404, even if it exists
    }
    return t, err
}

// roles - a function returning middleware
var rank = map[string]int{"viewer": 1, "editor": 2, "admin": 3}

func requireRole(minRole string) func(http.HandlerFunc) http.HandlerFunc {
    return func(next http.HandlerFunc) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
            user := r.Context().Value(userKey).(*User)
            var role string
            err := pool.QueryRow(r.Context(),
                `SELECT role FROM project_members
                 WHERE project_id = $1 AND user_id = $2`,
                r.PathValue("projectId"), user.ID).Scan(&role)
            if errors.Is(err, pgx.ErrNoRows) {
                writeError(w, 404, "project not found")          // not a member → 404
                return
            }
            if rank[role] < rank[minRole] {
                writeError(w, 403, "insufficient role")          // member but weak → 403
                return
            }
            next(w, r)
        }
    }
}

mux.HandleFunc("DELETE /projects/{projectId}/tasks/{id}",
    auth(requireRole("editor")(deleteTask)))
```

La pieza que merece quedarse: **`requireRole("editor")` es una función que devuelve una función.** En los tres dialectos la misma idea — parametrizas *qué* rol sin duplicar *cómo* se comprueba. Es el factory pattern aplicado al middleware, y la primera vez que funciones-que- devuelven-funciones dejan de ser un truco y se vuelven herramienta.

> Explicación de: función
>
> Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Prompt listo para tu AI

```
Implementa autorización en mi API [Express / FastAPI / Go] sobre la auth
del cap anterior. Requisitos: (1) ownership por query - todas las queries
de tasks filtran WHERE user_id, lo ajeno devuelve 404 indistinguible de
inexistente (nunca 403 en recursos ajenos); (2) roles por proyecto en
tabla project_members(user_id, project_id, role viewer|editor|admin) con
un factory requireRole(minRole) reutilizable; (3) viewer puede leer,
solo editor+ muta, no-miembro ni siquiera ve el 404; (4) el 403 reservado
para miembros con rol insuficiente. Dame el código y la tabla de qué
status devuelve cada caso.
```

26.6. ¿Por qué la autorización vive en dos sitios?

Lo único que necesitas llevarte: hay dos fronteras — *la query* (que filtra por dueño, tu primera línea y la más barata) y *el middleware de rol* (que decide capacidades en recursos compartidos). Juntas forman la regla: **nunca confíes en que el cliente no mandará el id de otro** — la autorización es del servidor, siempre, en cada request.

Otras cimentaciones: modelos de autorización

Modelo	Qué es	Qué te cuesta	Cuándo elegirla
Ownership (user_id)	"Solo lo tuyo"	Nada — una columna y un WHERE	El 90 % de los casos; Taskflow hoy
Roles por recurso	`members.role`	Una tabla + checks	Compartir proyectos (este cap.)
Roles globales	`users.role = 'admin'`	Granularidad nula — es todo o nada	Un panel admin, pocos roles
RBAC/permisos	roles → conjuntos de permisos	Tablas de permisos, matriz	Muchos roles, enterprise
ABAC / policies	"puede editar SI es suyo Y está publicado"	Motor de políticas (OPA, Casbin)	Reglas complejas y combinables
OAuth scopes	Permisos en el token de terceros	Delegas el modelo	APIs públicas (cap. 19)

La escalera honesta: ownership → roles por recurso → (si la empresa crece) RBAC/ABAC. No construyas el último escalón primero.

26.7. Errores comunes

Error	Por qué pasa	Fix
IDOR: el id de la URL no se verifica	"Está autenticado, qué más"	`WHERE user_id = $2` en TODA query de recursos
403 en recursos ajenos	"No es tuyo, te aviso" → confirmas que existe	404 — indistinguible de inexistente
Rol global para permisos por recurso	`user.role = 'editor'` edita TODOS los proyectos	El rol vive en la membresía
Chequeo solo en el frontend	El botón "borrar" oculto para viewers — la API sigue abierta	La UI oculta; el servidor *enforcea*
`if task.user_id != user.id` después de leer	Funciona, pero leíste lo ajeno y es un `if` olvidable	La query filtra — el error es estructural, no de memoria

26.8. Buenas prácticas

- **Autorización en la query primero** — `WHERE user_id` es gratis y estructural; los `if` posteriores son plan B.
- **404 para lo ajeno, 403 para lo insuficiente** — aprende la diferencia; es de las pocas decisiones de API que también es seguridad.
- **Roles pequeños y legibles** (`viewer < editor < admin` con ranking) antes que strings sueltos comparados a mano.

> Explicación de: string
>
> Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

- **Audita endpoints al crecer:** cada ruta nueva pasa por el checklist "¿quién puede hacer esto?" — el mini reto formaliza el hábito.

26.9. Ejercicio

1. Escribe la query de `deleteTask` con ownership por filtro (no por if).
2. Un `viewer` comparte proyecto contigo e intenta `DELETE /projects/5/tasks/9`. ¿Qué responde tu API y por qué 403 y no 404 aquí?
3. Un extraño (no miembro) intenta el mismo DELETE. ¿Por qué ahora sí es 404?

> Soluciones
>
> 1.
> ```
> DELETE FROM tasks WHERE public_id = $1 AND user_id = $2;
> -- check rows affected: 0 → NotFoundError (not yours OR doesn't exist)
> ```
>
> 2. **403**: es miembro del proyecto — ya sabes que el proyecto existe y que él pertenece; ocultar

la existencia no aplica. Lo que falla es el *nivel*: viewer no basta para mutar. El 403 dice "te conozco, sé que ves esto, pero tu rol no alcanza".

3. **404**: no siendo miembro, ni siquiera debe confirmar que el proyecto 5 existe — para él la respuesta es idéntica a "proyecto inexistente". Si devolvieras 403, acabas de confirmarle que el proyecto existe: enumeración gratuita.

26.10. Mini reto

Auditar tus endpoints: ¿cuáles dejan adivinar que un recurso existe?

Soluciones

La auditoría pregunta por cada endpoint con `{id}`: *¿qué responde cuando el recurso existe pero no es tuyo?* Endpoints sospechosos típicos:

- `GET /tasks/42` → 403 "forbidden" en vez de 404 → acabas de confirmar que la tarea 42 existe.
- `POST /projects/5/join` → 403 vs 404 distinguibles → sabes qué proyectos existen aunque no puedas entrar.
- Mensajes de error distintos: "task not found" vs "not your task" — diferencia de texto = diferencia de información.

Regla de la auditoría: para un recurso privado, **"no existe" y "no es tuyo" deben producir respuestas byte a byte idénticas** — mismo status, mismo body. Si puedes distinguirlas, un atacante también.

26.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

26.12. Lo que deberías saber hacer ahora

- ☐ Separar autenticación (401) de autorización (403/404)
- ☐ Enforzar ownership en la query, no en el if
- ☐ Modelar roles por recurso con tabla de membresía
- ☐ Construir un factory `requireRole` en tu stack
- ☐ Justificar cuándo 403 y cuándo 404 (anti-enumeración)

27 18. Necesitamos que robar una sesión no sea suficiente

La cookie se filtró: ¿qué puede hacer el atacante? ¿puedes revocarla?

> En una frase
>
> El JWT del cap. 15 tiene un agujero: **no se puede revocar** — quien lo robe lo usa hasta que expire, y "cerrar sesión en todos mis dispositivos" es imposible. Este capítulo añade la capa que falta: sesiones persistidas y revocables, refresh token rotation (que *detecta* el robo), rate limiting en login, y CORS explicado de verdad.

27.1. El problema

El JWT stateless es escalable pero ciego: el servidor verifica la firma y confía — sin consultar nada. Eso significa que **no hay interruptor**: el token robado vale hasta su `exp`, el logout no invalida nada en el servidor, y "cerrar todas las sesiones" (tu laptop robada) no existe como operación. Además el login es el endpoint más atacado del mundo: sin límite de intentos, la fuerza bruta prueba contraseñas a máquina.

> Explicación de: firma / HMAC
>
> Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

> Explicación de: autenticación
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

> Explicación de: sesión
>
> Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada

llamada.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

27.2. Cómo lo resuelve un equipo

El diseño que la industria convergió — **híbrido**:

- **Access token:** JWT stateless, vida corta (15 min). Escala gratis, no se consulta nada — y si se roba, vive poco.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

- **Refresh token:** opaco (no JWT, un string aleatorio), vida larga (30 días), **guardado en la BD**. Sirve solo para pedir access nuevos — y como vive en tabla, se puede `DELETE` = revocación real.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

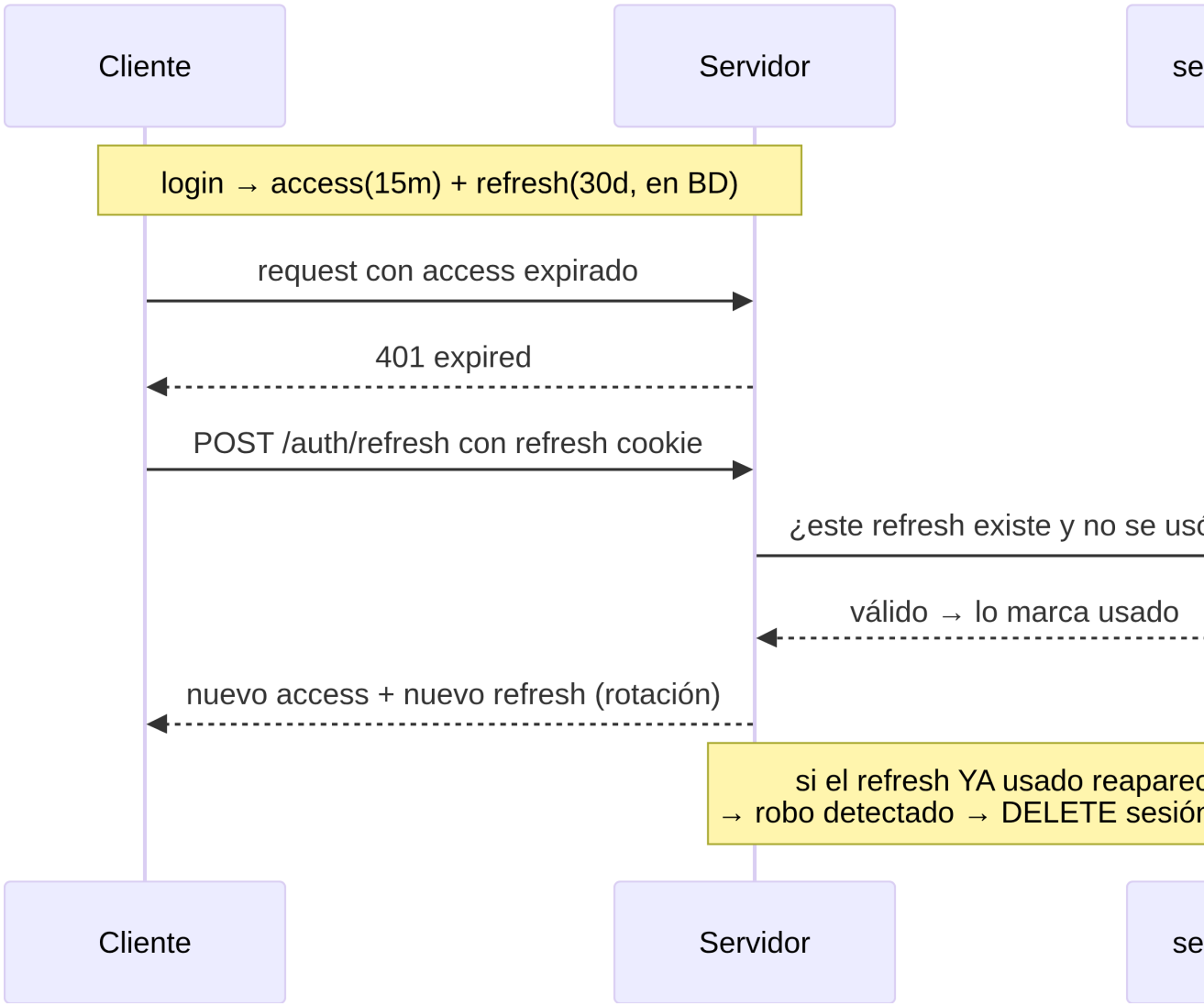
- **Rotación:** cada vez que el refresh se usa, se emite uno nuevo y el viejo muere. Si un token *ya usado* reaparece → alguien lo robó → se invalida la sesión entera (detección, no solo expiración).
- **Rate limiting** en `/auth/*`: 5 intentos por minuto por IP — la fuerza bruta se vuelve impráctica.
- **CORS:** el navegador pregunta "¿este origen puede llamarte?" y el servidor responde con headers — no es autenticación, es política del navegador.

27.3. Conceptos nuevos

- U JWT stateless vs sesión en BD: una query a cambio de revocación real

- U “Cerrar sesión en todos mis dispositivos”: imposible con JWT puro, trivial con tabla de sesiones
- U Refresh rotation + reuse detection: el token viejo reutilizado delata al ladrón
- U Rate limiting: el 429 como amigo — fuerza bruta vuelta impráctica
- U CORS en serio: `Origin` pregunta, `Access-Control-Allow-*` responde — es el navegador, no tu API

27.4. La explicación visual



27.5. Implementación

La tabla de sesiones y el refresh con rotación — la parte que hace revocable lo que el JWT no puede:

```
CREATE TABLE sessions (
  id           BIGSERIAL PRIMARY KEY,
  user_id      BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  refresh_hash TEXT NOT NULL UNIQUE,     -- hash del refresh, no el token
  expires_at   TIMESTAMPTZ NOT NULL,
  used_at      TIMESTAMPTZ,              -- rotación: cuándo se canjeó
  revoked_at   TIMESTAMPTZ,              -- logout / "cerrar todas"
  created_at   TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

¿Por qué `refresh_hash`? Misma lección del cap. 14: **los tokens también son secretos** — si la tabla `sessions` se filtra, los refresh en claro son sesiones regaladas. Se guarda el hash; lo que viaja al cliente es el token opaco.

> Explicación de: hash / bcrypt
>
> Un **hash** es una función de un solo sentido: `contraseña →$2b10…`. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

> Explicación de: secreto
>
> Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

El refresh con rotación + detección de reuso (patrón, válido en los tres stacks):

```
def refresh_session(refresh_token: str):
    row = conn.execute(
        "SELECT * FROM sessions WHERE refresh_hash = %s",
        (hash(refresh_token),),
    ).fetchone()

    if not row:
        raise UnauthorizedError("invalid session")
    if row["used_at"] or row["revoked_at"] or row["expires_at"] < now():
        # a used/expired token came back - someone stole it
        revoke_session_family(row["user_id"])     # kill the whole session
        raise UnauthorizedError("session reuse detected")
    mark_used(row["id"])                          # this refresh dies now
    return issue_access(row["user_id"]), issue_refresh(row["user_id"])
```

Y el rate limit del login — en los tres stacks es un middleware delante del handler (mismo patrón del cap. 16):

> Explicación de: middleware
>
> Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

```
POST /auth/login → rate_limit(5 req/min per IP) → handler
  6th attempt → 429 {"error": {"code": "RATE_LIMITED"}}
  + Retry-After: 42
```

CORS sin magia: tu frontend corre en `localhost:5173` y tu API en `:8000` — orígenes distintos. El navegador manda `Origin:` y, para requests "no simples", primero un `OPTIONS` (el *preflight*): "¿puedo hacer `POST` con `Content-Type: json` desde este origen?". El servidor responde con headers o el navegador bloquea la respuesta **aunque el request llegó**:

```
Access-Control-Allow-Origin: http://localhost:5173   ← ese origen exacto, no *
Access-Control-Allow-Credentials: true               ← porque viajan cookies
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Methods: GET, POST, PATCH, DELETE
```

Detalle crítico: `Allow-Origin: *` + `Allow-Credentials: true` es combinación **prohibida** por el estándar (el navegador la rechaza) — con cookies siempre se nombra el origen exacto. Y CORS no protege tu API de llamadas fuera del navegador (curl no pregunta) — protege al *usuario* de sitios malignos.

> Explicación de: cookie
>
> Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

> Prompt listo para tu AI

```
Implementa autenticación revocable para mi API [Express / FastAPI / Go]:
access JWT de 15min + refresh opaco de 30d persistido en tabla sessions
(hash del token, used_at, revoked_at, expires_at), endpoint
POST /auth/refresh con ROTATION (cada uso emite refresh nuevo e invalida
el viejo) y reuse detection (si un refresh usado reaparece, revoca la
sesión entera), POST /auth/logout que revoque, "cerrar todas las sesiones"
que revoque todas las del usuario, rate limit de 5 req/min en /auth/*,
y config CORS para mi frontend en localhost:5173 con credentials. Dame
el esquema de sessions, los endpoints, y explícame el flujo de rotación.
```

27.6. ¿Por qué este híbrido y no "solo JWT" o "solo sesiones"?

Lo único que necesitas llevarte: el access stateless hace el trabajo pesado (millones de requests sin tocar la BD) y el refresh persistido da el control (revocación real, detección de robo). JWT puro no puede desloguear; sesión pura paga una consulta por request. El híbrido cobra la consulta **una vez cada 15 minutos** — el precio justo por el botón "cerrar sesión en todos mis dispositivos".

i Otras cimentaciones: arquitecturas de sesión

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
JWT puro	Todo stateless	Sin revocación real	APIs internas, tokens de corta vida
Sesión opaca en BD	Una query por request	El lookup siempre	Apps donde el control manda
Access+refresh híbrido	Stateless + tabla para refresh	La complejidad de este capítulo	El default de apps serias
Refresh en Redis	Lookup más barato que Postgres	Otra pieza de infra	Sesiones calientes a escala
Todo tercero (Auth0, Clerk, Supabase Auth)	Ellos lo operan	Costo + lock-in + tu user table queda afuera	MVP donde auth no es tu diferencial

27.7. Errores comunes

Error	Por qué pasa	Fix
Refresh en claro en la BD	"Es solo un token" — es *la* sesión	`refresh_hash`, como las contraseñas
Refresh JWT stateless también	Recién reinventaste el problema	Refresh opaco persistido — eso es lo que se revoca
Sin rotación	El refresh robado vale 30 días	Rotación + reuse detection
`Allow-Origin: *` con cookies	Funciona en el tutorial	Inválido con credentials; nombra el origen
Rate limit solo en login	`/refresh` y `/register` también se abusan	Todo `/auth/*` limitado
CORS "arreglado" con `*` y silencio	No entendiste que es política del navegador	Entender qué pregunta el preflight

27.8. Buenas prácticas

- **Refresh opaco, aleatorio, hasheado, rotable** — las cuatro propiedades que hacen sesión seria.

- **Reuse detection = alarma gratis:** un refresh usado que reaparece es un robo; revoca la familia y marca el evento en logs.

> Explicación de: log / logging estructurado
>
> Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

- **Rate limit por IP *y* por cuenta** — el botnet tiene mil IPs pero la cuenta objetivo es una.
- **`Retry-After` en el 429** — el cliente bien portado espera lo que le dices.
- **CORS con lista explícita de orígenes** — localhost:5173 en dev, tu dominio en prod, nunca `*` con cookies.

27.9. Ejercicio

1. Diseña el flujo de "cerrar sesión en todos mis dispositivos" con la tabla `sessions`.
2. Explica por qué la rotación detecta el robo — qué ven el ladrón y la víctima cuando ambos usan el mismo refresh.
3. ¿Por qué CORS no te protege de curl/Postman/scripts? ¿A quién protege?

> Soluciones
>
> 1. `UPDATE sessions SET revoked_at = now() WHERE user_id = $1 AND revoked_at IS NULL` — todos los refresh del usuario mueren; los access siguen vivos hasta su `exp` (~15 min, el precio del híbrido). El usuario re-loguea en su dispositivo actual tras confirmar contraseña (operación sensible = re-auth).
> 2. El ladrón usa el refresh robado → recibe uno nuevo y el viejo queda `used_at`. La víctima intenta usar el suyo (el mismo viejo, si fue robado a ella — o el ladrón reusa uno): quien sea segundo dispara "token ya usado" → se revoca la sesión entera → ambos quedan fuera y hay que re-loguear. El robo queda *detectado*, no solo expirado.
> 3. CORS lo enforcea el *navegador*: curl y scripts de servidor no hacen preflight ni respetan los headers — ven la respuesta igual. CORS protege al **usuario con navegador** de un sitio maligno que haría requests con *sus* cookies; no es una frontera de la API sino una política del navegador. Por eso la seguridad real es SameSite+httpOnly+rate limit; CORS es el candado de la vitrina.

27.10. Mini reto

Listar los endpoints donde un 429 debería existir en cualquier API seria.

> Soluciones
>
> Los que cuestan recursos o revelan información por repetición:
>
> - `POST /auth/login` — fuerza bruta de contraseñas (5/min por IP + por email).

- `POST /auth/register` — creación masiva de cuentas.
- `POST /auth/refresh` — adivinación de tokens.
- `POST /auth/forgot-password` — spam de emails + enumeración de usuarios registrados.
- `POST /auth/verify` / códigos OTP — fuerza bruta de códigos de 6 dígitos (aquí el límite debe ser *muy* bajo: ~5 por código).
- Endpoints caros: exportaciones, reportes, búsquedas pesadas — cuestan CPU/dinero por llamada.

Patrón: limita donde **cada intento cuesta algo tuyo** (CPU, email, dinero) o **revela algo** (¿existe este email? ¿es válido este código?).

27.11. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

27.12. Lo que deberías saber hacer ahora

- ☐ Explicar por qué JWT puro no se revoca y el híbrido cómo lo arregla
- ☐ Diseñar la tabla de sesiones con hash, rotación y revocación
- ☐ Implementar reuse detection como alarma de robo
- ☐ Poner rate limiting donde importa y explicar el 429
- ☐ Responder un preflight CORS correctamente y explicar a quién protege

Parte VI

Sección II · Integraciones y concurrencia

28 19. Necesitamos llamar a otra API desde el servidor

Tu backend ahora es cliente de otro backend

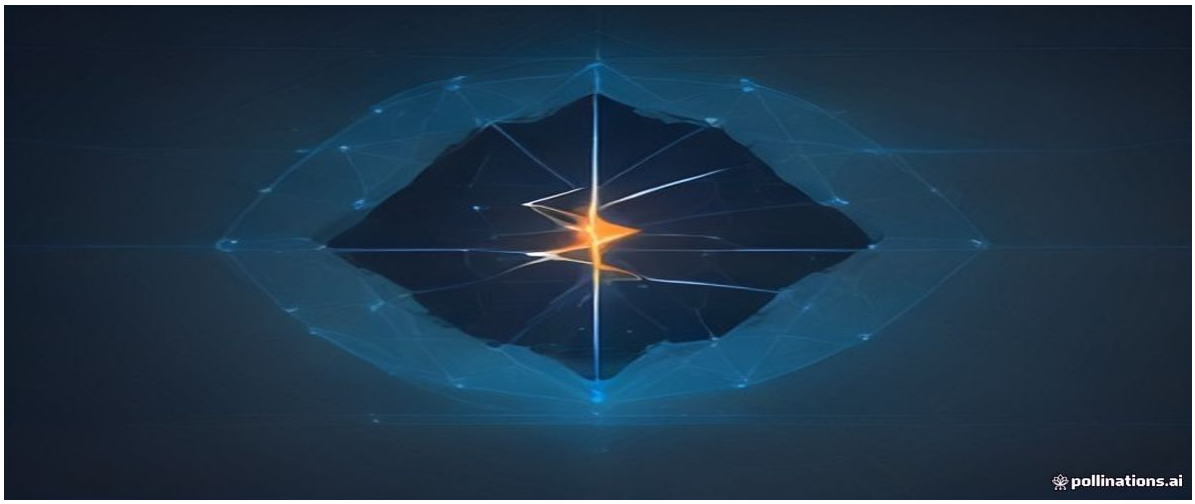


Figura 28.1: Parte 4 — Integraciones y concurrencia

> **En una frase**
>
> Hasta aquí tu API *respondía*. Ahora también *llama*: cuando se crea una tarea, hay que avisar a Slack/email — una API externa que no controlas. Hacer ese request bien no es `fetch(url)` y ya: es **timeout** (el que los tutoriales omiten), **reintento con backoff**, y una API tuya que sigue viva aunque la de ellos esté caída.

28.1. El problema

"Cuando se crea una tarea, notifica a Slack". Fácil, ¿no? `fetch` dentro del handler y listo. Hasta que: Slack tarda 30 segundos → tu usuario espera 30 segundos. Slack devuelve 500 → tu endpoint devuelve 500 aunque la tarea sí se creó. Slack está caído un día → tu registro de tareas "está caído" un día. **Una dependencia sin reglas convierte sus problemas en tus problemas.**

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: registry (Docker)

Un **registry** es el almacén de imágenes (Docker Hub, ECR, Artifact Registry): el CI empuja `nexus:v42`, el servidor la jala. Es el intermediario entre "se construyó" y "está corriendo".

28.2. Cómo lo resuelve un equipo

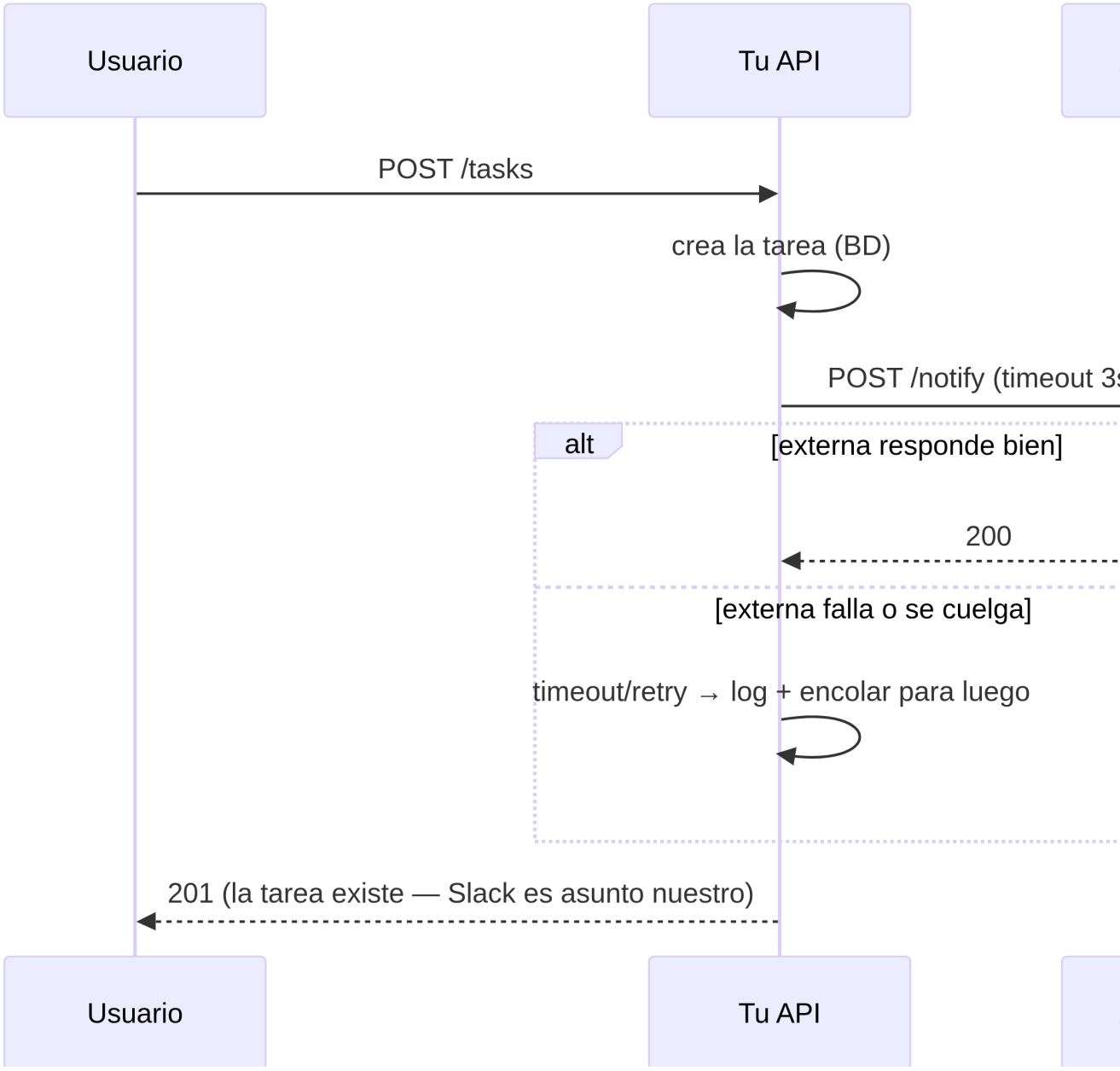
Un equipo serio trata la API externa como lo que es: **algo que falla, lento, y a veces miente**. Tres reglas antes de escribir el `fetch`:

1. **Timeout siempre** — "¿cuánto estoy dispuesto a esperar?" se decide *ahora* (2–5s típico), no cuando el thread quede colgado.
2. **Reintenta lo transitorio, con pausa creciente** — un 503 puede ser un parpadeo; reintentar al segundo puede salvarlo. Reintentar 50 veces por segundo *amplifica* el incendio.
3. **Degrada elegante** — si la notificación falla, la tarea ya se creó: responde 201, registra el fallo, reintenta después (cap. 21). El usuario no paga la caída de Slack.

28.3. Conceptos nuevos

- U El backend como cliente HTTP: `fetch` nativo vs `httpx` vs `net/http.Client`
- U Timeouts: el concepto que los tutoriales omiten y producción exige
- U Reintentos con backoff: reintentar sin amplificar el incendio
- U Degradación elegante: el servicio externo devuelve 500, tu API sigue viva
- U API keys en servidor (nunca en frontend): el patrón BFF

28.4. La explicación visual



La notificación puede fallar; la creación no. Por eso la llamada externa va **después** de persistir y su fallo no se traduce en error al usuario.

28.5. Implementación

Un pequeño cliente con timeout + retry, usado desde el servicio:

28.5.1. TypeScript

```typescript
async function notifyExternal(text: string): Promise<void> {
  const delays = [0, 500, 1500];                // retry with backoff
  for (let attempt = 0; attempt < delays.length; attempt++) {
    if (delays[attempt]) await new Promise(r => setTimeout(r, delays[attempt]));
    try {
      const res = await fetch(process.env.SLACK_WEBHOOK_URL!, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ text }),
        signal: AbortSignal.timeout(3000),      // the part tutorials skip
      });
      if (res.ok) return;
      if (res.status < 500) return;             // 4xx won't fix itself
    } catch { /* timeout or network: retryable */ }
  }
  throw new Error("external notify failed after retries");
}
```

28.5.2. Python

```python
import httpx

def notify_external(text: str) -> None:
    delays = [0, 0.5, 1.5]                       # retry with backoff
    for i, delay in enumerate(delays):
        if delay: time.sleep(delay)
        try:
            res = httpx.post(settings.slack_webhook_url,
                             json={"text": text}, timeout=3.0)
            if res.is_success: return
            if res.status_code < 500: return     # 4xx won't fix itself
        except httpx.HTTPError:
            pass                                 # timeout/network: retryable
    raise ExternalServiceError("notify failed after retries")
```

28.5.3. Go

```go
var client = &http.Client{Timeout: 3 * time.Second} // no default timeout!

func notifyExternal(text string) error {
    body, _ := json.Marshal(map[string]string{"text": text})
    for attempt, delay := range []time.Duration{0, 500, 1500} {
        if delay > 0 { time.Sleep(delay * time.Millisecond) }
        res, err := client.Post(webhookURL, "application/json", bytes.NewReader(body))
        if err == nil {
            defer res.Body.Close()
            if res.StatusCode < 300 { return nil }
            if res.StatusCode < 500 { return nil } // 4xx won't fix itself
        }
        _ = attempt
    }
    return errors.New("notify failed after retries")
}
```

Prompt listo para tu AI

```
Crea un módulo para llamar una API externa (webhook de Slack) desde mi
servicio en [Express/FastAPI/net-http]. Requisitos: timeout de 3s por
intento, máximo 3 intentos con backoff exponencial, no reintentar errores
4xx, lanzar un error de dominio ExternalServiceError si todo falla, la
URL viene de variable de entorno validada al arranque, y la llamada se
usa DESPUÉS de persistir la entidad - su fallo no debe cambiar la
respuesta al cliente. Con tests que simulen timeout y 500.
```

28.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: las tres implementaciones son la misma receta — *timeout por intento, backoff entre intentos, no reintentar lo que no se arregla solo.* Cambia la sintaxis, no el diseño.

Detalle: dónde difieren los defaults

- **TS:** `fetch` no tenía timeout hasta `AbortSignal.timeout()` — por eso años de código legacy cuelga forever. Axios lo trae desde siempre (`timeout: 3000`).
- **Python:** `requests` no tiene timeout por defecto (¡peligro histórico!); `httpx` lo pone en 5s de fábrica — por eso usamos httpx.
- **Go:** `http.Client{}` vacío = **sin timeout** — el error de producción más clásico de Go. `http.Client{Timeout: ...}` es obligatorio.

Librerías de retry existen en los tres (`backoff`, `tenacity`, `avast/retry-go`) — úsalas cuando el patrón se repite, pero entiende la receta manual primero: es 10 líneas y es lo que todo equipo espera que sepas escribir.

28.7. Errores comunes

Error	Por qué pasa	Fix
Llamada externa sin timeout	El default del cliente es "esperar siempre"	Timeout explícito por intento
Retry sin pausa ni límite	"Si falla, reintenta"	Backoff + tope de intentos; 4xx no se reintenta
Fallo externo → 500 al usuario	El error de Slack burbujea al handler	Persistir primero; el fallo externo se loguea y se encola
API key en el frontend	"La necesito para llamar la API"	La llamas desde tu backend — la key jamás sale del servidor (BFF)
Loguear el request completo	Debugging cómodo	La key/secret queda en logs — loguea status y duración, no headers

28.8. Buenas prácticas

- **Timeout, retry, dedupe** — el trío mínimo de toda integración. Sin los tres, no vas a prod.
- **Persiste antes de llamar** — la tarea existe en tu BD aunque la notificación nunca llegue; el trabajo pendiente se reintenta (cap. 21).
- **Mide la dependencia**: duración y tasa de fallo de cada llamada externa van a logs (cap. 23) — un proveedor lento es tu endpoint lento.

> Explicación de: log / logging estructurado
>
> Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

- **Un adapter, no `fetch` disperso**: `notifyExternal()` es la *única* puerta a Slack — si cambia la API de ellos, cambias un archivo (y es lo que mockeas en cap. 29).

28.9. Ejercicio

1. El cliente de arriba reintenta 4xx si quitas el `if`. ¿Por qué es incorrecto reintentar un 400 o un 401?
2. ¿Qué pasa si `SLACK_WEBHOOK_URL` no está definida? ¿Cuándo debería descubrirse ese problema — al arrancar o en el primer request?
3. Diseña la respuesta al cliente si la notificación falla tras todos los reintentos: ¿qué status, qué body, qué se loguea?

28.10. Mini reto

Responder: ¿qué ve tu usuario si la API externa está caída?

Soluciones

Ejercicio.

1. Un 4xx significa que *tu request* está mal (body inválido, key mala) — reintentar solo repite el mismo request defectuoso y amplifica el problema. Se reintenta lo transitorio: 5xx, timeouts, errores de red.
2. El `!`/validación al arranque (cap. 24): la app debe *reventar al boot* si falta la config — descubrirlo en el primer request es descubrirlo a las 3am en prod.
3. `201` + la tarea creada — la notificación es asíncrona por naturaleza. Se loguea `ERROR` con contexto (task_id, intentos, último status externo) y se encola el reintento. El usuario ya tiene lo que pidió.

Mini reto. Exactamente lo mismo que si estuviera viva: `201` con su tarea. La caída de Slack es problema *operativo* tuyo (logs, alertas, reintentos), no del usuario — degradación elegante = el usuario nunca se entera, tú sí.

28.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: concurrencia vs paralelismo

Concurrencia = manejar muchas cosas en progreso (atender 1000 requests intercalando). **Paralelismo** = ejecutar varias a la vez en varios núcleos. Un camarero con 10 mesas es concurrente; 10 camareros son paralelos.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: webhook

Un **webhook** es una llamada HTTP invertida: en vez de tú preguntarle a Stripe "¿llegó el pago?", Stripe te llama a tu endpoint cuando pasa. Se verifica la firma (es público) y se responde 200 rápido.

28.12. Lo que deberías saber hacer ahora

- ☐ Escribir una llamada externa con timeout, retry y backoff en tu stack
- ☐ Explicar por qué 4xx no se reintenta y 5xx sí
- ☐ Diseñar qué ve el usuario cuando la dependencia está caída
- ☐ Mantener API keys solo en el servidor y fuera de los logs
- ☐ Envolver la integración en un adapter con nombre de negocio

29 20. Necesitamos concurrencia de verdad

Tres runtimes, tres modelos distintos — el capítulo estrella

> **En una frase**
>
> Tu endpoint necesita datos de dos APIs externas: en secuencia son 2×300ms = 600ms; en paralelo son ~300ms. Los tres lenguajes resuelven esto con modelos *distintos* — y entender las diferencias es de lo que más pagan en una entrevista y en producción.

29.1. El problema

`GET /dashboard` debe traer: tus tareas + las notificaciones del servicio externo + los contadores del reporte. Tres llamadas que *no dependen entre sí* — hacerlas una tras otra es pagar el tiempo tres veces cuando podrías pagarlo una. La concurrencia no es un lujo: es devolver el tiempo que el usuario no tenía por qué prestarte.

29.2. Cómo lo resuelve un equipo

Un equipo serio primero **clasifica la espera**: ¿el tiempo se va en *I/O* (esperar red, disco, BD) o en *CPU* (calcular)? Concurrencia resuelve I/O — mientras una llamada espera, la otra viaja. Para CPU necesitas *paralelismo real* (varios núcleos trabajando a la vez), que no todos los runtimes ofrecen igual. La regla de bolsillo: **esperas juntas, cuentas separadas**.

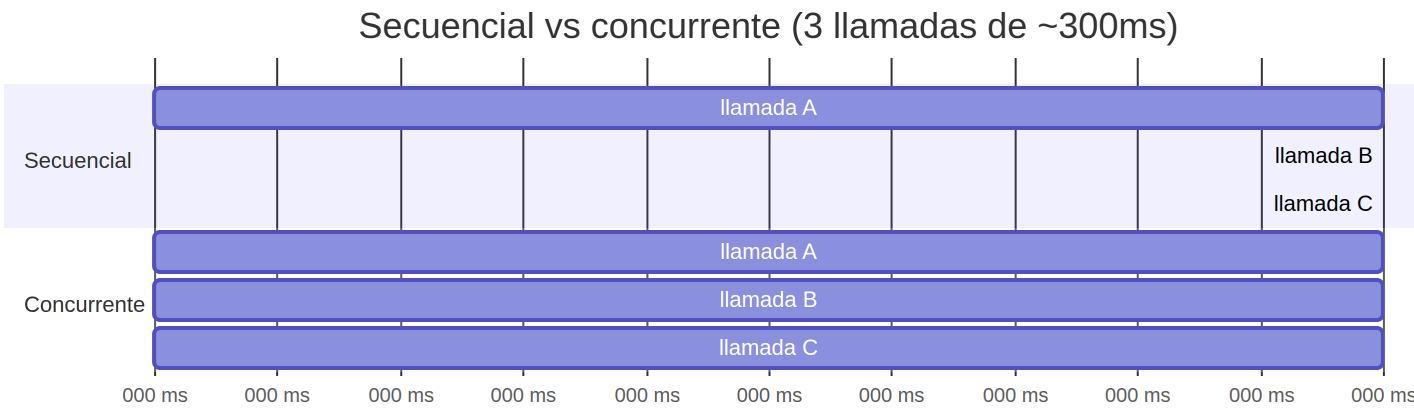
> **Explicación de: CPU / núcleos**
>
> El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). “CPU-bound” = el límite es el cálculo, no la espera.

Luego decide el límite: “en paralelo” no es “infinitas a la vez” — mil llamadas concurrentes a la API de alguien son un ataque, no una optimización.

29.3. Conceptos nuevos

- TS Event loop + `Promise.all` — concurrencia cooperativa
- Py `asyncio` + `gather` — el event loop de Python y sus reglas distintas
- Go Goroutines + `WaitGroup`/`errgroup` — concurrencia barata y paralelismo real
- U El GIL y los threads: qué puede y qué no cada runtime en paralelo — la explicación honesta
- U Cuándo la concurrencia importa (I/O) y cuándo es ruido (CPU)

29.4. La explicación visual



900ms vs ~300ms — mismo trabajo, distinto orden. Las tres llamadas *inician* juntas; el total es el de la más lenta, no la suma.

29.5. Implementación

El mismo endpoint — traer tres cosas en paralelo — en los tres modelos:

29.5.1. TypeScript

```
app.get("/dashboard", auth, async (req, res) => {
  // Promise.all: starts all three, resolves when ALL resolve
  const [tasks, notifications, stats] = await Promise.all([
    listTasks(req.user.id),              // queries the DB
    fetchNotifications(req.user.id),     // external API (cap. 19 client)
    getStats(req.user.id),               // report query
  ]);
  res.json({ tasks, notifications, stats });
```

```
  // if one rejects → all rejects → 500 (or Promise.allSettled to degrade)
});
```

29.5.2. Python

```python
@app.get("/dashboard")
async def dashboard(user=Depends(get_current_user)):
    # asyncio.gather: the event loop interleaves the awaits
    tasks, notifications, stats = await asyncio.gather(
        list_tasks(user.id),              # must be async to interleave!
        fetch_notifications(user.id),     # httpx.AsyncClient
        get_stats(user.id),
    )
    return {"tasks": tasks, "notifications": notifications, "stats": stats}
```

29.5.3. Go

```go
func dashboard(w http.ResponseWriter, r *http.Request) {
    user := userFromCtx(r.Context())
    g, ctx := errgroup.WithContext(r.Context())
    var tasks []Task
    var notes []Notification
    var stats Stats
    g.Go(func() error { var err error; tasks, err = listTasks(ctx, user.ID); return err })
    g.Go(func() error { var err error; notes, err = fetchNotes(ctx, user.ID); return err })
    g.Go(func() error { var err error; stats, err = getStats(ctx, user.ID); return err })
    if err := g.Wait(); err != nil {      // real parallelism - one goroutine per call
        writeError(w, 502, `{"error":{"code":"UPSTREAM"}}`); return
    }
    writeJSON(w, 200, map[string]any{"tasks": tasks, "notifications": notes, "stats": stats})
}
```

Prompt listo para tu AI

```
Optimiza mi endpoint GET /dashboard en [Express/FastAPI/net-http]: hoy
hace 3 llamadas independientes en secuencia (2 queries a Postgres + 1
API externa con el cliente con timeout del cap. 19). Requisitos:
ejecutarlas en paralelo con [Promise.all/asyncio.gather/errgroup], que
un fallo de UNA dependencia no tumbe las otras dos (degradación por
sección: notifications puede venir como null), timeout total de 5s, y
tests que midan que el tiempo total   el de la llamada más lenta.
```

29.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres lanzan las esperas juntas y recogen los resultados al final — `Promise.all`/`gather`/`errgroup` son la misma idea con distinta maquinaria debajo.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Detalle: los tres modelos de concurrencia

Runtime	Modelo	¿Paralelismo real en CPU?
Node	Un hilo + event loop: cuando algo espera I/O, el hilo atiende otra cosa	No — para CPU hay `worker_threads`
Python	`asyncio`: mismo event loop, pero *solo* funciones `async` cooperan	No en threads (el **GIL**: un solo hilo ejecuta Python a la vez) — para CPU hay `multiprocessing`
Go	Goroutines: miles de hilos lógicos baratos repartidos en núcleos reales	**Sí** — es la diferencia de diseño de Go

Dos matices honestos: (1) en Python, `await` sobre código *no async* (`time.sleep`, `requests`) **bloquea el loop entero** — por eso `httpx` y drivers async; (2) en Node/Python la concurrencia es *cooperativa*: nadie te quita el hilo hasta que sueltas en un `await`; en Go el scheduler lo reparte — por eso Go escala a CPU y los otros no.

Otras cimentaciones: cuándo la concurrencia no es la respuesta

Alternativa	Qué es	Cuándo
`Promise.allSettled`	Espera a todas aunque algunas fallen	Degradación por sección (un widget caído no tumba el dashboard)
Semáforo/`errgroup.SetLimit`	Paralelo con tope (ej. 10 a la vez)	Lotes grandes — 500 items en paralelo = DoS a tu propia dependencia
Worker threads / multiprocessing	Núcleos reales para CPU	Resize de imágenes, hashing masivo — *eso* sí es paralelismo

Cola de trabajos	Ni siquiera concurrente en el request	El trabajo no cabe en la paciencia HTTP (cap. 21)

29.7. Errores comunes

Error	Por qué pasa	Fix
`await` dentro de un `for`	"Lo hago uno a uno y funciona"	Acumula promesas → `Promise.all`/`gather`
Bloquear el event loop	`requests.get` dentro de `async def`	Librerías async (httpx, asyncpg) o el executor
Concurrencia sin límite	"Lanzo las 5.000 en paralelo"	Semáforo/`SetLimit` — tu dependencia también tiene rate limit
Compartir memoria sin cuidado	Dos goroutines escriben el mismo map	En Go: canales o mutex — `go run -race` lo detecta gratis
Pensar que "concurrencia = CPU más rápido"	Confundir espera con cómputo	I/O → concurrencia; CPU → paralelismo real u otro proceso

29.8. Buenas prácticas

- **Independencia primero**: solo se paraleliza lo que no se necesita entre sí — si B usa el resultado de A, no es candidata.
- **Define el comportamiento ante fallo parcial** antes de elegir: `all` (una falla, falla todo) vs `allSettled`/degradación (sección vacía, página viva).
- **Contexto propagado**: cancelación y deadline del request viajan a las llamadas (el `ctx` de Go, AbortSignal, el event loop) — si el cliente se fue, para qué seguir trabajando.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

- **Mide el antes/después**: "optimizamos" sin medir es superstición — log de duración por llamada (cap. 23).

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

29.9. Ejercicio

1. En el dashboard, ¿qué debería pasar si `fetchNotifications` falla pero las tareas y stats salen bien? Elige `all` vs `allSettled` y defiéndelo.
2. Tienes que llamar a la API externa 200 veces (una por tarea). ¿Por qué `Promise.all` de las 200 es una mala idea y qué pondrías en su lugar?
3. En Python, el handler usa `time.sleep(0.3)` dentro de un endpoint `async def`. ¿Qué le pasa a *los demás* requests mientras duerme?

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

29.10. Mini reto

Medir la diferencia entre secuencial y concurrente con dos llamadas lentas — en tu lenguaje.

Soluciones

Ejercicio.

1. `allSettled` (o catch por llamada): el dashboard muestra `notifications: null` y la página vive — un widget caído no debe tumbar los otros dos. `all` sería correcto solo si la página *no tiene sentido* incompleta.
2. 200 concurrentes = 200 conexiones abiertas a la vez: la API externa te banea por rate limit y tu event loop se ahoga. La forma: un semáforo/`errgroup.SetLimit(10)` o una cola con N workers — en paralelo *con tope*.
3. **Todos** esperan: `time.sleep` no es `await` — bloquea el único hilo del event loop y ningún otro request avanza esos 300ms. La versión async es `await asyncio.sleep(0.3)` — el loop atiende a los demás mientras duerme. Es *la* trampa de asyncio.

Mini reto. Con `asyncio.sleep`/`setTimeout`/`time.Sleep` de ~300ms en dos llamadas: secuencial 600ms, concurrente 300ms. Si tu "concurrente" mide 600ms, el await está dentro de un loop o la llamada es bloqueante — los dos errores de arriba.

29.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: goroutine

Una **goroutine** es el hilo ultraligero de Go: `go f()` lanza una función en paralelo gastando casi nada — se pueden tener millones. Es la superpotencia de Go: concurrencia como palabra del lenguaje.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña →$2b10…`. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

29.12. Lo que deberías saber hacer ahora

- ☐ Paralelizar llamadas independientes en tu stack (`all`/`gather`/`errgroup`)
- ☐ Explicar la diferencia entre concurrencia (I/O) y paralelismo (CPU)
- ☐ Decir qué es el GIL y por qué los threads de Python no paralelizan
- ☐ Ponerle límite a la concurrencia cuando hay N grande
- ☐ Elegir entre "todo falla junto" y "degradación por sección" con criterio

30 21. Necesitamos responder ya y terminar el trabajo después

El email tarda 3 segundos; el usuario no debe esperarlos

> **En una frase**
>
> El registro crea el usuario en 80ms… y luego manda el email de bienvenida que tarda 3s. La respuesta correcta no es "hacerlo más rápido": es **responder ya y terminar el trabajo después** — con el límite honesto de que un trabajo en memoria muere si el proceso muere, y por eso existen las colas de verdad.

30.1. El problema

`POST /auth/register` hace tres cosas: guardar el usuario (80ms), enviar el email (3s), notificar a Slack (variable). El usuario solo necesita

> **Explicación de: variable**
>
> Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

> **Explicación de: autenticación**
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

una respuesta: "tu cuenta existe". Los otros 3 segundos son trabajo que nadie está esperando — y sin embargo los está pagando en latencia, en timeouts del proxy, y en riesgo de que el email falle y parezca que el registro falló.

Explicación de: registry (Docker)

Un **registry** es el almacén de imágenes (Docker Hub, ECR, Artifact Registry): el CI empuja `nexus:v42`, el servidor la jala. Es el intermediario entre "se construyó" y "está corriendo".

Explicación de: reverse proxy

Un **reverse proxy** (nginx, ALB, Cloudflare) es el portero: recibe todo el tráfico en el 443, termina TLS y reparte a tu app interna. La app nunca mira a internet directamente — el proxy la protege.

30.2. Cómo lo resuelve un equipo

Un equipo serio separa **lo que el usuario espera** de **lo que el sistema debe hacer**: el response lleva solo lo primero; lo segundo va a un trabajo en segundo plano. La pregunta de diseño es una sola — *"¿el usuario necesita esto para continuar?"* No → background.

Y acepta el trade-off con los ojos abiertos: trabajo en memoria del proceso = si el proceso muere, el trabajo muere. Para "email que sería feo perder" se llega a colas de verdad (broker + worker + reintentos); para "estadística que se recalcula sola", el background in-process alcanza y sobra.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

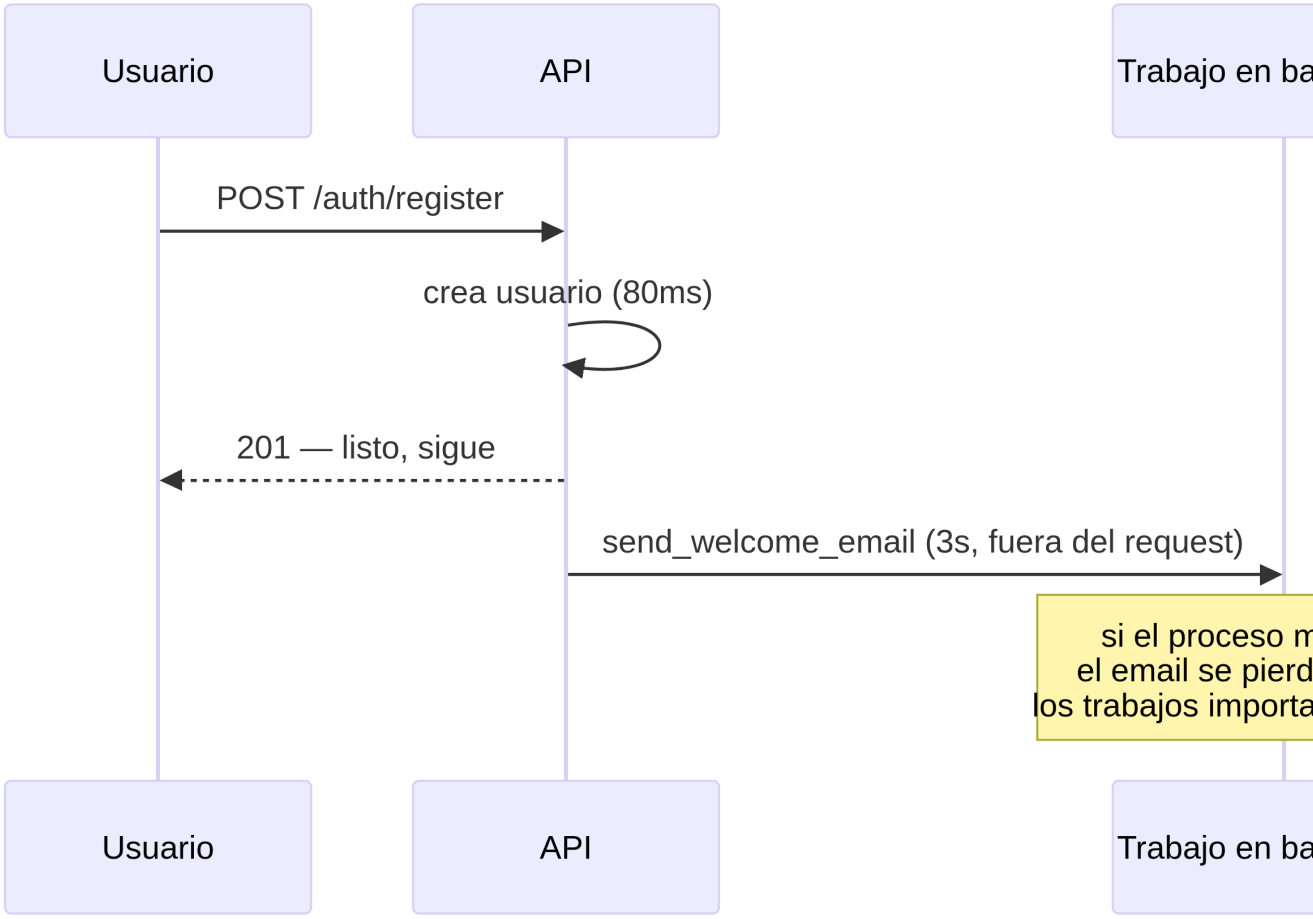
Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

30.3. Conceptos nuevos

- U Background work post-respuesta en cada stack
- TS `setImmediate`/cola propia · Py `BackgroundTasks` · Go `go func()` — y el mismo límite en los tres
- U Colas de verdad (concepto): broker, worker, reintentos, idempotencia — BullMQ/RQ/asynq como panorama
- U El patrón `job_id` + polling: procesos largos sin timeout

30.4. La explicación visual



30.5. Implementación

Responder primero, trabajar después — el mismo patrón en los tres:

30.5.1. TypeScript

```
app.post("/auth/register", async (req, res, next) => {
  try {
    const user = await registerUser(req.body);      // what the user waits for
    res.status(201).json(toPublicUser(user));       // respond FIRST
    setImmediate(() => {                            // work AFTER
      sendWelcomeEmail(user.email).catch(err =>
        logger.error({ err }, "welcome email failed")); // errors die here
```

```
    });
  } catch (err) { next(err); }
});
```

30.5.2. Python

```python
@app.post("/auth/register", status_code=201)
def register(body: RegisterIn, bg: BackgroundTasks):
    user = register_user(body)                      # what the user waits for
    bg.add_task(send_welcome_email, user.email)     # runs AFTER the response
    return UserPublic.model_validate(user)
    # FastAPI guarantees the task runs post-response - still in-process
```

30.5.3. Go

```go
func register(w http.ResponseWriter, r *http.Request) {
    user, err := registerUser(r.Context(), parseBody(r))
    if err != nil { writeError(w, 422, errJSON); return }
    writeJSON(w, 201, toPublic(user))               // respond first
    go func(email string) {                         // work after
        ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
        defer cancel()
        if err := sendWelcomeEmail(ctx, email); err != nil {
            slog.Error("welcome email failed", "err", err)
        }
    }(user.Email)                                   // careful: detached ctx, not r.Context()
}
```

Prompt listo para tu AI

```
En mi endpoint POST /auth/register [Express/FastAPI/net-http] el envío
del email de bienvenida hace que el response tarde 3s. Refactoriza para:
responder 201 apenas persista el usuario, ejecutar el email en
background post-respuesta con su propio contexto/timeout y manejo de
error que lo loguee sin tumbar nada, y documentar el límite (se pierde
si el proceso muere) más cómo se migraría a una cola BullMQ/RQ cuando el
volumen lo exija.
```

30.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres separan *responder* de *terminar* — y los tres comparten el mismo límite: el trabajo vive en la memoria del proceso. Más allá de cierto volumen/criticidad, la respuesta correcta es una cola de verdad, no más `go func()`.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Detalle: de in-process a cola de verdad

Nivel	Qué es	Cuándo basta	Qué pierdes
In-process	`setImmediate` / `BackgroundTasks` / `go func()`	Volumen bajo, trabajo rehacible (emails, métricas)	Muere con el proceso; sin reintentos; sin visibilidad
Cola (BullMQ/RQ/asynq)	Job persistido en Redis/BD + worker separado	Trabajo que no debe perderse, reintentos, picos	Operar el broker y el worker
Managed (SQS, Cloud Tasks, Pub/Sub)	La nube opera la cola	Ya estás en la nube y quieres cero ops	Lock-in del proveedor

Y el patrón para trabajos que duran *minutos* (exportar, reportes): `POST /exports` → `202 {"job_id": "j_9x"}` → el cliente consulta `GET /jobs/j_9x` → `{"status": "running"}` → `{"status": "done", "url": "..."}`. Ningún request HTTP dura minutos — el *trabajo* sí, y tiene su propio recurso.

30.7. Errores comunes

Error	Por qué pasa	Fix
Background que usa el contexto del request	"Funcionaba en dev"	El request muere al responder — ctx propio con su timeout
Error del job sin capturar	`go func` con panic → tumba el proceso	El job captura su propio error y lo loguea
Creer que in-process = durable	"Se encoló, llegará"	Si importa, cola real — el proceso puede morir mañana
Encolar dentro de la transacción	El job corre antes del COMMIT y no ve la fila	Enqueue *después* del commit
Reintentar el email infinitamente	Sin contador de intentos	Max N intentos + dead-letter: los venenosos se apartan

30.8. Buenas prácticas

- **La pregunta antes que el código**: "¿el usuario espera esto?" — si no, es job. Esa sola pregunta diseña el endpoint.

- **202 Accepted** cuando el trabajo *es* la respuesta (exports, procesamiento): "aceptado, avísame" + `job_id` para consultar.
- **Idempotencia en el worker** (cap. 22): los jobs se reintentan — "enviar email" debe poder correr dos veces sin mandar dos correos.
- **Observa la cola**: profundidad, duración, fallos — una cola que crece es un sistema muriendo en silencio (cap. 23).

30.9. Ejercicio

1. Ordena estos trabajos del registro: ¿cuáles en el request y cuáles en background? — crear el usuario, enviar email de bienvenida, registrar "user_registered" en analytics, validar que el email no exista.
2. ¿Por qué el `go func` de arriba usa `context.Background()` y no `r.Context()`?
3. Diseña `POST /exports` para "descargar todas mis tareas en CSV" (toma 2 minutos): rutas, códigos, y qué guarda la tabla `jobs`.

30.10. Mini reto

Diseñar el endpoint de "exportar mis datos" (toma minutos) sin timeout.

Soluciones

Ejercicio.

1. Request: validar email único + crear usuario (lo que el usuario espera). Background: email de bienvenida + evento de analytics — ninguno bloquea la respuesta.
2. `r.Context()` se cancela cuando el response se envía — el job moriría al instante de nacer. `context.Background()` (más su propio timeout) le da vida propia.
3. `POST /exports` → crea fila `jobs(id, user_id, type, status)` + encola → `202 {"job_id": "j_9x"}`. `GET /jobs/j_9x` → `{"status":"running"}` luego `{"status":"done","download_url":"..."}`. El CSV va a object storage (cap. 22), no al response.

Mini reto. El patrón `job_id` + polling de arriba: ningún HTTP dura minutos, pero un *job* sí — y tiene su propio recurso que el cliente consulta. Bonus: con WebSockets (cap. 38) el servidor puede *empujar* "tu export está listo" en vez de que el cliente pregunte.

30.11. Vocabulario técnico del capítulo

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

30.12. Lo que deberías saber hacer ahora

- ☐ Separar lo que el usuario espera de lo que el sistema debe hacer
- ☐ Implementar trabajo post-respuesta en tu stack con su manejo de error
- ☐ Explicar el límite de los jobs in-process y cuándo pasar a cola real
- ☐ Diseñar un proceso largo con `job_id` + polling (o push)
- ☐ Encolar después del commit, no dentro de la transacción

31 22. Necesitamos recibir lo que otros nos envían

Webhooks entrantes y uploads: el endpoint público peligroso

> **En una frase**
>
> Hasta ahora tus endpoints respondían a *tu* frontend logueado. Ahora el mundo te llama: Stripe avisa que un pago llegó, el usuario sube su avatar. Dos superficies nuevas — **webhooks** (requests firmados de otro servidor) y **uploads** (datos que no son JSON) — y las dos son el endpoint más atacado de tu API si las tratas como las demás.

31.1. El problema

El proveedor de pagos avisa `payment.completed` con un POST a tu `/webhooks/payments`. Problemas: ¿cómo sabes que el POST vino *de ellos* y no de alguien probando `/webhooks/payments` con `{"amount": 999999}`? ¿Qué pasa cuando el mismo webhook llega dos veces (ellos reintentan si tardas)? Y el avatar: `avatar.png` que en realidad es un `.exe` de 50MB disfrazado. Todo lo que entra necesita verificación *antes* de procesar.

31.2. Cómo lo resuelve un equipo

Un equipo serio trata estos endpoints con tres reglas:

1. **Verifica antes de creer** — el webhook viene firmado (HMAC con secreto compartido): se verifica la firma *sobre el body crudo*, antes de parsear. El upload se valida por contenido (magic bytes), no por nombre ni `Content-Type` declarado.
2. **Responde rápido, procesa después** — el proveedor reintenta si no respondes en segundos; el procesamiento va a una cola (cap. 21).
3. **Idempotencia** — el mismo evento puede llegar 5 veces; procesarlo debe tener el efecto de procesarlo 1 vez (dedupe por `event_id`).

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

31.3. Conceptos nuevos

- U Webhooks entrantes: verificar firma, responder rápido, procesar después
- U Idempotencia: el mismo webhook llega dos veces — `event_id` y deduplicación en BD
- U Uploads: multipart, tamaños máximos, validación de tipo real (no el nombre)
- Ops Dónde van los archivos: disco vs object storage — la decisión, no la implementación

31.4. La explicación visual

Verificar → deduplicar → responder → *entonces* procesar. En ese orden.

31.5. Implementación

El webhook verificado — el patrón completo en compacto:

31.5.1. TypeScript

```
app.post("/webhooks/payments",
  express.raw({ type: "application/json" }),     // RAW body - signature needs bytes
```

```
  async (req, res) => {
    const expected = crypto.createHmac("sha256", WEBHOOK_SECRET)
                           .update(req.body).digest("hex");
    const sig = req.headers["x-signature"] as string;
    if (!sig || !crypto.timingSafeEqual(Buffer.from(sig), Buffer.from(expected)))
      return res.sendStatus(401);                  // verify BEFORE parsing
    const event = JSON.parse(req.body.toString());
    const inserted = await dedupeInsert(event.id); // UNIQUE(event_id)
    if (!inserted) return res.sendStatus(200);     // already processed: ok
    enqueue("process_payment", event);             // fast response, work after
    res.sendStatus(200);
  });
```

31.5.2. Python

```
@app.post("/webhooks/payments")
async def payments_webhook(request: Request):
    body = await request.body()                    # raw bytes for the signature
    expected = hmac.new(WEBHOOK_SECRET, body, hashlib.sha256).hexdigest()
    sig = request.headers.get("x-signature", "")
    if not hmac.compare_digest(sig, expected):
        raise HTTPException(401)                   # verify BEFORE parsing
    event = json.loads(body)
    if not dedupe_insert(event["id"]):             # UNIQUE(event_id)
        return {"ok": True}                        # already processed
    enqueue("process_payment", event)              # fast response, work after
    return {"ok": True}
```

31.5.3. Go

```
func paymentsWebhook(w http.ResponseWriter, r *http.Request) {
    body, _ := io.ReadAll(http.MaxBytesReader(w, r.Body, 1<<20)) // cap size
    mac := hmac.New(sha256.New, webhookSecret)
    mac.Write(body)
    if !hmac.Equal([]byte(r.Header.Get("X-Signature")),
                   []byte(hex.EncodeToString(mac.Sum(nil)))) {
        w.WriteHeader(401); return                 // verify BEFORE parsing
    }
    var event PaymentEvent
    json.Unmarshal(body, &event)
    if !dedupeInsert(event.ID) {                   // UNIQUE(event_id)
        w.WriteHeader(200); return                 // already processed
    }
    enqueue("process_payment", event)              // fast response, work after
```

```
    w.WriteHeader(200)
}
```

Y el upload, las reglas sin depender del stack:

```
POST /me/avatar (multipart)
→ MaxBytes/límite: 5MB - el servidor rechaza antes de llenar el disco
→ Leer los primeros bytes: 89 50 4E 47 = PNG real (magic bytes),
  no confiar en filename ni Content-Type declarado
→ Guardar con nombre generado (uuid.png), nunca el nombre del usuario
→ Destino: object storage (S3/Blob/GCS) - el disco del servidor no escala
```

Prompt listo para tu AI

```
Implementa POST /webhooks/payments en [Express/FastAPI/net-http] que
reciba eventos de un proveedor. Requisitos: verificar la firma HMAC
SHA-256 del header X-Signature sobre el body CRUDO antes de parsear,
deduplicar por event_id con constraint UNIQUE en Postgres, responder
200 rápido y encolar el procesamiento real, límite de tamaño del body,
y tests: firma inválida → 401, mismo evento dos veces → procesado una
sola, y proveedor-caído → reintento seguro.
```

31.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el patrón es idéntico en los tres — *body crudo → firma → dedupe → 200 rápido → procesar después*. Lo que cambia es *cómo* obtienes el body crudo (Express lo re-serializa sin el middleware `raw`, por eso va explícito).

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Otras cimentaciones: dónde van los archivos

Destino	Qué es	Qué te cuesta	Cuándo
Disco local	Carpeta del servidor	Muere con el contenedor; no escala a 2 instancias	Dev, prototipo

Object storage (S3/Blob/GCS)	Servicio de archivos de la nube	Configurar + URLs firmadas	Prod casi siempre
CDN delante	Cacheo global del archivo	Una capa más	Archivos públicos muy leídos

La regla: **en prod el archivo vive fuera del proceso** — dos instancias de tu API deben ver el mismo avatar.

31.7. Errores comunes

Error	Por qué pasa	Fix
Verificar la firma sobre el JSON parseado	"Ya lo tengo como objeto"	La firma cubre los *bytes* — parsear puede reordenar/reformatear
Procesar antes de responder	El proveedor reintenta a los 5s	200 rápido + cola — el patrón del cap. 21
Sin deduplicación	"¿Por qué cobraría dos veces?"	`event_id` UNIQUE — el mismo evento, una sola ejecución
Confiar en `filename`/`Content-Type`	Lo declara el cliente	Magic bytes + extensión generada por ti
Sin límite de tamaño	Upload de 50GB = disco lleno	MaxBytes/corte en el middleware
Guardar en disco del contenedor	"Funciona en mi máquina"	Object storage — el disco de prod es efímero

31.8. Buenas prácticas

- **`timingSafeEqual`/`compare_digest`/`hmac.Equal`** para comparar firmas — el **`==`** normal puede filtrar por tiempo de respuesta.
- **Registra `event_id` + tipo en logs**, nunca el payload entero — los webhooks traen datos personales ajenos.
- **El handler del webhook es *tonto***: verifica, deduplica, encola. La lógica vive en el worker — testeable y reintentable.
- **Replay endpoint interno**: cuando un evento se procesó mal, querrás re-ejecutarlo desde la BD de eventos sin pedirle al proveedor nada.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

31.9. Ejercicio

1. ¿Por qué la deduplicación va en la BD (`UNIQUE(event_id)`) y no en memoria (un `Set` de ids procesados)?
2. El proveedor manda el mismo evento 5 veces seguidas. Traza qué devuelve tu endpoint cada vez y qué efecto total tiene.
3. ¿Qué valida *realmente* que `avatar.png` es una imagen: su nombre, su `Content-Type`, o sus primeros bytes? ¿Y por qué los otros dos no sirven?

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

31.10. Mini reto

Hacer que procesar el mismo webhook 5 veces tenga el efecto de procesarlo 1 vez.

Soluciones

Ejercicio.

1. Memoria muere con el proceso y no se comparte entre instancias — el quinto retry puede llegar a otra réplica de tu API. La constraint UNIQUE en Postgres es deduplicación *real*: sobrevive reinicios y escala horizontal.
2. Primera: firma ok → inserta `event_id` → 200 + encola. Segunda a quinta: firma ok → `dedupe_insert` devuelve "ya existe" → 200 sin encolar. Efecto total: procesado una vez; el proveedor recibió 200 las cinco veces (deja de reintentar).
3. Los primeros bytes (magic bytes: `89 50 4E 47` = PNG, `FF D8` = JPEG). El nombre y el `Content-Type` los declara el cliente — un `.exe` renombrado declara `image/png` sin ruborizarse.

Mini reto. La tabla `webhook_events(event_id UNIQUE, payload, processed_at)`: insertar el `event_id` *es* la deduplicación — el segundo INSERT choca con la constraint y sabes que ya lo

viste. Idempotencia = mismo efecto sin importar cuántas veces llegue.

31.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña →`$2b10…‘. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: CDN

Una **CDN** (Content Delivery Network) es una red de copias: tu estático se cachea en servidores cercanos al usuario. El de Buenos Aires no viaja a Virginia por una imagen — la recibe del nodo local.

31.12. Lo que deberías saber hacer ahora

- ☐ Verificar una firma HMAC sobre el body crudo antes de parsear
- ☐ Deduplicar webhooks por `event_id` con constraint UNIQUE
- ☐ Responder 200 rápido y procesar en cola
- ☐ Validar uploads por contenido (magic bytes), tamaño y nombre generado
- ☐ Elegir dónde viven los archivos (disco vs object storage) con criterio

Parte VII

Sección II · Robustez

32 23. Necesitamos ver qué está pasando sin estar mirando

Sin logs, «algo falla a veces» es todo lo que sabrás jamás

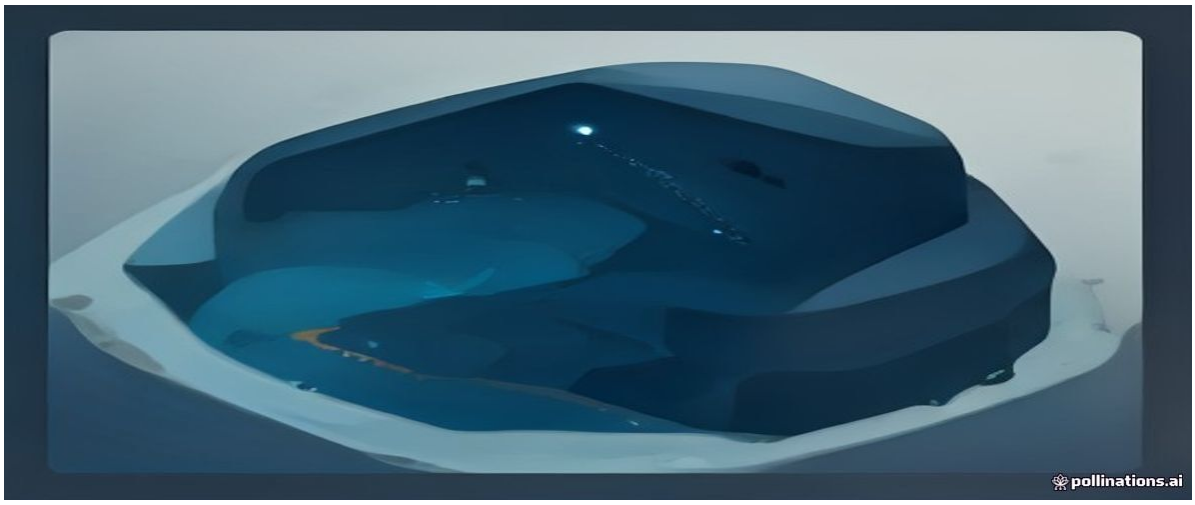


Figura 32.1: Parte 5 — Robustez

> **En una frase**
>
> En producción no hay consola que mirar ni breakpoint que poner: cuando un usuario dice "a veces falla", tu única ventana es lo que la app *escribió*. Logging real no es `console.log` suelto: son **líneas estructuradas con request_id** que te dejan reconstruir un request completo a las 3am sin despertar a nadie.

32.1. El problema

"El guardado falló ayer a las 15:14 con un usuario de Argentina." Sin logs esa frase es el fin de la investigación. Con logs malos — `print("aqui")`, `console.log(err)` sin contexto — es peor: miles de líneas que no responden *quién, qué endpoint, cuánto tardó, qué falló*. El problema no es que falle; es que no puedas *ver* por qué.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

32.2. Cómo lo resuelve un equipo

Un equipo serio trata los logs como **la fuente de datos de la verdad operativa**: cada request genera un `request_id` al entrar, todas las líneas de ese request lo llevan, y cada línea es un objeto JSON que una máquina puede filtrar (`request_id="r_9x"`, `level="error"`, `duration_ms>1000`). Tres preguntas guían qué loguear: ¿quién (user_id, request_id), qué (endpoint, acción, resultado), cuánto (duración, tamaño). Y una regla que no se negocia: **passwords, tokens y datos personales jamás tocan un log.**

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

32.3. Conceptos nuevos

- U Logging real: niveles, por qué `print`/`console.log` no es logging
- TS `pino`/`winston` · Py `logging` · Go `log/slog` — estructurado gratis en stdlib
- U Qué loguear y qué NUNCA loguear (passwords, tokens, datos personales)
- U Logs estructurados (JSON): producción parsea logs, no los lee
- U Request ID propagado por middleware en los tres

32.4. La explicación visual

Un request deja una huella rastreable — todas sus líneas comparten el mismo `request_id`:

```
{"ts":"...","level":"info","request_id":"r_9x2","msg":"request started","method":"POST","path":"/task
{"ts":"...","level":"info","request_id":"r_9x2","msg":"task created","task_id":"t_77","duration_ms":1
{"ts":"...","level":"error","request_id":"r_9x2","msg":"notify failed","service":"slack","duration_ms
{"ts":"...","level":"info","request_id":"r_9x2","msg":"request done","status":201,"duration_ms":3050}
```

Filtrar `request_id=r_9x2` = reconstruir la película completa. Eso es lo que convierte “a veces falla” en “Slack tardó 3s y el timeout saltó”.

32.5. Implementación

Middleware que genera el request_id + log estructurado por request:

32.5.1. TypeScript

```
import pino from "pino";
const logger = pino();                              // JSON to stdout by default

app.use((req, res, next) => {
  req.id = crypto.randomUUID();
  const start = Date.now();
  res.on("finish", () =>
    logger.info({ request_id: req.id, method: req.method, path: req.path,
                  status: res.statusCode, duration_ms: Date.now() - start },
                "request done"));
  next();
});
// in the service: logger.error({ request_id: req.id, err }, "task failed")
```

32.5.2. Python

```
import logging, json, uuid, time

class JsonFormatter(logging.Formatter):
    def format(self, record):
        return json.dumps({"ts": self.formatTime(record), "level": record.levelname,
                           "msg": record.getMessage(), **getattr(record, "ctx", {})})

@app.middleware("http")
async def log_requests(request, call_next):
    request.state.request_id = str(uuid.uuid4())
    start = time.time()
    response = await call_next(request)
    logger.info("request done", extra={"ctx": {
        "request_id": request.state.request_id, "method": request.method,
        "path": request.url.path, "status": response.status_code,
        "duration_ms": int((time.time() - start) * 1000)}})
    return response
```

32.5.3. Go

```
func logging(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        rid := uuid.NewString()
        ctx := context.WithValue(r.Context(), reqIDKey, rid)
        start := time.Now()
        rw := &statusWriter{ResponseWriter: w, status: 200}
        next.ServeHTTP(rw, r.WithContext(ctx))
        slog.Info("request done", "request_id", rid, "method", r.Method,
                  "path", r.URL.Path, "status", rw.status,
                  "duration_ms", time.Since(start).Milliseconds())
    })
}
// slog.New(slog.NewJSONHandler(os.Stdout, nil)) - structured out of the box
```

Prompt listo para tu AI

```
Agrega logging estructurado a mi API [Express/FastAPI/net-http].
Requisitos: middleware que genere request_id por request y lo propague
a todos los logs de ese request, formato JSON a stdout con
ts/level/method/path/status/duration_ms, niveles usados correctamente
(info para requests, error para fallos con el error serializado), y una
lista de campos que NUNCA se loguean (password, tokens, Authorization
header, body completo de /auth/*). Dame el middleware + ejemplo de uso
en un servicio.
```

32.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el patrón es uno — *id por request, contexto en cada línea, JSON que las máquinas filtran.* `pino`, `logging` y `slog` son tres caras de la misma necesidad: producción *parsea* logs, no los lee.

Detalle: niveles, qué loguear, qué nunca

Niveles = para quién es la línea: `debug` (tú, desarrollando), `info` (hechos normales que importan: request done, job encolado), `warn` (anómalo pero manejado: retry, degradación), `error` (algo se rompió y alguien debe mirarlo). Si *todo* es `error`, nada es alarma.
Nunca en un log: passwords, tokens/JWT, header `Authorization`, números de tarjeta, payloads completos de auth — un log filtrado es un incidente de seguridad, no solo feo. Se loguea `user_id`, no `email`; `token present`, no el token.
Más allá de logs (de crecer): métricas (números en el tiempo → alarmas), tracing (el viaje del request entre servicios) — decisión 23 del Ap. K. Los logs son la base: gratis, inmediatos, y suficientes durante mucho tiempo.

32.7. Errores comunes

Error	Por qué pasa	Fix
`print`/`console.log` en prod	Es lo que funciona en dev	Logger con niveles + formato — stdout es la interfaz, no el print
Loguear el body completo	"Para debuguear"	Secretos y PII en texto plano → loguea campos elegidos
Líneas sin contexto	`logger.error(err)` suelto	Siempre `request_id` + la acción — una línea debe contar su historia sola
Nivel `error` para todo	"Para que se vea"	Alarmas que lloran lobo: nadie mira errores que son normales
Puntos suspensivos en el medio	`logger.info("creando...")` y nada más	Log de *resultado* (created/failed) con duración — el intento sin cierre es ruido

32.8. Buenas prácticas

- **JSON a stdout**: la app escribe líneas JSON; el *colector* (Docker, systemd, el agente de la nube) las recoge — la app no abre archivos.

> Explicación de: la nube / cloud
>
> **La nube** son computadoras de otro que rentas por hora: AWS, Azure, GCP te prestan máquinas, redes y servicios administrados. Pagas por no comprar hardware — y por no administrarlo tú.

- **Request_id en la respuesta** (`X-Request-Id` header): cuando el usuario reporta "me falló", te da el hilo exacto.
- **Loguea decisiones, no solo errores**: "task created", "payment processed", "rate limit hit" — los hechos normales son lo que reconstruye la película.
- **Duración siempre**: `duration_ms` en cada línea — los problemas de rendimiento se ven antes de que estallen.

32.9. Ejercicio

1. De esta línea sobra algo: `logger.info({"email": u.email, "password_hash": u.password_hash, "token": jwt}, "user registered")`. ¿Qué va y qué queda?
2. El usuario reporta "me dio error a las 15:14". Con logs JSON + request_id, ¿cuál es tu primera query de búsqueda?
3. ¿Por qué `console.log(err)` no basta aunque "imprime el error"?

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

32.10. Mini reto

Dado un log "500 a las 3:14", reconstruir qué pasó usando el request ID.

Soluciones

Ejercicio.

1. Van fuera `password_hash` (nunca) y `token` (nunca — es una llave viva). `email` es dato personal: en muchos contextos se loguea `user_id` en su lugar. Queda: `{user_id, request_id, msg}`.
2. `request_id` si el usuario lo tiene (del header `X-Request-Id` que vio en la respuesta); si no: `level=error AND ts 15:14 AND user_id=42` — y con el `request_id` de esa línea, filtras *todas* las líneas del request para ver la película completa.
3. Porque no responde las preguntas que sí importan: ¿qué request era? ¿de qué usuario? ¿en qué endpoint? ¿después de qué? Un error sin contexto es un síntoma; un error con `request_id` es una historia.

Mini reto. `level=error AND ts 3:14` → encuentras la línea del 500 con su `request_id` → filtras todo ese id → ves: request started → query a BD → timeout de la API externa → error → status 500. La historia se reconstruye sin tocar el código ni reproducir el bug — eso es observabilidad.

32.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: background job / cola

Un **job** es trabajo que se hace sin que el usuario espere: mandar el email, generar el PDF. Va a una **cola** y un *worker* lo procesa en segundo plano — la respuesta HTTP sale inmediata.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: Docker

Docker empaqueta tu app con todo lo que necesita (runtime, librerías, config) en una **imagen** — el mismo paquete corre idéntico en tu laptop y en producción. Es la respuesta a "en mi máquina sí funcionaba".

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

32.12. Lo que deberías saber hacer ahora

- ☐ Configurar logging JSON con niveles en tu stack
- ☐ Generar y propagar un `request_id` por middleware
- ☐ Decir qué campos jamás van a un log
- ☐ Reconstruir un request completo filtrando por `request_id`
- ☐ Usar `duration_ms` y niveles para que las líneas cuenten historias

33 24. Necesitamos la misma app en tres lugares distintos

Tu máquina, CI y prod tienen BD, secretos y orígenes distintos

> **En una frase**
>
> El mismo código corre en tu laptop (`localhost:5432`), en CI (una BD desechable) y en producción (RDS con TLS y un secreto que jamás toca tu editor). La diferencia no puede vivir en el código: vive en **variables de entorno**, validadas al arrancar para que la app reviente *antes* de aceptar tráfico si falta algo — no a las 3am.

33.1. El problema

`DATABASE_URL = "postgres://postgres:postgres@localhost:5432/taskflow"` funciona perfecto… en tu máquina. En prod esa línea es: el host que no existe, la contraseña que está en el repo para siempre, y el código que hay que *editar para desplegar* (cada deploy una rama distinta — el caos garantizado). Y el modo silencioso de morir: sin validación, la app arranca feliz y explota en el *primer* request que toca la BD.

> **Explicación de: repositorio**
>
> Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

> **Explicación de: puerto**
>
> Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = “la app que escucha en el departamento 3000”. Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

> **Explicación de: localhost / 127.0.0.1**
>
> **localhost** significa “esta misma máquina” — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el “servidor” y el “cliente” viven en tu laptop: por eso todo es

`localhost:3000`.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

33.2. Cómo lo resuelve un equipo

Un equipo serio aplica dos ideas de 12-factor:

1. **La configuración vive en el entorno, no en el código** — el mismo artefacto (binario/imagen) corre en dev, CI y prod; lo que cambia es el set de variables. Ningún secreto toca el repositorio: `.env` está en `.gitignore`, `.env.example` documenta las llaves sin valores.
2. **Fail fast al arranque** — al boot, la app *parsea y valida* toda su config: si falta `DATABASE_URL` o `JWT_SECRET`, revienta inmediatamente con un error claro. Un deploy que no arranca es un deploy que no hace daño; uno que arranca roto es una 3am.

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

33.3. Conceptos nuevos

- U Variables de entorno y 12-factor: la configuración vive fuera del código
- TS `process.env` + zod · Py `pydantic-settings` · Go `os.Getenv` + validación manual
- U `.env`, `.env.example`, y por qué `.env` jamás entra a git
- U Fail fast: la app revienta al arrancar si falta `DATABASE_URL`, no a las 3 AM

33.4. La explicación visual

```
                MISMO artefacto (mismo binario/imagen/commit)

   ENTORNO      dev (tu laptop)      CI (tests)          prod
   DATABASE_URL localhost:5432     postgres://ci_db     postgres://rds...?sslmode=require
   JWT_SECRET   dev-secret-123     test-secret          (secret manager)
   CORS_ORIGIN  http://localhost   http://localhost      https://taskflow.app
                :5173              :5173
   LOG_LEVEL    debug              info                 info
```

El código nunca pregunta "¿dónde estoy?" — pregunta "¿cuál es mi `DATABASE_URL`?" y el entorno responde.

33.5. Implementación

Módulo `config` que valida todo al arranque — si algo falta, la app no abre el puerto:

33.5.1. TypeScript

```ts
import { z } from "zod";

const Env = z.object({
  DATABASE_URL: z.string().url(),
  JWT_SECRET: z.string().min(32),
  PORT: z.coerce.number().default(3000),
  CORS_ORIGIN: z.string().url(),
  NODE_ENV: z.enum(["development", "test", "production"]).default("development"),
});

export const env = Env.parse(process.env);   // throws at boot if anything's wrong
// import-time crash → "DATABASE_URL: Required" - before the port opens
```

33.5.2. Python

```python
from pydantic_settings import BaseSettings
from pydantic import Field

class Settings(BaseSettings):
    database_url: str
    jwt_secret: str = Field(min_length=32)
    port: int = 8000
```

```
    cors_origin: str
    environment: str = "development"

    class Config:
        env_file = ".env"                        # dev convenience; prod injects real env

settings = Settings()                            # ValidationError at import → app never starts
```

33.5.3. Go

```
type Config struct {
    DatabaseURL string
    JWTSecret   string
    Port        int
    CORSOrigin  string
}

func LoadConfig() (Config, error) {
    cfg := Config{
        DatabaseURL: os.Getenv("DATABASE_URL"),
        JWTSecret:   os.Getenv("JWT_SECRET"),
        CORSOrigin:  os.Getenv("CORS_ORIGIN"),
        Port:        8080,
    }
    if cfg.DatabaseURL == "" { return cfg, errors.New("DATABASE_URL is required") }
    if len(cfg.JWTSecret) < 32 { return cfg, errors.New("JWT_SECRET must be 32+ chars") }
    if cfg.CORSOrigin == "" { return cfg, errors.New("CORS_ORIGIN is required") }
    return cfg, nil
}
// main(): cfg, err := LoadConfig(); if err != nil { log.Fatal(err) } - never reaches ListenAndServe
```

Prompt listo para tu AI

```
Centraliza la configuración de mi API [Express/FastAPI/net-http] en un
módulo config. Requisitos: leer DATABASE_URL, JWT_SECRET, PORT,
CORS_ORIGIN y ENV de variables de entorno; validarlas al arranque
(zod/pydantic-settings/manual) con tipos y mínimos (JWT_SECRET  32
chars) y que la app falle al boot con mensaje claro si falta alguna;
defaults solo para cosas seguras (PORT); .env para dev en .gitignore y
.env.example con las llaves documentadas sin valores; el resto del
código importa config, nunca lee process.env/os.environ directamente.
```

33.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: un solo módulo lee el entorno, lo valida completo al arranque, y el resto del código importa `config` — nunca toca `process.env`/`os.environ` directamente. Los defaults solo existen para lo seguro (`PORT`); los secretos *no* tienen default.

i Otras cimentaciones: dónde vive la config

Alternativa	Qué es	Qué te cuesta	Cuándo
Variables de entorno	Valores por proceso, estándar 12-factor	Nada tipado por naturaleza — por eso se valida al boot	El default, todos los entornos
`.env` file	Las env vars escritas en un archivo para dev	Tentador de commitear → `.gitignore` siempre	Solo dev/CI local
Secret manager	AWS SM, Vault, Doppler — secretos con rotación y auditoría	Servicio extra	Prod serio (nu-04)
Archivo de config (yaml/toml)	Config versionada en el repo	Los secretos no pueden entrar ahí	Config *no* secreta (features, límites)

Regla: **secretos → entorno/secret manager; preferencias → lo que quieras; nada secreto → jamás al repo.**

33.7. Errores comunes

Error	Por qué pasa	Fix
`.env` commiteado	"Es solo para dev"	`.gitignore` + `.env.example` — y si ya entró, rotar el secreto (el historial lo guarda)
Leer env vars dispersas por el código	`os.environ["X"]` en 14 archivos	Un módulo `config` — la única puerta al entorno
Defaults para secretos	`os.getenv("JWT_SECRET", "changeme")`	Sin default → la app no arranca sin secreto real
Validar al primer uso	"Si falta, el request falla"	Validar al boot — el deploy roto se detecta al deployar, no a las 3am
Config distinta por entorno en el código	`if prod: url = ...`	El código no sabe dónde está — el entorno decide

33.8. Buenas prácticas

- **`.env.example` actualizado en cada PR** que agrega una variable — es la documentación viva de qué necesita la app para arrancar.
- **El objeto config se inyecta**, no se importa globalmente donde se pueda evitar — facilita tests (otra config) y deja explícitas las dependencias.

> Explicación de: dependencia
>
> Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

- **Nombres estables entre entornos**: `DATABASE_URL` es `DATABASE_URL` en dev y en prod — lo que cambia es el *valor*.
- **Loguea la config *no* secreta al arranque** (puerto, entorno, orígenes) — "¿en qué entorno está corriendo esto?" nunca debe ser una adivinanza. Los secretos, jamás (cap. 23).

33.9. Ejercicio

1. ¿Por qué `JWT_SECRET` no puede tener valor por defecto aunque `PORT` sí?
2. Tu app arrancó en prod y funciona… hasta el primer login, donde explota porque `JWT_SECRET` está vacío. ¿Qué práctica de este capítulo lo hubiera evitado y en qué momento?
3. Escribe el `.env.example` completo de Taskflow (todas las llaves que el proyecto usa hasta ahora).

> Explicación de: autenticación
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

33.10. Mini reto

Quitar todos los valores hardcodeados de Taskflow y arrancar en un entorno limpio.

> Soluciones
>
> **Ejercicio.**
>
> 1. `PORT` con default es una conveniencia inofensiva: si falta, 3000 sirve. `JWT_SECRET` con default es un *secreto conocido* — cualquiera que lea tu repo firma tokens válidos. Los secretos no tienen fallback: o hay secreto real, o la app no arranca.
> 2. **Fail fast al boot**: el `Env.parse`/`Settings()`/`LoadConfig()` habría reventado *en el deploy* con "JWT_SECRET is required" — descubierto al deployar (rollback inmediato), no en el

primer login de un usuario real.

3.
```
DATABASE_URL=postgres://user:pass@host:5432/dbname
JWT_SECRET=change-me-32-chars-minimum-xxxx
PORT=3000
CORS_ORIGIN=http://localhost:5173
SLACK_WEBHOOK_URL=https://hooks.slack.com/services/...
WEBHOOK_SECRET=
LOG_LEVEL=info
NODE_ENV=development
```

Mini reto. `grep` por `localhost`, `5432`, `secret`, `http://` en el código → todo valor que cambia entre entornos pasa a env var → `config` validado → arrancar con `env -i` (entorno vacío) debe fallar con mensajes claros nombrando cada variable faltante — esa es la prueba real de que nada quedó hardcodeado.

33.11. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

33.12. Lo que deberías saber hacer ahora

- ☐ Centralizar la config en un módulo validado al arranque
- ☐ Explicar qué va a env vars, qué a `.env`, qué jamás al repo
- ☐ Escribir un `.env.example` que documente cada variable

☐ Hacer que la app falle al boot (no al primer request) sin `DATABASE_URL`
☐ Correr el mismo artefacto en dev, CI y prod cambiando solo el entorno

34 25. Necesitamos que los errores escalen como en un equipo real

El frontend no puede programar contra el caos

> **En una frase**
>
> El cap. 6 te enseñó el formato `{error:{code}}`. Ahora el problema es de escala: 40 endpoints, errores de BD, de dominio, de terceros — si cada uno inventa su traducción, el frontend programa contra el caos. La solución es una **jerarquía de errores de dominio + un handler global** que es la única frontera donde un error se convierte en HTTP.

34.1. El problema

`users` devuelve `{message: "nope"}`, `tasks` devuelve un stack trace, `projects` devuelve 200 con `{error: null, data: null}` cuando algo falla. El frontend necesita un `if` por endpoint — y el día que cambias un texto ("not found" → "no encontrado"), un `if (err.message === "not found")` de alguien explota en producción. Los mensajes son para humanos; **las máquinas necesitan códigos estables**.

> **Explicación de: null / None / nil**
>
> **null** (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

> **Explicación de: return**
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

> **Explicación de: producción**
>
> **Producción** (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

34.2. Cómo lo resuelve un equipo

Un equipo serio separa dos mundos con una frontera clara:

- **Dominio**: el servicio lanza `TaskNotFoundError`, `ForbiddenError`, `ValidationError` — errores que *hablan el idioma del negocio* y no saben nada de HTTP.
- **Transporte**: un handler global (middleware de error / exception handler / mapper central) traduce cada error de dominio a su status + código estable. Es el *único* lugar donde vive esa tabla.

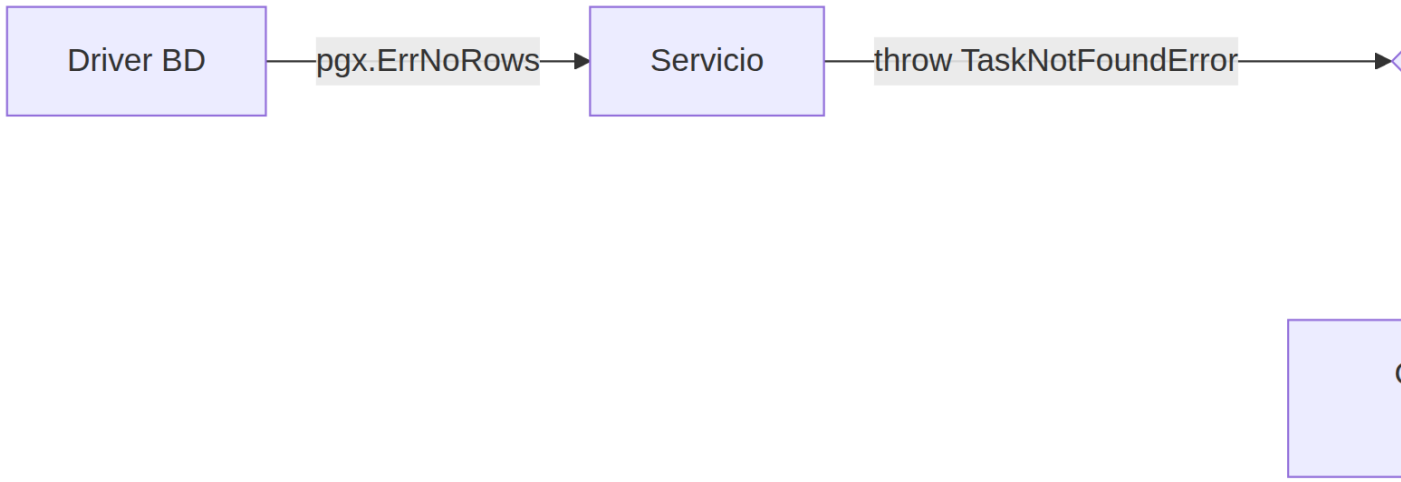
> Explicación de: dominio
>
> Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

El contrato hacia afuera: `{"error": {"code": "TASK_NOT_FOUND", "message": "..."}}` — el `code` es API (estable, testeable), el `message` es texto (puede cambiar, traducirse).

34.3. Conceptos nuevos

- U Jerarquía de errores de dominio: `DomainError → NotFoundError → TaskNotFoundError`
- TS Clases `extends Error` · Py Excepciones propias · Go Sentinel errors + `errors.Is/As`
- U Handlers globales: la frontera entre "error de negocio" y "respuesta HTTP"
- U El contrato de error: `{"error": {"code": "TASK_NOT_FOUND"}}` — códigos estables vs mensajes cambiantes
- U 500: qué se devuelve (poco) vs qué se loguea (todo)

34.4. La explicación visual



El servicio lanza *significado*; el handler traduce; el cliente ve contrato; el detalle feo queda en logs.

> Explicación de: log / logging estructurado
>
> Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

34.5. Implementación

Jerarquía + mapper central en los tres stacks:

34.5.1. TypeScript

```
export class DomainError extends Error {
  constructor(public code: string, message: string, public status = 400) {
    super(message);
  }
}
export class NotFoundError extends DomainError {
  constructor(what: string) { super(`${what.toUpperCase()}_NOT_FOUND`, `${what} not found`, 404); }
}
```

```
// the ONLY translation point - registered last
app.use((err, req, res, next) => {
  if (err instanceof DomainError) {
    return res.status(err.status).json({ error: { code: err.code, message: err.message } });
  }
  logger.error({ err, request_id: req.id }, "unhandled error");   // everything goes to logs
  res.status(500).json({ error: { code: "INTERNAL", message: "internal error" } });
});
// service: throw new NotFoundError("task") - knows no HTTP
```

34.5.2. Python

```
class DomainError(Exception):
    code = "DOMAIN"
    status = 400

class NotFoundError(DomainError):
    status = 404
    def __init__(self, what: str):
        self.code = f"{what.upper()}_NOT_FOUND"
        super().__init__(f"{what} not found")

@app.exception_handler(DomainError)
async def domain_handler(request, exc: DomainError):
    return JSONResponse({"error": {"code": exc.code, "message": str(exc)}},
                        status_code=exc.status)

@app.exception_handler(Exception)
async def unhandled_handler(request, exc: Exception):
    logger.exception("unhandled", extra={"ctx": {"request_id": request.state.request_id}})
    return JSONResponse({"error": {"code": "INTERNAL", "message": "internal error"}},
                        status_code=500)
```

34.5.3. Go

```
var ErrNotFound = errors.New("not found")

type DomainError struct {
    Code    string
    Message string
    Status  int
}
func (e *DomainError) Error() string { return e.Message }

func writeErr(w http.ResponseWriter, err error) {
```

```
    var de *DomainError
    switch {
    case errors.Is(err, pgx.ErrNoRows):                    // driver error → domain
        de = &DomainError{"TASK_NOT_FOUND", "task not found", 404}
    case errors.As(err, &de):
        // already a domain error
    default:
        slog.Error("unhandled", "err", err)
        de = &DomainError{"INTERNAL", "internal error", 500}
    }
    writeJSON(w, de.Status, map[string]any{
        "error": map[string]string{"code": de.Code, "message": de.Message}})
}
```

Prompt listo para tu AI

```
Implementa un sistema de errores coherente para mi API
[Express/FastAPI/net-http]. Requisitos: jerarquía DomainError con
status+code (NotFoundError, ValidationError, ForbiddenError,
UnauthorizedError, ConflictError, ExternalServiceError), un handler
global que sea la ÚNICA frontera dominio→HTTP y devuelva siempre
{"error":{"code","message"}} más request_id en logs para errores
inesperados, el 500 genérico sin stack trace en la respuesta, y tests
que verifiquen que NINGÚN endpoint filtra stack trace ni devuelve otro
formato.
```

34.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres hacen lo mismo — errores que *significan* algo en el dominio, un solo lugar que los traduce a HTTP, y un contrato estable para el cliente. Cambia el mecanismo (herencia vs excepciones vs sentinel errors), no el diseño.

Detalle: los tres mecanismos

- **TS**: clases `extends Error` + `instanceof` — la jerarquía es literal (`NotFoundError` *es un* `DomainError`).
- **Python**: excepciones con atributos de clase + un handler por tipo — FastAPI elige el handler por la clase de la excepción.
- **Go**: sin excepciones — el error es un *valor* que viaja por return; `errors.Is/As` pregunta "¿es este tipo?" y `writeErr` es el mapper manual. Más verboso, más explícito: el error está en la firma.

RFC 9457 (problem details) es la alternativa "estándar" al contrato propio: `{"type","title","status","detail"}` — misma idea, vocabulario IETF. Lo que importa no es el formato elegido sino que sea *uno solo y estable*.

34.7. Errores comunes

Error	Por qué pasa	Fix
`if (err.message === "task not found")` en el cliente	Los mensajes eran la API	Códigos estables: `code` es contrato, `message` es decoración
Stack trace en el body del 500	El framework lo hacía en dev	Handler global: al cliente "INTERNAL", al log todo
Errores de BD burbujeando al handler	`pgx.ErrNoRows` llega crudo	El servicio traduce driver→dominio en su frontera
Status 200 con error dentro	"Para no romper el frontend"	El status *es* parte del contrato — 4xx/5xx existen por algo
Handler de error por endpoint	Copia pega del try/catch	UN handler global + errores de dominio que viajan solos

34.8. Buenas prácticas

- **Catálogo de códigos**: `TASK_NOT_FOUND`, `VALIDATION`, `UNAUTHORIZED`, `FORBIDDEN`, `RATE_LIMITED`, `CONFLICT`, `EXTERNAL_SERVICE`, `INTERNAL` — documentados, estables, testeados. Son API pública: se agregan, no se renombran.
- **El 500 no improvisa**: mismo formato que el resto — el cliente no tiene un parser para "cuando todo explota".
- **Errores de dominio pequeños y específicos**: `NotFoundError` con el *qué* — el código sale solo (`TASK_NOT_FOUND`), sin decidirlo en cada throw.
- **El handler global también loguea**: request_id + error completo + contexto (cap. 23) — el cliente ve poco, tú ves todo.

34.9. Ejercicio

1. ¿Por qué `message` no puede ser el contrato aunque sea "estable hoy"? Da dos razones.
2. El driver lanza `pgx.ErrNoRows`. ¿Quién lo traduce a `TASK_NOT_FOUND` — el storage, el servicio o el handler global? Defiende tu respuesta.
3. Agrega `ConflictError` (409, para email duplicado) a la jerarquía: ¿qué `code` debería producir `POST /auth/register` con email ya tomado y por qué no 422?

> Explicación de: autenticación
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

34.10. Mini reto

Garantizar que ningún endpoint filtra un stack trace — con tests.

Soluciones

Ejercicio.

1. (a) Los mensajes cambian: traducción, mejora de copy, un typo — cada cambio es un breaking change invisible. (b) Las máquinas comparan strings con edge cases (espacios, mayúsculas); un `code` es un identificador *diseñado* para compararse.
2. **El servicio** — es su frontera: el storage reporta "no había fila" (hecho técnico), el servicio decide "eso significa task no encontrada" (significado de negocio). El handler global solo ve errores de dominio ya traducidos. Si el driver llega al handler crudo, cambiar de Postgres a MySQL cambiaría tu API pública.
3. `{"error":{"code":"EMAIL_TAKEN","message":"email already registered"}}` con 409 — no es un body malformado (422 sería "tu JSON está mal"), es un *conflicto de estado*: el recurso ya existe. 409 es el status honesto para eso.

Mini reto. Un test que pega a cada endpoint con inputs que exploten (DB caída, error forzado) y verifica: status 500 **y** body parsea como `{error:{code:"INTERNAL"}}` **y** no contiene "at", "Traceback", ".go:", ni "node_modules". El test es la garantía permanente — si alguien rompe el handler global, este test lo grita antes que prod.

34.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: estado (state)

El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: recurso (REST)

Un **recurso** es la "cosa" que tu API maneja: `tasks`, `users`, `projects`. Las URLs los nombran (`/tasks/5`), los verbos actúan sobre ellos. Diseñar REST = nombrar bien tus recursos.

34.12. Lo que deberías saber hacer ahora

- ☐ Diseñar una jerarquía de errores de dominio en tu stack
- ☐ Centralizar la traducción dominio→HTTP en un handler global
- ☐ Mantener `code` estable y `message` libre en el contrato
- ☐ Traducir errores del driver a dominio en la frontera correcta
- ☐ Testear que ningún endpoint filtra stack trace ni formato raro

35 26. Necesitamos defendernos de lo que el frontend no puede ver

El checklist de seguridad antes de exponer una API

> **En una frase**
>
> Todo lo anterior fue *construir* seguridad pieza por pieza. Este capítulo es **auditar**: recorrer tus endpoints reales con el OWASP API Top 10 en la mano y responder "¿dónde rompería esto si fuera el atacante?". La mayoría de las defensas ya las tienes — este capítulo es demostrarlo y tapar lo que falta.

35.1. El problema

La API funciona, los tests pasan… y alguien pega en el chat un `?user_id=1` que devuelve datos de otro usuario. La seguridad no es una feature que se agrega: es una *propiedad* que se audita — cada endpoint, cada input, cada error. OWASP no es teoría de conferencia: es la lista de las formas reales en que APIs como la tuya se rompen cada semana.

> **Explicación de: array / lista / slice**
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> **Explicación de: return**
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

35.2. Cómo lo resuelve un equipo

Un equipo serio hace la auditoría *sobre los endpoints que existen*, no sobre un checklist abstracto: por cada ruta — ¿quién puede llamarla, qué devuelve, qué pasa con input malicioso, qué loguea? La OWASP API Top 10 traducida a Taskflow es la tabla de abajo: cada amenaza mapea a un control que

ya construimos en los capítulos 4–18 — la auditoría es verificar que cada control está *activo en cada endpoint*, no en los que te acuerdas.

35.3. Conceptos nuevos

- U OWASP API Top 10 traducido a tus endpoints: BOLA, auth rota, exposición de datos
- U Inyección (SQL, comandos): por qué parámetros/ORM es tu defensa y dónde aún puedes fallar
- U Headers de seguridad en una API pura: HSTS, X-Content-Type-Options
- U Rate limiting generalizado: qué endpoints se protegen primero
- U Validación como seguridad: el 422 también es una muralla

35.4. La auditoría — OWASP API Top 10 contra Taskflow

Amenaza OWASP	En Taskflow significa	Tu control (capítulo)	Estado
BOLA — Broken Object Level Authorization	`GET /tasks/{id}` devuelve tareas ajenas	`WHERE user_id` en la query + 404 a lo ajeno (cap. 17)	por endpoint — auditar cada uno
Broken Authentication	login sin límite, tokens eternos	bcrypt, JWT corto, rate limit, refresh rotado (cap. 14–18)	
Excessive Data Exposure	`GET /me` filtra `password_hash`	Contratos de salida whitelist (cap. 5, 16)	
Unrestricted Resource Consumption	body gigante, uploads infinitos, listas sin paginar	Límites de body/upload, paginación, rate limit (cap. 12, 18, 22)	parcial
Broken Function Level Authorization	`viewer` llamando `DELETE /projects`	`requireRole` por endpoint (cap. 17)	auditar uno por uno
SSRF	"dame la URL del avatar a descargar" → tu servidor visita `169.254.169.254`	Allowlist de hosts si algún día aceptas URLs	diseño
Security Misconfiguration	headers ausentes, debug on, errores verbosos	Handler global + headers (abajo)	este capítulo
Injection	`WHERE email = '" + email + "'`	Queries parametrizadas *siempre* (cap. 9)	por disciplina
Improper Inventory	`/v1/tasks` viejo sin auth aún vivo	Versionar y matar lo viejo — inventario de rutas	proceso
Unsafe Consumption of APIs	Creer lo que el webhook dice sin verificar	Firma HMAC + dedupe (cap. 22)	

Los no son permanentes: se ganan *por endpoint y por PR* — por eso la auditoría es una tabla viva, no un recuerdo.

35.5. Implementación

Capítulo agnóstico — dos piezas que faltaban:

Headers de seguridad (un middleware, cualquier stack):

```
Strict-Transport-Security: max-age=31536000      # HTTPS siempre
X-Content-Type-Options: nosniff                  # no adivines el tipo
X-Frame-Options: DENY                            # tu API no se embebe
Cache-Control: no-store                          # respuestas de API no se cachean
```

El checklist por endpoint (la auditoría operativa):

```
GET /tasks/{id}
  [ ] auth requerida (middleware)
  [ ] ownership en la query (WHERE user_id)
  [ ] 404 para lo ajeno, no 403 (anti-enumeración)
  [ ] solo campos del contrato público
  [ ] query parametrizada
  [ ] rate limit si es sensible
  [ ] errores por el handler global (sin stack trace)
```

Prompt listo para tu AI

```
Audita la seguridad de mi API [Express/FastAPI/net-http] contra OWASP
API Top 10. Para cada endpoint: identifica la amenaza aplicable (BOLA,
function-level auth, data exposure, injection, resource consumption),
verifica el control que YA existe en el código (ownership en query,
requireRole, contrato de salida, parametrización, rate limit, headers),
y genera la tabla endpoint × amenaza × control × estado con las
correcciones concretas de lo que falte. Incluye tests de regresión para
cada control (viewer→403, ajeno→404, injection→literal).
```

35.6. ¿Por qué así y no "instalar un plugin de seguridad"?

Lo único que necesitas llevarte: la seguridad de API no es una librería que se enchufa — son ~8 decisiones ya tomadas que deben estar *activas en cada endpoint*. El plugin pone headers; no pone `WHERE user_id` en tu query.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

i Detalle: lo que el framework sí te da

Te lo da el framework/librería	Sigue siendo tuyo
Headers de seguridad (helmet, secure-headers)	BOLA: `WHERE user_id` en cada query
Parseo seguro de JSON/multipart	Contratos de salida (whitelist)
TLS terminado por el proxy	Rate limiting por endpoint sensible
Query builders parametrizados	Roles por recurso, dedupe de webhooks

La regla honesta: las herramientas cubren el *transporte*; la seguridad de *tu dominio* (quién toca qué) la escribes tú o no existe.

35.7. Errores comunes

Error	Por qué pasa	Fix
"Ya está securizado" por memoria	El control existe en 7 de 9 endpoints	La tabla endpoint×control — auditar, no recordar
Confiar la seguridad al frontend	"El botón está oculto para viewers"	El navegador miente; la API enforza — siempre
CORS como defensa	"Solo mi dominio puede llamar"	CORS protege al *usuario*, curl no pide permiso (cap. 18)
Errores verbosos en prod	Debug útil	Stack trace = mapa del código para el atacante → handler global
Rate limit solo en login	"Ahí era el problema"	También refresh, register, forgot-password, OTP, endpoints caros
Endpoints viejos vivos	`/v1/tasks` quedó sin auth tras la migración	Inventario de rutas: lo que no se audita, se expone

35.8. Buenas prácticas

- **La tabla de auditoría vive en el repo** (docs/SECURITY.md): cada endpoint nuevo agrega su fila en el mismo PR — seguridad como checklist mergeable, no como auditoría anual.
- **Tests de regresión de seguridad** (cap. 28): `viewer → 403`, `ajeno → 404`, `inyección → literal` son tests permanentes — el control que no se testea se rompe sin ruido.
- **Principio de privilegio mínimo** en todo: el usuario de BD de la app no es `postgres`; el token no vive 30 días; el endpoint no devuelve campos "por si acaso".

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

- **Asumir que el cliente es hostil**: toda entrada se valida (cap. 4) — el 422 *es* seguridad, no solo UX.

35.9. Ejercicio

1. Un pentester reporta: "`GET /tasks/41` da 403 y `GET /tasks/42` da 404 — así sé que la 41 existe y es de otro". ¿Qué está mal y cómo se arregla?
2. Tu API acepta `GET /avatar?url=...` y el servidor descarga esa URL. ¿Qué amenaza OWASP es y cuál es la mitigación?
3. De la tabla de auditoría: ¿por qué "BOLA" no se puede arreglar con un middleware como sí se arreglan los headers?

35.10. Mini reto

Encontrar el endpoint donde un `viewer` podría hacer algo que no debería.

Soluciones

Ejercicio.

1. El 403 delata existencia (enumeración). Fix: lo ajeno responde 404 idéntico a lo inexistente — la regla del cap. 17. `WHERE public_id AND user_id` produce 0 filas en ambos casos → mismo 404, sin pista.
2. **SSRF** (Server-Side Request Forgery): el atacante hace que *tu servidor* visite `http://169.254.169.254/latest/meta-data` (la metadata de la nube, con credenciales) o tu red interna. Mitigación: allowlist de hosts permitidos, nunca URLs arbitrarias; si es avatar de terceros, descargar en un worker aislado sin acceso a metadata.
3. Porque BOLA es *semántica*: el middleware no sabe que la tarea 41 pertenece al usuario 7 — esa relación vive en la BD y se enforza en la query. Headers/cookies son transporte; ownership es dominio. Por eso se audita por endpoint y se testea como regresión.

Mini reto. Los sospechosos de siempre: `PATCH/DELETE` de proyectos (¿verifica rol editor/admin o solo membresía?), `POST /projects/{id}/members` (¿quién puede invitar?), endpoints de exportación (¿viewer exporta datos que no ve?), y cualquier ruta agregada "rápido" sin pasar por `requireRole`. El patrón del hallazgo: endpoint que verifica *quién eres* pero no *qué puedes hacer aquí*.

35.11. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: parámetro / argumento

El **parámetro** es el hueco declarado en la función (`def f(x)` — `x` es parámetro); el **argumento** es el valor concreto que le pasas (`f(42)` — `42` es argumento). Mismo dato, dos momentos: declaración vs uso.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: cookie

Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña` →$2b10…‘. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques “esto realmente vino de Stripe”.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: ORM

Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

35.12. Lo que deberías saber hacer ahora

☐ Auditar tus endpoints contra OWASP API Top 10 con una tabla real
☐ Explicar BOLA y por qué la defensa vive en la query
☐ Mantener headers de seguridad y errores no-verbosos en prod
☐ Convertir cada control de seguridad en un test de regresión
☐ Reconocer SSRF, enumeración y exposición de datos en código real

Parte VIII

Sección II · Testing

36 27. Necesitamos confiar en el código que no estamos mirando

Sin tests, el miedo es la arquitectura

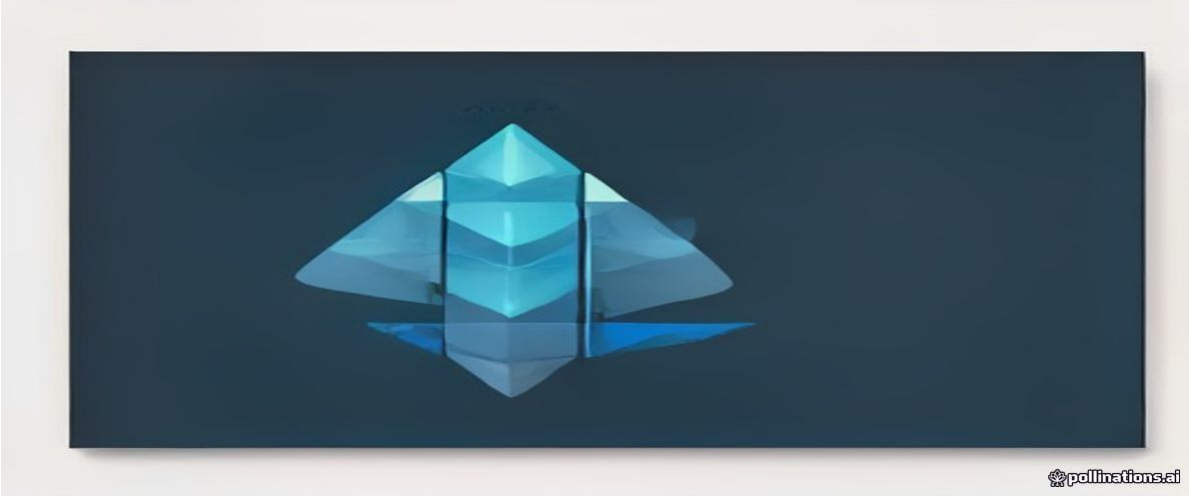


Figura 36.1: Parte 6 — Testing

> **En una frase**
>
> Cada refactor rompe algo en otro lugar, y sin tests te enteras por un usuario. La suite no existe para "cumplir cobertura": existe para que puedas **cambiar el código sin miedo** — muchos tests baratos en la capa de servicio, pocos caros en los bordes, y cada test respondiendo "¿qué rompería si esto se borra?".

36.1. El problema

El refactor del cap. 7 separó handler de servicio — ¿y quién garantiza que `complete_task` sigue dando el punto una sola vez, que el título vacío se rechaza, que la prioridad 99 no entra? "Lo probé a mano" es la respuesta que dura hasta el próximo commit de otro. El miedo a tocar código viejo no es prudencia: es la deuda de no tener tests.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

36.2. Cómo lo resuelve un equipo

Un equipo serio testea **donde viven las decisiones**: la capa de servicio (reglas de negocio puras, sin HTTP ni BD si se puede) — ahí un test es una función, un input, un assert. La pirámide: muchos unitarios rápidos, menos de integración, pocos end-to-end. Y el criterio que evita los tests de juguete: cada test debe *matar un bug posible* — si borras una línea del servicio y ningún test se queja, esa línea no está testeada.

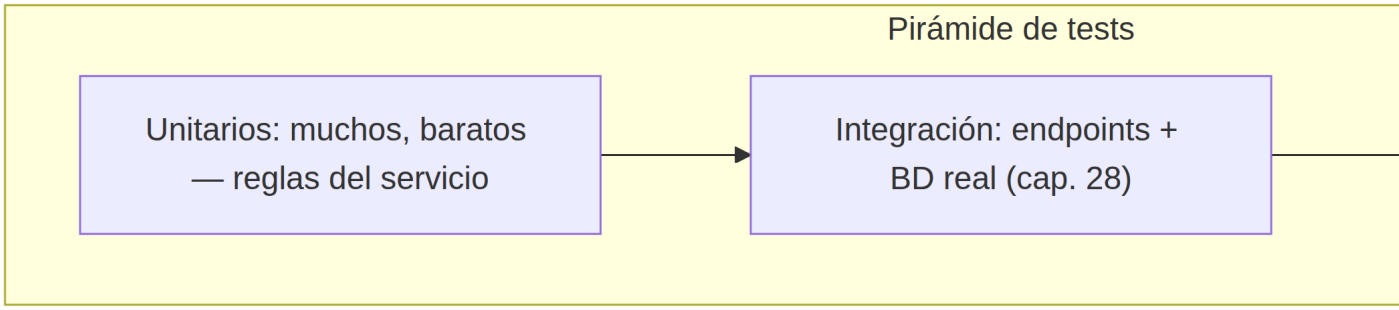
Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

36.3. Conceptos nuevos

- U Qué testear y qué no: la pirámide en backend (servicios > endpoints > e2e)
- TS vitest · Py pytest + `conftest` · Go `go test` + table-driven tests (estilo exportable)
- U Fixtures: `beforeEach` vs conftest vs helpers — preparar mundo, ejecutar, limpiar
- U Parametrize/table tests: un test, veinte casos

36.4. La explicación visual



La base ancha: el servicio es donde viven las reglas (“no borrar tarea con subtareas”, “el punto se da una vez”) — ahí cada test es barato y preciso. El e2e certifica que las piezas *conectan*, no que cada regla existe.

36.5. Implementación

Tests del servicio — unidad pura en los tres estilos:

36.5.1. TypeScript

```typescript
import { describe, it, expect } from "vitest";

describe("createTask", () => {
  it("rejects empty title", () => {
    expect(() => createTask(userId, "")).toThrow(ValidationError);
  });

  it.each([["  "], ["a".repeat(201)]])("rejects bad title: %j", (title) => {
    expect(() => createTask(userId, title)).toThrow(ValidationError);
  });

  it("trims and persists a valid title", () => {
    const t = createTask(userId, "  pay rent  ");
    expect(t.title).toBe("pay rent");
    expect(t.done).toBe(false);
  });
});
```

36.5.2. Python

```python
import pytest

class TestCreateTask:
    def test_rejects_empty_title(self):
        with pytest.raises(ValidationError):
            create_task(user_id=1, title="")

    @pytest.mark.parametrize("title", ["  ", "a" * 201])
    def test_rejects_bad_title(self, title):
        with pytest.raises(ValidationError):
            create_task(user_id=1, title=title)

    def test_trims_and_persists(self):
        t = create_task(user_id=1, title="  pay rent  ")
        assert t.title == "pay rent"
        assert t.done is False
```

36.5.3. Go

```go
func TestCreateTask(t *testing.T) {
    cases := []struct {
        name    string
        title   string
        wantErr bool
    }{
        {"empty title", "", true},
        {"blank title", "   ", true},
        {"too long", strings.Repeat("a", 201), true},
        {"valid", "pay rent", false},
    }
    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            task, err := createTask(1, tc.title)   // table-driven: one test, all cases
            if tc.wantErr {
                if err == nil { t.Fatal("expected error") }
                return
            }
            if err != nil { t.Fatal(err) }
            if task.Title != "pay rent" || task.Done { t.Fatalf("bad task: %+v", task) }
        })
    }
}
```

Prompt listo para tu AI

```
Escribe la suite de tests unitarios del servicio de tareas en mi stack
[vitest/pytest/go test]. Reglas a cubrir: título requerido (vacío,
solo-espacios, >200 chars), trim antes de persistir, prioridad por
defecto 3, done inicia false, complete_task solo otorga el punto UNA vez
(segundo complete → conflicto), no se puede borrar tarea con subtareas.
Usa [it.each/parametrize/table-driven] para los casos límite, fixtures
para el usuario de prueba, y nombres de test que lean como
especificación ("rejects_empty_title").
```

36.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: los tres estilos son el mismo test — *mundo preparado* → *acción* → *assert*. La tabla de casos (parametrize / table-driven) es la técnica universal para los límites: un test, veinte inputs.

Detalle: fixtures en cada mundo

- **TS/vitest**: `beforeEach` prepara el mundo por test — cada test nace en un estado limpio y conocido.
- **Python/pytest**: `conftest.py` declara fixtures que se *inyectan por nombre* — `def test_x(db, user)` recibe lo que necesita sin llamarlo.
- **Go**: sin fixtures de framework — helpers `newTestService(t)` que construyen el mundo explícitamente. Más verboso, más visible.

El estilo Go (table-driven) es exportable a cualquier lenguaje: cuando tus casos son datos + expectativa, una tabla > diez funciones copiadas.

36.7. Errores comunes

Error	Por qué pasa	Fix
Tests que testean el framework	`expect(res.status).toBe(200)` sin lógica propia	Testear *tu* regla: "viewer → 403", no "Express devuelve 200"
Acoplar al implementación	El test llama funciones privadas	Test contra el contrato: input → output/efecto
Fixtures compartidos mutables	"El user del test anterior ya existe"	Mundo limpio por test (beforeEach/fixture scope)
Tests que siempre pasan	`expect(true).toBe(true)` tras un refactor	El test debe fallar si borras la línea que protege
Cobertura como meta	"Lleguemos a 90 %"	Cobertura donde importa: reglas, fronteras de error, auth

36.8. Buenas prácticas

- **Nombres que son especificación**: `rejects_empty_title`, `second_complete_returns_conflict` — leer la suite es leer las reglas del negocio.
- **Un assert de significado por test** (o pocos): el test que verifica 14 cosas no dice cuál falló.
- **El test como documento de diseño** (TDD, Ap. G tickets 41–42): escribir "debería rechazar X" antes de tener X es diseñar la API desde su consumidor.
- **Rápido o no se corre**: la suite unitaria corre en segundos — si tarda minutos, los devs la saltan y es decoración.

EXTRA · Los tres niveles que nombran los roadmaps

EXTRA Cuando un roadmap o una entrevista dice "tipos de pruebas", se refiere a —

- **Pruebas unitarias** — una función aislada, sin BD ni red (cap-27)
- **Pruebas de integración** — la API contra la BD real (cap-28)
- **Pruebas funcionales / e2e** — el sistema completo como un usuario (cap-27, la punta cara

de la pirámide)

La pirámide del libro ordena exactamente esto: muchas unitarias, las suficientes de integración, pocas funcionales.

36.9. Ejercicio

1. Ordena por dónde testear: "el email único se enforza", "el JSON parsea", "un viewer no puede borrar", "la título se trimea". ¿Cuál va en unit, cuál en integración, cuál no se testea?
2. Este test existe: `it("works", () => { expect(createTask(1,"x").id).toBeTruthy() })`. ¿Qué le falta para ser un buen test?
3. Escribe la tabla de casos para `complete_task`: todos los estados desde los que se puede llamar y qué debe pasar.

Explicación de: roles / RBAC

Los **roles** agrupan permisos: *viewer* lee, *editor* escribe, *admin* gestiona. RBAC = el permiso depende del rol que tienes *en ese recurso* — puedes ser admin de un equipo y viewer de otro.

Explicación de: estado (state)

El **estado** es todo lo que el programa recuerda: las variables, la sesión, lo que muestra la UI. *Stateless* (sin estado) = el servidor no recuerda nada entre requests — cada request trae todo lo necesario.

36.10. Mini reto

Escribir el test que hubiera detectado tu último bug.

Soluciones

Ejercicio.

1. "Título se trimea" → **unit** (regla pura del servicio). "Email único" → **integración** (la constraint vive en la BD — cap. 28). "Viewer no puede borrar" → **integración** (necesita auth+roles reales). "El JSON parsea" → **no se testea**: es el framework, no tu código.

2. Todo: no dice *qué* debería pasar ni qué bug mata. Un buen test nombra la regla (`creates_task_with_default_priority`), verifica el *significado* (`priority === 3`, `done === false`), y falla si la regla se rompe — este test pasa aunque el servicio haga cualquier cosa que devuelva algo con id.

estado inicial	llamada	esperado
done=false, subtareas=0	complete	done=true, punto+1
done=true	complete	409 CONFLICT — ya completada
done=false, subtareas pendientes	complete	409 — subtareas abiertas
tarea de otro usuario	complete	404 (anti-enumeración)

3.

Mini reto. El patrón es siempre el mismo: reproduce el bug como test *que falla* (`it("returns 404 for foreign task")` → hoy devuelve 200 con datos ajenos) → arregla el código → el test queda como guarda permanente. Cada bug real merece su test: la suite es la memoria de todos los errores que ya no puedes volver a cometer.

36.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: constraint / restricción

Una **constraint** es una regla que la BD hace cumplir: `NOT NULL`, `UNIQUE`, `CHECK (price > 0)`, `FOREIGN KEY`. Es la última línea de defensa — el código puede tener bugs; la constraint no perdona.

Explicación de: trazas (tracing)

Una **traza** sigue un request a través de todo el sistema: entró por el gateway → llamó auth → consultó la BD → tardó 340ms en la query. Cuando algo anda lento, la traza dice exactamente dónde.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo

al reiniciar.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: TDD

TDD (Test-Driven Development) = escribir el test *antes* del código: primero defines qué debe pasar (rojo), luego lo haces pasar (verde), luego limpias. El test es la especificación que corre.

Explicación de: cobertura (coverage)

La **cobertura** es el % de tu código que los tests ejecutan. Útil como detector de "esto nadie lo prueba"; inútil como meta — 90 % cubierto no significa que los tests afirmen algo correcto.

36.12. Lo que deberías saber hacer ahora

- ☐ Escribir tests de servicio en tu stack (vitest/pytest/go test)
- ☐ Usar tables/parametrize para casos límite sin copiar tests
- ☐ Preparar el mundo con fixtures (beforeEach/conftest/helpers)
- ☐ Decidir qué se testea en unit vs integración vs e2e
- ☐ Escribir el test primero cuando la regla está clara (TDD)

37 28. Necesitamos testear la API completa sin romper la BD

Los endpoints tocan Postgres: ¿tests contra qué base de datos?

> **En una frase**
>
> El test unitario cubre la regla; el test de API cubre el **contrato completo**: ruta + middleware + auth + validación + query + respuesta, con Postgres de verdad. Las dos preguntas prácticas: ¿contra *qué* base de datos (una dedicada de test, con rollback por test), y ¿cómo se ejercita la API sin levantar el servidor (supertest/TestClient/httptest)?

37.1. El problema

Los tests del cap. 27 prueban `create_task` directo — pero el bug del cap. 17 (la tarea ajena que devolvía 200) *vivía en la integración*: el servicio funcionaba, la query no filtraba por dueño. Un bug de esos solo lo caza un test que pase **por el endpoint de verdad**: auth real, query real, BD real. Y ahí aparecen las preguntas de siempre: ¿los tests corren contra mi BD de dev? ¿qué pasa con los datos que crean?

> **Explicación de: base de datos**
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

> **Explicación de: query / consulta**
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

> **Explicación de: autenticación**
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

37.2. Cómo lo resuelve un equipo

Un equipo serio monta la infra de test una vez y la reutiliza siempre:

1. **BD de test dedicada**: `taskflow_test` — otra base en el mismo Postgres de Docker, con las mismas migraciones. Los tests *destruyen datos*: nunca contra dev, jamás contra prod.
2. **Aislamiento por transacción**: cada test corre dentro de una transacción que hace **rollback** al terminar — el test escribe, verifica, y la BD queda como si nada. Aislamiento gratis, sin scripts de limpieza, y rápido.
3. **La API sin socket**: supertest/TestClient/httptest llaman al

> Explicación de: migración
>
> Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

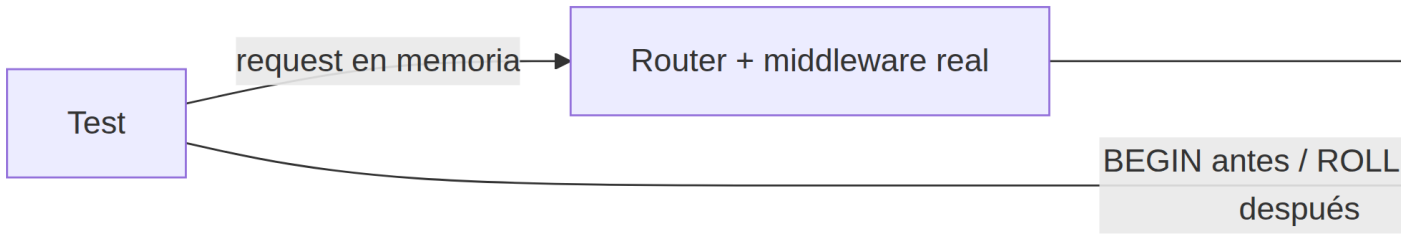
> Explicación de: transacción
>
> Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

router directo en memoria — HTTP real, sin puerto, sin `npm start` paralelo.

37.3. Conceptos nuevos

- U BD de test dedicada: `DATABASE_URL` apuntando a `taskflow_test`
- D El truco de la transacción con rollback: tests aislados, rápidos, sin limpieza
- TS supertest · Py `TestClient` · Go `httptest` — ejercitar la API sin socket real
- U Testear lo protegido: fabricar usuarios, sesiones y tokens en fixtures
- U Tests que son documentación: "viewer no puede mutar → 403" permanente

37.4. La explicación visual



Todo es real menos el socket y la durabilidad de los datos — por eso el test corre rápido y deja la BD limpia sola.

37.5. Implementación

El setup + un test que atraviesa auth → validación → BD → contrato:

37.5.1. TypeScript

```ts
// setup: DATABASE_URL=postgres://localhost:5432/taskflow_test (env de test)
import request from "supertest";

describe("POST /tasks", () => {
  it("creates a task for the authenticated user", async () => {
    const { token, user } = await seedUser();           // fixture: user+token
    const res = await request(app)
      .post("/tasks")
      .set("Cookie", `token=${token}`)
      .send({ title: "pay rent" });
    expect(res.status).toBe(201);
    expect(res.body.title).toBe("pay rent");
    expect(res.body.public_id).toMatch(/^[0-9a-f-]{36}$/);
    expect(res.body.password_hash).toBeUndefined();     // the leak test
  });

  it("rejects invalid body with 422 contract", async () => {
    const res = await request(app).post("/tasks")
      .set("Cookie", `token=${token}`).send({ title: "" });
    expect(res.status).toBe(422);
    expect(res.body.error.code).toBe("VALIDATION");
  });
});
```

37.5.2. Python

```python
from fastapi.testclient import TestClient

@pytest.fixture
def client(db_tx):                      # db_tx: connection inside a transaction
    app.dependency_overrides[get_db] = lambda: db_tx
    yield TestClient(app)               # test runs; fixture rolls back after

def test_create_task(client, auth_headers):
    res = client.post("/tasks", json={"title": "pay rent"}, headers=auth_headers)
    assert res.status_code == 201
    assert res.json()["title"] == "pay rent"
    assert "password_hash" not in res.text            # the leak test
```

```python
def test_validation_error_contract(client, auth_headers):
    res = client.post("/tasks", json={"title": ""}, headers=auth_headers)
    assert res.status_code == 422
    assert res.json()["error"]["code"] == "VALIDATION"
```

37.5.3. Go

```go
func TestCreateTask(t *testing.T) {
    tx := beginTx(t)                    // test transaction
    defer tx.Rollback()                 // cleanup is a rollback - free isolation
    user, token := seedUser(t, tx)
    srv := httptest.NewServer(newRouter(testDeps{tx: tx}))
    defer srv.Close()

    req, _ := http.NewRequest("POST", srv.URL+"/tasks",
        strings.NewReader(`{"title":"pay rent"}`))
    req.Header.Set("Cookie", "token="+token)
    req.Header.Set("Content-Type", "application/json")
    res, _ := http.DefaultClient.Do(req)

    if res.StatusCode != 201 { t.Fatalf("got %d", res.StatusCode) }
    var body map[string]any
    json.NewDecoder(res.Body).Decode(&body)
    if body["password_hash"] != nil { t.Fatal("leaks internal field") } // the leak test
}
```

Prompt listo para tu AI

```
Monta tests de integración para mi API [Express/FastAPI/net-http]
contra Postgres. Requisitos: BD dedicada taskflow_test con las mismas
migraciones, aislamiento por transacción con rollback por test (sin
scripts de limpieza), ejercitar el router sin levantar socket
(supertest/TestClient/httptest), fixtures para crear usuario+token
válido y un segundo usuario, y tests de contrato: create→201 con
public_id, validación→422 {error:{code}}, sin auth→401, recurso
ajeno→404, viewer→403, y un test que verifique que password_hash NUNCA
aparece en ninguna respuesta.
```

37.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el patrón es uno — *BD de test + rollback por test + router en memoria*. Las herramientas cambian, la arquitectura del test no.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Detalle: las dos decisiones de infra

- **Rollback por test vs truncate por test**: el rollback es más rápido y no toca secuencias — pero solo funciona si tu código *acepta la conexión/transacción inyectada* (otra paga de la DI del cap. 16). Truncate entre tests es la alternativa simple cuando inyectar la tx es imposible.
- **Fixtures de identidad**: `seedUser()` crea usuario + sesión + token *en la misma tx del test* — el token es real (pasa por tu middleware de verdad), no un mock de auth. El segundo fixture (`otherUser`) es el que permite escribir el test más importante: `GET /tasks/{ajena} → 404`.

37.7. Errores comunes

Error	Por qué pasa	Fix
Tests contra la BD de dev	"Es la que está corriendo"	`taskflow_test` dedicada — los tests borran datos, es su trabajo
Limpieza manual por test	`DELETE FROM tasks` al final	Rollback de la tx — la BD queda sola
Mockear la auth	`req.user = fake`	Token real de fixture — si no, el middleware no se testea
Tests dependientes del orden	"El test 3 usa lo del test 2"	Cada test construye su mundo — orden aleatorio debe pasar
Testear solo el camino feliz	"201 funciona"	El contrato incluye los errores: 401/403/404/422 son API también

37.8. Buenas prácticas

- **El test de la fuga**: `"password_hash" not in res.text` en cada endpoint que toca usuarios — la regresión de seguridad más barata que existe (cap. 26).
- **Tests de permisos con dos fixtures**: `user` y `otherUser` — el 404-a-lo-ajeno y el 403-de-viewer son tests, no esperanzas.
- **El contrato se testea, no la implementación**: status + código de error + campos del body — el test no sabe qué query corriste.
- **CI corre la suite**: BD de test en el pipeline (un Postgres en Docker de un comando) — los tests que solo corren "en mi máquina" no existen para el equipo.

37.9. Ejercicio

1. ¿Por qué el rollback-por-test necesita que la app reciba la conexión inyectada? ¿Qué pasa si el servicio abre su propia conexión del pool?
2. Escribe (en pseudocódigo) el test "viewer no puede mutar" para `PATCH /projects/{id}` — ¿qué fixtures necesita?
3. ¿Por qué testear `res.body.error.code === "VALIDATION"` y no `res.body.error.message === "title is required"`?

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

37.10. Mini reto

Demostrar con un test que el doble-submit devuelve 409 tras éxito.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Soluciones

Ejercicio.

1. Si el servicio pide su propia conexión al pool, sus writes viven en *otra* transacción — que sí hace COMMIT y queda persistida: el rollback del test no la alcanza y la BD se ensucia. La DI de la conexión es el precio de entrada del patrón (cap. 16 lo paga).
2. Fixtures: `project` con `owner` (admin) y `viewer` (miembro con rol viewer) + tokens de ambos. El test: `PATCH` con token de viewer → `403` (miembro sin permiso — no 404: sí pertenece al proyecto); el mismo PATCH con token de owner → `200` (el contraste prueba que la ruta funciona y el rol es lo que cambió).
3. Porque `code` es el contrato estable y `message` es texto para humanos que cambiará (cap. 25): el test contra el mensaje se rompe cuando alguien mejora la redacción — falso negativo que enseña a ignorar los tests.

Mini reto. `POST /tasks` con el mismo payload dos veces (o con `Idempotency-Key`, cap. 22): primera → `201`; segunda → `409 {"error":{"code":"CONFLICT"}}` (o `200` con el mismo recurso si la diseñaste idempotente). El test documenta la decisión — cualquiera de las dos, pero *decidida y testeada*, no accidental.

37.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: sesión

Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada llamada.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

37.12. Lo que deberías saber hacer ahora

- ☐ Levantar una BD de test dedicada con tus migraciones
- ☐ Aislar tests con transacción+rollback (o truncate) sin limpieza manual
- ☐ Ejercitar el router en memoria (supertest/TestClient/httptest)
- ☐ Fabricar usuarios/tokens/roles en fixtures para tests de permisos
- ☐ Testear el contrato completo: éxito, validación, auth, ownership, fuga de campos

38 29. Necesitamos testear lo que depende de terceros

La API externa no puede recibir 500 llamadas en cada test run

> **En una frase**
>
> Tu servicio notifica a Slack — ¿cada `npm test` va a pegarle a Slack de verdad? No: se reemplaza la dependencia en *la frontera correcta* — tu adapter `notifyExternal` del cap. 19, no la librería HTTP. Mockear bien es responder: "¿esto verifica que **yo** llamé bien, o que **ellos** funcionan?".

38.1. El problema

El test del registro crea el usuario… y envía el email real, llama a Slack real, cobra en Stripe real. Tests lentos, que fallan cuando el tercero se cae, que gastan rate limit, y que envían correos de mentira a usuarios de verdad. Pero el extremo opuesto es igual de inútil: mockear tan adentro que el test solo verifica que el mock funciona.

> **Explicación de: mock / stub**
>
> Un **mock** es un doble de pruebas: finge ser el servicio externo (Stripe, SMTP) para que el test no cobre ni mande emails. Se mockea en la *frontera* (tu adaptador), nunca la BD — fingir la BD es mentir el test.

> **Explicación de: registry (Docker)**
>
> Un **registry** es el almacén de imágenes (Docker Hub, ECR, Artifact Registry): el CI empuja `nexus:v42`, el servidor la jala. Es el intermediario entre "se construyó" y "está corriendo".

38.2. Cómo lo resuelve un equipo

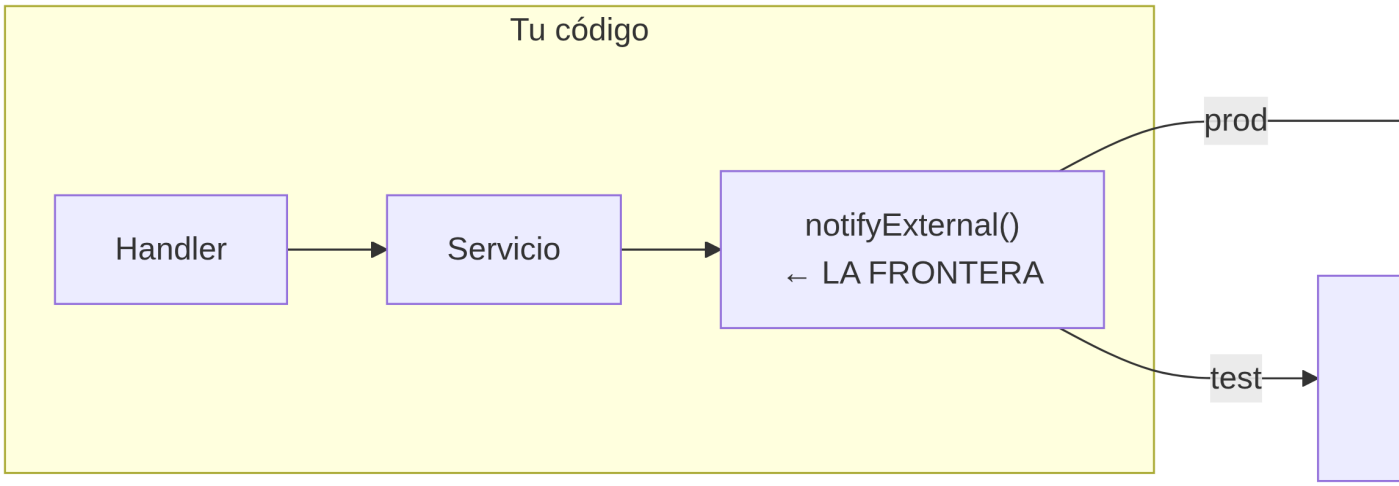
Un equipo serio mockea en **la frontera que él diseñó**: el adapter con nombre de negocio (`notifyExternal`, `chargeCard`) — no `fetch`, no `httpx`, no `net/http`. Razón: lo que quieres verificar es *tu* contrato ("¿llamaste con el texto correcto? ¿degradaste si falló?"), no la librería HTTP de alguien.

Y la regla que evita el teatro: **un mock que siempre responde bien solo prueba el día que todo salió bien** — el camino que importa es el del fallo.

38.3. Conceptos nuevos

- TS `vi.mock` · Py `monkeypatch/unittest.mock` · Go interfaces inyectadas
- U La frontera del mock: qué sustituir y qué no — Go fuerza DI por diseño; TS/Py la simulan
- U Cobertura: qué significa 40 %/80 % y por qué 100 % puede ser peor que 70 % bien elegido
- Ops Tests como requisito de merge: cobertura mínima en CI

38.4. La explicación visual



El mock vive donde *tu* diseño ya separó la integración (cap. 19) — por eso el adapter con nombre de negocio era importante: hoy es la costura donde cosen los tests.

38.5. Implementación

Sustituir el adapter en los tres mecanismos:

38.5.1. TypeScript

```
vi.mock("./notify", () => ({
  notifyExternal: vi.fn(),                    // the adapter, not fetch
}));

it("degrades when slack is down", async () => {
```

```
  vi.mocked(notifyExternal).mockRejectedValue(new ExternalServiceError("down"));
  const res = await request(app).post("/tasks").set("Cookie", token)
    .send({ title: "x" });
  expect(res.status).toBe(201);                  // external failure  your failure
  expect(notifyExternal).toHaveBeenCalledWith(expect.stringContaining("x"));
});
```

38.5.2. Python

```
def test_degrades_when_slack_down(client, monkeypatch):
    calls = []
    def fake_notify(text): calls.append(text); raise ExternalServiceError("down")
    monkeypatch.setattr("app.services.notify.notify_external", fake_notify)

    res = client.post("/tasks", json={"title": "x"}, headers=auth_headers)
    assert res.status_code == 201                # external failure  your failure
    assert calls and "x" in calls[0]
    # FastAPI alternative: app.dependency_overrides[notify_dep] = lambda: fake_notify
```

38.5.3. Go

```
// the service takes the dependency as an interface - the seam is in the signature
type Notifier interface{ Notify(ctx context.Context, text string) error }

type fakeNotifier struct{ err error; got []string }
func (f *fakeNotifier) Notify(_ context.Context, text string) error {
    f.got = append(f.got, text); return f.err
}

func TestDegradesWhenSlackDown(t *testing.T) {
    fn := &fakeNotifier{err: errors.New("down")}
    srv := httptest.NewServer(newRouter(testDeps{notifier: fn}))
    res := postTask(t, srv, token, `{"title":"x"}`)
    if res.StatusCode != 201 { t.Fatalf("external failure leaked: %d", res.StatusCode) }
    if len(fn.got) != 1 { t.Fatal("notify not called") }
}
```

Prompt listo para tu AI

```
Diseña los tests de mi integración con [Slack/email/pagos] en
[Express/FastAPI/net-http]. Requisitos: sustituir el adapter
notifyExternal (no fetch/httpx/http.Client) usando
[vi.mock/monkeypatch/dependency_overrides/interfaz inyectada]; cubrir
los tres caminos - éxito (llamó con el payload correcto), fallo
transitorio (reintentó y degradó sin tumbar mi respuesta), y fallo
permanente (error de dominio registrado, usuario afectado según lo
diseñado). Verificar también que NO se llama cuando la regla dice que
no (ej. no notificar si la tarea falló validación).
```

38.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el mock vive en la frontera de *tu* diseño — el adapter. Si tienes que mockear `fetch` o `httpx` para testear tu lógica, la señal real es que te falta el adapter (cap. 19), no que falta el mock.

Detalle: los tres mecanismos de sustitución

- **TS**: `vi.mock` reemplaza el *módulo* — funciona aunque el servicio haga `import` directo; potente pero mágico: el mock reemplaza por nombre de archivo.
- **Python**: `monkeypatch` cambia el atributo en runtime; FastAPI además tiene `dependency_overrides` — la DI del cap. 16 convertida en costura de tests oficial.
- **Go**: sin mocks de framework — la *interfaz* en la firma es la costura; inyectar `fakeNotifier` es solo pasar otra implementación. Más explícito: la testabilidad se ve en la firma del servicio.

Cobertura honesta: 40 % cubriendo servicios+auth vale más que 90 % de getters. 100 % suele comprar tests que no matan bugs — el número es un mapa de lo *no* cubierto, no una nota del examen.

38.7. Errores comunes

Error	Por qué pasa	Fix
Mockear la librería HTTP	"Intercepto fetch"	Mockea tu adapter — si no existe, eso es el hallazgo
Mock que siempre devuelve éxito	El camino feliz es el fácil	Test del fallo: 500, timeout, respuesta rara — ahí vive tu lógica
Assert sobre el mock, no el efecto	`expect(mock).toHaveBeenCalled()` y ya	Ambos: llamó bien **y** mi sistema respondió según lo diseñado

Error	Por qué pasa	Fix
Mockear tu propia capa de servicio	Para “testear el handler”	El handler con servicio mockeado no testea nada real — baja al adapter
100 % cobertura como meta	Métrica bonita	Cobertura en reglas y fronteras; getters no matan bugs

38.8. Buenas prácticas

- **Tres caminos por integración**: éxito, fallo transitorio, fallo permanente — la degradación elegante del cap. 19 se *demuestra* aquí, no se espera.
- **El fake también graba**: `got []string` — verificar *con qué* te llamaron importa tanto como que te llamaron.
- **Tests de no-llamada**: “si la validación falla, NO se notifica” — los efectos colaterales ausentes también son contrato.
- **Un test de humo real** (opcional, aparte): una llamada verdadera al sandbox del tercero en CI nocturno — el mock dice que llamaste bien; solo el humo dice que la API de ellos no cambió.

38.9. Ejercicio

1. ¿Por qué mockear `fetch`/`httpx` es la frontera *incorrecta* aunque técnicamente funcione?
2. Tu test mockea `notifyExternal` y verifica que se llamó. ¿Qué dos cosas importantes NO está verificando?
3. En Go no hay `vi.mock` — ¿por qué se dice que “la DI es el mock de Go” y qué implica para cómo escribes los servicios?

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

38.10. Mini reto

El test de “servicio externo caído → la API responde bien igual”.

Soluciones

Ejercicio.

1. Tres razones: (a) testeas la librería, no tu contrato — si el mock de `fetch` simula mal la API real, el test miente; (b) el test se acopla a *cómo* llamas (URL, headers) — refactorizar el cliente rompe tests que no deberían enterarse; (c) el adapter es *tu* frontera semántica:

`notifyExternal("task created")` es lo que tu negocio necesita, no `fetch(url, opts)`.

2. (a) **Con qué** se llamó — el payload correcto (un typo en el texto pasa el test); (b) **qué hizo tu sistema con el resultado** — el assert del mock no verifica tu 201, tu 502, ni tu degradación. El mock verifica la llamada; el test debe verificar la *respuesta*.
3. La interfaz en la firma *es* la costura: `NewService(n Notifier)` recibe la implementación — en prod el cliente real, en test el fake. Implica que los servicios declaran sus dependencias como interfaces pequeñas (1-3 métodos) en vez de esconder `http.Post` adentro — la testabilidad se *diseña* en la firma, no se parchea después.

Mini reto. El test de los ejemplos de arriba: fake que lanza/retorna error → POST /tasks → assert `201` (o el status diseñado) + assert de que el usuario recibe su tarea + assert de que el error quedó registrado. Es el test que *demuestra* la degradación elegante del cap. 19 — sin él, "degrada bien" es una esperanza.

38.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: método

Un **método** es una función que vive dentro de un objeto/clase: `task.save()` — `save` es un método de `task`. La diferencia con una función suelta: el método conoce al objeto que lo contiene (`this`/ `self`).

Explicación de: cobertura (coverage)

La **cobertura** es el % de tu código que los tests ejecutan. Útil como detector de "esto nadie lo prueba"; inútil como meta — 90 % cubierto no significa que los tests afirmen algo correcto.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

Explicación de: timeout

Un **timeout** es la fecha límite de una espera: "si la BD no responde en 5s, falla". Sin timeout, un servicio caído convierte tu espera en infinita — tu app muere de pie esperando.

38.12. Lo que deberías saber hacer ahora

- ☐ Mockear en la frontera del adapter (vi.mock/monkeypatch/interfaz)
- ☐ Cubrir éxito, fallo transitorio y fallo permanente de una integración
- ☐ Verificar con qué te llamaron *y* cómo respondió tu sistema
- ☐ Diseñar servicios con dependencias inyectables para que sean testeables
- ☐ Leer la cobertura como mapa de huecos, no como nota

Parte IX

Sección III · Nube — Fundamentos

39 N1. Qué es un servidor de verdad

On-premise, datacenters, VPS y bare metal — la máquina siempre existe

> **En una frase**
>
> "La nube" no existe: son **computadoras de otros en datacenters**. Este capítulo mapea el espectro completo — tu laptop, un servidor en tu closet (on-premise), un VPS alquilado, bare metal — y qué responsabilidad carga cada uno. Al final vas a conectarte por SSH a una máquina remota y dejar tu app corriendo ahí, como se hizo durante 20 años.

39.1. El problema

"Sube la app a la nube" — ¿a dónde exactamente? Tu API corre en tu laptop porque tu laptop es una computadora encendida con un proceso escuchando en un puerto. Para que otros la usen necesitas **otra computadora, encendida 24/7, con una IP pública**. Toda la industria del hosting se reduce a: ¿de quién es esa máquina, dónde está, y quién la mantiene?

> **Explicación de: dirección IP**
>
> Una **IP** es la dirección numérica de una máquina en la red: `203.0.113.10`. Los servidores tienen IP pública; dentro de una VPC tienen IPs privadas que internet no ve.

> **Explicación de: puerto**
>
> Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

> **Explicación de: proceso**
>
> Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

39.2. Cómo lo resuelve un equipo

Nadie empieza comprando un rack. El camino típico de un producto:

1. **Desarrollo:** tu laptop. IP `localhost`, apagas cuando quieres.
2. **Primer deploy:** un VPS de $5/mes (DigitalOcean, Hetzner). Una porción de máquina ajena con IP pública.
3. **Crecimiento:** cloud (AWS/Azure/GCP) — misma idea, más servicios alrededor.
4. **Escala o regulación:** datacenter propio o *bare metal* (máquina física completa alquilada). Raro en productos pequeños.

> Explicación de: la nube / cloud
>
> **La nube** son computadoras de otro que rentas por hora: AWS, Azure, GCP te prestan máquinas, redes y servicios administrados. Pagas por no comprar hardware — y por no administrarlo tú.

> Explicación de: localhost / 127.0.0.1
>
> **localhost** significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

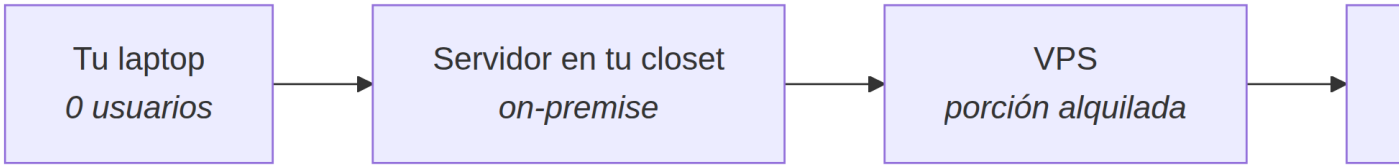
> Explicación de: deploy / despliegue
>
> Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

La lección no es "usa X" sino **entender qué responsabilidad heredas en cada opción**: hardware, red, electricidad, parches del SO, backups.

39.3. Conceptos nuevos

- U **On-premise:** tu propio hardware en tu edificio — control total, costo total
- Ops **Colocation:** tu máquina vive en un datacenter ajeno (ellos: luz, red, seguridad física)
- Ops **VPS (Virtual Private Server):** una porción virtual de máquina física — DigitalOcean, Hetzner, EC2 básico
- Ops **Bare metal:** una máquina física completa solo para ti, alquilada
- U La máquina siempre existe: "serverless" también corre en una — solo que no es tu problema

39.4. La explicación visual



Modelo	Máquina de…	Datacenter de…	Tú administras	Costo típico
On-premise	tuya	tuyo (tu closet)	TODO	Hardware + luz + tu tiempo
Colocation	tuya	ajeno	SO en adelante	$100-500/mes por rack
VPS	ajena (compartida)	ajeno	SO en adelante	$5-50/mes
Cloud VM	ajena	ajeno	SO en adelante	Por hora/segundo
Bare metal	ajena (exclusiva)	ajeno	SO en adelante	$100+/mes

La regla de oro: **a más control, más responsabilidad.** No hay opción "mejor" — hay opción que *te cuesta lo que puedes pagar*.

39.5. Implementación

El camino clásico que todo backend debería hacer **una vez**: alquilar un VPS, entrar por SSH y dejar la app corriendo. Aunque luego uses PaaS o serverless, entenderás qué abstraen.

Explicación de: IaaS / PaaS / SaaS

IaaS: rentas la máquina, administras todo lo de adentro. **PaaS**: rentas la plataforma, solo traes tu código. **SaaS**: usas el software hecho (Gmail, RDS administrado). Menos control, menos trabajo.

Explicación de: serverless / función

Serverless = no piensas en servidores: subes una *función* y el proveedor la ejecuta por evento, escalando de 0 a 1000 sola. Pagas por ejecución. Lambda (AWS), Cloud Functions (GCP), Azure Functions.

Paso 1 — provisionar. En DigitalOcean/Hetzner creas un "Droplet"/VPS: Ubuntu, el plan más barato, y subes tu **llave SSH pública** (la alternativa — password — es la primera mala práctica que evitaremos).

Paso 2 — entrar. SSH es "la terminal de otra máquina":

```
# From your terminal - run the remote machine's shell
ssh ubuntu@203.0.113.10
# Now every command runs on THAT machine, not yours
```

Paso 3 — subir tu app (con `git clone` o `scp`/`rsync`):

```
# On the VPS: clone and install
git clone https://github.com/you/taskflow.git
cd taskflow && npm ci
```

Paso 4 — correrla... y que sobreviva. Si la arrancas con `node dist/index.js` y cierras la terminal, el proceso muere con tu sesión. Necesitas un **gestor de procesos** — algo que la reinicie si crashea y la arranque si el servidor reinicia:

> Explicación de: terminal / consola
>
> La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

```
# systemd - the OS's own process manager (recommended)
sudo systemctl enable taskflow    # start on boot
sudo systemctl start taskflow
sudo systemctl status taskflow    # is it alive?
journalctl -u taskflow -f         # its logs

# PM2 - same idea, Node-specific and friendlier
pm2 start dist/index.js --name taskflow
pm2 save && pm2 startup           # survive reboots
```

Paso 5 — abrir el puerto justo. Tu app escucha en 8000, pero el mundo espera HTTP en 80. Además, exponer el puerto de la app directo es mala idea — por eso va nginx delante (reverse proxy, cap. 33). El firewall mínimo:

> Explicación de: reverse proxy
>
> Un **reverse proxy** (nginx, ALB, Cloudflare) es el portero: recibe todo el tráfico en el 443, termina TLS y reparte a tu app interna. La app nunca mira a internet directamente — el proxy la protege.

```
sudo ufw allow OpenSSH   # port 22 - NEVER lock yourself out
sudo ufw allow 'Nginx Full'
sudo ufw enable
```

Resultado: `http://203.0.113.10` responde tu API, corriendo en una máquina que no ves, en un datacenter que no conoces. **Eso es "la nube"** — con tu responsabilidad encima.

Prompt listo para tu AI

```
Guíame para desplegar mi API [Express/FastAPI/Go] en un VPS Ubuntu por
primera vez. Dame los pasos en orden: crear la llave SSH si no tengo,
provisionar el VPS, conectar por SSH, subir el código, instalar
dependencias, y dejar el proceso vivo con systemd (con el archivo .service
completo). Requisitos: NO uses password para SSH (solo llaves), crea un
usuario no-root para la app, configura ufw abriendo solo lo necesario, y
explícame por qué cada capa de seguridad existe.
```

39.6. ¿Por qué cada modelo existe?

Lo único que necesitas llevarte: el espectro mueve una sola variable — *quién sufre cuando algo físico se rompe*. On-premise: tú, a las 3am, manejando al datacenter. VPS: ellos, tú solo el SO. Todo lo demás (precio, control, compliance) se deriva de esa única pregunta.

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Detalle: cuándo cada cimentación

- **On-premise** sigue existiendo por tres razones reales: regulación (bancos, salud — el dato no puede salir), costo a escala masiva (Dropbox *salió* de AWS ahorrando millones), y latencia (fábricas, hospitales sin internet confiable).
- **VPS** es el sweet spot de proyectos pequeños: $5/mes, IP propia, aprendes Linux real.
- **Cloud** paga cuando necesitas los *servicios alrededor* de la máquina (redes, balanceadores, servicios administrados — caps. siguientes).
- **Bare metal** es para cargas que no toleran la virtualización o la vecindad de otros tenants.

39.7. Errores comunes

Error	Por qué pasa	Fix
SSH con password y root abierto	"Es solo un VPS de prueba" — los bots escanean port 22 en minutos	Llaves SSH, `PermitRootLogin no`, usuario no-root
El proceso muere al cerrar la terminal	Lo corriste con `node app.js` en tu sesión SSH	systemd o PM2
Todo el tráfico directo al puerto 8000	"Funciona, no lo toco"	ufw + nginx delante; el puerto de la app queda interno

Error	Por qué pasa	Fix
No saber *dónde* está la máquina	La región importa: latencia y leyes del dato	Elige región cerca de tus usuarios

39.8. Buenas prácticas

- **Llaves SSH siempre; password nunca.** La llave pública va al VPS al crearlo; la privada no sale de tu máquina.
- **Un usuario por servicio**, no root. Si tu app se compromete, el atacante hereda los permisos de *ese* usuario — que sean pocos.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

- **Firewall que niega por defecto** y abre solo 22/80/443.
- **Documenta la máquina:** IP, región, qué corre ahí. Un servidor olvidado es una factura olvidada y un agujero de seguridad.

EXTRA · Fundamentos de internet (el checklist del roadmap)

EXTRA Antes de servidores, los roadmaps piden entender internet misma — cada ítem mapea a un capítulo:

- ¿Cómo funciona internet? — paquetes que viajan entre IPs
- ¿Qué es HTTP y HTTPS? — *cap-01, cap-33*
- ¿Qué es una dirección IP? — *este capítulo*
- ¿Qué es un nombre de dominio? ¿DNS? — *cap-33, nu-04*
- ¿Qué es hosting? — *nu-02*
- ¿Qué es SMTP? — el protocolo de correo; por eso mandar email es un job externo (*cap-21*)

39.9. Ejercicio

1. Ordena estas opciones de menor a mayor responsabilidad tuya: bare metal, PaaS, VPS, on-premise, serverless.
2. Tu app corre con `npm start` dentro de tu sesión SSH. Nombra **dos** formas en que esto falla sin que toques nada.
3. ¿Por qué el firewall se configura ANTES de exponer la app?

Soluciones

1. Serverless < PaaS < VPS < bare metal < on-premise. (En serverless ni siquiera ves el SO; en on-premise hasta la electricidad es tuya.)
2. (a) Tu sesión SSH se cae o la cierras → el proceso muere con ella.
   (b) El servidor reinicia por un parche de AWS/DigitalOcean → nada vuelve a arrancar tu app. Ambas se arreglan con un process manager (systemd/PM2).
3. Porque el orden es defensivo: primero cierras puertas, luego abres la que necesitas. Si expones la app y *después* configuras el firewall, quedó una ventana donde cualquiera entra — y los bots no esperan.

39.10. Mini reto

Este script de deploy circula por internet. Encuentra las **tres** decisiones peligrosas:

```
ssh root@203.0.113.10
git clone https://github.com/you/taskflow.git && cd taskflow
npm install
export DB_PASSWORD="super-secret-prod-pw"
node dist/index.js &
```

Soluciones

1. **`root` por SSH** — el usuario con todos los permisos expuesto a internet (debería estar `PermitRootLogin no` + usuario normal).
2. **Secreto en la línea de comandos** — `export DB_PASSWORD=...` queda en el historial del shell y visible en `ps` mientras corre (cap. 24 y N4 cubren dónde van los secretos de verdad).
3. **`node ... &` y a rezar** — el proceso queda huérfano: muere con la sesión SSH, nadie lo reinicia si crashea, nada lo arranca al reiniciar el servidor. Falta systemd/PM2.

39.11. Vocabulario técnico del capítulo

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: latencia / throughput

Latencia = cuánto tarda UNA operación (ms). **Throughput** = cuántas haces por segundo. Se pelean: bajar latencia no siempre sube throughput. La latencia la siente el usuario; el throughput lo paga tu factura.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

39.12. Lo que deberías saber hacer ahora

- ☐ Explicar el espectro on-premise → VPS → cloud → bare metal
- ☐ Decir qué es un datacenter y quién lo opera en cada modelo
- ☐ Conectar por SSH a una máquina remota con llaves
- ☐ Dejar un proceso vivo con systemd o PM2
- ☐ Nombrar las 3 capas mínimas de seguridad de un VPS expuesto

40 N2. IaaS, PaaS y SaaS: la frontera de responsabilidad

Quién administra qué — la pregunta que responde todo modelo de nube

> En una frase
>
> La única pregunta que importa en la nube: **¿hasta dónde llega tu responsabilidad?** IaaS te da la máquina y administras todo desde el SO; PaaS te da la plataforma y solo subes tu código; SaaS solo la usas. Mismo problema, distinta línea donde terminas tú y empieza el proveedor.

40.1. El problema

En N1 viste el VPS: tú instalas Ubuntu, Node, nginx, systemd, firewall. Funciona, pero ¿qué pasa si **no quieres** administrar todo eso? Cada modelo de nube es una respuesta distinta a *"¿qué parte de la pila quiero olvidar?"*. No son opciones buenas/malas — son fronteras distintas entre tu trabajo y el del proveedor.

> Explicación de: la nube / cloud
>
> **La nube** son computadoras de otro que rentas por hora: AWS, Azure, GCP te prestan máquinas, redes y servicios administrados. Pagas por no comprar hardware — y por no administrarlo tú.

> Explicación de: firewall / security group
>
> Un **firewall** (security group en AWS) es la lista de puertos que aceptan tráfico: "443 abierto a todos, 5432 solo desde la app". Cada puerto abierto es una puerta a vigilar — abre lo mínimo.

40.2. Cómo lo resuelve un equipo

La analogía clásica es la pizza:

Modelo	Análogo	Tú traes	Ellos traen
On-premise	Cocinar en casa	Todo: horno, ingredientes, mesa	La receta quizá
IaaS	Cocinar en cocina rentada	Ingredientes y cocinar	Local, horno, luz
PaaS	Pizza congelada en casa ajena	Elegirla y calentarla	Cocina, horno, pizza hecha
FaaS	Delivery	Pedirla por teléfono	Todo lo demás
SaaS	Restaurante	Sentarte a comer	Absolutamente todo

Traducido a la pila técnica (hardware → virtualización → SO → runtime → app → datos):

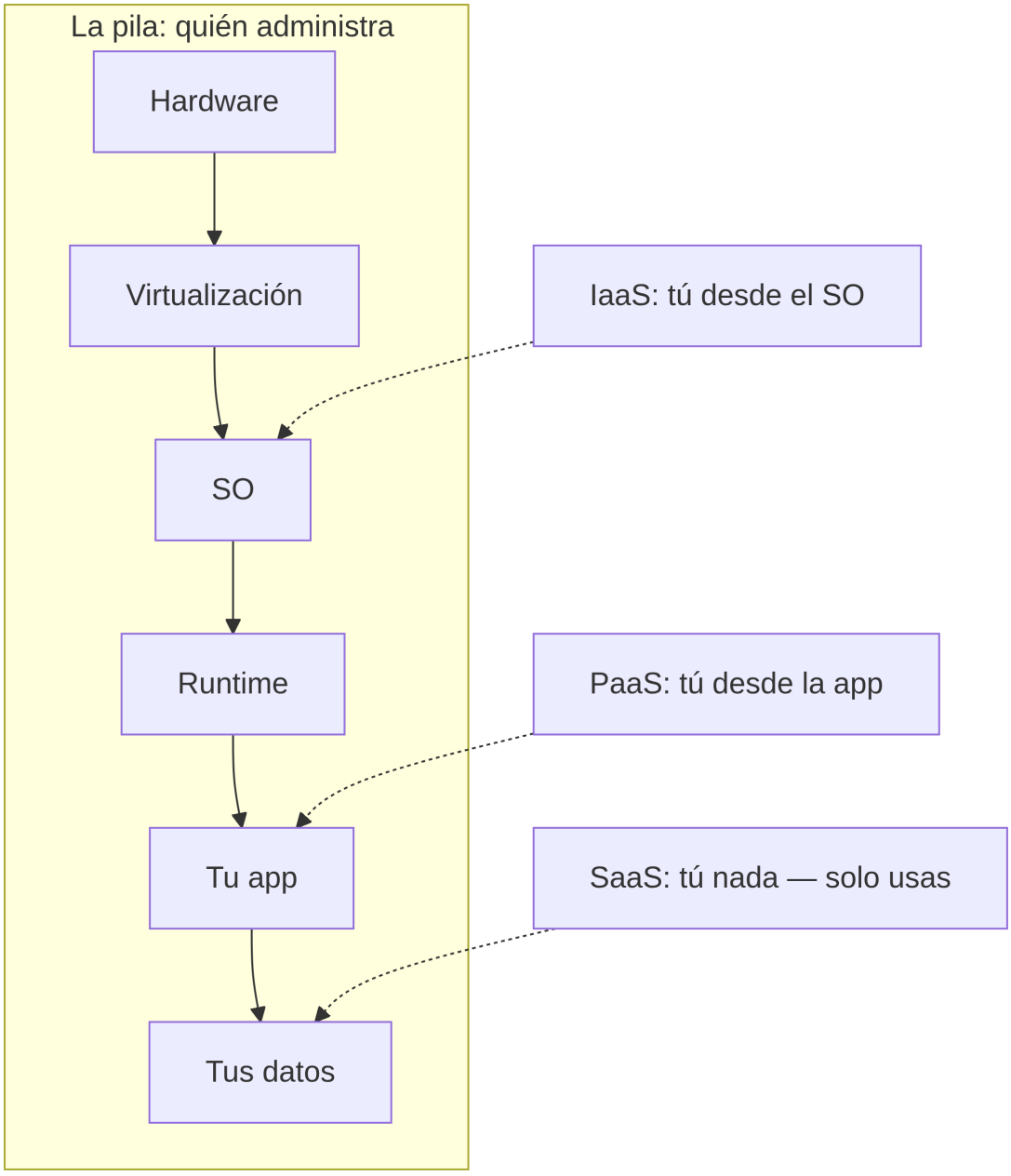
> Explicación de: runtime
>
> El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

- **IaaS** (EC2, Azure VM, Droplet): ellos hardware + virtualización. Tú desde el SO: parches, runtime, nginx, app, datos. *"Te alquilo la cocina vacía."*
- **PaaS** (Heroku, Render, Railway, App Engine): ellos hasta el runtime. Tú `git push` con tu código. *"Cocina equipada: trae tu receta."*
- **FaaS/serverless** (Lambda, Functions): ellos hasta el handler. Tú una función. *"Delivery por evento."*
- **SaaS** (Gmail, Notion, Stripe): ellos todo. Tú eres el usuario.

40.3. Conceptos nuevos

- U **IaaS:** EC2, VMs — tú desde el SO en adelante
- U **PaaS:** Heroku, Render, App Engine, Elastic Beanstalk — solo tu app
- U **SaaS:** Gmail, Notion — solo lo usas
- Ops **CaaS** (Containers as a Service: ECS, Cloud Run) y **FaaS** (Lambda) — puntos intermedios: el primero sube un contenedor, el segundo sube una función

40.4. La explicación visual



Capa	On-prem	IaaS	CaaS	PaaS	FaaS	SaaS
App	tú	tú	tú	tú	tú (función)	ellos
Runtime	tú	tú	tú	ellos	ellos	ellos
SO	tú	tú	ellos	ellos	ellos	ellos

Capa	On-prem	IaaS	CaaS	PaaS	FaaS	SaaS
Virtualización	tú	ellos	ellos	ellos	ellos	ellos
Hardware	tú	ellos	ellos	ellos	ellos	ellos

La lectura: cada paso a la derecha te quita trabajo operativo y te cuesta control + dinero + dependencia del proveedor (*lock-in*).

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

40.5. Implementación

Misma app — Taskflow — en los dos extremos que ya conoces:

40.5.1. IaaS (lo que hiciste en N1)

```
# On your VPS - YOU own everything from the OS up
sudo apt update && sudo apt install -y nodejs npm nginx
git clone https://github.com/you/taskflow.git && cd taskflow
npm ci && npm run build
sudo systemctl enable taskflow && sudo systemctl start taskflow
# + firewall, + TLS certs, + log rotation, + OS patches... all yours
```

40.5.2. PaaS (Render/Railway/Heroku)

```
# On your laptop - you own only the code
git push render main
# They: build, runtime, process manager, TLS, restarts, OS patches
# You see: "Deploy live at https://taskflow.onrender.com"
```

En IaaS ejecutaste ~10 comandos operativos; en PaaS, **uno**. El PaaS interiorizó todo lo del capítulo N1 (systemd, nginx, firewall, parches) detrás de `git push`. *Eso* es lo que pagas.

Explicación de: IaaS / PaaS / SaaS

IaaS: rentas la máquina, administras todo lo de adentro. **PaaS**: rentas la plataforma, solo traes tu código. **SaaS**: usas el software hecho (Gmail, RDS administrado). Menos control, menos trabajo.

Prompt listo para tu AI

```
Quiero desplegar mi API de tareas [Express/FastAPI/Go] en un PaaS
[Render / Railway / Heroku / Fly.io]. Dame: el archivo de configuración
necesario (Procfile, render.yaml o equivalente), qué cambios necesita mi
código (puerto desde variable de entorno, build command), las variables
de entorno que debo configurar en el dashboard, y cómo hacer deploy.
Después explícame exactamente qué partes del servidor de N1 está
administrando el PaaS por mí.
```

40.6. ¿Por qué existen tantos modelos?

Lo único que necesitas llevarte: elige la frontera según *lo que tu equipo puede administrar bien*, no según lo que "debería" usarse. Un equipo de 2 sin sysadmin no compra IaaS — compra PaaS y se dedica a su producto. Un equipo con 50 ingenieros y requisitos raros sí justifica IaaS. El modelo correcto es el que minimiza *tu* costo total, no el que minimiza la factura del proveedor.

Otras cimentaciones: la línea se mueve

Alternativa	Qué es	Qué te cuesta	Cuándo elegirla
PaaS	Subes código	Control limitado, precio/unidad más alto, lock-in moderado	Equipos pequeños, MVPs, la opción por defecto sana
CaaS	Subes contenedor	Debes saber Docker (cap. 30-32)	Cuando PaaS se queda corto pero no quieres VMs
IaaS	Subes SO completo	Todo el trabajo de N1 + parches + seguridad	Control fino, compliance, escala con equipo ops
FaaS	Subes función	Cold starts, límites de tiempo, lock-in alto	Eventos, webhooks, tráfico irregular (cap. N5)
SaaS+backend	Auth0/Supabase/Firebase hacen tu backend	Dependencia fuerte, pero velocidad extrema	MVPs donde el backend no es tu diferencial

40.7. Errores comunes

Error	Por qué pasa	Fix
IaaS “para aprender cloud” en el primer deploy	Se confunde “nube” con “VM”	Empieza en PaaS; baja a IaaS cuando tengas razón concreta
“PaaS = no hay ops”	Siguen existiendo: config, secretos, costos, límites	PaaS quita *sysadmin*, no *responsabilidad*
Elegir por factura y no por tiempo	El VPS de $5 te cuesta 10h/mes de ops	Costo real = factura + horas de equipo
Miedo al lock-in prematuro	“¿Y si cambio de proveedor?” en un MVP	El lock-in se paga; la velocidad perdida no se recupera

40.8. Buenas prácticas

- **La app no debe saber en qué modelo corre:** puerto desde env var, configuración por entorno, estado fuera del proceso — así puedes cambiar de frontera después.

> Explicación de: entorno (environment)
>
> Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

> Explicación de: proceso
>
> Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que “se caiga el proceso” = el programa murió.

- **Documenta qué frontera elegiste y por qué** — el próximo deploy parte de esa decisión, no de cero.
- **PaaS no es “para novatos”:** la mayoría de productos reales vive en PaaS durante años. Administrar VMs no es un logro, es un costo.

40.9. Ejercicio

1. Clasifica: Gmail, EC2, Heroku, un Droplet, Notion, Lambda, Cloud Run.
2. Tu startup de 3 personas lanza su API. ¿IaaS o PaaS? Justifica con la frontera, no con el precio.
3. ¿Qué capa comparten IaaS y PaaS como responsabilidad del proveedor?

> Explicación de: serverless / función
>
> **Serverless** = no piensas en servidores: subes una *función* y el proveedor la ejecuta por evento, escalando de 0 a 1000 sola. Pagas por ejecución. Lambda (AWS), Cloud Functions (GCP), Azure Functions.

Soluciones

1. SaaS: Gmail, Notion · IaaS: EC2, Droplet · PaaS: Heroku · FaaS: Lambda · CaaS: Cloud Run (punto medio: subes un contenedor, ellos el SO y runtime).
2. PaaS. Con 3 personas, cada hora de sysadmin es una hora que no va al producto. La frontera correcta es "nosotros código, ellos plataforma" hasta que exista una *razón concreta* (requisito de red, costo a escala, compliance) que la mueva.
3. Ninguna más allá del hardware — espera, trampa: IaaS delega hardware+virtualización; PaaS delega además SO+runtime. La capa compartida como *del proveedor* en ambos es hardware+virtualización.

40.10. Mini reto

Un equipo dice: *"Pasamos de Heroku a EC2 para ahorrar $200/mes"*. Tres meses después tienen un sysadmin parcial. Haz la cuenta honesta.

Soluciones

La factura bajó $200, pero el costo subió: si el sysadmin cuesta aunque sea 5h/mes a $50/h = $250 + el tiempo de los devs en parches/incidentes + el riesgo de errores operativos. El ahorro era ilusorio — compararon factura contra factura e ignoraron las horas. Regla: migra de frontera cuando el *costo total* (factura + tiempo + riesgo) mejora, no cuando solo una línea baja.

40.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: HTTPS / TLS / certificado

HTTPS = HTTP viajando cifrado por **TLS**: nadie en el camino puede leer ni alterar el tráfico. El **certificado** es la credencial que prueba que el servidor es quien dice — lo emite una autoridad y hay que renovarlo.

Explicación de: Docker

Docker empaqueta tu app con todo lo que necesita (runtime, librerías, config) en una **imagen** — el mismo paquete corre idéntico en tu laptop y en producción. Es la respuesta a "en mi máquina sí funcionaba".

Explicación de: webhook

Un **webhook** es una llamada HTTP invertida: en vez de tú preguntarle a Stripe "¿llegó el pago?", Stripe te llama a tu endpoint cuando pasa. Se verifica la firma (es público) y se responde 200 rápido.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

40.12. Lo que deberías saber hacer ahora

- ☐ Dibujar la frontera de responsabilidad de IaaS/PaaS/SaaS
- ☐ Ubicar un servicio concreto en el modelo correcto (incl. CaaS/FaaS)
- ☐ Calcular el costo real de un modelo: factura + horas + riesgo
- ☐ Decidir la frontera correcta para un equipo dado

Parte X

Sección III · Docker

41 30. Necesitamos que cualquier máquina ejecute esto igual

«En mi máquina funciona» deja de ser una excusa

Figura 41.1: Parte 7 — Docker

> **i** En una frase
>
> El nuevo dev tardó dos días en arrancar el proyecto: versión de Node distinta, Postgres mal configurado, una variable que nadie documentó. **Docker empaqueta la app completa — runtime, dependencias, código — en una imagen que corre igual en cualquier máquina.** La receta es el `Dockerfile`; la imagen es la clase; el contenedor es la instancia.

41.1. El problema

"En mi máquina funciona" significa exactamente eso: funciona *en tu máquina* — con tu Node 20, tu Postgres 16, tu OS. La laptop del compañero tiene Node 18; el servidor tiene Ubuntu sin Node instalado; CI tiene otra cosa. Cada entorno es una lotería de versiones. La pregunta que Docker responde: **¿cómo hago que "mi máquina" viaje con el código?**

Explicación de: Docker

Docker empaqueta tu app con todo lo que necesita (runtime, librerías, config) en una **imagen** — el mismo paquete corre idéntico en tu laptop y en producción. Es la respuesta a "en mi máquina sí funcionaba".

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

41.2. Cómo lo resuelve un equipo

Un equipo serio empaqueta *el entorno como código*: un `Dockerfile` es la receta declarativa — "parte de Node 20, instala dependencias, copia el código, corre `node dist/index.js`" — que cualquier máquina con Docker ejecuta idéntico. El onboarding pasa de dos días de wiki a `docker compose up`. Y el mismo artefacto que el dev corre es el que prod corre — la paridad dev/prod del cap. 24 llevada al extremo.

Explicación de: Docker Compose

Docker Compose describe un sistema de varios contenedores en un YAML: la app + Postgres + Redis, sus redes y volúmenes. `docker compose up` levanta el entorno completo de desarrollo con un comando.

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

41.3. Conceptos nuevos

- Ops Qué es un contenedor (sin la teoría de namespaces): máquina mínima reproducible
- Ops Imagen vs contenedor vs Dockerfile: la analogía clase/instancia
- Ops Dockerfile por lenguaje: `tsc`→runtime vs venv vs **binario estático de ~15MB**

- Ops `.dockerignore`: qué nunca entra en una imagen

41.4. La explicación visual

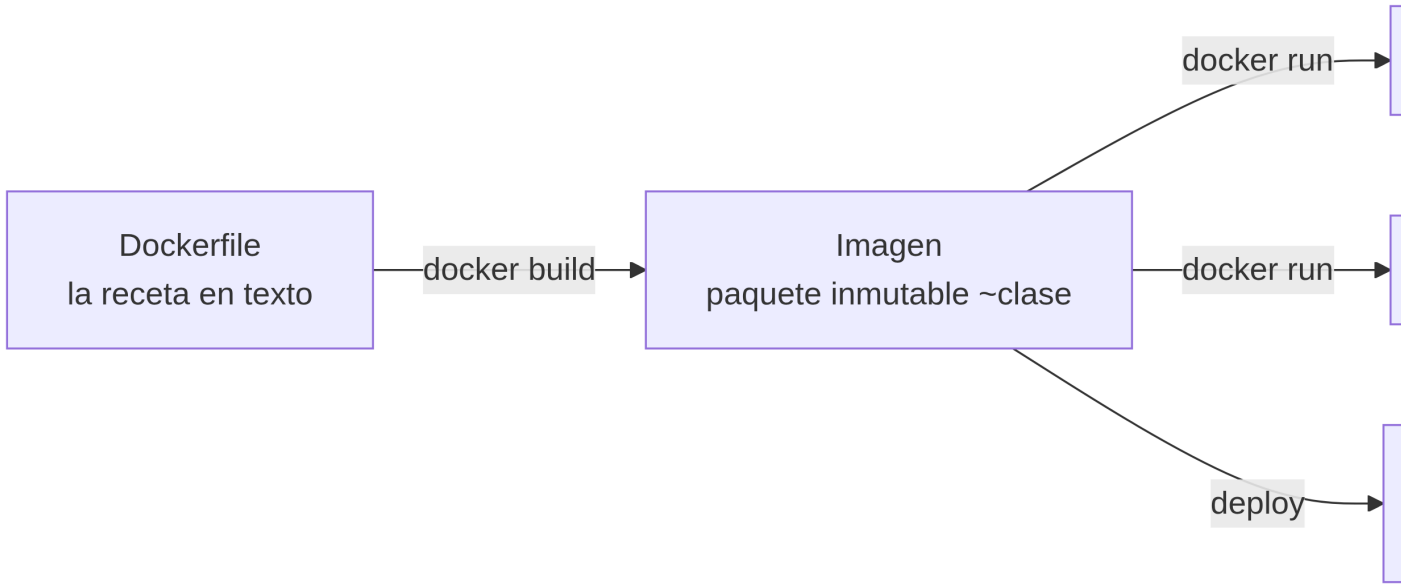


Imagen vs contenedor = clase vs instancia: la imagen es el paquete inmutable (runtime + deps + código); cada `docker run` crea un contenedor vivo de esa imagen. Cambiar código = construir *nueva imagen* — los contenedores son descartables por diseño.

Explicación de: contenedor

Un **contenedor** es un proceso con mochila propia: corre aislado con su filesystem y sus librerías, pero comparte el kernel de la máquina — por eso arranca en milisegundos donde una VM tarda minutos.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

41.5. Implementación

El primer Dockerfile por stack — funcional (la versión de producción viene en el cap. 32):

41.5.1. TypeScript

```
FROM node:20-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci                          # lockfile exact, not "npm install"
COPY . .
RUN npm run build                   # tsc → dist/
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

```
# .dockerignore - what never enters the image
node_modules
dist
.env
.git
```

41.5.2. Python

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
# .dockerignore
.venv
__pycache__
.env
.git
```

41.5.3. Go

```
FROM golang:1.22 AS build
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
```

```
COPY . .
RUN CGO_ENABLED=0 go build -o server ./cmd/server

FROM gcr.io/distroless/static        # the runtime is just the binary
COPY --from=build /app/server /server
EXPOSE 8080
CMD ["/server"]                      # ~15MB total - static binary, no OS needed
```

Correrlo:

```
docker build -t taskflow-api .       # recipe → image
docker run -p 3000:3000 --env-file .env taskflow-api
```

Prompt listo para tu AI

```
Escribe el Dockerfile de mi API [Express+TS/FastAPI/Go net-http] para
desarrollo. Requisitos: imagen base slim oficial, instalar dependencias
por lockfile (npm ci / requirements.txt / go mod download) en capa
separada para cachear, compilar si aplica (tsc/go build), .dockerignore
con node_modules|venv|.env|.git, EXPOSE del puerto correcto, y CMD que
arranque el servidor en 0.0.0.0. Explícame cada instrucción y cómo
construir y correr la imagen.
```

41.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el Dockerfile es siempre la misma receta — *base* → *deps* → *código* → *run*. Lo que cambia es cuánto necesita el runtime: TS/Python llevan su intérprete dentro de la imagen; Go compila un binario que no necesita nada más — por eso su imagen final cabe en ~15MB.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Detalle: capas, caché y tamaños

- **Capas y caché**: cada instrucción es una capa cacheada. Por eso `COPY package*.json` + `RUN npm ci` van *antes* de `COPY . .` — las deps solo se reinstalan cuando el lockfile cambia, no en cada build.
- **Tamaños honestos**: `node:20-slim` ~200MB + node_modules; Python similar + venv; Go `distroless` ~15MB — el binario estático *es* todo el runtime. La diferencia no es estética: menos imagen = menos superficie de ataque y deploys más rápidos.
- `CGO_ENABLED=0`: fuerza el binario 100 % estático — sin esa flag, Go enlaza libc y el `distroless` no la tiene.

41.7. Errores comunes

Error	Por qué pasa	Fix
`COPY . .` antes de instalar deps	El orden "natural" de escribir	Deps primero (cachea), código después (cambia seguido)
`.env` dentro de la imagen	`COPY . .` lo incluye	`.dockerignore` — la imagen que lleva secretos es un incidente
`node_modules` copiado al build	Sin `.dockerignore`	Excluirlo — la imagen instala las suyas para *su* OS
`npm install` en vez de `npm ci`	Es el comando que conoces	`ci` = lockfile exacto, reproducible; `install` puede resolver distinto
Correr en dev lo que es de dev	`--reload`, debugger, sourcemaps	La imagen de prod es otro Dockerfile (cap. 32)

41.8. Buenas prácticas

- **Bases oficiales y slim**: `node:20-slim`, `python:3.12-slim` — el `-slim` quita lo que no necesitas; Alpine existe pero sus libc distintas muerden a veces.
- **`.dockerignore` desde el día uno**: `.env`, `.git`, `node_modules`, `dist` — lo que no entra no puede filtrarse ni pesar.
- **Un proceso por contenedor**: el contenedor *es* tu app — si muere, muere el contenedor y el orquestador lo nota.

Explicación de: Kubernetes (k8s)

Kubernetes es el orquestador de contenedores: le dices "quiero 3 réplicas de esta imagen" y él las distribuye, reinicia las que mueren y las expone. Poderoso y complejo — se usa cuando las máquinas ya son manada.

- **Tags con sentido**: `taskflow-api:1.4.2` / `:commit-sha` — `latest` es el tag que "nadie sabe qué versión es".

EXTRA · Containerización / virtualización

EXTRA El roadmap externo lista — **Docker** (el estándar de contenedores, este capítulo), **Kubernetes** (orquestador de manadas de contenedores — nu-05 lo ubica), **rkt** (histórico, ya descontinuado — buen recordatorio de que las listas envejecen y los conceptos quedan).

41.9. Ejercicio

1. ¿Por qué `COPY package*.json` + `RUN npm ci` van *antes* de `COPY . .`? ¿Qué se rompe si inviertes el orden?

2. Lista tres cosas que tu `.dockerignore` debe excluir y el daño concreto de cada una si entra.
3. La imagen de Go pesa ~15MB y la de Node ~250MB+. ¿Qué hay *dentro* de cada una que explica la diferencia?

41.10. Mini reto

Reducir el tamaño de la imagen a la mitad explicando cada decisión (spoiler: Go gana).

Soluciones

Ejercicio.

1. La caché de capas: deps cambian poco (lockfile estable) → su capa se reutiliza en cada build. Código cambia siempre → su capa va al final. Invertir el orden: cada cambio de código invalida también la capa de deps → `npm ci` completo en cada build (minutos en vez de segundos).
2. `.env` (secretos *dentro* de la imagen — cualquiera con la imagen los tiene), `node_modules/.venv` (deps del OS equivocado + peso — la imagen instala las suyas), `.git` (historial completo = secretos históricos + peso).
3. Node/Python llevan el *intérprete completo* + stdlib + deps en la imagen (~200-400MB). Go compila a binario estático: la imagen es literalmente el ejecutable + nada — `distroless` no tiene ni shell.

Mini reto. Los pasos típicos: base `slim` en vez de full (−150MB), multi-stage para dejar el toolchain fuera de la imagen final (cap. 32), `.dockerignore` agresivo, `npm ci --omit=dev` (sin devDependencies), y en Go el salto a `distroless`/scratch. Cada decisión responde "¿esto lo necesita el *runtime* o solo el *build*?" — lo del build no viaja.

41.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

41.12. Lo que deberías saber hacer ahora

- ☐ Explicar imagen vs contenedor vs Dockerfile con la analogía clase/instancia
- ☐ Escribir un Dockerfile funcional para tu stack con capas ordenadas para caché
- ☐ Configurar `.dockerignore` para que secretos y deps no entren
- ☐ Construir y correr la imagen con `docker build`/`run` + env-file
- ☐ Explicar por qué Go produce imágenes ~15MB y los otros no

42 31. Necesitamos la app y Postgres levantándose juntas

El entorno completo en un archivo

> **En una frase**
>
> `docker run postgres`, `docker run app`, crear la red a mano, acordarse del orden… funciona *si no se te olvida un paso*. `docker-compose.yml` declara **el entorno completo — app + BD + su red + sus volúmenes — en un archivo** que se levanta con un comando y se versiona con el código.

42.1. El problema

La receta manual: crear una red, lanzar Postgres con 5 flags, esperar a que esté listo, lanzar la app apuntando al host correcto, repetir mañana. Cada dev lo hace distinto, nadie documenta el orden, y "Postgres arrancó" no significa "Postgres acepta conexiones" — la app crashea en el arranque por llegar 2 segundos temprano.

42.2. Cómo lo resuelve un equipo

Un equipo serio declara el entorno como código: `docker-compose.yml` dice qué servicios existen, cómo se conectan, qué volúmenes persisten sus datos y qué espera a qué. El onboarding es `git clone` + `docker compose up`. Y como el archivo vive en el repo, *el entorno también tiene code review.*

> **Explicación de: volumen (Docker)**
>
> Un **volumen** es almacenamiento que sobrevive al contenedor: el contenedor muere y renace, el volumen (los datos de Postgres) sigue ahí. Sin volumen, `docker compose down` borra tu base de datos.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

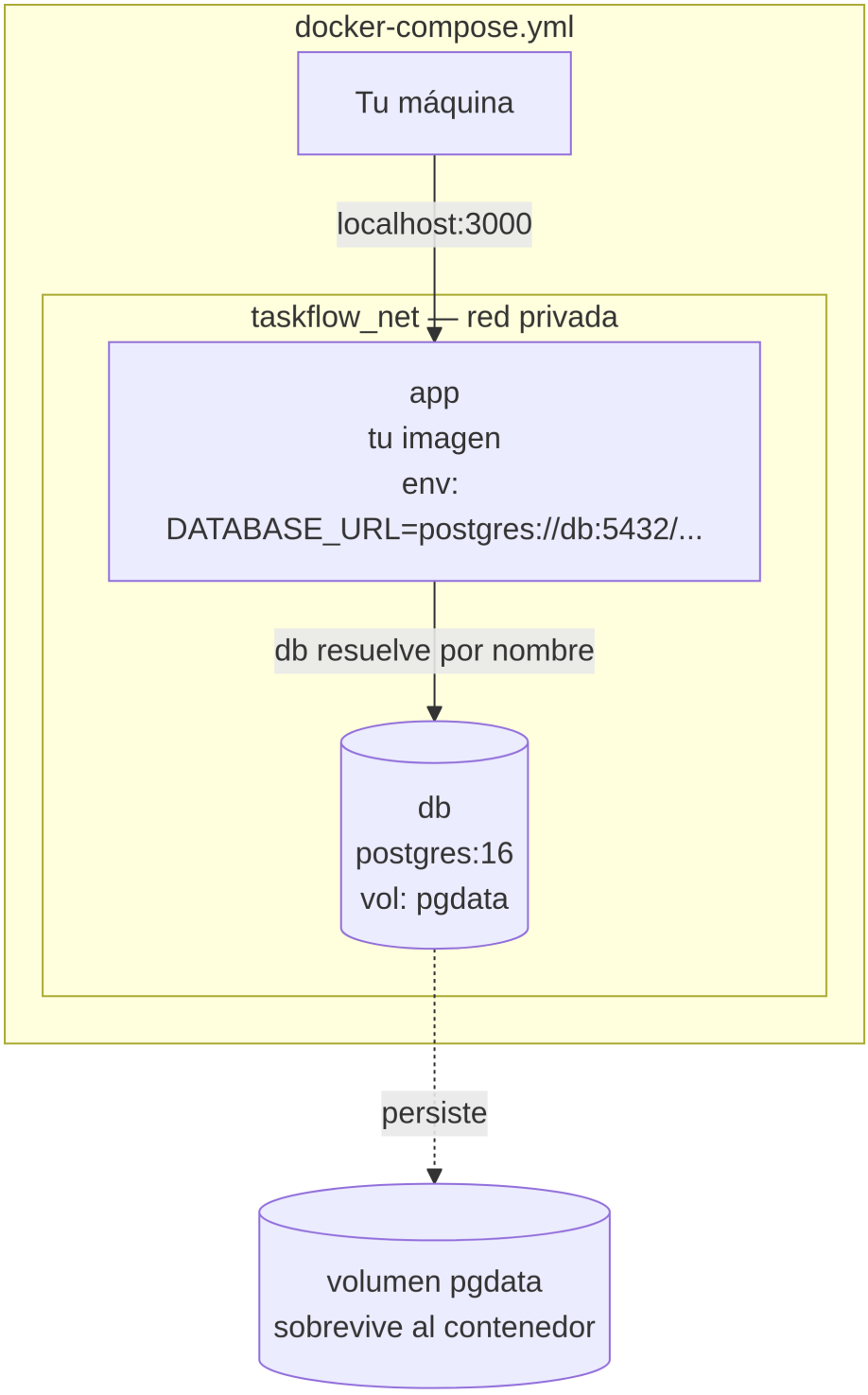
Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

42.3. Conceptos nuevos

- Ops docker-compose: servicios, redes, volúmenes, `depends_on`
- Ops Volúmenes: dónde viven los datos cuando el contenedor muere
- Ops `depends_on` + healthcheck: “Postgres listo” no es “Postgres arrancado”
- Ops El workflow diario: `compose up`, logs, rebuild — `.env` dentro de compose

42.4. La explicación visual



La magia silenciosa: dentro de la red de compose, **`db` es un nombre DNS** — la app conecta a `postgres://db:5432` sin saber IPs. El volumen `pgdata` guarda los datos *fuera* del contenedor: `docker compose down` mata contenedores, no datos.

> Explicación de: contenedor
>
> Un **contenedor** es un proceso con mochila propia: corre aislado con su filesystem y sus librerías, pero comparte el kernel de la máquina — por eso arranca en milisegundos donde una VM tarda minutos.

> Explicación de: puerto
>
> Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

> Explicación de: DNS
>
> El **DNS** es la agenda telefónica de internet: traduce `nexus.com` → `203.0.113.10`. Tu dominio apunta vía DNS a tu load balancer o servidor — por eso "propagación de DNS" toma minutos.

42.5. Implementación

El `docker-compose.yml` completo — agnóstico de lenguaje (cambia la imagen del servicio `app`):

> Explicación de: imagen (Docker)
>
> Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

```yaml
services:
  app:
    build: .                          # the Dockerfile from cap. 30
    ports: ["3000:3000"]              # host:container
    environment:
      DATABASE_URL: postgres://taskflow:taskflow@db:5432/taskflow
      JWT_SECRET: ${JWT_SECRET}       # from your shell's env / .env file
      CORS_ORIGIN: http://localhost:5173
    depends_on:
      db:
        condition: service_healthy    # wait for READY, not just started
    command: ["sh", "-c", "npm run migrate && node dist/index.js"]

  db:
    image: postgres:16
```

```
    environment:
      POSTGRES_USER: taskflow
      POSTGRES_PASSWORD: taskflow
      POSTGRES_DB: taskflow
    volumes:
      - pgdata:/var/lib/postgresql/data   # data outlives the container
    healthcheck:                          # what "ready" means
      test: ["CMD-SHELL", "pg_isready -U taskflow"]
      interval: 2s
      timeout: 3s
      retries: 10

volumes:
  pgdata:
```

El workflow diario:

```
docker compose up --build     # build images + start everything
docker compose logs -f app    # follow one service's logs
docker compose down           # stop and remove containers (data persists)
docker compose down -v        # ALSO wipe the volume - fresh DB
docker compose exec db psql -U taskflow   # jump inside the DB container
```

Prompt listo para tu AI

```
Escribe el docker-compose.yml de mi proyecto [Express+TS/FastAPI/Go] +
Postgres 16. Requisitos: servicio app que buildea del Dockerfile local,
expone el puerto, recibe DATABASE_URL apuntando al servicio db por
nombre, espera a db con depends_on + healthcheck (pg_isready), corre
las migraciones antes de arrancar el servidor, credenciales de dev en
environment, JWT_SECRET desde .env del host, volumen nombrado para los
datos, y los comandos del workflow diario (up, logs, exec psql, down -v).
```

42.6. ¿Por qué así y no "un script bash que lanza todo"?

Lo único que necesitas llevarte: compose convierte *pasos* en *declaración* — describes el estado final ("estos servicios, esta red, estos volúmenes") y Docker calcula cómo llegar. El script describe pasos; el YAML describe la realidad — y se puede leer, revisar y versionar.

Explicación de: Docker

Docker empaqueta tu app con todo lo que necesita (runtime, librerías, config) en una **imagen** — el mismo paquete corre idéntico en tu laptop y en producción. Es la respuesta a "en mi máquina sí funcionaba".

Otras cimentaciones: orquestar el entorno de dev

Alternativa	Qué es	Qué te cuesta	Cuándo
docker-compose	Declarativo, un archivo, en el repo	Solo local/dev — no es orquestador de prod	El default para dev
Makefile/script	Comandos encadenados	Frágil: orden manual, sin healthchecks	Proyecto de un servicio
Devcontainers	El IDE vive *dentro* del contenedor	Config extra, IDE dependiente	Onboarding cero-instalación
Kubernetes local	kind/minikube — prod-like	Overkill para dev diario	Cuando prod es k8s y quieres paridad

42.7. Errores comunes

Error	Por qué pasa	Fix
`depends_on: [db]` sin healthcheck	"Ya depende de db"	`depends_on` solo ordena el *arranque* — `service_healthy` espera a que Postgres acepte conexiones
`localhost` en DATABASE_URL dentro de compose	Es el host que siempre funciona	Dentro de la red de compose el host es `db` — el nombre del servicio es DNS
Datos dentro del contenedor	Sin volumen declarado	`docker compose down` = adiós BD → volumen nombrado siempre
Secretos en el compose commiteado	`POSTGRES_PASSWORD: prod123`	Dev passwords ok en compose; lo real viene de `${VAR}` del entorno
Rebuild manual olvidado	"Cambié el código y no se ve"	`up --build` o `watch` — la imagen no se regenera sola

42.8. Buenas prácticas

- **Las migraciones corren en el `command` de `app`** (o un servicio `migrate` one-shot): `git clone` + `up` = entorno *funcional*, no solo servicios vivos.
- **Un `.env` para compose** con las llaves no-secretas del entorno de dev — compose lo lee automáticamente para `${VAR}`.
- **Servicios con nombres de dominio**: `db`, `api`, `frontend` — el DNS interno los resuelve; nunca hardcodees IPs.

- `docker compose exec` para entrar (psql, shell) — el contenedor es una máquina que se puede inspeccionar, no una caja sellada.

42.9. Ejercicio

1. ¿Por qué `depends_on` sin `condition` no evita el crash de arranque? ¿Qué hace `service_healthy` que el default no hace?
2. Corriste `docker compose down` y tus datos siguen ahí al próximo `up`. ¿Dónde vivían? ¿Qué flag los habría borrado?
3. La app conecta a `postgres://db:5432` — ¿quién resuelve `db` a una IP y por qué no puedes usar `localhost`?

Explicación de: dirección IP

Una **IP** es la dirección numérica de una máquina en la red: `203.0.113.10`. Los servidores tienen IP pública; dentro de una VPC tienen IPs privadas que internet no ve.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

42.10. Mini reto

Un compañero clona el repo y levanta todo con un solo comando.

Soluciones

Ejercicio.

1. `depends_on` por defecto espera a que el contenedor *exista* (`service_started`), no a que Postgres acepte conexiones — la app arranca contra una BD aún inicializándose → crash. `service_healthy` espera al `healthcheck` (`pg_isready` devuelve ok) = "listo de verdad".
2. En el volumen nombrado `pgdata` — los volúmenes viven fuera del ciclo de vida del contenedor. `docker compose down -v` los borra (la BD fresca del siguiente `up`).
3. La red interna de compose tiene DNS: cada servicio es alcanzable por su nombre. `localhost` dentro del contenedor de la app es *el contenedor mismo* — no tu máquina, no el contenedor de Postgres.

Mini reto. Es la prueba de fuego del capítulo: `git clone` + `docker compose up --build` debe bastar — build de la imagen, Postgres con healthcheck, `depends_on` ordenado, migraciones en el `command`, `.env.example` para las llaves que faltan. Si necesita un paso manual, ese paso pertenece al YAML.

42.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

42.12. Lo que deberías saber hacer ahora

- ☐ Escribir un docker-compose.yml con app + Postgres + volumen + healthcheck
- ☐ Explicar por qué `db` funciona como hostname y `localhost` no
- ☐ Correr migraciones automáticamente al levantar el entorno
- ☐ Usar el workflow diario: up, logs, exec, down / down -v
- ☐ Distinguir "Postgres arrancado" de "Postgres listo" (healthcheck)

43 32. Necesitamos una imagen que producción acepte

Funciona, pero pesa 1.2 GB y corre como root

> **En una frase**
>
> El Dockerfile del cap. 30 *funciona* — y es inaceptable para prod: lleva el toolchain completo (tsc, devDependencies, pip), corre como root, y arranca sin migraciones ni healthcheck. La imagen de producción se construye **en etapas** (multi-stage): la herramienta de build se queda atrás; solo viaja lo que el runtime necesita.

43.1. El problema

"Funciona" no es el checklist de producción. La imagen de dev pesa 1.2GB (toolchain + dev deps + cache de paquetes), corre como `root` (un proceso escapado = root en el host), no verifica que la app esté viva, y asume que la BD ya tiene el esquema — el primer deploy en un entorno limpio crashea contra tablas inexistentes.

> **Explicación de: deploy / despliegue**
>
> Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

> **Explicación de: caché / Redis**
>
> Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

> **Explicación de: imagen (Docker)**
>
> Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

43.2. Cómo lo resuelve un equipo

Un equipo serio separa **build** de **runtime**: el multi-stage build usa una imagen gorda para *compilar* y copia solo el artefacto a una imagen mínima para *correr*. Encima: usuario no-root, healthcheck declarado, entrypoint que corre migraciones antes de servir, y la misma imagen en staging y prod — lo que cambia son las env vars (cap. 24: el artefacto no sabe dónde está).

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

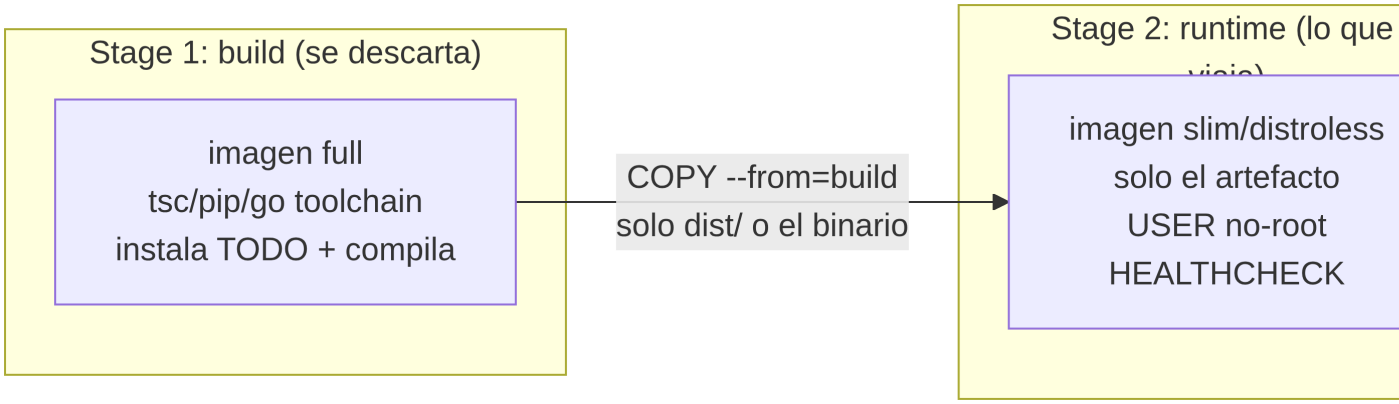
Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

43.3. Conceptos nuevos

- Ops Multi-stage builds: herramientas de build fuera de la imagen final
- Ops Misma imagen en staging y prod, variables distintas (conexión con el cap. de config)
- Ops Migraciones al arranque: entrypoint que corre `migrate` antes de servir
- Ops No-root, healthcheck, señales y shutdown limpio: lo que plataforma revisa

43.4. La explicación visual



El stage de build puede pesar 2GB — no importa: **se descarta**. Lo que cuenta es lo que el `COPY --from` cruza a la imagen final.

43.5. Implementación

Dockerfile de producción por stack:

43.5.1. TypeScript

```
FROM node:20-slim AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci                          # all deps incl. dev (tsc lives here)
COPY . .
RUN npm run build

FROM node:20-slim
WORKDIR /app
ENV NODE_ENV=production
COPY package*.json ./
RUN npm ci --omit=dev               # prod deps only - no tsc, no vitest
COPY --from=build /app/dist ./dist
USER node                           # not root
EXPOSE 3000
HEALTHCHECK --interval=30s --timeout=3s \
  CMD node -e "fetch('http://localhost:3000/health').then(r=>process.exit(r.ok?0:1)).catch(()=>proces
CMD ["sh", "-c", "npm run migrate && node dist/index.js"]   # migrate, then serve
```

43.5.2. Python

```
FROM python:3.12-slim AS build
WORKDIR /app
RUN pip install --upgrade pip
COPY requirements.txt .
RUN pip install --prefix=/install --no-cache-dir -r requirements.txt

FROM python:3.12-slim
WORKDIR /app
COPY --from=build /install /usr/local
COPY . .
RUN useradd -m app && chown -R app /app
USER app                            # not root
EXPOSE 8000
HEALTHCHECK --interval=30s --timeout=3s \
  CMD python -c "import urllib.request;urllib.request.urlopen('http://localhost:8000/health')"
CMD ["sh", "-c", "alembic upgrade head && uvicorn app.main:app --host 0.0.0.0 --port 8000"]
```

43.5.3. Go

```
FROM golang:1.22 AS build
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -ldflags="-s -w" -o server ./cmd/server

FROM gcr.io/distroless/static
COPY --from=build /app/server /server
COPY --from=build /app/migrations /migrations
USER nonroot                          # distroless has no root shell anyway
EXPOSE 8080
# no shell in distroless → migrations run as a separate one-shot job,
# or embed golang-migrate into the binary and run at boot
CMD ["/server"]
```

Prompt listo para tu AI

```
Convierte mi Dockerfile a producción [Node+TS/Python/Go]. Requisitos:
multi-stage (build con toolchain → runtime solo con artefacto),
dependencias solo de producción en la imagen final, usuario no-root,
HEALTHCHECK contra /health, entrypoint que corre migraciones antes de
servir (o job separado si no hay shell), ENV de producción, y la misma
imagen servirá en staging y prod - solo cambian las env vars. Dame el
Dockerfile + el checklist de lo que cambió y por qué.
```

43.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el patrón es universal — *compilar gordo, correr flaco*. Lo que cambia es qué es "el artefacto": `dist/` + deps de prod (Node), el site-packages instalado (Python), o un binario que es literalmente todo (Go — por eso su stage final puede ser `distroless`, sin shell ni OS).

Detalle: el checklist de producción

Requisito	Qué compra	En el Dockerfile de arriba
Multi-stage	Imagen final sin toolchain	`AS build` + `COPY --from`
Prod deps only	Menos superficie y peso	`npm ci --omit=dev`
No-root	Un exploit no es root en el host	`USER node` / `useradd` / `nonroot`

HEALTHCHECK	El orquestador sabe si estás vivo	`HEALTHCHECK CMD ...`
Migrate antes de servir	El esquema viaja con la versión	`migrate && serve` en el CMD
Graceful shutdown	Termina requests antes de morir	`SIGTERM` → cerrar server + pool (app-side)

Shutdown limpio: cuando el orquestador hace rolling deploy manda `SIGTERM` — la app debe parar de aceptar requests, terminar los en curso (5-10s de gracia), cerrar el pool de BD y salir. Un `process.exit(0)` instantáneo corta requests a la mitad.

43.7. Errores comunes

Error	Por qué pasa	Fix
El toolchain en la imagen final	"Build y run en la misma"	Multi-stage — `COPY --from` solo el artefacto
`USER root` por omisión	Es el default	`USER` explícito — plataformas serias lo exigen
Imagen distinta por entorno	`Dockerfile.prod` vs `Dockerfile.staging`	UNA imagen + env vars — la paridad es el punto
Migraciones olvidadas	"La corrí a mano una vez"	En el entrypoint o job — el esquema viaja con el deploy
`SIGTERM` ignorado	El proceso muere a la fuerza	Handler: stop accepting → drain → close pool → exit

43.8. Buenas prácticas

- **La imagen es inmutable y versionada** (`:1.4.2`, `:sha-abc123`) — nunca `latest` en prod: el rollback es cambiar el tag, no rebuildar.
- **Build una vez, promocionar la misma imagen** dev→staging→prod — si se rebuilda por entorno, no estás probando lo que despliegas.
- **Healthcheck vs la app**: el `HEALTHCHECK` del Dockerfile es el fallback; el orquestador (cap. 33) usa `/health` del cap. 20 del Ap. G — ligero para el balanceador, `/health/deep` para ti.
- **Escanea la imagen** (`docker scout`, Trivy en CI, cap. 34): las CVEs de la base son tuyas también — las bases slim/distroless tienen menos que escanear y menos que temer.

43.9. Ejercicio

1. ¿Por qué `npm ci --omit=dev` en el stage final si `dist/` ya está compilado? ¿Qué queda dentro y qué se excluye?

2. El CMD corre `migrate && serve`. ¿Qué pasa si dos réplicas arrancan a la vez y ambas intentan migrar? ¿Cómo lo resuelve el mundo real?
3. Enumera qué hace tu app *idealmente* al recibir `SIGTERM` antes de salir.

Explicación de: lenguaje compilado vs interpretado

Compilado (Go): el código se traduce a binario una vez y ese binario corre solo — rápido, deploy simple. **Interpretado** (Python, JS): un runtime lee el código en cada ejecución — flexible, pero necesita el runtime instalado.

43.10. Mini reto

Auditar la imagen contra un checklist de producción.

Soluciones

Ejercicio.

1. `dist/` es el JS compilado, pero el runtime aún necesita las deps de producción (`express`, `pg`, `zod` — imports en runtime). `--omit=dev` excluye `typescript`, `vitest`, `@types/*` — lo que solo servía para compilar/testear.
2. Las migraciones serias toman un *advisory lock* (pg_migrate/golang- migrate lo hacen): una réplica migra, las demás esperan — la migración es idempotente y ordenada. Alternativa de plataforma: correr migraciones como **job separado** antes de rotar instancias (release phase de Heroku, init job de k8s) — la app solo sirve.
3. (a) Dejar de aceptar conexiones nuevas, (b) terminar las en curso con gracia (~10s), (c) cerrar el pool de BD y workers, (d) `exit(0)`. Requests cortados a la mitad son los errores fantasma de cada deploy.

Mini reto. El checklist: multi-stage , prod-deps only , no-root , healthcheck , migraciones automáticas , `.dockerignore` , tag versionado , SIGTERM manejado , imagen escaneada sin CVEs críticas , misma imagen en todos los entornos . Cada es una línea concreta de trabajo — la imagen de prod es un checklist, no una sensación.

43.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

43.12. Lo que deberías saber hacer ahora

- ☐ Escribir un Dockerfile multi-stage para tu stack
- ☐ Explicar qué viaja del stage de build al de runtime y por qué
- ☐ Correr la app como no-root con healthcheck declarado
- ☐ Integrar migraciones al arranque sin romper el multi-replica
- ☐ Manejar SIGTERM para shutdown limpio con drenado de requests

Parte XI

Sección III · Producción

44 33. Necesitamos servir esto a usuarios reales

uvicorn –reload es para tu máquina

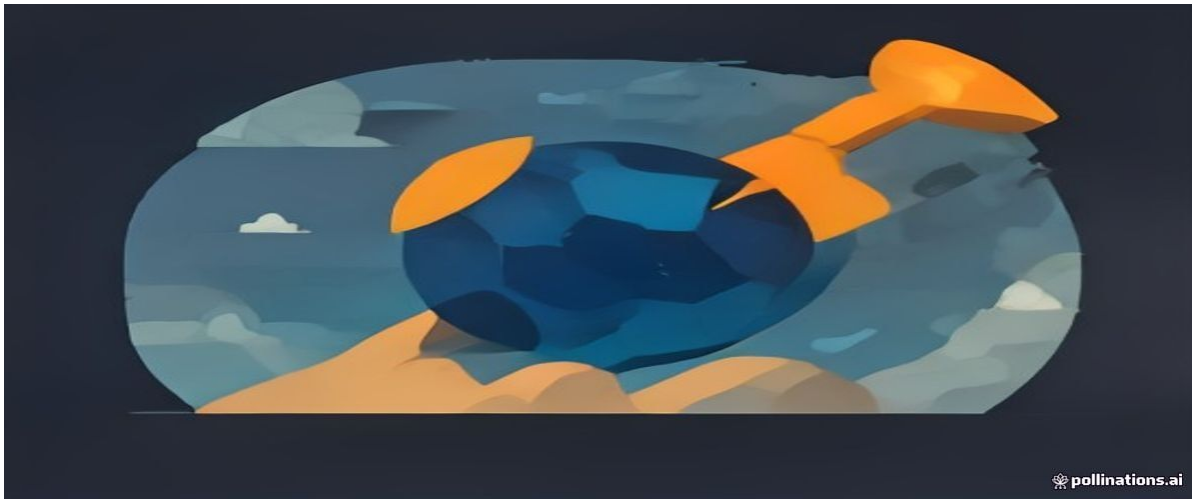


Figura 44.1: Parte 8 — Deployment

> **i** En una frase
>
> `npm run dev`, `uvicorn --reload`, `go run` — todo eso es para *tu* máquina: un proceso, sin HTTPS, que muere y nadie lo revive. Producción es otra arquitectura: **workers para usar todos los núcleos, un reverse proxy que termina TLS, y health checks que le dicen al orquestador si estás vivo** — y en Go, la mitad de eso ya viene en el binario.

44.1. El problema

La imagen de prod corre… y un solo proceso de Node/Python usa **un solo núcleo** de los 8 del servidor — estás pagando 8 y usando 1. Encima: ¿quién termina el HTTPS? ¿quién te revive si el proceso crashea? ¿cómo sabe el balanceador que el contenedor nuevo ya acepta tráfico? Servir a usuarios reales es una topología, no un comando.

Explicación de: HTTPS / TLS / certificado

HTTPS = HTTP viajando cifrado por **TLS**: nadie en el camino puede leer ni alterar el tráfico. El **certificado** es la credencial que prueba que el servidor es quien dice — lo emite una autoridad y hay que renovarlo.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

44.2. Cómo lo resuelve un equipo

Un equipo serio pone tres piezas delante de tu código:

1. **Workers/procesos**: N procesos detrás del mismo puerto (o la plataforma escala N contenedores — misma idea) → los núcleos trabajan todos.
2. **Reverse proxy** (nginx/Traefik/el balanceador de la nube): termina TLS (el candado), sirve estáticos si los hay, y reenvía a tu app — que solo habla HTTP interno.
3. **Health checks**: el orquestador pregunta `/health` cada X segundos — si no respondes, te saca del tráfico y te reinicia.

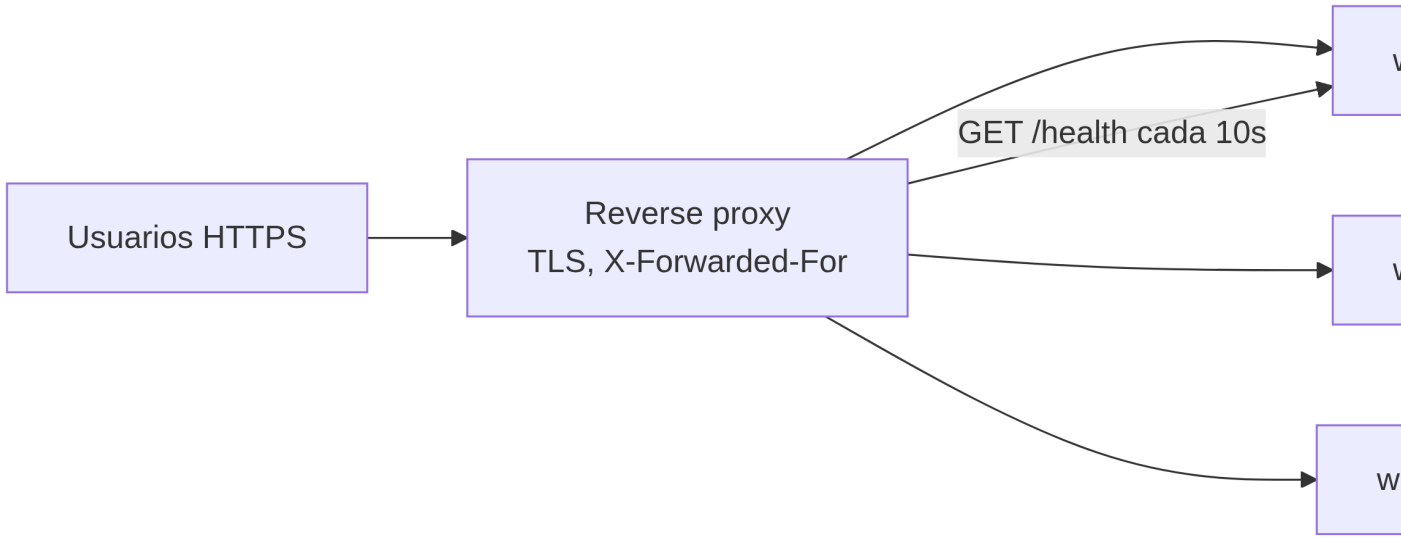
Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

44.3. Conceptos nuevos

- U Workers y procesos: por qué un proceso no usa tus 8 núcleos (GIL/single-thread, mencionado honesto)
- TS PM2/cluster · Py gunicorn+uvicorn workers · Go un solo binario que ya usa todos los núcleos
- Ops Reverse proxy y HTTPS: nginx/Traefik delante — TLS, `X-Forwarded-*`, `COOKIE_SECURE`
- Ops Health checks reales: `/health` para el orquestador vs `/health/deep` para ti

44.4. La explicación visual



El proxy es la *única* cara pública: habla HTTPS afuera, HTTP adentro, y distribuye entre workers. Tu app nunca ve TLS — pero sí necesita saber que el request original era HTTPS (por eso `X-Forwarded-*`).

44.5. Implementación

Cómo cada stack usa todos los núcleos:

44.5.1. TypeScript

```
# PM2: process manager - N processes, restart on crash, logs
pm2 start dist/index.js -i max --name taskflow   # one process per core
pm2 save && pm2 startup                          # survive reboots
```

```
# nginx in front: TLS + proxy to the node processes
server {
  listen 443 ssl;
  server_name api.taskflow.dev;
  ssl_certificate     /etc/letsencrypt/live/api.taskflow.dev/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/api.taskflow.dev/privkey.pem;
  location / {
    proxy_pass http://localhost:3000;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_set_header X-Forwarded-Proto https;   # the app reads this for secure cookies
  }
}
```

44.5.2. Python

```
# gunicorn manages N uvicorn workers - each is an event loop on its own process
gunicorn app.main:app -k uvicorn.workers.UvicornWorker \
  --workers 4 --bind 0.0.0.0:8000
# rule of thumb: workers   (2 × cores) + 1 for I/O-bound APIs
```

```
Same nginx config - proxy_pass http://localhost:8000
Why workers: the GIL means ONE Python process uses ONE core;
N processes = N cores (cap. 20's honest concurrency table)
```

44.5.3. Go

```
// nothing extra: ONE Go binary already uses all cores -
// the runtime schedules goroutines across OS threads (GOMAXPROCS = cores)
func main() {
    srv := &http.Server{
        Addr: ":8080", Handler: router,
        ReadTimeout: 5 * time.Second, WriteTimeout: 10 * time.Second,
        IdleTimeout: 60 * time.Second,          // production timeouts
    }
    log.Fatal(srv.ListenAndServe())              // that's the whole prod server
}
```

```
Same nginx in front if you want TLS/static - or terminate TLS at
the cloud load balancer and run the binary behind it.
```

Prompt listo para tu AI

```
Prepara mi API [Express/FastAPI/Go] para servir usuarios reales detrás
de nginx. Requisitos: N workers/procesos usando todos los núcleos
[PM2 -i max / gunicorn+uvicorn workers / nada extra en Go - explícame
por qué], config de nginx que termine TLS y haga proxy_pass con
X-Forwarded-For y X-Forwarded-Proto, la app leyendo el proto original
para cookies Secure, timeouts de servidor configurados, y /health
ligero para el balanceador + /health/deep con check de BD para
diagnóstico.
```

44.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: Node y Python escalan a núcleos *multiplicando procesos* (un proceso = un núcleo útil); Go ya paraleliza dentro de un proceso — por eso "un binario" es una respuesta completa, no una simplificación.

Detalle: procesos vs el modelo de cada runtime

Stack	Para usar 8 núcleos	Por qué
Node	`pm2 -i 8` o 8 contenedores	Un proceso = un event loop = un hilo
Python	`gunicorn --workers N`	El GIL: un proceso ejecuta un hilo de Python a la vez — N procesos = N núcleos
Go	`./server` — ya está	Las goroutines se reparten en threads OS sobre todos los núcleos

En la nube esto se simplifica: el orquestador (ECS, Cloud Run, App Service) escala *contenedores*, no workers — corres 1 proceso por contenedor y escala horizontal. Los workers importan cuando *tú* operas la máquina (VPS, cap. nu-01) — y para entender qué estás pagando.
`X-Forwarded-Proto` y las cookies: tras el proxy, tu app ve HTTP — sin ese header, "¿el request original era HTTPS?" es imposible de saber y las cookies `Secure` se rompen o se desactivan. El proxy lo declara; la app lo confía *solo desde el proxy* (`trust proxy` en Express, el middleware correspondiente en los demás).

44.7. Errores comunes

Error	Por qué pasa	Fix
`uvicorn --reload` en prod	"Es como lo corro en dev"	Reload watches files + un proceso — prod es gunicorn/PM2/orquestador
Un proceso en máquina de 8 núcleos	El default	Workers × núcleos — o contenedores × réplicas
La app habla TLS ella misma	"Pongo el cert en Express"	TLS en el proxy — la app ve HTTP interno + `X-Forwarded-Proto`
`/health` que toca la BD	"Chequeo todo"	El balanceador pega cada 10s × N instancias — `/health` barato, `/health/deep` aparte
Confiar `X-Forwarded-*` de cualquiera	`trust proxy` global	Solo confiar cuando viene *del proxy* — el cliente puede falsificarlo directo

44.8. Buenas prácticas

- **Un proceso por contenedor** en orquestadores: la plataforma escala réplicas; meter PM2 dentro del contenedor es orquestar dentro de la orquestación — señales y healthchecks se confunden.
- **Timeouts en el servidor** (Read/Write/Idle): el servidor sin timeouts guarda conexiones muertas hasta morir él.
- **Graceful + healthchecks en el deploy**: el orquestador marca "unhealthy" → te reinicia; marca "starting" → no te manda tráfico aún. Tu `/health` *es* el contrato con la plataforma.
- **El proxy también cachea/comprime**: gzip y estáticos en nginx — tu app no paga esos ciclos.

> EXTRA · Servidores web del mundo real
>
> EXTRA Los roadmaps piden conocer servidores web — el trio real:
>
> - **Nginx** — el reverse proxy estándar: sirve estáticos rapidísimo y reparte a tu app
> - **Apache** — el veterano; `.htaccess` y hosting clásico
> - **Reverse proxy** — el *rol*, no un producto: terminar TLS, servir estáticos, balancear; Nginx/Caddy/Traefik lo juegan
>
> Tu app (Node/Python/Go) nunca mira internet directamente — el proxy es su fachada.

44.9. Ejercicio

1. Tienes un VPS de 8 núcleos con FastAPI. ¿Cuántos procesos corren tu código en un setup serio y qué los levanta?
2. ¿Por qué `/health` NO debe verificar la BD, si "salud real" suena mejor?
3. Tras el proxy, Express ve `req.secure === false` aunque el usuario entró por HTTPS. ¿Qué falta y dónde?

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

44.10. Mini reto

Explicar el camino completo de un request desde el navegador hasta tu handler en prod.

> Soluciones
>
> **Ejercicio.**
>
> 1. ~(2×8)+1 17 es la regla teórica para I/O-bound; en la práctica 4-8 workers + autoscale según

medición — los levanta `gunicorn -k uvicorn.workers.UvicornWorker`. Si el orquestador es de contenedores: 1 proceso por contenedor × N réplicas.
2. El balanceador pega cada ~10s a *todas* las instancias — si `/health` toca la BD, multiplicas carga y, peor: un flap de BD marca TODAS las instancias unhealthy y el orquestador las tumba a la vez — conviertes un problema de BD en una caída total. `/health` = "el proceso sirve"; `/health/deep` = diagnóstico para ti.
3. `app.set("trust proxy", ...)` + el proxy enviando `X-Forwarded-Proto: https`. Sin lo primero, Express ignora el header (correcto — no confiar en headers del cliente); sin lo segundo, la cookie `Secure` nunca se setea.

Mini reto. DNS → IP del balanceador/proxy → handshake TLS (el candado) → nginx/balanceador termina TLS → `proxy_pass` a un worker sano (preguntó `/health` recién) → tu router → middleware (request_id, auth, rate limit) → handler → servicio → pool → Postgres. Nueve saltos, y cada uno es un capítulo del libro — por eso la cadena es repasable.

44.11. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: goroutine

Una **goroutine** es el hilo ultraligero de Go: `go f()` lanza una función en paralelo gastando casi nada — se pueden tener millones. Es la superpotencia de Go: concurrencia como palabra del lenguaje.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

Explicación de: DNS

El **DNS** es la agenda telefónica de internet: traduce `nexus.com` → `203.0.113.10`. Tu dominio apunta vía DNS a tu load balancer o servidor — por eso "propagación de DNS" toma minutos.

Explicación de: dirección IP

Una **IP** es la dirección numérica de una máquina en la red: `203.0.113.10`. Los servidores tienen IP pública; dentro de una VPC tienen IPs privadas que internet no ve.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: cookie

Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

44.12. Lo que deberías saber hacer ahora

- ☐ Correr N workers en tu stack (PM2/gunicorn) y explicar por qué Go no los necesita
- ☐ Configurar un reverse proxy con TLS + `X-Forwarded-*`
- ☐ Distinguir `/health` (barato, orquestador) de `/health/deep` (BD, diagnóstico)
- ☐ Confiar en `X-Forwarded-Proto` solo desde el proxy
- ☐ Narrar el camino completo navegador→handler en producción

45 34. Necesitamos que nadie rompa main

Si depende de la memoria, fallará

> En una frase
>
> "¿Pasaste los tests antes del merge?" — en un equipo real nadie pregunta: **el pipeline corre lint + typecheck + tests + build en cada PR, y si algo falla, el merge está bloqueado.** CI no es burocracia: es la memoria del equipo convertida en máquina — lo que se olvida en la cabeza, no se olvida en el YAML.

45.1. El problema

El flujo manual: tú corres los tests (cuando te acuerdas), el compañero pushea sin correrlos "porque era un cambio chiquito", main se rompe el viernes y nadie sabe desde qué commit. Y el deploy: `ssh` + `git pull` desde la laptop de una persona — si esa laptop se pierde, nadie despliega. Todo lo que depende de *acordarse* acaba fallando.

> Explicación de: deploy / despliegue
>
> Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

> Explicación de: commit
>
> Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

45.2. Cómo lo resuelve un equipo

Un equipo serio convierte las verificaciones en **eventos automáticos**: cada PR dispara el pipeline — instala deps, corre migraciones sobre un Postgres efímero, corre la suite completa, buildea — y GitHub marca el PR en verde o rojo. El gate de merge ("CI verde obligatorio") convierte la regla de equipo en regla de máquina. Y CD continúa: merge → build de imagen → registry → deploy — ningún `ssh` desde laptops.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice “crea esta columna” (up) y “bórrala” (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

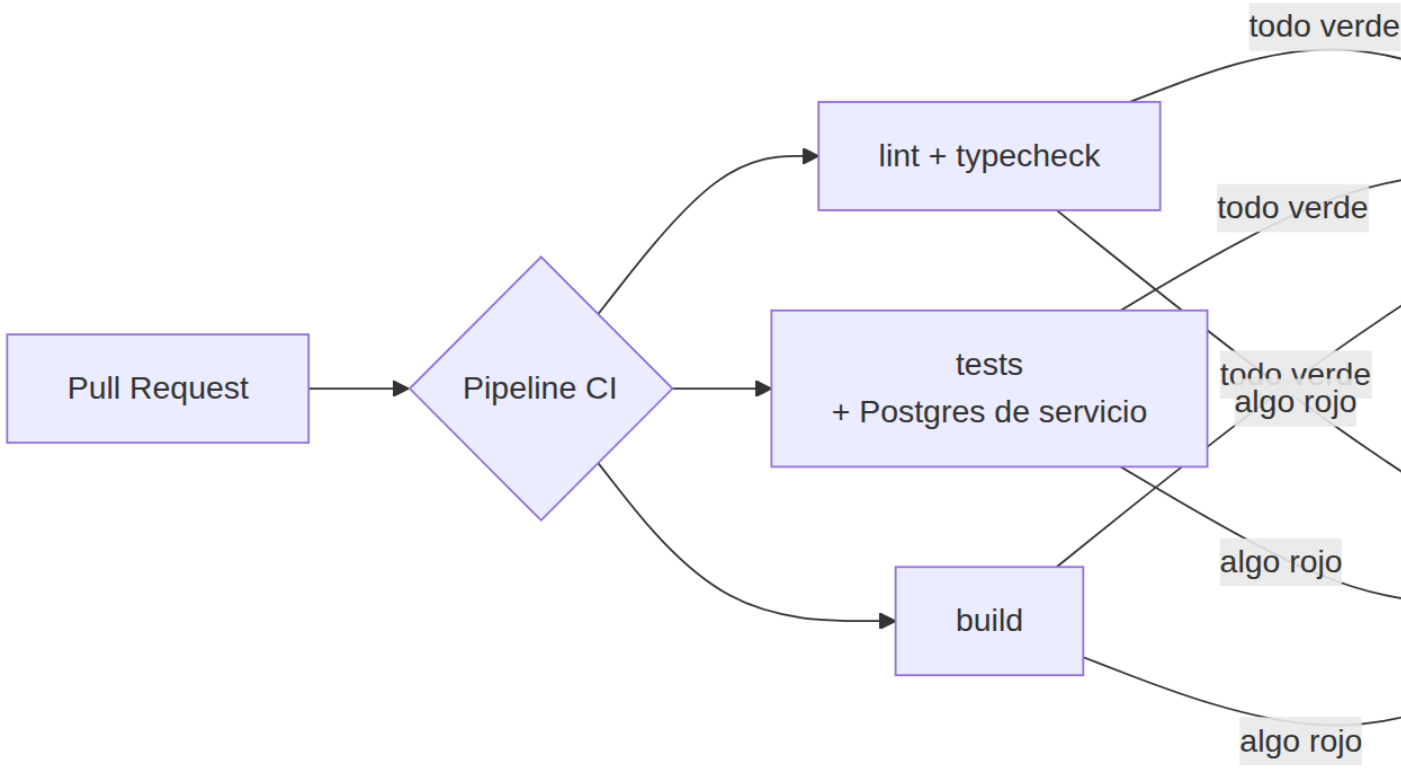
Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

45.3. Conceptos nuevos

- Ops CI con GitHub Actions: lint + typecheck + tests + build por stack
- Ops La BD en CI: Postgres como servicio del pipeline — el esquema es reproducible
- U Gates de merge: cobertura mínima, CI verde obligatorio — la conversación de equipo
- Ops CD: build de imagen → registry → deploy — las piezas y su orden

45.4. La explicación visual



El pipeline es el mismo para los tres stacks — lo que cambia son los comandos de cada paso (`npm run lint` / `ruff` / `go vet`), no la forma.

45.5. Implementación

Capítulo agnóstico — el pipeline de GitHub Actions que sirve para los tres (los comandos por stack van comentados):

```
# .github/workflows/ci.yml
name: ci
on:
  pull_request:
  push: { branches: [main] }

jobs:
  verify:
    runs-on: ubuntu-latest
    services:
      postgres:                                # a REAL Postgres for the integration tests
```

```
      image: postgres:16
      env: { POSTGRES_PASSWORD: ci, POSTGRES_DB: taskflow_test }
      ports: ["5432:5432"]
      options: >-
        --health-cmd "pg_isready" --health-interval 2s --health-retries 10
  env:
    DATABASE_URL: postgres://postgres:ci@localhost:5432/taskflow_test
    JWT_SECRET: ci-secret-key-for-tests-only-32chars
  steps:
    - uses: actions/checkout@v4

    - uses: actions/setup-node@v4          # TS: setup-node · Py: setup-python · Go: setup-go
      with: { node-version: 20, cache: npm }

    - run: npm ci                          # Py: pip install -r requirements.txt
    - run: npm run lint                    # Py: ruff check · Go: go vet ./...
    - run: npm run typecheck               # Py: mypy · Go: (compiler does it)
    - run: npm run migrate                 # the schema is reproducible - that's the point
    - run: npm test                        # Py: pytest · Go: go test ./...
    - run: npm run build                   # proves it compiles, not just that tests pass

image:
  needs: verify
  if: github.ref == 'refs/heads/main'      # only after merge to main
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - run: docker build -t registry.example.com/taskflow:${{ github.sha }} .
    - run: docker push registry.example.com/taskflow:${{ github.sha }}
    # deploy step: platform-specific (ECS update / kubectl set image / Render hook)
```

Prompt listo para tu AI

```
Crea el workflow de GitHub Actions para mi API [Express+TS/FastAPI/Go]
+ Postgres. Requisitos: en cada PR y push a main correr lint, typecheck,
migraciones y tests contra un Postgres 16 de servicio con healthcheck
y DATABASE_URL+JWT_SECRET de test como env vars; un segundo job que solo
corre tras merge a main: build de la imagen Docker taggeada con el sha
del commit y push al registry; caché de dependencias; y los comandos
correctos para mi stack en cada paso.
```

45.6. ¿Por qué así y no “cada quien corre los tests en su máquina”?

Lo único que necesitas llevarte: CI convierte la regla en *infraestructura* — no hay que acordarse ni confiar en que cada uno corrió la suite: el PR lo dice, y main solo acepta código verificado. La laptop

que despliega es un riesgo; el pipeline que despliega es un proceso.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Otras cimentaciones: dónde corre el pipeline

Alternativa	Qué es	Cuándo
GitHub Actions	YAML en el repo, runners de GitHub	El default si el repo está en GitHub
GitLab CI / Azure DevOps	Lo mismo en otras plataformas	Si el equipo ya vive ahí
Pre-commit hooks	Verificación *antes* de pushear	Complemento: atrapa lo rápido (lint) sin esperar CI
Jenkins self-hosted	CI que tú operas	Empresas con requisitos de infra propia — pagas el ops

45.7. Errores comunes

Error	Por qué pasa	Fix
CI que no bloquea el merge	"Está para informar"	Branch protection: status check requerido — el gate es la mitad del valor
Tests conectando a una BD compartida	"Ya existe el staging"	Postgres efímero del pipeline — reproducible, aislado, sin estado
Secretos de prod en el CI	"Necesitaba el deploy"	Secrets del repo + OIDC — nunca en el YAML ni en logs
Pipeline de 20 minutos	Todo en un solo job	Paralelizar lint/typecheck/test + caché de deps — CI lento es CI que se salta
Tests de CI distintos a los locales	"En mi máquina pasan"	Mismos comandos en ambos — el pipeline corre lo que tú corres

45.8. Buenas prácticas

- **El pipeline corre lo mismo que tú corres** — si tu test local es `npm test`, el paso es `npm test`: una sola forma de verificar, no dos dialectos.
- **Las migraciones corren en CI** sobre la BD efímera — el test "¿el esquema se construye de cero?" se responde en cada PR, gratis.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

- **CD separado del CI**: verificar desplegar — el build de imagen solo tras merge a main, taggeado con el sha (rollback = tag anterior, cap. 35).
- **Secretos solo en el almacén de secretos** del repo/plataforma — el YAML referencia, nunca contiene.

45.9. Ejercicio

1. ¿Por qué el pipeline corre *migraciones + tests* y no solo tests? ¿Qué bug del mundo real detecta eso que "tests con BD ya poblada" no detecta?
2. Ordena el CD: ¿por qué el deploy va *después* del build de imagen y no en paralelo?
3. Tu compañero propone correr la suite contra staging "porque ya tiene datos reales". ¿Dos problemas con eso?

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

45.10. Mini reto

Hacer fallar el build a propósito con un test roto y ver el gate funcionar.

Soluciones

Ejercicio.

1. Las migraciones en CI verifican que el esquema *se construye desde cero* — detecta la migración que funciona sobre tu BD de dev "porque la columna ya existía" pero falla en una base vacía. El bug clásico: migración que asume estado que solo existe en tu máquina.
2. El deploy consume el *artefacto* del build — deployar sin imagen es deployar nada, y deployar una imagen del build que aún corre es una carrera: puedes publicar algo que no compiló. `needs: verify` es la cola ordenada.
3. (a) Los tests *escriben y borran* — correrlos contra datos reales de staging contamina lo que otros ven/usan; (b) staging es compartido: dos PRs corriendo se pisan datos entre sí → falsos rojos que enseñan a ignorar CI. La BD efímera es aislada y gratis.

Mini reto. Cambia un `expect` a algo falso, abre un PR, y mira: CI rojo → merge bloqueado por el gate → el botón gris que dice "checks failing". Revierte el test: verde. El experimento de 5

minutos que convierte “el gate existe” en algo que viste funcionar — y demuestra por qué ningún “era un cambio chiquito” pasa sin tests.

45.11. Vocabulario técnico del capítulo

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

Explicación de: Docker

Docker empaqueta tu app con todo lo que necesita (runtime, librerías, config) en una **imagen** — el mismo paquete corre idéntico en tu laptop y en producción. Es la respuesta a "en mi máquina sí funcionaba".

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

45.12. Lo que deberías saber hacer ahora

☐ Escribir un workflow de GitHub Actions con lint+typecheck+tests+build

☐ Levantar Postgres como servicio efímero del pipeline con healthcheck
☐ Configurar el gate de merge (CI verde obligatorio)
☐ Separar CI (verificar) de CD (build imagen → registry → deploy)
☐ Mantener secretos fuera del YAML y la BD de test efímera

46 35. Necesitamos cambiar producción sin tumbarla

Deploy a las 4 PM del viernes, con migración. ¿Qué puede salir mal?

> **En una frase**
>
> Deployar es reemplazar el avión en vuelo: la versión vieja y la nueva conviven unos minutos — y la BD es *una sola* para las dos. Por eso los deploys sin downtime no son un flag de la plataforma: son una **disciplina — migraciones compatibles hacia atrás (expand/contract), rollback que sabe qué se revierte y qué no, y ojos en los logs después del deploy.**

46.1. El problema

`git push` a prod a las 4 PM del viernes: la plataforma levanta la versión nueva mientras apaga la vieja. En esos 90 segundos, la v1.4 y la v1.5 conviven — y ambas hablan con la *misma* base de datos. Si la migración de la v1.5 renombró `name` → `full_name`, la v1.4 que sigue viva escribe `name` en una columna que ya no existe: errores por todas partes, en el mejor momento posible.

> **Explicación de: producción**
>
> **Producción** (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

> **Explicación de: base de datos**
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

> **Explicación de: migración**
>
> Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no

se editan una vez aplicadas.

46.2. Cómo lo resuelve un equipo

Un equipo serio asume la coexistencia: durante el deploy, **dos versiones del código comparten una BD** — por tanto la migración debe ser segura para *las dos*. La técnica se llama **expand/contract**:

> Explicación de: deploy / despliegue
>
> Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

1. **Expand**: agregar lo nuevo sin tocar lo viejo (columna nueva, tabla nueva — nunca renombrar ni borrar).
2. Código que escribe en *ambos* (o migra datos en segundo plano).
3. **Contract**: cuando todo usa lo nuevo, otra migración retira lo viejo — un deploy *después*, no en el mismo.

Y el seguro: el rollback de *código* es cambiar el tag de imagen; el rollback de *datos* no existe gratis — los datos que la v1.5 escribió en `full_name` no se des-escriben al volver a v1.4.

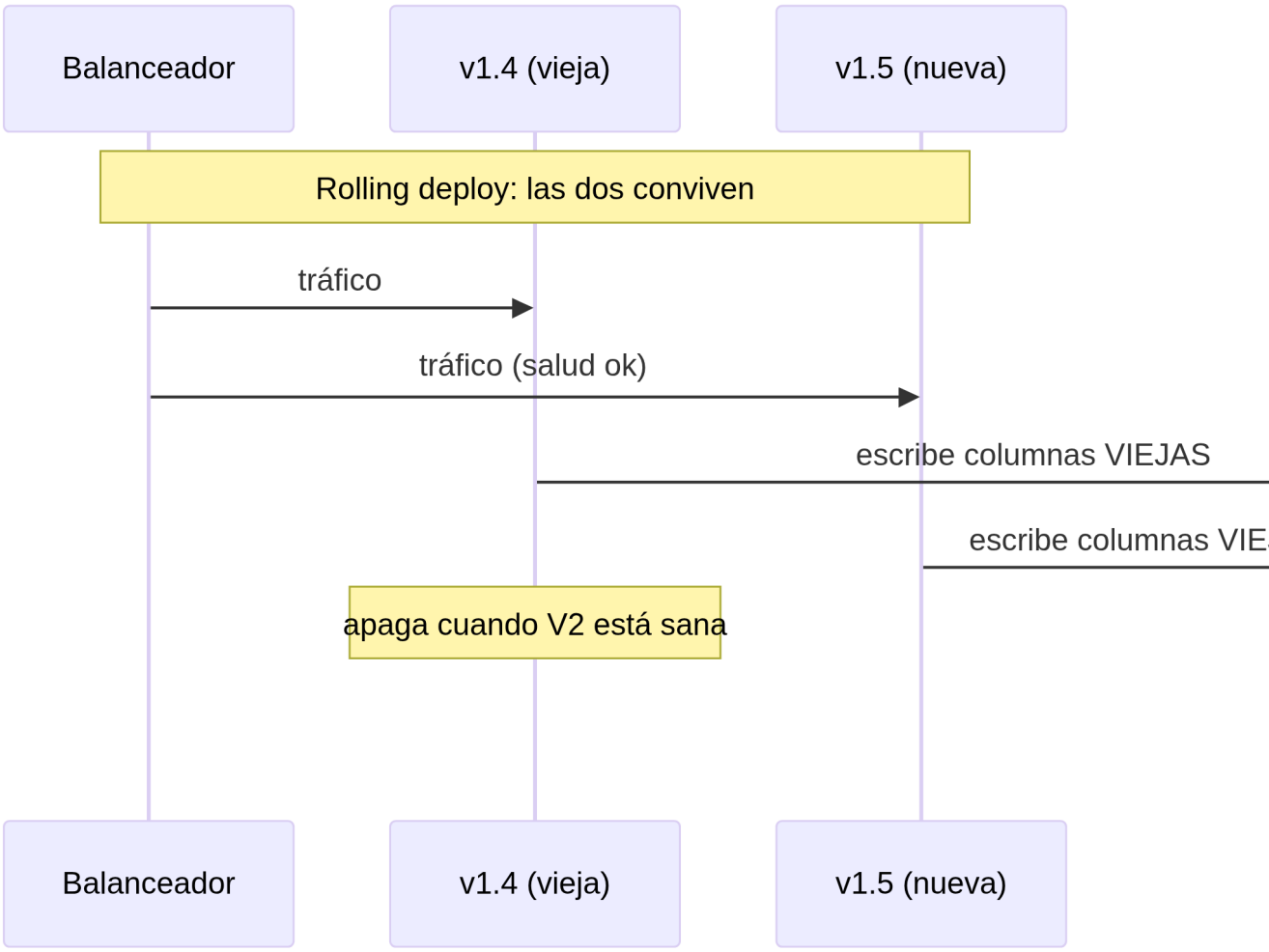
> Explicación de: rollback
>
> Un **rollback** es volver atrás: la migración se revierte, el deploy regresa a la versión anterior. Todo cambio serio tiene plan de rollback *antes* de ejecutarse — si no, el plan es rezar.

> Explicación de: imagen (Docker)
>
> Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

46.3. Conceptos nuevos

- Ops Deploys sin downtime (concepto): rolling, qué exige de tu app y de tus migraciones
- D Migraciones compatibles: agregar columna sí, renombrar con datos en prod no — expand/contract
- Ops Rollback: qué se revierte (código) y qué no se revierte gratis (datos)
- Ops Post-deploy: logs, errores, latencia — observabilidad mínima viable

46.4. La explicación visual



La regla de oro: **la BD debe entender a la versión vieja y a la nueva a la vez** — toda migración se evalúa contra esa pregunta.

46.5. Implementación

Capítulo agnóstico — la receta expand/contract del cambio clásico "`name` → `first_name` + `last_name`":

```
-- Paso 1 (deploy N): EXPAND - add only, nothing removed
ALTER TABLE users ADD COLUMN first_name TEXT;
ALTER TABLE users ADD COLUMN last_name  TEXT;
```

```sql
-- Paso 2 (mismo deploy N): el código escribe las TRES columnas
-- (name sigue poblada para la v1.4 que aún vive)
-- + backfill en segundo plano:
UPDATE users SET first_name = split_part(name, ' ', 1),
                 last_name  = split_part(name, ' ', 2)
WHERE first_name IS NULL;

-- Paso 3 (deploy N+2, cuando TODO lee las nuevas): CONTRACT
ALTER TABLE users ALTER COLUMN first_name SET NOT NULL;
ALTER TABLE users DROP COLUMN name;
```

Tres deploys para "renombrar una columna" — parece excesivo hasta que comparas con la alternativa: el deploy que la renombra de golpe tumba la versión vieja en plena rotación.

Prompt listo para tu AI

```
Diseña el plan de deploy sin downtime para este cambio: renombrar
users.name → users.first_name + last_name en mi API [stack] con Postgres
y rolling deploys. Dame el plan expand/contract completo: qué
migraciones en qué orden, qué escribe cada versión del código mientras
conviven, cómo se hace el backfill sin bloquear la tabla (batches),
cuándo es seguro el DROP COLUMN, y qué se revierte (y qué NO) si hay
que hacer rollback en cada etapa.
```

46.6. ¿Por qué así y no "apagar, migrar, encender"?

Lo único que necesitas llevarte: el downtime programado funciona mientras tienes 40 usuarios; con usuarios reales el deploy es en caliente — y en caliente la BD es el punto único compartido, así que toda la disciplina es *mantener la BD compatible con dos versiones*.

Otras cimentaciones: estrategias de deploy

Estrategia	Qué es	Qué te cuesta	Cuándo
Rolling	Reemplaza instancias de a una	Convivencia de versiones (este capítulo)	El default en orquestadores
Blue-green	Dos entornos completos; el balanceador cambia de golpe	Doble infra mientras dura	Rollback instantáneo crítico

Canary	La nueva versión recibe 1 % del tráfico primero	Más plomería de routing	Cambios riesgosos a escala
Recreate	Apagar todo, subir nuevo	Downtime real	Dev/staging, apps internas

46.7. Errores comunes

Error	Por qué pasa	Fix
Migración destructiva en el deploy	`DROP`/`RENAME` con v1.4 viva	Expand/contract — destructivo solo cuando nada viejo corre
Migración que bloquea la tabla	`ALTER` pesado en horario	Operaciones baratas en el hot path (`ADD COLUMN` es instantáneo en PG 11+); lo pesado, por pasos/ventana
Rollback = "deployar la imagen vieja"	Olvida que los datos ya cambiaron	Saber qué se revierte: código sí, datos escritos no — por eso expand/contract también es rollback-friendly
Deployar y mirar el techo	"Salió verde el pipeline"	Post-deploy: logs, tasa de errores, latencia los primeros 15 min (cap. 23)
Viernes 4 PM sin poder revertir	Ritmo	Deployar cuando hay equipo despierto — la valentía no es un plan

46.8. Buenas prácticas

- **Toda migración pasa el test**: "¿la versión anterior del código sigue funcionando con este esquema?" — si no, va por pasos.

> Explicación de: esquema (schema)
>
> El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

- **Backfill en batches** (`UPDATE ... WHERE id BETWEEN` o por lotes) — un UPDATE de millones de filas de golpe bloquea la tabla que atiende tráfico.
- **El deploy taggeado = el rollback**: `registry/app:sha-viejo` — revertir código es cambiar una etiqueta, no rebuildar nada.
- **Post-deploy checklist**: error rate, p95, logs de excepciones nuevas — los 15 minutos después del deploy son donde vive el downtime real.

46.9. Ejercicio

1. ¿Por qué `RENAME COLUMN name TO full_name` en un solo deploy tumba la versión vieja aunque la nueva esté lista?
2. Diseña el expand/contract para *eliminar* la columna `legacy_flag` — ¿qué va primero, el código o la migración?
3. La v1.5 lleva 20 minutos en prod escribiendo en `first_name` cuando hay que hacer rollback a v1.4. ¿Qué se pierde y qué no?

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

46.10. Mini reto

Planear "dividir name en first_name/last_name" sin downtime.

Soluciones

Ejercicio.

1. Porque durante el rolling, la v1.4 (que consulta `name`) sigue viva mientras la migración ya la renombró → sus queries fallan. Renombrar es destructivo *para la otra versión* — siempre hay dos versiones durante la ventana del deploy.
2. Código primero: deploy que *deja de leer* `legacy_flag` (la columna queda huérfana pero presente — la v1.4 sigue feliz). Cuando nada la lee, la migración `DROP COLUMN` en el deploy siguiente — contract es el último paso, nunca el primero.
3. Se revierte el *código* (imagen vieja, un tag) — los datos escritos en `first_name` durante esos 20 min **no** se revierten: v1.4 no los lee (lee `name`, que v1.5 también mantuvo escrita por diseño). Si la migración hubiera sido destructiva, el rollback sería una restauración de backup, no un tag.

Mini reto. El plan de la sección Implementación de arriba: expand (agregar columnas) → dual-write + backfill en batches → todo el código lee lo nuevo → contract (NOT NULL + DROP). Cada paso es deploy seguro porque cada esquema intermedio sirve a ambas versiones — *esa* es la frase que se responde en la entrevista.

46.11. Vocabulario técnico del capítulo

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: latencia / throughput

Latencia = cuánto tarda UNA operación (ms). **Throughput** = cuántas haces por segundo. Se pelean: bajar latencia no siempre sube throughput. La latencia la siente el usuario; el throughput lo paga tu factura.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: staging

Staging es el ensayo general: una copia de producción (misma config, datos falsos o anonimizados) donde pruebas el deploy antes de hacerlo de verdad. Si falla ahí, falló gratis.

Explicación de: load balancer

Un **load balancer** reparte el tráfico entre tus réplicas: request entra → lo manda a la instancia menos ocupada. Además detecta cuál está caída (health checks) y deja de enviarle.

Explicación de: Kubernetes (k8s)

Kubernetes es el orquestador de contenedores: le dices "quiero 3 réplicas de esta imagen" y él las distribuye, reinicia las que mueren y las expone. Poderoso y complejo — se usa cuando las máquinas ya son manada.

Explicación de: backup / respaldo

Un **backup** es una copia de los datos en otro lugar, hecha seguido y *probada* (un backup que nunca se restauró no es backup, es esperanza). Regla 3-2-1: 3 copias, 2 medios, 1 fuera del sitio.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: registry (Docker)

Un **registry** es el almacén de imágenes (Docker Hub, ECR, Artifact Registry): el CI empuja `nexus:v42`, el servidor la jala. Es el intermediario entre "se construyó" y "está corriendo".

Explicación de: CI/CD / pipeline

CI (integración continua): cada push corre tests y build automático. **CD** (despliegue continuo): si todo pasa, se despliega solo. El **pipeline** es la cadena de pasos — GitHub Actions es el motor típico.

46.12. Lo que deberías saber hacer ahora

- ☐ Explicar por qué dos versiones conviven durante un rolling deploy
- ☐ Diseñar migraciones expand/contract para cambios destructivos
- ☐ Hacer backfills en batches sin bloquear la tabla caliente
- ☐ Distinguir qué se revierte (código/tag) de qué no (datos)
- ☐ Definir el checklist post-deploy: errores, latencia, logs nuevos

Parte XII

Sección III · Proveedores cloud

47 N3. AWS, Azure y GCP: el mismo supermercado con tres logos

Los tres gigantes venden lo mismo — aprende las equivalencias

> En una frase
>
> AWS, Azure y Google Cloud ofrecen **los mismos servicios con distintos nombres**: EC2 = Azure VM = Compute Engine. Aprende las ~10 categorías, no el logo — y sabrás moverte en cualquiera de los tres.

47.1. El problema

"Hay que usar AWS" suena a aprender un universo: 200+ servicios, siglas por todas partes, una consola que parece cabina de avión. La verdad liberadora: **de esos 200 servicios usas 10**, y esos 10 existen idénticos en los tres proveedores. El trabajo no es aprender AWS — es aprender las categorías (compute, storage, base de datos, DNS, identidad, colas…) y luego traducir nombres.

> Explicación de: la nube / cloud
>
> **La nube** son computadoras de otro que rentas por hora: AWS, Azure, GCP te prestan máquinas, redes y servicios administrados. Pagas por no comprar hardware — y por no administrarlo tú.

> Explicación de: background job / cola
>
> Un **job** es trabajo que se hace sin que el usuario espere: mandar el email, generar el PDF. Va a una **cola** y un *worker* lo procesa en segundo plano — la respuesta HTTP sale inmediata.

> Explicación de: terminal / consola
>
> La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: DNS

El **DNS** es la agenda telefónica de internet: traduce `nexus.com` → `203.0.113.10`. Tu dominio apunta vía DNS a tu load balancer o servidor — por eso "propagación de DNS" toma minutos.

47.2. Cómo lo resuelve un equipo

Nadie memoriza el catálogo. Un equipo real:

1. **Elige UN proveedor** — casi siempre por razones externas: dónde ya está la empresa, créditos gratis, qué sabe el equipo.
2. **Aprende sus 10 servicios núcleo** en ese proveedor.
3. **Guarda la tabla de equivalencias** para cuando toque otro.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

El proveedor dominante es AWS (~30 % del mercado), luego Azure (~20 %, fuerte en empresas con Microsoft), luego GCP (~10 %, fuerte en datos/ML). Los PaaS modernos (Render, Vercel, Fly.io) viven *encima* de estos tres.

Explicación de: IaaS / PaaS / SaaS

IaaS: rentas la máquina, administras todo lo de adentro. **PaaS**: rentas la plataforma, solo traes tu código. **SaaS**: usas el software hecho (Gmail, RDS administrado). Menos control, menos trabajo.

47.3. Conceptos nuevos

- Ops **Compute:** EC2 / Azure VM / Compute Engine — las VMs del cap. N1
- Ops **Storage de objetos:** S3 / Blob / Cloud Storage — archivos infinitos por HTTP
- D **BD administrada:** RDS / Azure SQL / Cloud SQL — Postgres sin ser DBA (cap. N4)
- Ops **Serverless:** Lambda / Azure Functions / Cloud Functions (cap. N5)
- Ops **IAM:** identidad y permisos — quién puede tocar qué, en los tres
- Ops **Regiones y zonas:** dónde vive físicamente tu recurso (latencia + leyes)

47.4. La explicación visual

La tabla de equivalencias — la herramienta más útil del capítulo:

Categoría	AWS	Azure	GCP
Compute (VM)	EC2	Virtual Machines	Compute Engine
Storage objetos	S3	Blob Storage	Cloud Storage
BD relacional	RDS	Azure SQL Database	Cloud SQL
BD NoSQL	DynamoDB	Cosmos DB	Firestore
Serverless	Lambda	Azure Functions	Cloud Functions
Contenedores	ECS/EKS	AKS	GKE/Cloud Run
DNS	Route 53	Azure DNS	Cloud DNS
CDN	CloudFront	Azure CDN	Cloud CDN
Identidad	IAM	Entra ID (ex-AD)	Cloud IAM
Secretos	Secrets Manager	Key Vault	Secret Manager
Colas	SQS	Service Bus	Pub/Sub
Observabilidad	CloudWatch	Azure Monitor	Cloud Monitoring

Fíjate el patrón: las categorías son universales porque resuelven problemas universales. Azure renombra más (Entra ID era "Active Directory"), GCP usa nombres descriptivos ("Compute Engine"), AWS usa siglas ("EC2" = Elastic Compute Cloud). Mismo supermercado.

47.5. Implementación

El mismo trabajo — *subir el frontend estático de Taskflow (Sección I) a un bucket público* — en los tres proveedores. Mira cómo cambian solo los nombres:

47.5.1. AWS

```
# awscli - one profile configured with `aws configure`
aws s3 mb s3://taskflow-web-you              # make bucket
aws s3 sync ./dist s3://taskflow-web-you     # upload the build
aws s3 website s3://taskflow-web-you --index-document index.html
# + CloudFront distribution in front for HTTPS + caching
```

47.5.2. Azure

```
# az cli - `az login` first
az storage account create -n taskflowwebyou -g my-rg
az storage blob service-properties update \
  --account-name taskflowwebyou --static-website --index-document index.html
az storage blob upload-batch -s ./dist -d '$web' --account-name taskflowwebyou
```

47.5.3. GCP

```
# gcloud cli - `gcloud init` first
gsutil mb gs://taskflow-web-you
gsutil -m rsync -r ./dist gs://taskflow-web-you
gsutil web set -m index.html gs://taskflow-web-you
gsutil iam ch allUsers:objectViewer gs://taskflow-web-you
```

Tres comandos equivalentes: *crear contenedor de objetos* → *subir build* → *habilitar como sitio web*. La categoría (static hosting en object storage) es la misma; solo cambia el CLI.

> Explicación de: objeto / dict / struct
>
> Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

> Explicación de: build
>
> El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

> Explicación de: CLI
>
> Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

El tour mental de cualquier consola: todas se organizan igual — *región* arriba a la derecha (dónde vive el recurso), *buscador* de servicios (las 10 categorías), *IAM* (quién eres y qué puedes tocar), *Billing* (dónde mirar cada semana). Domina esas 4 zonas y cualquier consola deja de intimidar.

> Prompt listo para tu AI

```
Quiero desplegar [mi frontend estático / mi API / ambos] en [AWS /
Azure / GCP] usando el free tier. Dame los comandos CLI exactos en
orden: crear los recursos mínimos, subir el código, exponer la URL, y -
igual de importante - los comandos para DESTRUIR todo al terminar y no
generar costos. Explícame qué hace cada recurso que creamos en
términos de las categorías universales (compute/storage/identidad).
```

47.6. ¿Por qué existen tres?

Lo único que necesitas llevarte: compiten por el mismo pastel, así que convergen a los mismos servicios — la diferencia real no es técnica sino de *ecosistema*: Azure entra por las empresas que ya pagan Microsoft, GCP por equipos de datos/ML, AWS por ser el primero y tener la comunidad más grande. Para ti, la habilidad transferible es la categoría, no la consola.

i Otras cimentaciones: elegir proveedor

Criterio	Qué significa	Peso real
El que ya usa tu empresa/cliente	No eliges — vas ahí	El 80 % de los casos
Créditos/free tier	AWS Activate, Azure for Startups, $300 de GCP	Alto al empezar
Ecosistema de tu stack	.NET → Azure natural; ML → GCP Vertex	Medio
Multi-cloud "por si acaso"	Usar dos proveedores a la vez	Casi nunca vale el doble de complejidad
Proveedores alternos	Cloudflare, DigitalOcean, Hetzner, OVH	Excelentes para cosas específicas y precio

El mito del multi-cloud: *"si AWS cae, tengo Azure"* cuesta duplicar todo tu conocimiento e infraestructura. Lo que sí se hace: diseñar sin lock-in innecesario (contenedores portables, Postgres estándar) para que migrar sea *posible*, no diario.

47.7. Errores comunes

Error	Por qué pasa	Fix
La factura sorpresa	Dejaste una VM/bucket/NAT corriendo "por si acaso"	Billing alerts día 1 + destruir lo que no uses
Bucket S3 público con datos privados	"Public" suena a "mi sitio web"	Bloqueo de acceso público por defecto; revisa dos veces
Trabajar como root/admin	La cuenta raíz puede TODO — incluso vaciar tu cuenta	IAM user con mínimos permisos + MFA
Todo en una región lejana	"us-east-1 por defecto" aunque tus usuarios estén en LATAM	Región cerca de usuarios (latencia) y legal del dato
Aprender "AWS" en vez de categorías	Memorizas siglas que no transfieren	Esta tabla de equivalencias

47.8. Buenas prácticas

- **Billing alert antes que cualquier recurso.** Todos tienen presupuestos con alerta por email — configúrala en tu primera sesión.
- **MFA en la cuenta raíz + usuario IAM para el día a día.** La cuenta raíz queda guardada como la escritura de la casa.
- **Tags en cada recurso** (`project:taskflow`, `env:dev`) — en 3 meses no sabrás qué VM era de qué.
- **Destruye los experimentos.** El tutorial terminó = recursos borrados. La nube cobra por existir, no por usar.

47.9. Ejercicio

1. Traduce a Azure y GCP: EC2, S3, RDS, Lambda, Route 53.
2. Tu app necesita: una VM, una Postgres administrada, storage para imágenes de usuarios, y DNS. Nombra los 4 servicios en AWS.
3. ¿Por qué "multi-cloud desde el día 1" casi nunca es buena idea?

Explicación de: serverless / función

Serverless = no piensas en servidores: subes una *función* y el proveedor la ejecuta por evento, escalando de 0 a 1000 sola. Pagas por ejecución. Lambda (AWS), Cloud Functions (GCP), Azure Functions.

Soluciones

1. EC2 → Azure VM / Compute Engine · S3 → Blob Storage / Cloud Storage · RDS → Azure SQL / Cloud SQL · Lambda → Azure Functions / Cloud Functions · Route 53 → Azure DNS / Cloud DNS.
2. EC2 (compute), RDS (Postgres administrada), S3 (imágenes como objetos), Route 53 (DNS). Son 4 de las ~10 categorías — ya nombraste medio AWS.
3. Porque duplicas el costo de conocimiento y complejidad para cubrir un evento raro (la caída total de un proveedor). Lo correcto: evitar lock-in *innecesario* (tecnologías portables) y aceptar que migrar es un proyecto, no un interruptor.

47.10. Mini reto

Encuentra los **tres** problemas en este "deploy a AWS":

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

```
aws configure    # entered ROOT account keys
aws s3 mb s3://company-user-data
aws s3 sync ./backups s3://company-user-data
aws s3api put-bucket-acl --bucket company-user-data --acl public-read
# done! left the t2.large test instance running "just in case"
```

Soluciones

1. **Llaves de la cuenta raíz** — pueden hacer TODO incluyendo cerrar la cuenta; deben ser credenciales de un usuario IAM con permisos mínimos.
2. **`public-read` en datos de usuarios** — el bucket queda listable y descargable por cualquiera en internet. Es el error que produce las filtraciones masivas de "S3 bucket expuesto" en las noticias.
3. **Instancia corriendo "por si acaso"** — una t2.large son ~$70/mes de nada. Regla de la nube: se cobra por existir.

47.11. Vocabulario técnico del capítulo

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: CDN

Una **CDN** (Content Delivery Network) es una red de copias: tu estático se cachea en servidores cercanos al usuario. El de Buenos Aires no viaja a Virginia por una imagen — la recibe del nodo local.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

Explicación de: contenedor

Un **contenedor** es un proceso con mochila propia: corre aislado con su filesystem y sus librerías, pero comparte el kernel de la máquina — por eso arranca en milisegundos donde una VM tarda minutos.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: HTTPS / TLS / certificado

HTTPS = HTTP viajando cifrado por **TLS**: nadie en el camino puede leer ni alterar el tráfico. El **certificado** es la credencial que prueba que el servidor es quien dice — lo emite una autoridad y hay que renovarlo.

47.12. Lo que deberías saber hacer ahora

- ☐ Traducir un servicio de un proveedor a los otros dos
- ☐ Nombrar las ~10 categorías universales que todos venden
- ☐ Configurar billing alerts y un usuario IAM antes de crear recursos
- ☐ Explicar por qué el multi-cloud prematuro es una trampa

48 N4. Servicios administrados: paga por no administrar

Managed Postgres, storage, DNS, CDN y secretos — lo que la nube te quita

> **En una frase**
>
> El valor real de la nube no son las máquinas — son los **servicios administrados**: una Postgres con backups y failover automáticos (RDS), un storage prácticamente indestructible (S3), DNS global, CDN, y un lugar diseñado para secretos. Pagas para *no* ser DBA ni sysadmin a las 3 de la mañana.

48.1. El problema

En N1 administraste un VPS: SO, nginx, systemd. Ahora imagina hacer lo mismo con **PostgreSQL**: instalarla, configurar backups diarios, probar que esos backups restauran (los backups que nunca se prueban no existen), montar una réplica por si el servidor muere, aplicar parches de seguridad sin tumbar el servicio, cifrar los discos. Eso es un *empleo de tiempo completo* llamado DBA — y tu equipo de 3 personas no tiene uno.

48.2. Cómo lo resuelve un equipo

La respuesta de la industria: **comprar la administración, no solo la máquina**. Un servicio administrado te da el software funcionando y el proveedor absorbe: parches, backups, failover, réplicas, escalado de disco. Tú recibes una **connection string** y un panel.

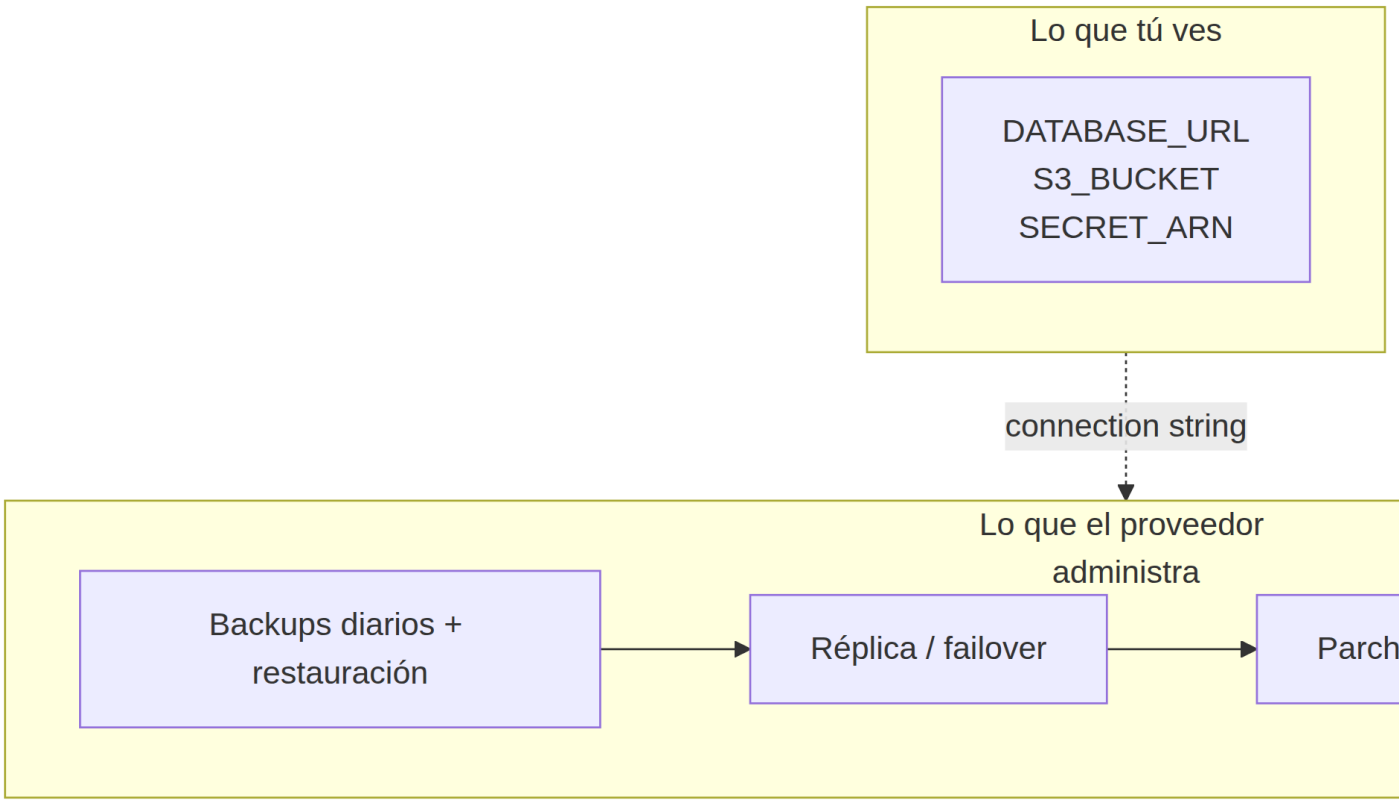
> **Explicación de: string**
>
> Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

El intercambio es el patrón de toda esta sección: *menos control, menos trabajo, más factura por unidad.* Para casi todo equipo, el trato sale ganando — las horas de DBA cuestan más que la diferencia de precio.

48.3. Conceptos nuevos

- D **BD administrada** (RDS, Cloud SQL, Azure SQL, Supabase, Neon, PlanetScale): backups, réplicas y parches incluidos
- Ops **Object storage** (S3, Blob, GCS): archivos por HTTP, durabilidad de "11 nueves" — NO es un filesystem
- Ops **DNS administrado** (Route 53, Cloudflare) y **CDN** (CloudFront, Cloudflare): tu contenido copiado cerca de cada usuario
- Ops **Secretos administrados** (Secrets Manager, Key Vault, Doppler): dónde van las API keys de verdad
- Ops **Observabilidad administrada** (CloudWatch, Datadog): logs/métricas sin montar el stack

48.4. La explicación visual



Servicio	Qué te quita de encima	Qué NO es
RDS	Backups, failover, parches, réplicas, cifrado	"Postgres en una VM" — es Postgres *con equipo de ops incluido*

Servicio	Qué te quita de encima	Qué NO es
S3	Discos, RAID, capacidad, replicación	"Una carpeta en la nube" — es key→objeto por HTTP, sin directorios reales
CDN	Servidores en 100+ ciudades	"Un acelerador mágico" — es caché geográfica de tus estáticos
Secrets Manager	Rotación, auditoría, cifrado de secretos	"Un .env caro" — rota credenciales sin redeploy

48.5. Implementación

Conectar Taskflow a una Postgres administrada — el flujo es idéntico en los tres proveedores porque **la app no sabe quién administra la base**:

Paso 1 — provisionar (consola o CLI, 5 minutos):

```
# AWS example - creates a managed Postgres
aws rds create-db-instance \
  --db-instance-identifier taskflow-db \
  --db-instance-class db.t3.micro \
  --engine postgres --master-username app \
  --manage-master-user-password     # secret auto-created in Secrets Manager
```

Paso 2 — la app solo ve una URL. Esto es lo elegante: tu código de los caps. 8–13 no cambia *nada* — cambia una variable de entorno:

> Explicación de: variable
>
> Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

> Explicación de: variable de entorno
>
> Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

> Explicación de: entorno (environment)
>
> Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

```
# .env - the ONLY difference between local and managed Postgres
# Local:          DATABASE_URL=postgresql://app:dev@localhost:5432/taskflow
# Managed (RDS):  DATABASE_URL=postgresql://app:****@taskflow-db.abc.us-east-1.rds.amazonaws.com:5432
```

Paso 3 — el secreto vive en el servicio de secretos, no en tu `.env` ni en el historial del shell (¿recuerdas el mini reto de N1?):

```
aws secretsmanager get-secret-value --secret-id taskflow/db
# → {"DATABASE_URL": "postgresql://..."}   fetched at boot, never in a file
```

El contrato mental: el proveedor te entrega un *endpoint + credencial auditable*; tú entregas la app configurada por entorno. La frontera de N2 aplicada a la base de datos.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Prompt listo para tu AI

```
Migra mi API de tareas de la Postgres local a una base de datos
administrada [RDS / Cloud SQL / Neon / Supabase]. Dame: los pasos para
provisionar la instancia (tier gratuito), cómo correr mis migraciones
existentes contra ella, qué cambia en mi configuración (pista: solo una
env var), dónde guardar la connection string (secret manager, no .env
commiteado), y cómo verificar backups automáticos habilitados. Incluye
el paso que la mayoría olvida: cerrar el acceso público y abrir solo
desde mi app.
```

48.6. ¿Por qué no administrarlo todo tú?

Lo único que necesitas llevarte: un servicio administrado convierte *conocimiento operativo* (que tardas años en ganar) en *factura mensual* (que puedes pagar hoy). La pregunta correcta nunca es "¿puedo hacerlo yo?" — es "¿quiero mantenerlo yo a las 3am dentro de dos años?".

Otras cimentaciones: administrado vs autogestionado

Alternativa	Qué te cuesta	Cuándo elegirla
Managed DB (RDS/Cloud SQL)	~2× el precio de la VM equivalente, menos control fino	Casi siempre — es la cimentación por defecto

Postgres en tu VPS/contenedor	Toda la operación: backups, failover, parches	Aprendizaje, presupuesto cero, o requisitos muy raros
DBaaS nueva gen (Neon, Supabase, Turso)	Dependencia de un proveedor más joven	MVPs: free tiers generosos, DX excelente
Self-hosted + herramientas (CloudNativePG)	Operación Kubernetes + DBA-light	Escala grande donde el 2× de RDS pesa

El punto medio moderno: **Neon/Supabase/Turso** — Postgres administrada con free tier real y DX de startup. Para Taskflow en producción inicial, son la opción más sensata: administrado *y* barato.

48.7. Errores comunes

Error	Por qué pasa	Fix
RDS expuesta a internet	"Así la pruebo desde mi laptop"	Red privada + acceso solo desde la app (o bastion)
Secreto en `.env` commiteado	El repo es eterno aunque borres el commit	Secret manager + rotar lo que ya se filtró
S3 usado como filesystem	`s3://bucket/uploads/` como "carpeta" con millones de archivos y listados	Es key→objeto: prefijos sí, `ls` barato no
Pensar "administrado = infalible"	RDS cae también; solo cae menos	Backups + réplica + plan de restauración igualmente
Facturación por sorpresa en S3	Cobran por GB, requests y *egress* (salida de datos)	Lifecycle policies + revisar egress en la factura

48.8. Buenas prácticas

- **La app se configura por entorno, no por proveedor:** `DATABASE_URL` apunta a localhost en dev y a RDS en prod — mismo código.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

- **Backups verificados:** programa una restauración de prueba. Un backup que nunca restauraste es una esperanza, no un plan.
- **Menor privilegio también en la nube:** la BD de la app solo acepta conexiones de la app, no de `0.0.0.0/0`.
- **Un secreto, un lugar:** mezclar `.env`, Secrets Manager y variables del CI dispersa la responsabilidad — elige uno por entorno.

EXTRA · Las tres capas de caché del roadmap

EXTRA Cuando un roadmap dice "aprende caching", son tres capas distintas —

- **CDN** — la copia en el borde, cerca del usuario (*este capítulo*)
- **Caché del lado del servidor** — Redis o Memcached: resultados de queries caras, sesiones, rate limits
- **Caché del lado del cliente** — el navegador guarda estáticos con headers de expiración

El libro lo deja como decisión (Ap. K #14): el caché se agrega cuando un problema *medido* lo pide — antes es complejidad sin beneficio.

48.9. Ejercicio

1. Nombra **cuatro** cosas que RDS hace por ti y que tendrías que implementar a mano en un VPS.
2. Un junior dice: "S3 es como Google Drive del servidor". ¿Qué dos diferencias técnicas rompen la analogía?
3. ¿Dónde debe vivir `DATABASE_URL` en producción y por qué no en un `.env` del servidor?

Soluciones

1. Backups automáticos con restauración point-in-time, failover a réplica (si muere el nodo, otro toma el lugar), parches de seguridad aplicados sin downtime, réplicas de lectura, cifrado de disco, monitoreo. Cualquiera de las cuatro es correcta.
2. (a) S3 es key→objeto por HTTP — no hay directorios reales ni `ls` barato (listar millones de objetos cuesta). (b) La durabilidad es por diseño (11 nueves: objetos replicados entre instalaciones) — no es "un disco que puedes formatear", es un servicio de objetos.
3. En un secret manager (Secrets Manager, Key Vault, las env vars del PaaS). El `.env` del servidor queda en disco en texto plano, en backups del filesystem, y en la historia del repo si se commitea — el secret manager da cifrado, rotación y auditoría.

48.10. Mini reto

Tu Taskflow ya corre en un VPS (N1) y la Postgres vive **en la misma máquina**. Enumera tres escenarios donde esa decisión duele — y cuál es la primera señal de que debes moverla a un servicio administrado.

Soluciones

Escenarios: (a) el VPS muere → pierdes app **y** datos en un solo incidente; (b) el disco se llena de WAL/ backups improvisados → nadie los prueba hasta el desastre; (c) necesitas reiniciar/parchear el SO → también tumbas la base. La primera señal de migración: **el momento en que la base guarda datos que no puedes volver a generar** — típicamente el primer usuario real. Desde

ese día, “Postgres en la misma caja” es una deuda con vencimiento desconocido.

48.11. Vocabulario técnico del capítulo

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: DNS

El **DNS** es la agenda telefónica de internet: traduce `nexus.com` → `203.0.113.10`. Tu dominio apunta vía DNS a tu load balancer o servidor — por eso “propagación de DNS” toma minutos.

Explicación de: CDN

Una **CDN** (Content Delivery Network) es una red de copias: tu estático se cachea en servidores cercanos al usuario. El de Buenos Aires no viaja a Virginia por una imagen — la recibe del nodo local.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el

caché ya no vale) es lo difícil.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: cifrado / encriptación

El **cifrado** convierte datos en basura legible solo con la llave — y a diferencia del hash, *se puede revertir*. HTTPS cifra el cable; las contraseñas NO se cifran, se hashean (no hay por qué revertirlas).

48.12. Lo que deberías saber hacer ahora

- ☐ Decir qué te quita de encima un servicio administrado
- ☐ Explicar por qué S3 no es “un disco” y RDS no es “Postgres en una VM”
- ☐ Conectar tu app a una BD administrada cambiando solo una env var
- ☐ Justificar dónde viven los secretos en producción

49 N5. Serverless y edge: tu código sin tu servidor

Lambda, Functions, Workers y cuándo la nube corre por ti

> **En una frase**
>
> Serverless es la cimentación alternativa del Cap. 1 llevada al extremo: **no hay proceso escuchando — tu función despierta con cada request y muere al responder**. Pagas por ejecución, no por servidor encendido. Cuándo conviene y cuándo te sale carísimo es la lección del capítulo.

49.1. El problema

Tu API de tareas tiene tráfico irregular: 200 requests el lunes, 3 el martes, 5.000 cuando un cliente la descubre. En el modelo de N1–N2 pagas un servidor **encendido 24/7** que duerme la mayor parte del día — como dejar un taxi corriendo en la puerta por si algún día necesitas ir al aeropuerto. La pregunta de este capítulo: ¿y si el taxi solo existiera *mientras viajas*?

49.2. Cómo lo resuelve un equipo

El modelo **FaaS** (Function as a Service): subes una función — no una app, no un servidor — y el proveedor la ejecuta cuando algo la invoca (un HTTP request, un archivo subido, un cron). Cobran por milisegundo de ejecución. Sin requests, costo $0.

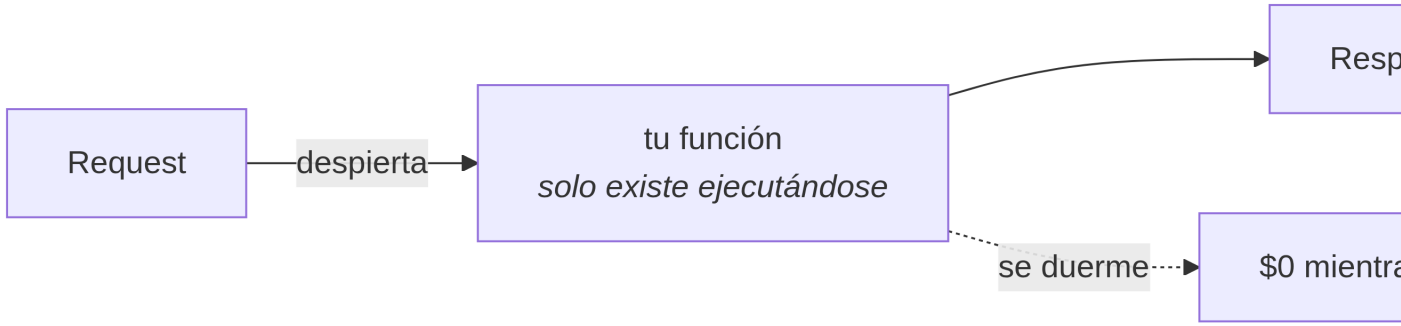
El caso que enamora: **webhooks, eventos, trabajos ocasionales**. El caso que arruina: una API con tráfico constante y alto, donde el precio por ejecución supera al servidor fijo — y donde los *cold starts* (la función "fría" tarda en despertar) pegan en cada usuario nuevo.

49.3. Conceptos nuevos

- Ops **FaaS:** Lambda, Azure Functions, Cloud Functions — función por evento
- Ops **Edge:** Cloudflare Workers, Vercel Edge, Deno Deploy — tu código corre en el datacenter *más cercano al usuario*

- U **Cold start:** la función dormida tarda ~100ms–2s en despertar — imperceptible o inaceptable según tu caso
- U **Sin estado:** cada ejecución nace nueva — la memoria y el disco NO sobreviven entre requests
- Ops **Vendor lock-in:** la función usa el runtime/eventos del proveedor — moverla es un proyecto

49.4. La explicación visual



Comparado con lo que ya conoces:

	Proceso (N1/N2)	Serverless
Qué existe	Un proceso escuchando 24/7	Una función que despierta por evento
Pagas por	Servidor encendido (aunque duerma)	Milisegundos de ejecución
Estado	Memoria/disco del proceso	NADA persiste entre requests
Escala	Tú la configuras	Automática: 1 o 10.000 ejecuciones
Latencia	Siempre caliente	Cold starts ocasionales

49.5. Implementación

El mismo endpoint — `GET /tasks` — como función serverless. Fíjate qué desaparece: el `listen()`, el puerto, el servidor. La función recibe un

> Explicación de: puerto
>
> Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

evento (el request envuelto por el proveedor) y devuelve un objeto:

49.5.1. TypeScript (Lambda)

```
// handler.ts - AWS Lambda + API Gateway
export const handler = async (event: any) => {
  const tasks = await listTasks();   // DB via DATABASE_URL from env
  return {
    statusCode: 200,
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(tasks),     // Lambda returns an OBJECT, not res.json()
  };
};
// No app.listen() - the platform invokes handler() per request
```

49.5.2. Python (Lambda)

```
# handler.py - AWS Lambda
def handler(event, context):
    tasks = list_tasks()              # DB via DATABASE_URL from env
    return {
        "statusCode": 200,
        "headers": {"Content-Type": "application/json"},
        "body": json.dumps(tasks),
    }
# No uvicorn, no port - AWS calls handler(event, context)
```

49.5.3. Go (Lambda)

```
// main.go - AWS Lambda
func handler(ctx context.Context, req events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse,
    tasks, err := listTasks(ctx)     // DB via DATABASE_URL from env
    if err != nil {
        return events.APIGatewayProxyResponse{StatusCode: 500}, nil
    }
    body, _ := json.Marshal(tasks)
    return events.APIGatewayProxyResponse{StatusCode: 200, Body: string(body)}, nil
}

func main() { lambda.Start(handler) } // the runtime calls handler() per request
```

¿Y si ya tienes una app completa y no quieres reescribirla? Los **adaptadores** envuelven tu app existente en un handler: `aws-serverless-express` (Express), `mangum` (FastAPI), `aws-lambda-go-api-proxy` (Go). Tu código de los caps. 2–7 viaja casi intacto — la ruta sigue siendo la ruta, pero ahora la invoca la plataforma en vez de un socket.

El edge es serverless + geografía: la función no corre en `us-east-1` sino en el punto de presencia *más cercano al usuario* (Cloudflare Workers corre en 300+ ciudades). Ideal para: auth checks, redirects, personalización ligera — código que debe responder en <50ms desde cualquier país.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Prompt listo para tu AI

```
Convierte mi endpoint GET /tasks de [Express/FastAPI/Go] a una función
serverless para [AWS Lambda / Cloudflare Workers / Vercel]. Requisitos:
usa el adaptador apropiado para no reescribir toda la app (o el handler
nativo si prefieres), la conexión a la base de datos debe funcionar con
el modelo sin estado (conexión por ejecución o pooler - explícame cuál
y por qué), y dame el archivo de configuración de despliegue. Al final
explícame qué partes de mi código asumen un proceso siempre vivo y
dejarían de funcionar.
```

49.6. ¿Por qué no es siempre la respuesta?

Lo único que necesitas llevarte: serverless gana cuando el tráfico es *irregular* (pagas solo lo que usas y escala solo) y pierde cuando el tráfico es *constante y alto* (una función cobra más por millón de requests que un servidor fijo) o cuando el trabajo es *largo* (límites de ~15 min) o *con estado* (websockets, sesiones en memoria). El cálculo es honesto, no ideológico.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: sesión

Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada llamada.

Otras cimentaciones: el mapa final de deployment

Alternativa	Modelo	Qué te cuesta	Cuándo elegirla
VPS (N1)	Proceso siempre vivo	Toda la ops	Aprendizaje, control total, precio fijo
PaaS (N2)	Subes código	Precio/unidad mayor	El default sano para la mayoría de APIs
Contenedores (cap. 30+)	Subes imagen	Sabes Docker + orquestación	Portabilidad entre proveedores
FaaS	Subes función	Cold starts, límites, lock-in, sin estado	Eventos, webhooks, tráfico irregular
Edge	Función en 300 ciudades	Runtime limitado (no todo Node corre)	Latencia global, middleware ligero

El espectro completo del libro en una tabla: **misma app, cinco cimentaciones**. La elección correcta depende del tráfico, el equipo y el tiempo de vida del proceso — nunca del hype.

49.7. Errores comunes

Error	Por qué pasa	Fix
"Serverless = gratis"	El free tier engaña; a 10M requests/mes Lambda puede costar más que un VPS	Haz la cuenta a tu volumen real
Estado en memoria entre requests	La función nueva no recuerda nada	Estado externo: DB, Redis, KV — como debe ser siempre
El monolito en UNA lambda	Una función gigante hace todo → cold start enorme, despliegue lento	Funciones por dominio, o app completa via adaptador sabiendo el costo
Conexión DB por request	Postgres no tolera 5.000 funciones abriendo conexiones	Pooler (RDS Proxy, PgBouncer, Neon serverless)
Ignorar cold starts en auth	El primer request del día tarda 2s	Keep-warm, provisioned concurrency, o acepta el trade

49.8. Buenas prácticas

- **Diseña sin estado de todas formas** — es buena práctica para todo backend (el proceso siempre vivo también puede morir), y te deja la puerta serverless abierta.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

- **Serverless para eventos primero:** webhooks, procesamiento de archivos, crons — ahí el modelo es perfecto antes de mover tu API entera.
- **Mide el cold start real** de tu función con sus dependencias — el de los benchmarks (una función vacía) no es el tuyo.
- **Vigila la factura por volumen:** configura alertas; el modelo escala solo… incluido el cobro.

49.9. Ejercicio

1. Clasifica cada caso como "serverless conviene" o "proceso fijo conviene": (a) webhook de pagos que llega 50 veces/día, (b) API con 2M requests/día estables, (c) resize de imágenes al subirse, (d) app de chat con websockets.
2. ¿Por qué tu función Lambda no puede guardar la sesión del usuario en una variable global?
3. ¿Qué dos diferencias hay entre `res.json(tasks)` (Express) y el `return {statusCode, body}` de Lambda?

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Soluciones

1. Conviene: (a) eventos irregulares — pagas ~nada entre webhooks;
   (c) procesamiento por evento — el caso clásico. Proceso fijo: (b) tráfico alto y estable — el costo por ejecución supera al servidor;
   (d) websockets necesitan conexiones persistentes — las funciones mueren al responder.
2. Porque cada ejecución puede correr en una *instancia distinta* (o en la misma reciclada después) — la variable global no es confiable: existe solo mientras esa ejecución viva y no la ven las demás. La sesión va en un store externo (Redis, DB, o JWT en el token).
3. (a) Lambda devuelve un objeto que la *plataforma* traduce a HTTP — no escribes en un socket; el body debe ser string serializado tú mismo. (b) No hay `listen()` ni puerto: la plataforma decide cuándo invocarte — tu código es un *handler*, no un servidor.

49.10. Mini reto

Un equipo migra su API completa a Lambda "para ahorrar". Su tráfico: 3M requests/día constantes, cada request toca Postgres. Dos semanas después la factura subió y los usuarios se quejan de lentitud intermitente. ¿Qué salió mal? (Pista: tres cosas.)

Soluciones

1. **El cálculo era al revés:** a 90M requests/mes constantes, el costo por ejecución + API Gateway supera a un contenedor/VM fija — serverless ahorra con tráfico *irregular*, no con carga sostenida.
2. **Cold starts:** los usuarios sentían los despertares — funciones con muchas dependencias tardan en arrancar; con tráfico constante igual aparecen cuando AWS recicla instancias o escala.
3. **Conexiones a Postgres:** miles de funciones abriendo conexiones nuevas agotaron la base (Postgres tolera ~cientos, no miles) — necesitaban un pooler (RDS Proxy) que olvidaron o cuya latencia extra empeoró todo.

Moraleja: serverless es una *herramienta con un nicho*, no un upgrade.

49.11. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

49.12. Lo que deberías saber hacer ahora

- ☐ Explicar el modelo serverless y su precio real (cold starts, límites, sin estado)
- ☐ Decidir cuándo conviene serverless vs un proceso siempre vivo
- ☐ Escribir o adaptar un endpoint como función en tu stack
- ☐ Nombrar las 5 cimentaciones de deployment y su criterio de elección

Parte XIII

Proyecto final · Nexus

50 36. Diseñar antes de construir: la arquitectura de Nexus

Te contratan para el backend de un Figma-lite. ¿Por dónde empiezas?

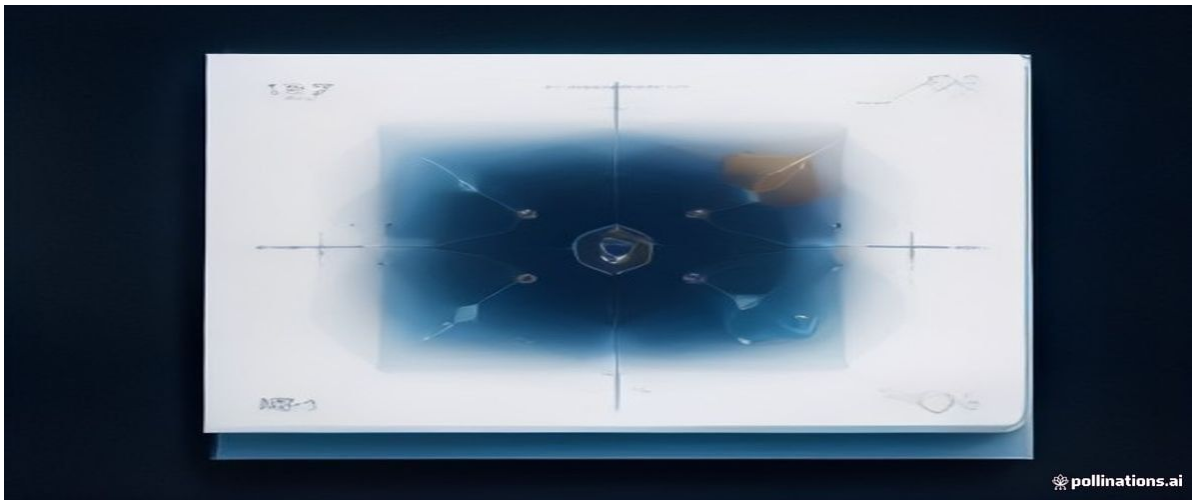


Figura 50.1: Parte 9 — Proyecto final: Nexus

> **En una frase**
>
> Nexus: un workspace colaborativo donde equipos editan documentos en tiempo real, con historial completo — un Figma-lite. Antes de la primera línea de código: **el dominio dibujado, el contrato escrito, y las decisiones documentadas**. Este capítulo es diseño en papel — el código llega en el 37.

50.1. El problema

Te llega el brief: "workspace por equipos; proyectos con documentos que varias personas editan a la vez; historial para volver atrás; miembros con roles". Es exactamente el tipo de requerimiento difuso que recibe un backend real — nadie te da el esquema, te dan *una intención*. El error de junior: empezar

por `POST /login`. El error caro: descubrir en la semana 3 que "editar a la vez" necesitaba un modelo distinto desde el día 1.

> Explicación de: esquema (schema)
>
> El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

> Explicación de: autenticación
>
> La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

50.2. Cómo lo resuelve un equipo

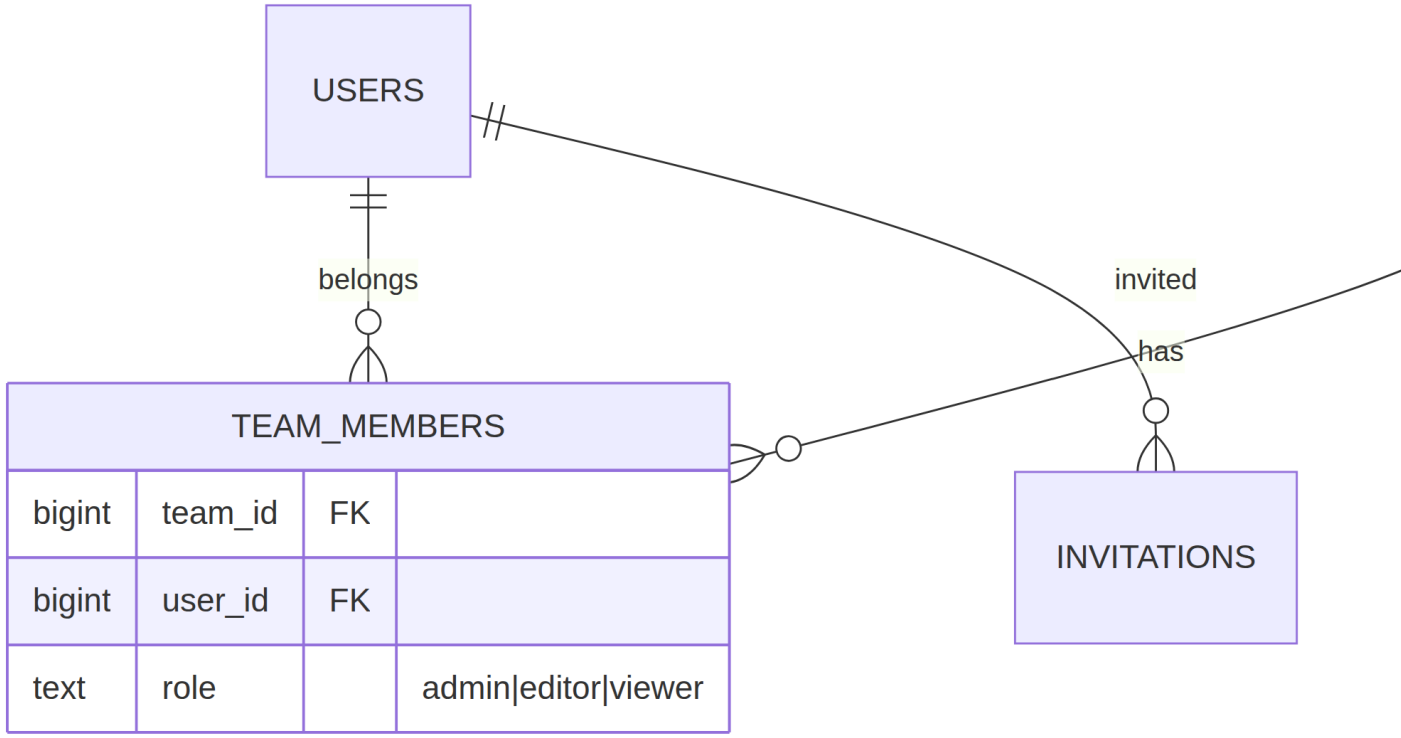
Un equipo serio responde al brief con tres artefactos antes de codificar:

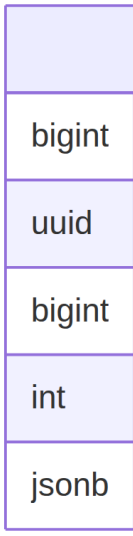
1. **El modelo** (diagrama ER): qué *cosas* existen y cómo se relacionan — esto responde "¿qué es Nexus?".
2. **El contrato** (rutas + schemas + códigos de error): cómo se habla con él — esto responde "¿cómo se usa?".
3. **Los ADRs** (media página por decisión): por qué se eligió así — esto responde "¿por qué así y no de otra forma?" para el humano del futuro que pregunte.

50.3. Conceptos nuevos

- U Del requerimiento al modelo: entidades, relaciones, ownership — el ER completo
- U ADR mínimo: documentar por qué capas, por qué esas fronteras
- U Diseñar el contrato primero: rutas, schemas, códigos de error — OpenAPI como fuente de verdad
- D `public_id` vs UUID interno y otras decisiones que parecen menores y no lo son

50.4. El modelo de Nexus





Lee las decisiones del dibujo:

- **Los roles viven en la membresía**, no en el usuario — eres `admin` *del equipo A* y `viewer` *del B* (cap. 17).
- **`documents.current_revision` es un puntero** — como `HEAD` en git: el documento sabe "estoy en la rev 12" y la historia vive aparte en `revisions`.
- **Cada revisión es un snapshot completo** — restore = leer una fila, no re-ejecutar la historia (cap. 37).

50.5. El contrato primero

```
POST   /auth/register | /auth/login | /auth/refresh | /auth/logout
GET    /teams                      → mis teams
POST   /teams/{id}/invitations     → admin only
POST   /invitations/{token}/accept
GET    /projects/{id}/documents
POST   /documents                  → editor+
GET    /documents/{id}             → member: content + current_revision
POST   /documents/{id}/revisions   → save new revision (the write hot path)
GET    /documents/{id}/revisions   → history list
POST   /documents/{id}/restore/{rev} → move the pointer back
WS     /documents/{id}/live        → presence + edits broadcast
```

Más los errores estables (cap. 25): `UNAUTHORIZED`, `FORBIDDEN`, `DOC_NOT_FOUND`, `NOT_A_MEMBER`, `ROLE_TOO_LOW`, `INVITATION_EXPIRED`, `VALIDATION`, `CONFLICT`, `INTERNAL`. El frontend puede programarse completo contra esta tabla *antes* de que exista el backend.

50.6. Los ADRs — tres decisiones, media página cada una

```
# ADR-01: Monolito modular por capas
Contexto: equipo chico, dominio cohesionado, realtime+HTTP en un proceso.
Decisión: monolito con capas (rutas/servicios/storage) del cap. 7.
Alternativas: microservicios (overhead de red sin el problema de escala
de equipos). Consecuencias: si una pieza crece 10× distinta, se extrae -
las fronteras ya existen.

# ADR-02: Snapshot + puntero para versionado (no solo diffs)
Contexto: restore debe ser O(1); el historial es append-only.
Decisión: snapshot completo por revisión + current_revision como puntero.
Alternativas: solo diffs (restore = replay N eventos), CRDT (proyecto
aparte). Consecuencias: +espacio en disco; -complejidad de restore.

# ADR-03: Last-Writer-Wins en edición concurrente
```

```
Contexto: dos cursores en el mismo documento.
Decisión: LWW con rev + broadcast - el merge fino por carácter es CRDT,
otro nivel de complejidad. Alternativas: OT/CRDT. Consecuencias: ediciones
en el mismo párrafo se pisan - aceptado y documentado.
```

Prompt listo para tu AI

```
Actúa como tech lead: diseña la arquitectura de Nexus, un workspace
colaborativo (teams → projects → documents con versionado y edición
concurrente). Entregables: diagrama ER con justificación de cada
relación y ON DELETE, contrato REST+WS completo (rutas, schemas,
códigos de error estables), modelo de membresías con roles por equipo,
y 3 ADRs de media página (estructura del proyecto, modelo de
versionado, resolución de conflictos) con alternativas descartadas y
consecuencias aceptadas. No escribas código - el diseño primero.
```

50.7. ¿Por qué diseñar en papel primero?

Lo único que necesitas llevarte: cada línea del ER y cada ADR es un problema resuelto *cuando costaba una oración* — en código cuesta un refactor, en prod una migración. El diseño no es burocracia: es el lugar barato de equivocarse.

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Detalle: las decisiones que parecen menores

- **`public_id` por recurso**: `documents/7f3a…` en URLs — el id secuencial expone volumen y permite enumerar (cap. 8).
- **`rev` entero por documento** (no UUID global): la historia es ordenada *por documento* — `rev=14` se compara, un UUID no.
- **`team_members.role` con CHECK** (`admin|editor|viewer`): el rol inválido lo rechaza la BD, no solo la validación.
- **Invitaciones con `token` + `expires_at` + `used_at`**: otro flujo con estado — el token opaco

viaja por email, el servidor verifica uno-uso-expira (mismo patrón que refresh tokens del cap. 18).

50.8. Errores comunes

Error	Por qué pasa	Fix
Empezar por el código	"El diseño sale solo"	ER + contrato antes — el dibujo se cambia en segundos
Roles en el usuario	"Es admin de la app"	El rol es *de la membresía* — contexto por equipo
Diseñar para el futuro imaginario	"Y si necesitamos microservicios"	Diseña para el brief de hoy + ADRs que documenten los bordes
Contrato después del código	"Ya veremos qué devuelve"	El contrato es la fuente de verdad — frontend y backend se construyen en paralelo contra él
Decisiones sin documentar	"Queda en la PR"	ADR de media página — el por qué que el código no dice

50.9. Buenas prácticas

- **El brief se reescribe como modelo**: cada sustantivo del requerimiento es un candidato a entidad; cada verbo, una relación o una ruta.
- **Decidir lo reversible rápido y lo irreversible lento** (Ap. K, decisión framework): el modelo de versionado es puerta de una vía — merece la página de análisis; el nombre de la ruta no.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

- **El contrato incluye los errores**: una ruta sin sus 4xx definidos está a medio diseñar.
- **El diseño se presenta**: un RFC de una página que el equipo puede comentar — las objeciones en papel son gratis.

50.10. Ejercicio

1. El brief dice "los documentos pueden estar en carpetas". Agrega la entidad al ER: ¿carpeta como tabla, como columna, o como tag? Defiende.

2. Escribe la ruta y el contrato de "quitar a un miembro del equipo": método, path, quién puede, qué devuelve si el miembro no existe, si el caller no es admin, si es el último admin.
3. ¿Por qué `REVISIONS` no tiene `public_id` en la tabla de arriba?

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: método

Un **método** es una función que vive dentro de un objeto/clase: `task.save()` — `save` es un método de `task`. La diferencia con una función suelta: el método conoce al objeto que lo contiene (`this`/ `self`).

50.11. Mini reto

Presentar el diseño como si fuera un RFC de equipo — y defender cada decisión.

Soluciones

Ejercicio.

1. Tabla `folders` solo si las carpetas tienen comportamiento propio (permisos, anidamiento). Si son solo agrupación visual: columna `folder`/`path` en `documents` o tabla `folders(id, project_id, name)` liviana. La respuesta seria pregunta primero: "¿la carpeta *hace* algo o solo *agrupa*?" — el modelo sigue al comportamiento.
2. `DELETE /teams/{id}/members/{user_id}` → admin only. Miembro inexistente → 404 (`MEMBER_NOT_FOUND`). Caller no-admin → 403. Último admin → 409 `LAST_ADMIN` — regla de negocio que se descubre al diseñar el contrato, no al crashearla.
3. Las revisiones no son recursos de primera clase expuestos por URL propia — se acceden *a través* del documento (`/documents/{id}/ revisions/14`). El `public_id` existe para lo que viaja solo por URL; la revisión viaja dentro del documento.

Mini reto. El RFC: contexto (el brief) → modelo (ER) → contrato → 3 ADRs → "lo que NO construimos" (CRDT, microservicios, comentarios en documento) → preguntas abiertas. Defender cada decisión = decir qué alternativa se descartó y qué precio se aceptó — eso es lo que un RFC real necesita, y lo que el Ap. K entrena.

Repaso de entrevista: las preguntas de este tema viven en el Apéndice H (diseño y arquitectura).

50.12. Vocabulario técnico del capítulo

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: concurrencia vs paralelismo

Concurrencia = manejar muchas cosas en progreso (atender 1000 requests intercalando). **Paralelismo** = ejecutar varias a la vez en varios núcleos. Un camarero con 10 mesas es concurrente; 10 camareros son paralelos.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: monolito

Un **monolito** es una app donde todo corre en un solo proceso: rutas, lógica, BD access. No es insulto — es el default correcto mientras el equipo sea chico. Se puede modular por dentro sin dividir el deploy.

Explicación de: microservicios

Microservicios = dividir la app en servicios independientes que se comunican por red. Ganas despliegue separado; pagas con distribución (transacciones entre servicios, latencia, observabilidad). Equipo de 5 no necesita esto.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: foreign key / clave foránea

Una **foreign key** es un puntero verificado: `tasks.project_id` apunta a `projects.id` y la BD *garantiza* que el proyecto existe. Es la diferencia entre "el dato dice 5" y "el dato apunta a algo real".

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

50.13. Lo que deberías saber hacer ahora

- ☐ Convertir un brief difuso en un diagrama ER con ownership claro
- ☐ Escribir el contrato completo (rutas + errores) antes del código
- ☐ Documentar decisiones en ADRs de media página con alternativas
- ☐ Detectar reglas de negocio al diseñar (LAST_ADMIN, uno-uso, expiración)
- ☐ Separar decisiones reversibles de puertas de una vía

51 37. El corazón: documentos versionados

Un «git interno»: snapshot + diff + puntero

> **En una frase**
>
> Guardar cada tecleo como un "cambio" y reconstruir sumándolos es frágil (la red cae a la mitad y el documento queda corrupto) y el undo en el navegador muere al cerrar la pestaña. Nexus guarda la historia como **git por dentro: cada guardado es un snapshot completo, el documento tiene un puntero (`current_revision`), y undo/redo/restore son mover ese puntero — atómicamente.**

51.1. El problema

El modelo "obvio": `documents.content` se sobrescribe + una tabla `edits` con los cambios. Falla dos veces: (1) el save parcial — la red cae entre el INSERT del edit y el UPDATE del content → el documento queda a mitad, inconsistente; (2) el restore — "volver a hace 40 cambios" = re-aplicar 40 diffs esperando que ninguno esté corrupto. Necesitamos que guardar sea **una operación atómica** y restaurar sea **una lectura**, no una replay.

51.2. Cómo lo resuelve un equipo

Un equipo serio copia el modelo que ya ganó — **git interno**:

- `revisions(document_id, rev, snapshot, author_id, created_at)` — la historia es *append-only* (cada guardado una fila nueva, jamás un UPDATE).
- `documents.current_revision` — el puntero `HEAD`: qué versión es la actual.
- Guardar = una transacción: `INSERT revision(rev = current+1)` + `UPDATE documents SET current_revision = rev` — todo o nada (cap. 13), con `SELECT ... FOR UPDATE` para que dos pestañas del mismo usuario no escriban el mismo `rev`.
- Undo/redo/restore = `UPDATE documents SET current_revision = $rev` — mover el puntero, O(1), sin tocar la historia. Editar tras undo = truncar el "futuro" — exactamente `git commit` tras un reset.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

51.3. Conceptos nuevos

- D Snapshot + diff + puntero: `documents`, `revisions`, `current_revision`
- D Guardado atómico: transacciones + `SELECT FOR UPDATE` contra doble-submit y pestañas
- U Undo/redo/restore como operaciones del servidor: mover el puntero, truncar el futuro
- D `JSONB` para datos que evolucionan: cuándo es decisión y cuándo pereza
- U El patrón de Google Docs / Figma / AutoCAD web — la analogía con git es exacta

51.4. La explicación visual

```
revisions (append-only)          documents

 doc 7 · rev 1 · {A}            id 7
 doc 7 · rev 2 · {A,B}          current_revision: 2     → rev 2
 doc 7 · rev 3 · {A,B,C}        content: {A,B,C} (cache)

  undo → puntero a 1 · editar tras undo → DELETE rev > 1 (truncate future)
  - igual que `git reset` + commit nuevo -
```

El snapshot es el *estado completo* en ese rev: restaurar es leer una fila. Los diffs existen como *optimización de transporte* (qué mandar por WebSocket), nunca como fuente de verdad.

Explicación de: WebSocket

WebSocket es una conexión que queda *viva*: en vez de preguntar- responder-cerrar como HTTP, ambos pueden hablar cuando quieran. Es como el servidor puede *empujar* — por eso sirve para chat y colaboración en vivo.

51.5. Implementación

El guardado atómico — el corazón de Nexus en los tres stacks:

51.5.1. TypeScript

```typescript
async function saveRevision(userId: number, docPublicId: string, content: unknown) {
  return await pool.tx(async (t) => {
    const { rows } = await t.query(
      `SELECT d.id, d.current_revision FROM documents d
       JOIN projects p ON p.id = d.project_id
       JOIN team_members m ON m.team_id = p.team_id
       WHERE d.public_id = $1 AND m.user_id = $2 AND m.role IN ('admin','editor')
       FOR UPDATE OF d`,                          // lock: two tabs can't race the same rev
      [docPublicId, userId]);
    if (!rows[0]) throw new NotFoundError("document");
    const rev = rows[0].current_revision + 1;
    await t.query(
      `DELETE FROM revisions WHERE document_id = $1 AND rev > $2`,   // truncate future
      [rows[0].id, rows[0].current_revision]);
    await t.query(
      `INSERT INTO revisions (document_id, rev, snapshot, author_id) VALUES ($1,$2,$3,$4)`,
      [rows[0].id, rev, JSON.stringify(content), userId]);
    await t.query(
      `UPDATE documents SET current_revision = $1, content = $2 WHERE id = $3`,
      [rev, JSON.stringify(content), rows[0].id]);
    return { rev };                                  // commit: all or nothing
  });
}
```

51.5.2. Python

```python
def save_revision(user_id: int, doc_public_id: str, content: dict) -> int:
    with pool.connection() as conn:
        with conn.transaction():                     # BEGIN
            row = conn.execute("""
                SELECT d.id, d.current_revision FROM documents d
                JOIN projects p ON p.id = d.project_id
                JOIN team_members m ON m.team_id = p.team_id
                WHERE d.public_id = %s AND m.user_id = %s
                  AND m.role IN ('admin','editor')
                FOR UPDATE OF d""",                   # lock against racing tabs
                (doc_public_id, user_id)).fetchone()
            if not row: raise NotFoundError("document")
            rev = row["current_revision"] + 1
            conn.execute(
                "DELETE FROM revisions WHERE document_id=%s AND rev>%s",
                (row["id"], row["current_revision"]))  # truncate future
            conn.execute(
                "INSERT INTO revisions (document_id,rev,snapshot,author_id)"
                " VALUES (%s,%s,%s,%s)",
                (row["id"], rev, Jsonb(content), user_id))
```

```
            conn.execute(
                "UPDATE documents SET current_revision=%s, content=%s WHERE id=%s",
                (rev, Jsonb(content), row["id"]))
            return rev                                  # commit on exit
```

51.5.3. Go

```
func saveRevision(ctx context.Context, userID int64, docPublicID string, content []byte) (int, error)
    tx, err := pool.Begin(ctx)
    if err != nil { return 0, err }
    defer tx.Rollback(ctx)                              // no-op after Commit

    var docID int64
    var cur int
    err = tx.QueryRow(ctx, `
        SELECT d.id, d.current_revision FROM documents d
        JOIN projects p ON p.id = d.project_id
        JOIN team_members m ON m.team_id = p.team_id
        WHERE d.public_id = $1 AND m.user_id = $2
          AND m.role IN ('admin','editor')
        FOR UPDATE OF d`, docPublicID, userID).Scan(&docID, &cur)
    if errors.Is(err, pgx.ErrNoRows) { return 0, ErrNotFound }
    if err != nil { return 0, err }

    if _, err = tx.Exec(ctx,
        "DELETE FROM revisions WHERE document_id=$1 AND rev>$2", docID, cur); err != nil {
        return 0, err                                   // truncate future
    }
    rev := cur + 1
    if _, err = tx.Exec(ctx,
        "INSERT INTO revisions (document_id,rev,snapshot,author_id) VALUES ($1,$2,$3,$4)",
        docID, rev, content, userID); err != nil { return 0, err }
    if _, err = tx.Exec(ctx,
        "UPDATE documents SET current_revision=$1, content=$2 WHERE id=$3",
        rev, content, docID); err != nil { return 0, err }
    return rev, tx.Commit(ctx)                          // all or nothing
}
```

Y restaurar — la operación más elegante del sistema:

```
-- restore: move the pointer. O(1). No replay, no corruption possible.
UPDATE documents d SET current_revision = $2
WHERE d.public_id = $1
  AND EXISTS (SELECT 1 FROM revisions r
              WHERE r.document_id = d.id AND r.rev = $2);
-- then content = SELECT snapshot FROM revisions WHERE rev = $2
```

Prompt listo para tu AI

```
Implementa el servicio de versionado de documentos de Nexus en
[Express+pg/FastAPI+psycopg/Go+pgx]. Modelo: revisions(document_id,
rev, snapshot JSONB, author_id) append-only + documents.current_revision
como puntero. Requisitos: saveRevision en una transacción con SELECT
FOR UPDATE del documento (dos pestañas no pueden escribir el mismo
rev), DELETE de revisiones futuras al editar tras undo, UPDATE del
puntero + cache de contenido en la misma tx, restore como UPDATE del
puntero (O(1), sin replay), permiso editor+ verificado EN la query, y
tests: save concurrente → revs secuenciales sin duplicados, editar tras
undo trunca el futuro, restore mueve el puntero sin tocar la historia.
```

51.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: la transacción hace el trabajo — *lock + truncate + insert + move pointer* es UN acto atómico en los tres stacks. Cambia el mecanismo (`pool.tx`/`conn.transaction`/`tx`), jamás el diseño.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Detalle: por qué snapshot y no solo diffs

Modelo	Restore	Espacio	Corrupción posible
Solo diffs (event sourcing)	Replay N eventos — O(N) y cada diff debe estar sano	Mínimo	Sí — un diff roto rompe todo lo posterior
Solo snapshot actual	No hay historia	Mínimo	El undo no existe
Snapshot por rev + puntero	Leer una fila — **O(1)**	Más disco (barato)	No — cada rev es autónoma

`JSONB` para el snapshot es *decisión*, no pereza: el contenido del documento tiene forma variable y evoluciona (hoy capas, mañana comentarios) — pero **los metadatos del versionado son relacionales puros** (rev, author, puntero). JSONB para el contenido; relacional para el control — la misma respuesta de cap-08 y Ap. K decisión 11.

51.7. Errores comunes

Error	Por qué pasa	Fix
Save como dos operaciones sueltas	INSERT luego UPDATE fuera de tx	Una transacción — la caída a mitad no existe
`rev` calculado en el cliente	"El frontend manda rev 15"	`rev = current_revision + 1` calculado *en el servidor* bajo lock
Editar tras undo sin truncar	El "futuro" viejo convive con el nuevo	`DELETE rev > current` antes del insert — una sola línea temporal
Diffs como fuente de verdad	"Para ahorrar espacio"	Snapshot completo — los diffs son transporte, no historia
Save sin verificar rol en la query	`if user.role == 'editor'` en memoria	El permiso va EN el SQL — la query devuelve 0 filas → 404

51.8. Buenas prácticas

- **La historia es append-only**: nunca UPDATE a una revisión — la historia que se puede editar no es historia, es rumor.
- **`SELECT FOR UPDATE` solo del documento** (`FOR UPDATE OF d`) — bloquear toda la tabla es la forma lenta de hacer lo mismo.
- **El puntero y el cache viven juntos**: `current_revision` + `content` en el mismo UPDATE — el doc siempre muestra el estado del rev apuntado.
- **`revisions` particionable**: cuando crezca, se pagina por rev y se archiva — el diseño append-only ya lo permite sin cambios.

51.9. Ejercicio

1. Dos pestañas del mismo usuario guardan a la vez: sin `FOR UPDATE`, ¿qué `rev` escriben ambas y qué queda en la tabla? ¿Por qué el lock lo evita?
2. ¿Por qué `DELETE FROM revisions WHERE rev > current` y no un flag `is_future`? ¿Qué rompe soft-delete aquí?
3. El cliente pide "dame el diff entre rev 12 y 14". Tienes snapshots completos. ¿Cómo respondes sin un sistema de diffs?

51.10. Mini reto

Explicar por qué snapshot completo y no solo diffs (O(1) restore vs replay).

Soluciones

Ejercicio.

1. Ambas leen `current_revision = 14` y ambas escriben `rev 15` — dos filas con el mismo rev (o la segunda choca con un UNIQUE y el usuario ve un error fantasma). Con `FOR UPDATE`: la segunda espera a que la primera haga commit, lee `15`, escribe `16` — secuencia real, sin carrera.
2. El flag deja el "futuro" en la tabla: queries que deben recordar filtrarlo, historia hinchada con ramas muertas. Git tampoco soft- deletea el futuro: lo elimina. `DELETE` es honesto — la línea temporal divergente deja de existir.
3. Computas el diff *al servir*: `jsonb` permite comparar campos, o devuelves ambos snapshots al cliente para que los compare (el diff es *presentación*, no almacenamiento). Guardar el estado, derivar el delta — nunca al revés.

Mini reto. Restore con solo-diffs = leer rev objetivo hacia atrás aplicando inversas, o desde un checkpoint + replay — O(distancia), frágil si cualquier diff se corrompió, y más código. Restore con snapshot = `SELECT snapshot WHERE rev = $1` — **O(1), indestructible, una línea de SQL**. El espacio extra es el precio; el disco es lo más barato del sistema.

Repaso de entrevista: las preguntas de este tema viven en el Apéndice H (bases de datos).

51.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: concurrencia vs paralelismo

Concurrencia = manejar muchas cosas en progreso (atender 1000 requests intercalando). **Paralelismo** = ejecutar varias a la vez en varios núcleos. Un camarero con 10 mesas es concurrente; 10 camareros son paralelos.

Explicación de: Git

Git es el historial de tu proyecto: cada *commit* es una foto del código con mensaje. Permite volver atrás, trabajar en ramas paralelas y fusionar el trabajo de varias personas sin pisarse.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo

al reiniciar.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: JOIN

Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: caché / Redis

Un **caché** es una copia rápida de algo costoso de obtener: el resultado de una query pesada, la sesión. Redis es el caché estándar. La regla de oro: cachear es fácil, *invalidar* (saber cuándo el caché ya no vale) es lo difícil.

51.12. Lo que deberías saber hacer ahora

- ☐ Modelar versionado como snapshot + puntero (el "git interno")
- ☐ Guardar en una transacción con `SELECT FOR UPDATE` anti-carreras
- ☐ Implementar undo/redo/restore como movimiento del puntero
- ☐ Truncar el futuro al editar tras undo — y explicar por qué
- ☐ Elegir JSONB para el contenido y relacional para el control, con criterio

52 38. Colaboración: equipos, invitaciones y tiempo real

«Que todos vean qué está pasando» exige realtime + roles

> **En una frase**
>
> Nexus deja de ser un CRUD bonito: ahora *varias personas* editan el mismo documento y **se ven hacerlo**. Eso exige tres piezas que el libro ya preparó: membresías con rol *en la relación*, invitaciones como tokens de un solo uso, y un canal WebSocket autenticado donde el servidor — por primera vez — *empuja* datos sin que le pregunten.

52.1. El problema

El brief: "Ana invita a Beto al equipo; ambos abren el mismo documento; cuando Ana mueve un elemento, Beto lo ve moverse". Tres problemas en uno: (1) ¿cómo entra Beto — un link mágico que expira, no una contraseña que Ana le dicta?; (2) ¿cómo viaja el cambio de Ana a Beto — HTTP es preguntar, esto es *empujar*?; (3) ¿qué pasa cuando ambos editan lo mismo a la vez?

52.2. Cómo lo resuelve un equipo

Un equipo serio separa las tres preguntas:

1. **Membresía**: `team_members(team_id, user_id, role)` — el rol vive en la relación (cap. 17), no en el usuario.
2. **Invitación**: `invitations(token, team_id, email, role, expires_at, used_at)` — un opaco de un solo uso que expira; aceptar crea la membresía. Otro flujo con estado — el mismo patrón del refresh token (cap. 18).
3. **Tiempo real**: WebSocket autenticado con la misma cookie; el servidor mantiene `documento → conexiones` y hace broadcast. El protocolo de mensajes es un contrato como cualquier ruta REST.
4. **Conflictos**: last-writer-wins — documentado en el ADR-03 (cap. 36); CRDT queda fuera a propósito.

Explicación de: cookie

Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

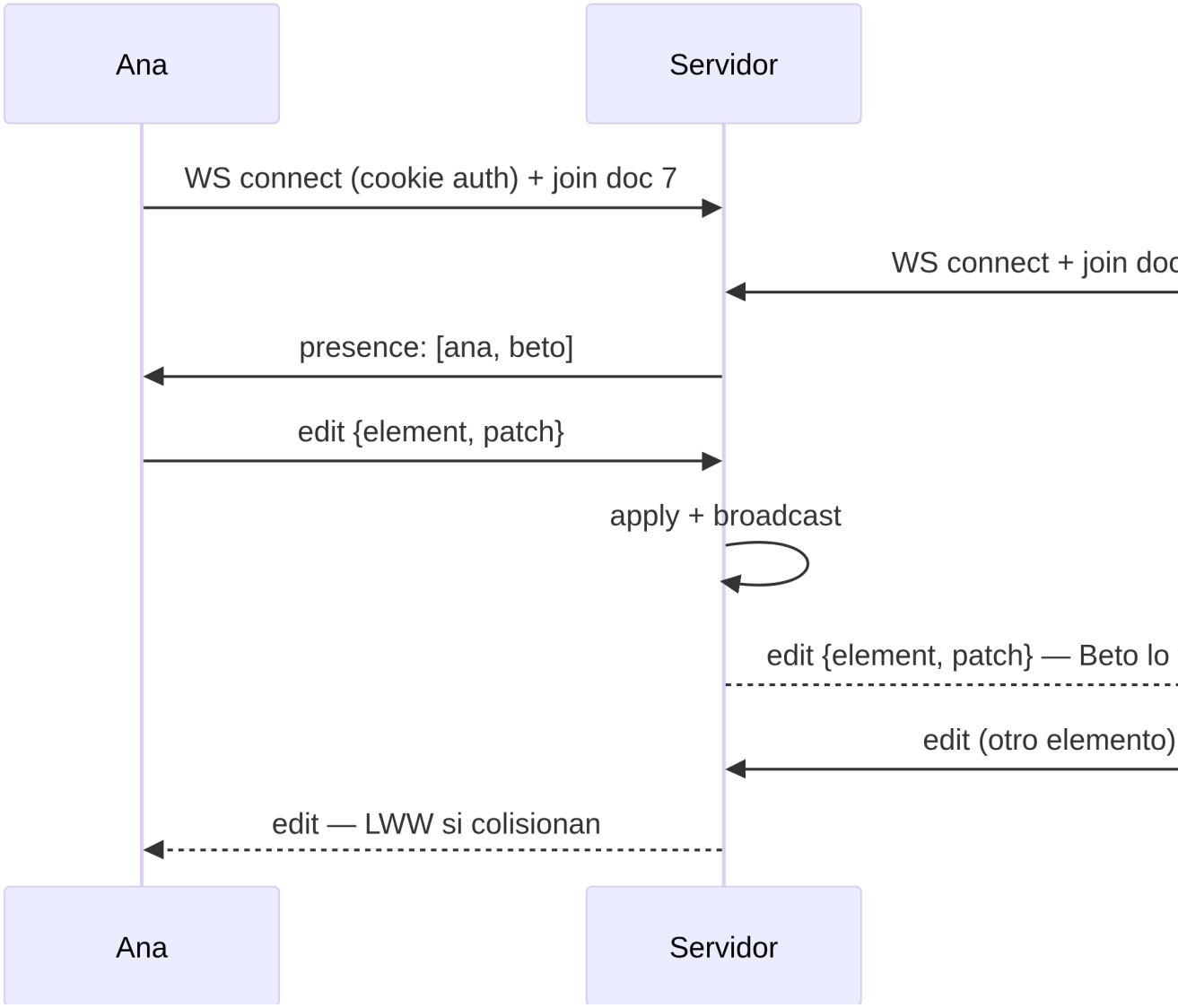
Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

52.3. Conceptos nuevos

- D Modelo de membresías: `teams`, `members` — roles en la relación
- U Invitaciones: tokens de un solo uso, expiración, aceptación — otro flujo con estado
- U WebSockets: endpoint autenticado, presencia, broadcast — qué mantiene el servidor ahora
- TS `ws` · Py FastAPI WebSocket · Go `coder/websocket` — conexiones vivas por runtime
- U Conflictos en vivo: last-writer-wins pragmático — cuándo alcanza y cuándo CRDT

52.4. La explicación visual



El servidor ahora tiene *estado de conexiones*: un mapa `documento → [sockets]` en memoria. Lo que el REST no tenía que recordar, el WS sí — por eso escalar WS a múltiples instancias es otra conversación (pub/sub entre ellas; para Nexus, una instancia basta).

> Explicación de: memoria (RAM)
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

52.5. Implementación

El endpoint WS autenticado + broadcast por documento:

52.5.1. TypeScript

```typescript
import { WebSocketServer } from "ws";

const rooms = new Map<string, Set<WebSocket>>();   // docId → live sockets

wss.on("connection", async (socket, req) => {
  const user = await userFromCookie(req.headers.cookie);   // SAME auth as REST
  const docId = getDocId(req.url);
  if (!user || !(await isMember(user.id, docId))) return socket.close();

  joinRoom(rooms, docId, socket);
  broadcast(docId, { type: "presence", users: roomUsers(rooms, docId) });

  socket.on("message", (data) => {
    const msg = JSON.parse(data.toString());
    if (msg.type === "edit") {
      broadcast(docId, { type: "edit", by: user.id, patch: msg.patch },
                socket);                                    // to the OTHERS
    }
  });
  socket.on("close", () => leaveRoom(rooms, docId, socket));
});
```

52.5.2. Python

```python
rooms: dict[str, set[WebSocket]] = {}            # docId → live sockets

@app.websocket("/documents/{doc_id}/live")
async def live(websocket: WebSocket, doc_id: str):
    user = await user_from_cookie(websocket.cookies)   # SAME auth as REST
    if not user or not await is_member(user.id, doc_id):
        return await websocket.close()
    await websocket.accept()
    rooms.setdefault(doc_id, set()).add(websocket)
    await broadcast(doc_id, {"type": "presence", "users": room_users(doc_id)})
    try:
        async for raw in websocket.iter_text():
            msg = json.loads(raw)
            if msg["type"] == "edit":
                await broadcast(doc_id,
```

```
                    {"type": "edit", "by": user.id, "patch": msg["patch"]},
                    exclude=websocket)                  # to the OTHERS
    finally:
        rooms[doc_id].discard(websocket)
```

52.5.3. Go

```
var rooms = NewRoomMap()   // docID → set of *websocket.Conn (mutex-guarded)

func live(w http.ResponseWriter, r *http.Request) {
    user := userFromCtx(r.Context())             // SAME auth middleware as REST
    docID := r.PathValue("id")
    if user == nil || !isMember(r.Context(), user.ID, docID) {
        w.WriteHeader(403); return
    }
    conn, err := websocket.Accept(w, r, nil)
    if err != nil { return }
    rooms.Join(docID, conn)
    defer rooms.Leave(docID, conn)
    broadcast(docID, Message{Type: "presence", Users: rooms.Users(docID)})
    for {
        var msg Message
        if err := wsjson.Read(r.Context(), conn, &msg); err != nil { return }
        if msg.Type == "edit" {
            broadcastExcept(docID, Message{Type: "edit", By: user.ID, Patch: msg.Patch}, conn)
        }
    }
}
```

Prompt listo para tu AI

```
Implementa el canal realtime de Nexus en [ws/FastAPI/coder-websocket]:
WS /documents/{id}/live autenticado con la misma cookie httpOnly del
REST (rechazar sin sesión o sin membresía), mapa documento→conexiones
thread-safe, mensajes con contrato {type: presence|edit, ...}, broadcast
de edits a las OTRAS conexiones del documento, limpieza al desconectar,
y last-writer-wins documentado. Tests/manual: dos clientes, uno edita
→ el otro recibe en <300ms; desconectar limpia la presencia.
```

52.6. ¿Por qué cada stack lo hace así?

Lo único que necesitas llevarte: el WS es una ruta más — *autenticada igual, autorizada igual* — solo que la conexión queda viva y el servidor guarda el mapa de quién escucha qué documento. Los tres hacen lo mismo: aceptar → join → loop de mensajes → broadcast → limpiar al cerrar.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Detalle: las decisiones de realtime

Decisión	Nexus elige	La alternativa y su precio
Auth del WS	Cookie httpOnly en el upgrade — *misma* auth	Token en query param: queda en logs/URLs
Conflictos	LWW (ADR-03)	CRDT/OT: merge por carácter — semanas de trabajo
Presencia	Mapa en memoria del proceso	Redis pub/sub: necesario solo con N instancias
El servidor empuja	WebSocket	SSE (solo servidor→cliente) o polling (latencia y carga)

LWW honesto: si Ana y Beto tocan el *mismo* elemento en el mismo instante, gana el último en llegar — el documento queda consistente (mismo estado en todos) aunque un cambio se pierda. Aceptable porque: las ediciones suelen ser en elementos *distintos*, el historial del cap. 37 lo recupera todo, y la alternativa es un proyecto de meses.

52.7. Errores comunes

Error	Por qué pasa	Fix
WS sin autenticación/autorización	"Es un socket, no una ruta"	Mismo middleware: cookie → user → isMember, antes del accept
Broadcast a *todos* incluido el emisor	`rooms.forEach(send)`	El emisor ya aplicó su cambio — eco = doble render
Mapa de rooms sin mutex (Go) / sin limpieza	Sockets muertos en el mapa	`on close → leave`; en Go el mapa va con sync.Mutex
Confiar el mensaje del cliente	`msg.user_id` declarado por el cliente	El `by` lo pone el servidor — el cliente solo manda el patch
Asumir que WS escala como HTTP	"Levanto 3 réplicas"	Las conexiones viven en UNA instancia — pub/sub entre ellas o sticky sessions

52.8. Buenas prácticas

- **El protocolo WS es contrato**: `{type, ...}` con tipos cerrados (presence, edit, cursor, error) — validado como cualquier body REST.
- **Presencia derivada del mapa**, no de mensajes "estoy aquí" del cliente — el servidor sabe quién está porque sabe quién está *conectado*.
- **El estado efímero en memoria, el durable en BD**: presencia = memoria (muere con el proceso, se reconstruye al reconectar); ediciones finales = revisions del cap. 37.

> Explicación de: proceso
>
> Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

- **Heartbeat/ping**: conexiones zombie (wifi que murió sin close) se purgan con ping periódico + timeout — el mapa no miente para siempre.

52.9. Ejercicio

1. ¿Por qué el `by` del mensaje edit lo pone el servidor y no viene en el mensaje del cliente?
2. Beto abre dos pestañas del mismo documento. ¿Cuántas entradas tiene en `rooms[doc]`? ¿Cómo aparece en presencia?
3. El equipo quiere "que Ana vea el cursor de Beto moviéndose". ¿Qué cambia respecto al broadcast de edits — y qué NO debe guardar el servidor de ese tráfico?

52.10. Mini reto

Dos navegadores deben verse cambios en <300ms: diseñar cómo verificarlo.

> Soluciones
>
> **Ejercicio.**
>
> 1. Porque el cliente miente gratis: `{"by": 42}` declarado por Beto es un claim, no un hecho. El servidor sabe *quién* es por la cookie del upgrade — el `by` se deriva de la autenticación, no del payload. Misma regla que el `user_id` del body (cap. 16).
> 2. Dos entradas (dos sockets) — la presencia por *usuario* deduplica: Beto aparece una vez aunque tenga N conexiones. El mapa guarda conexiones; la presencia muestra usuarios — dos vistas del mismo estado.
> 3. Los cursores son tráfico *efímero y de alta frecuencia*: broadcast sin persistir, sin rate-limit de mensaje pero con throttle en el cliente (~30/seg), y tipo `cursor` distinto de `edit` — el servidor solo reenvía, jamás guarda posiciones de cursor en BD (ruido puro).
>
> **Mini reto.** La verificación honesta: dos clientes WS reales (o un test con dos conexiones),

timestamp en el mensaje `edit` — el receptor mide `Date.now() - msg.ts`. Local será ~5ms; el número honesto se mide en staging con latencia real. Y el criterio: <300ms es "se siente en vivo" — el humano no distingue 50ms de 200ms en un movimiento.

Repaso de entrevista: las preguntas de este tema viven en el Apéndice I (frontend y tiempo real).

52.11. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: diccionario / map

Un **diccionario** (`dict` en Python, `map` en Go, objeto en JS) guarda parejas clave→valor: `{"ana": 30}`. Es la estructura que más aparece en JSON — un objeto JS *es* un diccionario.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

52.12. Lo que deberías saber hacer ahora

- ☐ Modelar membresías con rol en la relación, no en el usuario
- ☐ Diseñar invitaciones como tokens opacos de un solo uso con expiración
- ☐ Autenticar un WebSocket con la misma auth del REST
- ☐ Mantener un mapa documento→conexiones con broadcast y limpieza
- ☐ Defender last-writer-wins sabiendo qué se pierde y qué no

53 39. Endurecimiento: de funciona a publicable

La checklist final que separa portafolio de producción

> **En una frase**
>
> Nexus funciona en tu máquina: login, documentos, realtime, historial. “Funciona” no es “publicable” — entre ambos hay una **auditoría**: seguridad por endpoint, configuración sin secretos quemados, errores que no filtran, logs que reconstruyen, índices que aguantan datos reales, tests que cubren lo que importa y docs que permiten a otro levantarlo. Este capítulo es esa pasada — aplicada, no teórica.

53.1. El problema

Todo está “listo”… hasta que se lo enseñas a alguien. El README no levanta en una máquina limpia; hay un endpoint que olvidó `requireRole`; `GET /documents` devuelve el `content` entero de 2MB por documento; un EXPLAIN muestra que la lista escanea la tabla completa; y el log de errores está vacío porque “nunca falla”. La distancia entre

> **Explicación de: array / lista / slice**
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

> **Explicación de: return**
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = “aquí está el plato, salgo de la cocina”.

> **Explicación de: log / logging estructurado**
>
> Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

funciona y *publicable* se mide en findings — y se cierra con una auditoría sistemática, no con suerte.

53.2. Cómo lo resuelve un equipo

Un equipo serio audita por **categorías con evidencia**, no por vibes: cada hallazgo es fila de una tabla — *área, endpoint/archivo, riesgo, fix, test de regresión*. Las categorías son exactamente los capítulos de robustez del libro: seguridad (26), configuración (24), errores (25), observabilidad (23), performance (12), tests (27–29), y docs/operación. Nada nuevo que aprender — todo por aplicar.

53.3. Conceptos nuevos

- U Auditoría integral: todo lo de los caps. 23–26 aplicado sobre código real
- D Performance con datos reales: índices faltantes, N+1 escondidos, endpoints que escanean tablas
- U La suite de tests como contrato: cobertura donde importa, no donde impresiona
- Ops Documentación usable: README de setup, `openapi.json` publicado, runbook básico

53.4. La auditoría de Nexus — la tabla viva

Área	Hallazgo típico en Nexus	Control que debe existir	Cap.
Auth	`GET /documents/{id}` sin verificar membresía	`isMember` en la query → 404 a lo ajeno	17
Auth	`/auth/login` sin rate limit	5/min por IP+email → 429	18
Datos	`GET /me` filtra campos internos	Contrato whitelist por salida	5, 16
Roles	`viewer` puede `POST /revisions`	`requireRole('editor')` por ruta	17
Webhooks/WS	Socket sin auth ni isMember	Misma auth que REST en el upgrade	38
Config	`JWT_SECRET` con default	Fail-fast al boot, sin defaults	24
Errores	Un endpoint devuelve stack trace	Handler global único → `{error:{code}}`	25
Logs	Requests sin request_id	Middleware + JSON + nunca secretos	23
Perf	`GET /documents` hace N+1 por revisiones	Una query con agregación	12
Perf	`WHERE project_id` sin índice	Índice por patrón de acceso	12
Tests	Solo camino feliz	401/403/404/422 + fuga de campos	28
Docs	README dice “instala y corre”	Setup reproducible + openapi + runbook	—

53.5. Implementación

Capítulo agnóstico — los dos hallazgos que aparecen *siempre* en la pasada de performance con datos reales:

```
-- Finding 1: the documents list scans + N+1 per document
-- Before: one query per document to count its revisions
SELECT d.public_id, d.title, d.current_revision, d.updated_at,
       COUNT(r.rev) AS revision_count
FROM documents d
LEFT JOIN revisions r ON r.document_id = d.id
WHERE d.project_id = $1
GROUP BY d.id
ORDER BY d.updated_at DESC
LIMIT 21;  -- one query, no N+1

-- Finding 2: the access pattern without an index
CREATE INDEX CONCURRENTLY idx_documents_project_updated
  ON documents (project_id, updated_at DESC);
CREATE INDEX CONCURRENTLY idx_revisions_doc_rev
  ON revisions (document_id, rev DESC);
-- EXPLAIN ANALYZE before and after - the audit is measured, not felt
```

Y el seed para que "con datos reales" no sea una metáfora:

```
-- 100k rows in 10 seconds - the audit needs volume or it lies
INSERT INTO revisions (document_id, rev, snapshot, author_id)
SELECT 1, g, jsonb_build_object('elements', '[]'::jsonb), 1
FROM generate_series(1, 100000) g;
```

Prompt listo para tu AI

```
Audita mi API Nexus [stack] antes de publicarla. Recorre cada endpoint
y para cada uno verifica: auth (¿middleware?), autorización (¿rol/
ownership en la query?), exposición (¿solo campos del contrato?),
errores (¿formato {error:{code}} sin stack?), rate limit si aplica,
queries parametrizadas, índices para su patrón de acceso (EXPLAIN con
100k filas seedeadas), y si su comportamiento tiene test de regresión
(401/403/404/422 + fuga de campos). Salida: tabla de hallazgos
endpoint × riesgo × control × estado, ordenada por severidad, con el
fix y el test de cada uno.
```

53.6. ¿Por qué una auditoría y no "otra feature más"?

Lo único que necesitas llevarte: endurecer no agrega funciones — agrega *confianza verificable*: cada riesgo convertido en control testeado es una 3am que no pasa. La auditoría es la diferencia entre "creo

que está bien” y “puedo demostrar que está bien”.

> **i** Detalle: docs que se usan de verdad
>
> **README de setup honesto** (la prueba: alguien lo levanta sin ti):
>
> ```
> # Nexus API
> ## Setup
> 1. `docker compose up` - postgres + api + migraciones automáticas
> 2. `npm test` - suite completa contra taskflow_test
> 3. API en :3000 - OpenAPI en /docs
> ## Entorno
> Ver `.env.example` - todas las llaves requeridas documentadas.
> ```
>
> **Runbook básico** (la 3am tiene guion): “si la API no responde → /health; si responde pero falla → /health/deep (¿BD?); errores → filtra logs por request_id; lento → EXPLAIN de la query del endpoint”. Cuatro líneas que convierten pánico en procedimiento.
> **openapi.json publicado**: el contrato del cap. 36 generado y servido — el frontend y los testers consultan la fuente de verdad, no el código.

53.7. Errores comunes

Error	Por qué pasa	Fix
Auditar “de memoria”	El control existe en 7 de 9 rutas	Tabla por endpoint — lo que no se lista no se audita
Perf con la BD vacía de dev	“Va rápido en mi máquina”	Seed de 100k filas — el EXPLAIN sin datos miente
Coverage como selfie	“92 %!”	Cobertura donde importa: auth, permisos, errores — Ap. K lo repite
Docs escritos para quien ya sabe	“Obvio, npm install”	README probado por alguien ajeno — si no levanta, el README falla
Findings sin fix asignado	Auditoría como lista de lamentos	Cada hallazgo = fila con fix + test de regresión + dueño

53.8. Buenas prácticas

- **La tabla de auditoría vive en el repo** (`docs/AUDIT.md`) — cada PR que toca un endpoint actualiza su fila; seguridad mergeable.
- **Seed generoso y repetible** — `generate_series`/script: el volumen que revela N+1 y seq scans se crea en un comando.
- **Hallazgo → test**: cada fix sin test de regresión es un fix que vuelve — la auditoría deja tests, no solo notas.

- **Runbook de 5 líneas mínimo**: qué mirar, en qué orden — la observabilidad (cap. 23) convertida en procedimiento.

53.9. Ejercicio

1. De la tabla: ¿por qué "viewer puede POST /revisions" es BOLA-adjacente y no simplemente "un endpoint sin auth"?
2. El EXPLAIN de `GET /documents` muestra `Seq Scan` sobre 100k filas. ¿Qué índice agregas según el patrón de acceso de la query?
3. Ordena por severidad estos hallazgos y defiende el orden: endpoint sin auth / stack trace en 500 / README que no levanta / lista sin paginar.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

53.10. Mini reto

Pasar la checklist del apéndice F a Nexus y documentar cada hallazgo.

Soluciones

Ejercicio.

1. No es "sin auth" — el viewer *está autenticado y es miembro*; el problema es que su **rol** no permite mutar. Es function-level authorization (OWASP #5): la pregunta correcta no es "¿quién eres?" sino "¿qué puedes hacer *aquí*?" — `requireRole('editor')` en la ruta, y el test `viewer → 403` permanente.
2. La query filtra `project_id` y ordena por `updated_at DESC` → índice compuesto `(project_id, updated_at DESC)` — igualdad primero, orden después (cap. 12). El EXPLAIN posterior debe mostrar `Index Scan/Bitmap` — y si no lo muestra, el índice no

cuenta.

3. Orden defendible: (a) **endpoint sin auth** — exposición de datos a cualquiera, el peor; (b) **stack trace en 500** — mapa del código al atacante; (c) **lista sin paginar** — se degrada con datos, DoS accidental; (d) **README roto** — duele pero no expone. El criterio del orden: ¿qué daño hace *al mundo exterior* primero?

Mini reto. El resultado real: la tabla de hallazgos llena — cada fila `endpoint | riesgo | control | estado | test`. Nexus maduro tiene verde en casi todo porque los caps. 14–28 ya construyeron los controles; el ejercicio demuestra *que están activos en cada ruta* — que es lo que "auditoría" significa de verdad.

Repaso de entrevista: las preguntas de este tema viven en el Apéndice I (seguridad y operación).

53.11. Vocabulario técnico del capítulo

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: dirección IP

Una **IP** es la dirección numérica de una máquina en la red: `203.0.113.10`. Los servidores tienen IP pública; dentro de una VPC tienen IPs privadas que internet no ve.

Explicación de: volumen (Docker)

Un **volumen** es almacenamiento que sobrevive al contenedor: el contenedor muere y renace, el volumen (los datos de Postgres) sigue ahí. Sin volumen, `docker compose down` borra tu base de datos.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: puerto

Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: JOIN

Un **JOIN** combina tablas por su relación: “cada tarea con el nombre de su proyecto”. Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

Explicación de: autorización / permisos

La **autorización** responde “¿qué puedes hacer?”: eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

53.12. Lo que deberías saber hacer ahora

- ☐ Auditar una API por categorías con tabla de hallazgos y evidencia
- ☐ Sembrar volumen real y leer EXPLAIN para encontrar N+1 y seq scans
- ☐ Exigir test de regresión por cada control de seguridad
- ☐ Escribir README de setup + openapi publicado + runbook mínimo
- ☐ Convertir “creo que está bien” en “puedo demostrarlo”

54 40. Cierre: cómo piensa un backend developer

Qué stack elegirías para cada problema — la respuesta honesta

> **En una frase**
>
> Empezaste con `let tasks = []` y terminaste con un workspace colaborativo versionado, en tiempo real, auditado y desplegable. Este capítulo no agrega nada: **ordena lo que ya sabes** — el mapa completo del libro, la tabla final de qué pertenece al lenguaje vs al framework vs a la industria, y la respuesta honesta a "¿Node, Python o Go?".

54.1. El mapa completo

Todo el libro fue una sola pregunta reformulada: *"¿qué necesita el siguiente problema?"*

```
Fundamentos (0-0.5)     → terminal, HTTP, JSON - las piezas que todo usa
Frontend (fe-01..07)    → el cliente que consumes y que ahora también entiendes
API (1-7)               → rutas, contratos, validación, errores, capas
Datos (8-13)            → diseño, SQL, drivers, migraciones, índices, tx
Identidad (14-18)       → hashing, JWT, middleware, roles, sesiones vivas
Integraciones (19-22)   → salir, concurrencia, jobs, recibir webhooks
Robustez (23-26)        → logs, config, errores a escala, auditoría
Testing (27-29)         → pirámide, BD real, mocks en la frontera
Docker→Prod (30-35)     → imagen, compose, CI/CD, deploy sin downtime
Nube (nu-01..05)        → servidores, frontera de responsabilidad, providers
Proyecto (36-39)        → Nexus: todo junto, diseñado y endurecido
Repaso (ap-g..k)        → 50 tickets, entrevistas, preguntas decisivas
```

Cada eslabón existe porque el anterior dejó un problema sin resolver — esa cadena de necesidades *es* la arquitectura, no una lista de temas.

> **Explicación de: array / lista / slice**
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

54.2. La tabla final: ¿de quién es cada cosa?

Lo que confunde al junior es atribuir al lenguaje lo que es del framework, o al framework lo que es de la industria:

> Explicación de: framework
>
> Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Decisión	Del lenguaje	Del framework	De la industria (igual en todos)
Concurrencia	event loop / asyncio / goroutines	—	I/O-bound se paraleliza
Validación	—	zod / pydantic / manual	schema en la frontera
Auth	—	Depends / middleware / wrapper	JWT corto + refresh persistido
Errores	clases / excepciones / sentinel	exception handler / middleware	dominio → mapper → contrato
BD	—	drivers	pool, parametrizar, tx, índices
Deploy	binario vs runtime	—	imagen única, env vars, healthchecks
Testing	—	vitest / pytest / go test	pirámide + rollback-per-test

La columna derecha es el libro entero: lo que aprendiste no fue "Express" o "FastAPI" — fue la industria: contratos, pools, migraciones, transacciones, rate limits, auditorías. Los lenguajes y frameworks son la *sintaxis* de esas ideas; por eso puedes leer un backend en el stack que no practicas y reconocerle la forma.

> Explicación de: migración
>
> Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

> Explicación de: transacción
>
> Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

54.3. ¿Node, Python o Go? — la respuesta honesta

- **Node/TypeScript**: un solo lenguaje en front y back — el camino natural del fullstack pequeño; el ecosistema npm más grande; contrata más gente por oferta. Su precio: la concurrencia es cooperativa (un hilo) y el ecosistema se mueve rápido (dependencias que mueren).

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

- **Python/FastAPI**: el framework más didáctico y el ecosistema de datos/AI — si el producto toca ML, pandas o LLMs, estás en casa. Su precio: el GIL (workers por núcleo) y el ecosistema de empaquetado.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

- **Go**: rendimiento por núcleo, concurrencia real, un binario de deploy hermoso — para infra, workers, alto throughput. Su precio: más verboso, menos mágico (no hay framework que decida por ti).

El criterio, en orden: **qué sabe el equipo → qué pide el dominio → mercado de empleos local → gustos.** Y la verdad incómoda del capítulo: para un CRUD bien hecho, los tres son equivalentes — el cuello de botella será la BD, no el lenguaje.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

54.4. Lo que el libro dejó fuera — a propósito

```
Microservicios → el monolito modular primero (cap. 7, Ap. K #7)
CRDT/OT        → LWW documentado; el merge fino es un proyecto aparte
Colas serias   → in-process hasta que el volumen lo exija (cap. 21)
Caché/Redis    → cuando un problema MEDIDO lo pida (Ap. K #14)
Kubernetes     → la orquestación que necesitas cuando la necesitas
Event sourcing → diffs como transporte; snapshot como verdad (cap. 37)
```

No son vacíos: son **decisiones** — el senior sabe qué *no* construir todavía y por qué. Cada uno tiene su capítulo de decisión en el Ap. K.

54.5. Errores comunes (de carrera, esta vez)

Error	Por qué pasa	Fix
Aprender el siguiente framework	"Falta NestJS en mi CV"	Falta *fundamento*: el framework nuevo es un sub-tab, no un mundo
Portafolio de tutoriales	Proyectos copiados que no se pueden defender	UN proyecto con ADRs que puedes *discutir* en la entrevista — Nexus
Memorizar preguntas	Listas de "las 50 preguntas"	Los Ap. H/I/J + K: repaso activo + criterio — responder es razonar
Esperar a "saber suficiente"	El libro nunca termina	Ya sabes el mapa — ahora se aprende construyendo y leyendo código real

54.6. Buenas prácticas para seguir

- **Lee backends reales**: cal.com (Next/tRPC), immich (NestJS), miniflux (Go), paperless-ngx (Django) — mapea cada uno contra las capas del cap. 7: encontrarás los mismos huesos.
- **Reconstruye una pieza desde cero**: tu propio middleware de auth, tu propio versionado — la segunda implementación es la que enseña.
- **Usa la AI como lead, no como autor**: los "Prompt listo" del libro son tu formato — contexto + requisitos + criterios de aceptación; y el diff se revisa como PR de un junior (cap-terminal).

> Explicación de: terminal / consola
>
> La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

- **El ADR es un hábito, no un documento**: cada "¿por qué así?" merece un párrafo — tu yo del futuro es el equipo.

54.7. Ejercicio

1. Toma la tabla "¿de quién es cada cosa?" y agrega tres filas que el libro no lista explícitas (ej. logging, config, webhooks).
2. Tu equipo elige Go para un CRUD interno de 4 endpoints "porque es más rápido". ¿Qué le respondes con el criterio del capítulo?
3. De "lo que quedó fuera": elige uno y escribe la condición concreta bajo la cual *sí* lo introducirías en Nexus.

54.8. Mini reto

Elegir un backend open-source real y mapear sus capas contra la arquitectura del libro.

Soluciones

Ejercicio.

1. Ejemplos correctos: logging → industria (JSON + request_id en los tres); config → industria (12-factor + fail-fast); webhooks → industria (firma + dedupe + cola); rate limiting → industria; background jobs → industria (patrón) con mecanismo del framework (BackgroundTasks) o lenguaje (goroutine). La tabla crece — el criterio no cambia.
2. "Rápido para qué — ¿el cuello de botella es el lenguaje o la BD de 3 tablas? Si el equipo conoce TS y esto es un CRUD de 4 endpoints, Go compra performance que no necesitamos y paga verbosidad que sí pagamos. Si mañana pide 10k req/s de streaming, hablamos." — el criterio manda, no el hype.
3. Ejemplos defendibles: **cola seria** — cuando el volumen de emails/ jobs haga que perder uno sea inaceptable o los workers in-process mueran con deploys frecuentes; **Redis** — cuando una query medida supere X ms tras índices, o rate limiting multi-instancia lo exija; **CRDT** — cuando el merge por carácter sea requisito del producto (colaboración tipo Google Docs), no antes.

Mini reto. El ejercicio de cierre del libro: abrir `src/` de un backend real y etiquetar — "esto es el router (cap. 3), esto el schema (cap. 4), esto el pool (cap. 10), esto el middleware de auth (cap. 16), esto la migración (cap. 11)". Cuando un repo de 200 archivos se lee como "capas que ya conoces", el libro cumplió: **la arquitectura es la constante; la sintaxis es el acento.**

Repaso de entrevista: las preguntas de este tema viven en el Apéndice J (AI y conductual).

54.9. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: goroutine

Una **goroutine** es el hilo ultraligero de Go: `go f()` lanza una función en paralelo gastando casi nada — se pueden tener millones. Es la superpotencia de Go: concurrencia como palabra del lenguaje.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: sesión

Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada llamada.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

54.10. Lo que deberías saber hacer ahora

- ☐ Recorrer el mapa completo y decir qué problema resuelve cada pieza
- ☐ Separar qué es del lenguaje, del framework y de la industria
- ☐ Responder "¿Node, Python o Go?" con el criterio, no con el hype
- ☐ Nombrar qué dejó fuera el libro y la condición que lo activaría
- ☐ Leer un backend real mapeándolo a las capas conocidas

Fin del camino planeado — inicio del tuyo. El libro te enseñó el mapa; el territorio se camina construyendo. Nexus es tu prueba de que puedes; los apéndices son tu kit de repaso. A construir.

A A. Tres sintaxis, un programador

La tabla de traducción TS Python Go

> **En una frase**
>
> Solo lo que el libro usó, en tres columnas: tipos, módulos, errores, clases/structs, concurrencia — y los errores típicos del que viene de JS. Cuando leas un backend en el stack que no practicas, esta tabla es el diccionario.

A.1. Sintaxis esencial

Concepto	TypeScript	Python	Go
Función	`function f(x: number): number {}`	`def f(x: int) -> int:`	`func f(x int) int {}`
Bloques	`{ }`	indentación	`{ }`
Nulo	`null` / `undefined`	`None`	`nil`
Lista	`T[]` / `Array<T>`	`list[T]`	`[]T` (slice)
Mapa	`Record<K,V>` / `Map`	`dict[K,V]`	`map[K]V`
Módulo	`import { x } from "./m"`	`from m import x`	`import "m"`
Error	`throw` / `catch`	`raise` / `except`	`return err` (valor)
Async	`async/await` + `Promise.all`	`async/await` + `gather`	`go f()` + channels/errgroup
"Clase"	`class`	`class`	`struct` + métodos (sin herencia)

A.2. Tipos y estructuras

```
TS:      interface Task { id: number; title: string; done: boolean }
Python:  class Task(BaseModel): id: int; title: str; done: bool
Go:      type Task struct { ID int64; Title string; Done bool }
```

```
TS:      tasks.filter(t => !t.done).map(t => t.title)
Python:  [t.title for t in tasks if not t.done]
Go:      var titles []string; for _, t := range tasks { if !t.Done { titles = append(...) } }
```

A.3. Errores: el modelo mental de cada uno

```
TS:      excepciones que suben - throw new DomainError(...) → catch/handler
Python:  excepciones que suben - raise NotFoundError → except/handler
Go:      errores como VALOR - if err != nil { return err } → sin saltos
```

El que viene de JS a Go siempre pregunta "¿dónde está el catch?" — no hay: el error viaja por `return`, visible en cada firma. Verbosidad que compra visibilidad.

> Explicación de: return
>
> **return** hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

> Explicación de: firma / HMAC
>
> Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

A.4. Lo que NO se traduce directo

Pieza	Solo existe en	Por qué
`yield`/generadores	Python (y TS iterators)	Streams perezosos — Go usa canales/loops
`defer`	Go	"Ejecuta esto al salir de la función" — el `finally` que no olvidas
Context managers (`with`)	Python	Recurso que se cierra solo — TS usa `try/finally`, Go `defer`
Goroutines	Go	Concurrencia nativa barata — en los otros es modelo, no palabra clave
Punteros explícitos	Go (`*T` vs `T`)	Valor vs referencia visible en la firma

A.5. Errores típicos del que viene de JS

Trampa	En qué lenguaje muerde	El por qué
Default mutable `def f(items=[])`	Python	El default se crea UNA vez — compartido entre llamadas → `None` + crear dentro
`==` vs `is`	Python	`==` compara valor, `is` compara identidad — `x == []` no es `x is []`
`if (err)` vs `err == nil`	Go	`nil` es su propio tipo de valor — y el error se *retorna*, no se lanza
Comparar `struct` con `==`	Go	Funciona pero compara campos — con slices/maps adentro no compila
`interface{}`/`any` como escape	TS/Go	"Luego lo tipo" = nunca — el tipo débil contamina todo lo que lo toca
Slices que comparten array	Go	`append` puede mutar el backing array — copia si dudas

A.6. Mini reto

Traducir una función del proyecto a los otros dos lenguajes sin mirar la solución.

Soluciones

Toma `createTask(userId, title)` — validar, trim, insertar, devolver. La traducción correcta no es literal: en Go el error viaja por return (`(Task, error)`), en Python el tipo va en la firma con `-> Task`, en TS el schema de entrada se valida con zod antes. Si tu traducción usa `try/catch` en Go o `return err` en Python, estás escribiendo JS con acento — la forma correcta es la del idioma, no la tuya.

A.7. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: booleano

Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: concurrencia vs paralelismo

Concurrencia = manejar muchas cosas en progreso (atender 1000 requests intercalando). **Paralelismo** = ejecutar varias a la vez en varios núcleos. Un camarero con 10 mesas es concurrente; 10 camareros son paralelos.

Explicación de: módulo / import

Un **módulo** es un archivo de código que exporta cosas para que otros archivos las importen. Es la unidad de organización: en vez de un archivo gigante, el código vive en módulos con responsabilidad propia.

Explicación de: async / await / Promise

Async/await es la sintaxis para operaciones que tardan (red, disco): `await` pausa *esa función* sin congelar el programa entero. Una `Promise` es la "promesa" de un resultado futuro — el ticket que recibes mientras lo preparan.

Explicación de: tipos / tipado estático

Los **tipos** (`int`, `string`, `boolean`, `Task`) son el contrato de cada dato. *Tipado estático* (TypeScript, Go, Python con hints) = los tipos se revisan al escribir, no cuando explota en producción.

A.8. Lo que deberías saber hacer ahora

- ☐ Leer una función en cualquiera de los tres stacks y seguirla
- ☐ Explicar por qué Go no tiene `catch` y qué lo reemplaza
- ☐ Nombrar dos trampas de cada lenguaje para quien viene de JS
- ☐ Traducir una función respetando el *idioma*, no la letra

B B. SQL de bolsillo

Las 20 queries que un backend usa

> **En una frase**
>
> Cheatsheet agnóstico: CRUD, JOINs, agregaciones, índices, `EXPLAIN` — las 20 queries que cubren el 95 % de lo que un backend le pide a una BD relacional, agrupadas por lo que resuelven.

B.1. CRUD — las 4 de siempre

```
-- 1. Leer uno (siempre parametrizado: $1 /%s / ?)
SELECT id, email, name FROM users WHERE id = $1;

-- 2. Leer lista filtrada + ordenada + paginada
SELECT id, title, done FROM tasks
WHERE user_id = $1 AND done = false
ORDER BY created_at DESC
LIMIT 21;                          -- pide 21, muestra 20: el 21 dice "hay más"

-- 3. Insertar y recuperar lo generado
INSERT INTO tasks (user_id, title) VALUES ($1, $2)
RETURNING id, created_at;

-- 4. Actualizar / borrar SIEMPRE con WHERE
UPDATE tasks SET done = true WHERE id = $1 AND user_id = $2;   -- ownership en la query
DELETE FROM tasks WHERE id = $1 AND user_id = $2;
```

B.2. Relaciones — las que devuelven lo que la UI muestra

```
-- 5. JOIN: la tarea con su dueño
SELECT t.id, t.title, u.name AS owner
FROM tasks t JOIN users u ON u.id = t.user_id;
```

```sql
-- 6. LEFT JOIN: incluir aunque no haya hijo
SELECT p.id, p.name, COUNT(t.id) AS task_count
FROM projects p LEFT JOIN tasks t ON t.project_id = p.id
GROUP BY p.id;

-- 7. EXISTS: "¿hay alguno que cumpla?" - más barato que COUNT
SELECT EXISTS (SELECT 1 FROM team_members
               WHERE team_id = $1 AND user_id = $2 AND role = 'admin');

-- 8. IN: cualquiera de la lista
SELECT * FROM tasks WHERE project_id IN ($1, $2, $3);

-- 9. Subquery: filtrar por el resultado de otra pregunta
SELECT * FROM users WHERE id IN
  (SELECT user_id FROM tasks GROUP BY user_id HAVING COUNT(*) > 10);
```

B.3. Agregaciones — responder preguntas, no listar filas

```sql
-- 10. Contar y agrupar
SELECT status, COUNT(*) FROM tasks GROUP BY status;

-- 11. Agregado con condición (el "cuántos sin hacer" por usuario)
SELECT user_id, COUNT(*) FILTER (WHERE done = false) AS pending
FROM tasks GROUP BY user_id;

-- 12. HAVING: WHERE sobre grupos
SELECT team_id FROM team_members GROUP BY team_id HAVING COUNT(*) > 50;

-- 13. MAX/más reciente por grupo
SELECT DISTINCT ON (document_id) document_id, rev, created_at
FROM revisions ORDER BY document_id, rev DESC;
```

B.4. Transacciones — todo o nada

```sql
-- 14. La transacción de negocio
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = $1;
UPDATE accounts SET balance = balance + 100 WHERE id = $2;
COMMIT;          -- o ROLLBACK si cualquier pieza falla

-- 15. Lock de fila: "nadie más toque esto hasta que yo termine"
SELECT balance FROM accounts WHERE id = $1 FOR UPDATE;
```

```sql
-- 16. Upsert: insertar o actualizar según choque
INSERT INTO sessions (id, user_id, expires_at) VALUES ($1,$2,$3)
ON CONFLICT (id) DO UPDATE SET expires_at = EXCLUDED.expires_at;
```

B.5. Performance — ver qué hace la BD

```sql
-- 17. El índice que sigue el patrón de acceso
CREATE INDEX CONCURRENTLY idx_tasks_user_created
  ON tasks (user_id, created_at DESC);     -- igualdad primero, orden después

-- 18. Ver el plan: ¿usa el índice o escanea todo?
EXPLAIN ANALYZE SELECT * FROM tasks WHERE user_id = $1 ORDER BY created_at DESC;
-- Seq Scan en tabla grande = alarma; Index Scan/Bitmap = bien

-- 19. Estadísticas frescas (sin esto, el planner decide con datos viejos)
ANALYZE tasks;

-- 20. JSONB: la columna flexible cuando el esquema varía
SELECT snapshot->>'title' FROM revisions WHERE snapshot @> '{"done":true}';
```

B.6. Las reglas detrás de las queries

- **Parametrizado siempre** — `$1` / `%s` / `?` nunca se concatena; la inyección SQL se muere en esta línea (cap. 9).

> Explicación de: inyección SQL
>
> La **inyección SQL** es el clásico de romper la query: si concatenas input del usuario en el SQL, el usuario *escribe SQL*. `' OR 1=1--` borra tablas. La vacuna: queries parametrizadas (`$1`), siempre.

- **`WHERE` con ownership** — `AND user_id = $2` convierte "borra la tarea 5" en "borra la tarea 5 *si es tuya*" (cap. 17).
- **`LIMIT` con `ORDER BY`** — límite sin orden es un límite a la suerte: el orden sin índice es sort en memoria (cap. 12).

> Explicación de: memoria (RAM)
>
> La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

- **`EXPLAIN` antes de indexar** — el índice responde a un patrón de acceso medido, no a una corazonada.
- **`FOR UPDATE` dentro de la tx** — fuera de transacción no bloquea nada: el lock vive y muere con el BEGIN/COMMIT (cap. 13).

Explicación de: lock / bloqueo (FOR UPDATE)

Un **lock** de fila (`SELECT ... FOR UPDATE`) dice "esta fila es mía hasta que mi transacción termine": otros que la pidan esperan. Es como el candado del probador de ropa — evita que dos editen lo mismo.

B.7. Mini reto

Reescribir de memoria las 5 queries más usadas de Taskflow.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Soluciones

Las 5 que un CRUD real repite todo el día: (1) leer uno por id con ownership, (2) listar filtrado + orden + LIMIT 21, (3) insert con RETURNING, (4) update con doble WHERE (id + user_id), (5) EXISTS de membresía. Si salen de memoria con los `$1` en su sitio y el ownership en el WHERE — eso *es* el nivel que una entrevista pide.

B.8. Vocabulario técnico del capítulo

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: snapshot

Un **snapshot** es una foto completa del estado en un momento: el documento entero en la revisión 14. Restaurar = leer la foto, no reconstruir sumando cambios. Cuesta más disco; regala simplicidad.

Explicación de: JSON

JSON (JavaScript Object Notation) es el formato universal para mover datos entre programas: texto con llaves y corchetes, `{"title": "Comprar leche", "done": false}`. Casi cualquier lenguaje lo lee y lo escribe.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A y sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: JOIN

Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

Explicación de: paginación

Paginar es servir la lista en páginas (20 por 20) en vez de las 10 millones. `LIMIT 21` + cursor = la página siguiente se pide con la última fila vista, no con "salta 40000" (offset, que se degrada).

Explicación de: CRUD

CRUD = Create, Read, Update, Delete — las cuatro operaciones básicas sobre un recurso. `POST/GET/PUT/DELETE /tasks`. El 80 % de las APIs del mundo es CRUD bien hecho sobre varias tablas.

Explicación de: roles / RBAC

Los **roles** agrupan permisos: *viewer* lee, *editor* escribe, *admin* gestiona. RBAC = el permiso depende del rol que tienes *en ese recurso* — puedes ser admin de un equipo y viewer de otro.

B.9. Lo que deberías saber hacer ahora

- ☐ Escribir las 20 sin mirar la documentación
- ☐ Leer un EXPLAIN y nombrar Seq Scan vs Index Scan
- ☐ Saber qué query va dentro de tx y por qué `FOR UPDATE`
- ☐ Usar `FILTER`, `DISTINCT ON`, `ON CONFLICT` — las tres joyas de Postgres

C C. El mapa universal

Qué vive en el lenguaje, qué en el framework, qué en la industria

> **En una frase**
>
> La tabla de tres columnas con **todos** los conceptos del libro clasificados — ahora con evidencia en los tres stacks. Cuando cambies de stack, la columna derecha es tu equipaje completo; las otras dos son solo sintaxis nueva.

C.1. El mapa completo

Concepto	Del lenguaje	Del framework	De la industria (te lo llevas)	Cap.
Async/concurrencia	event loop / asyncio / goroutines	—	I/O-bound se paraleliza, CPU-bound no	20
Tipos en la frontera	anotaciones	zod / pydantic / decoder	validar antes de tocar	4
Routing	—	Express / FastAPI / ServeMux	recursos, verbos, contrato	3
Inyección de deps	—	Depends / closures / params	construir en el borde, pedir adentro	16
Errores	throw / raise / return err	handler global	dominio → mapper → `{error:{code}}`	6, 25
Auth	—	middleware/Depends/wrapper	bcrypt, JWT corto + refresh rotado	14–18
Autorización	—	Depends/factory	rol en la relación, ownership en la query	17
BD	—	pg / psycopg / pgx	pool, parametrizar, tx, FOR UPDATE	10, 13

Concepto	Del lenguaje	Del framework	De la industria (te lo llevas)	Cap.
Migraciones	—	dbmate / alembic / goose	versionadas, up+down, sin editar viejas	11
Índices/N+1	—	—	EXPLAIN, igualdad→orden, LIMIT 21	12
Integraciones	fetch / httpx / http.Client	—	timeout, retry, degradación, adapter	19
Jobs	—	setInterval / BackgroundTasks / ticker	"¿espera el usuario?" → si no, job	21
Webhooks	—	—	firma sobre body crudo, dedupe, 200 rápido	22
Logs	—	pino / logging / slog	JSON + request_id, nunca secretos	23
Config	—	dotenv / pydantic-settings / env	12-factor, fail-fast, sin defaults secretos	24
Seguridad	—	—	OWASP auditada por endpoint, 404 vs 403	26
Testing	—	vitest / pytest / go test	pirámide, rollback-per-test, mock en frontera	27–29
Deploy	binario vs runtime	—	imagen única, env vars, healthcheck, migrate-gate	30–35
Versionado	—	—	snapshot + puntero, append-only	37
Realtime	—	ws / WebSocket / coder	auth igual que REST, rooms, LWW	38

C.2. Cómo leer la tabla

La columna derecha es el 80 % del trabajo. Un senior en Node que lee Go no está "aprendiendo backend de nuevo" — está aprendiendo dónde Go pone el `err` y cómo se llaman sus paquetes. Las *decisiones* ya las tiene: parametrizar, transaccionar, auditar, testear, desplegar con compuertas.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

El framework decide menos de lo que crees. FastAPI no te obliga a usar Depends ni Express a usar middleware — pero la industria sí te obliga a validar, a autenticar una vez y a no dejar stack traces. El framework propone el *mecanismo*; tú mantienes el *control*.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

El lenguaje solo cambia la forma. `Promise.all/gather/errgroup` son la misma idea con tres acentos — como `throw/raise/return err`. Por eso el libro muestra los tres: para que veas que la idea sobrevive a la sintaxis.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

C.3. Mini reto

Agregar a la tabla un concepto que el libro no cubrió (ej. caché) y clasificarlo en las tres columnas.

Soluciones

Ejemplo resuelto — **caché**: del lenguaje, casi nada (quizá estructuras en memoria); del framework, un decorador/middleware opcional; **de la industria**: invalidar es lo difícil, medir antes de cachear, TTL + claves versionadas, Redis como pieza compartida entre instancias. Si tu fila pone "Redis" en la columna del lenguaje, vuelve al cap. 40: el producto no es del lenguaje — la *decisión de cachear* tampoco.

C.4. Vocabulario técnico del capítulo

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: asíncrono

Asíncrono = empezar algo sin esperar sentado a que termine: pides la pizza (async) y sigues trabajando; cuando llega, te avisan. Lo opuesto a síncrono (esperar parado). Vital cuando la espera es larga: red, disco, bases de datos.

Explicación de: goroutine

Una **goroutine** es el hilo ultraligero de Go: `go f()` lanza una función en paralelo gastando casi nada — se pueden tener millones. Es la superpotencia de Go: concurrencia como palabra del lenguaje.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña →`$2b10…‘. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques “esto realmente vino de Stripe”.

Explicación de: log / logging estructurado

Un **log** es el diario del programa: qué pasó, cuándo, con qué request. *Estructurado* = en JSON con campos (`request_id`, `user_id`), para filtrar por máquina y no con los ojos.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

C.5. Lo que deberías saber hacer ahora

☐ Clasificar cualquier concepto nuevo en lenguaje / framework / industria
☐ Explicar por qué la columna derecha es el equipaje que viaja
☐ Leer un backend en otro stack reconociendo los mismos huesos

D D. Glosario de backend de la vida real

Definiciones como las daría un equipo, no un examen

> **En una frase**
>
> Cada término del libro definido como lo explicaría un compañero en la daily — qué es, para qué existe, y la trampa que esconde. Ordenado por tema, pensado para consulta rápida y repaso en voz alta.

D.1. HTTP y APIs

Endpoint — La pareja `verbo + ruta` (`GET /tasks/5`). La API es la colección de endpoints; el contrato es lo que cada uno promete.

Contrato / schema — La forma acordada del request y del response. Existe para que front y back puedan trabajar sin llamarse. Trampa: documentarlo "después" — después nunca llega.

REST — Convención: recursos en la URL, verbos como acciones, JSON como formato, status codes como resultado. No es una ley, es el default que todos entienden.

Idempotencia — Repetir la operación N veces deja el sistema igual que con 1. `PUT` es idempotente por diseño; `POST` necesita `Idempotency-Key`. Trampa: asumir que el cliente no reintenta — siempre reintenta.

Rate limiting — Presupuesto de requests por identidad/ventana. Existe porque el abuso y los bugs de loop se parecen mucho. Devuelve `429` + `Retry-After`, no un 500 misterioso.

> **Explicación de: bucle (loop)**
>
> Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

CORS — Política del *navegador*: "¿desde qué orígenes puede JS leer esta respuesta?" No es seguridad contra servidores — `curl` ni lo ve. Trampa: `*` con credenciales, que los navegadores rechazan (bien).

OpenAPI — El contrato escrito en YAML/JSON que humanos y máquinas leen: documentación viva, clientes generados, tests de contrato. Es la fuente de verdad cuando existe antes que el código.

D.2. Datos

ORM — El traductor objetos tablas. Ahorra el SQL repetitivo y cobra con N+1 y queries opacas. Regla: ORM para el CRUD, SQL crudo para lo difícil — y siempre poder leer lo que genera.

> Explicación de: ORM
>
> Un **ORM** (Object-Relational Mapper) traduce entre objetos del código y filas de la tabla: `task.save()` en vez de `INSERT`. Cómodo para el CRUD, peligroso si no sabes qué SQL genera (el N+1 nace ahí).

> Explicación de: query / consulta
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Migración — Cambio de esquema versionado (`up`/`down`) aplicado en orden. Existe porque "la BD de cada quien" no escala a dos personas. Trampa: editar una migración ya aplicada — se crea una nueva, jamás se retoca el historial.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

> Explicación de: migración
>
> Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Transacción — Grupo de operaciones que se confirman o se revierten juntas (ACID). Existe porque el mundo falla a la mitad. `FOR UPDATE` = "esta fila es mía hasta el COMMIT".

> Explicación de: transacción
>
> Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

N+1 — La enfermedad de los loops: 1 query para la lista + N para los hijos. Se cura con JOIN/agregación — una query, no N.

Explicación de: JOIN

Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

Pool — Las conexiones a la BD reutilizadas entre requests, porque abrir una por request es caro. Tamaño real: 10–20, no 1000.

Índice — Estructura aparte (B-tree) que convierte búsquedas en saltos. Sigue el patrón de acceso: igualdad primero, orden después. Trampa: indexar todo — cada índice se paga en cada escritura.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

JSONB — Columna JSON indexable dentro del relacional. Decisión cuando el *contenido* varía por registro; pereza cuando es por no diseñar las tablas.

D.3. Identidad y seguridad

JWT — JSON firmado (no cifrado): el servidor lo emite y verifica sin consultar la BD. Contenido legible para cualquiera — nunca secretos adentro. La firma garantiza *integridad*, no *privacidad*.

Refresh token + rotación — El token largo guardado (hasheado) en BD que renueva el access corto. Rotación: cada uso entrega uno nuevo y quema el anterior — reuso detectado = familia entera revocada.

httpOnly cookie — Cookie que JavaScript no puede leer: el XSS que roba tokens no la alcanza. vs `localStorage`: cómodo pero legible por cualquier script que se cuele.

Explicación de: token

Un **token** es una credencial portable: una cadena que dice quién eres y hasta cuándo. El servidor la emite tras el login; el cliente la presenta en cada request en vez de la contraseña.

BOLA / IDOR — "¿Puedo pedir `/tasks/99` que es de otra persona?" El #1 de OWASP API. Fix: ownership *en la query* y 404 (no 403) para lo ajeno — no confirmes que existe.

HMAC / firma de webhook — Hash del body crudo con un secreto compartido: prueba de que el evento viene del proveedor y nadie lo tocó. Se verifica *antes* de parsear — el JSON re-serializado no firma igual.

Nonce — Número de un solo uso: en webhooks rechaza replays, en auth evita reuso. Pariente del idempotency key.

OWASP API Top 10 — La lista de las 10 formas reales en que mueren las APIs. No es teoría: es la tabla de auditoría por endpoint del cap. 26.

D.4. Operación

Reverse proxy — El portero (nginx/ALB) que recibe el 443, hace TLS y reparte a tu app en :3000. La app nunca ve internet directamente.

> Explicación de: puerto
>
> Un **puerto** es una puerta numerada dentro de una computadora: la IP llega al edificio, el puerto al departamento. `:3000` = "la app que escucha en el departamento 3000". Postgres suele usar 5432, HTTP el 80, HTTPS el 443.

12-factor — Las reglas de deploy: config por env vars, procesos sin estado, logs a stdout, mismo artefacto en todos los entornos. La razón por la que tu Dockerfile funciona igual en dev y prod.

> Explicación de: entorno (environment)
>
> Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Request ID — El ID único por request que viaja en logs, headers y respuestas de error. Es la diferencia entre "algo falló" y "esta línea de tiempo concreta falló".

Healthcheck — `/health` (¿vivo?) y `/health/deep` (¿BD accesible?). El orquestador decide restart/tráfico según el primero; los humanos diagnostican con el segundo.

Graceful shutdown — Al apagar: dejar de aceptar, terminar lo que corre, cerrar el pool, salir. Lo que convierte un deploy en invisible en vez de una ráfaga de 502.

> Explicación de: pool de conexiones
>
> Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Migrate-gate — Migrar en CI *antes* del deploy: el deploy de migración fallida nunca despierta — la imagen verde no sale si la migración no entró.

Runbook — El guion de la 3am: qué mirar, en qué orden, qué comando. La observabilidad convertida en procedimiento — cinco líneas que valen más que veinte gráficas.

D.5. Testing

Pirámide — Muchos unitarios (rápidos), algunos de API (la base real), pocos e2e (frágiles). Invertida = suite lenta que todos ignoran.

Rollback-per-test — Cada test corre en una tx que se revierte al final: tests aislados contra BD real sin sembrar/limpiar. El truco que hace barata la base de la pirámide.

Mock en la frontera — Fingir el *adaptador* externo (la función que llama a Stripe/SMTP), nunca `fetch` ni la BD. Mockear la BD es fingir que el SQL funciona — justo lo que el test debía probar.

D.6. Mini reto

Explicar 5 términos en voz alta sin mirar — con la trampa incluida.

Soluciones

La prueba honesta: si al explicar *idempotencia* mencionaste `Idempotency-Key` y los reintentos del cliente, lo tienes. Si solo salió "se puede repetir", vuelve al cap. 22 — la definición sin el *para qué* es trivia, no conocimiento.

D.7. Vocabulario técnico del capítulo

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: cookie

Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

Explicación de: JWT

Un **JWT** (JSON Web Token) es un token *firmado*: el servidor lo genera con su secreto y puede verificarlo sin consultar la BD. Ojo: está firmado, no cifrado — cualquiera puede leer su contenido, pero no modificarlo.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña` →$2b10…‘. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que “se caiga el proceso” = el programa murió.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

D.8. Lo que deberías saber hacer ahora

- ☐ Definir cada término con su “para qué existe” y su trampa
- ☐ Reconocer cuándo un término elegante esconde una decisión (JSONB, ORM)
- ☐ Usar el vocabulario correcto en una entrevista sin sonar a examen

E E. Estructura de proyecto ×3

El árbol de carpetas de Nexus comentado, por lenguaje

> **En una frase**
>
> La plantilla reusable: dónde vive cada capa (rutas, schemas, servicios, queries, middleware) en Node/TS, Python y Go. Mismos huesos, distinta sintaxis de carpetas — copia el árbol de tu stack y empieza.

E.1. TypeScript (Express)

```
nexus-api/
  src/
    routes/          # HTTP edge: parse → call service → format response
      tasks.ts
      auth.ts
    schemas/         # zod: the contract of each endpoint (input + output)
      task.ts
    services/        # business rules - no req/res here, ever
      taskService.ts
    db/
      pool.ts      # THE pool - created once, imported everywhere
      queries.ts   # parametrized SQL lives here, not in services
    middleware/
      auth.ts      # requireUser - cookie → user → res.locals
      errors.ts    # THE global handler - DomainError → {error:{code}}
    errors.ts        # DomainError hierarchy
    app.ts           # wiring: middleware → routes → error handler LAST
  migrations/         # dbmate: numbered, versioned, never edited
  tests/              # vitest + supertest, DB real, rollback-per-test
  .env.example        # every required key documented - the real .env is gitignored
  Dockerfile          # multi-stage: deps → build → runtime
  compose.yaml        # app + postgres for dev
```

E.2. Python (FastAPI)

```
nexus-api/
   app/
      routers/         # APIRouter per resource: the HTTP edge
         tasks.py
         auth.py
      schemas/         # pydantic: contract IS the framework's language
         task.py
      services/        # business rules - never sees Request/Response
         task_service.py
      db/
         pool.py      # connection pool - built at startup
         queries.py   # parametrized SQL
      deps.py          # Depends: current_user, get_db - injection lives here
      errors.py        # exception hierarchy + handlers
      main.py          # wiring: include_router + exception handlers
   migrations/          # alembic: versions/, env.py
   tests/               # pytest + TestClient + rollback fixture
   .env.example
   Dockerfile
   compose.yaml
```

E.3. Go

```
nexus-api/
   cmd/
      api/
          main.go      # THE entrypoint: config → pool → mux → server
   internal/            # unimportable from outside - enforced by the compiler
      http/
         routes.go    # ServeMux patterns: "GET /tasks/{id}"
         middleware.go # requireUser wrapping http.Handler
      tasks/
         service.go   # business rules
         queries.go   # parametrized SQL
      auth/
      errors/          # sentinel errors + the one mapper
   migrations/          # goose: .sql files, up/down
   tests/ or *_test.go  # go test + httptest, tx rollback
   .env.example
   Dockerfile           # multi-stage: builder → scratch/distroless binary
   compose.yaml
```

E.4. Cómo leer los árboles

Las capas son las mismas, los nombres cambian. `routes`/`routers`/ `http` = el borde HTTP. `schemas`/`schemas`/structs en `http` = el contrato. `services` = las reglas. `db`/`queries` = el SQL. La arquitectura del cap. 7 no es "de Express" — es de backend.

> Explicación de: query / consulta
>
> Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

El SQL tiene casa propia. En los tres, las queries viven en un archivo/capa aparte (`queries.ts`, `queries.py`, `queries.go`) — no esparcidas por los servicios. Cuando el DBA pregunta "¿qué le pide tu app a la BD?", la respuesta es un archivo, no un grep.

> Explicación de: base de datos
>
> Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

`internal/` en Go es el único muro real. El compilador prohíbe importarlo desde fuera — la privacidad que en los otros stacks es convención, en Go es ley.

El entrypoint es delgado en los tres. `app.ts`/`main.py`/`main.go` solo *conectan* piezas — si tiene lógica de negocio, el árbol falló.

E.5. Errores comunes

Error	Por qué pasa	Fix
Lógica en las rutas	"Es un endpoint pequeño"	La ruta parsea y delega — el servicio decide
SQL esparcido	Queries dentro de servicios	`queries.*` por dominio — un archivo por fuente de verdad
El pool recreado por request	`createPool()` dentro del handler	El pool nace una vez en el entrypoint y se inyecta
Carpetas por tipo técnico	`controllers/`, `models/` genéricos	Por dominio (tasks/, auth/) — el proyecto crece por recursos
Tests en otra galaxia	`tests/` sin espejo del src	El árbol de tests refleja el de src — encontrar el test = saber dónde

E.6. Mini reto

Arrancar un proyecto nuevo desde cero usando solo el árbol como guía.

Soluciones

El orden correcto: entrypoint delgado → pool inyectado → una ruta de ejemplo que recorre las 4 capas (ruta → schema → servicio → query) → error handler → un test de integración. Si en 30 minutos tienes `GET /health` + un endpoint CRUD completo con las capas separadas, el árbol funciona — y ese esqueleto es literalmente lo que un lead espera el día 1.

E.7. Vocabulario técnico del capítulo

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: cookie

Una **cookie** es un dato que el servidor pone en tu navegador y el navegador devuelve en cada request. `httpOnly` = JavaScript no puede leerla (el XSS no la roba); `Secure` = solo viaja por HTTPS.

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

Explicación de: servidor

Un **servidor** es una computadora que espera peticiones y las responde 24/7. Físicamente no es nada mágico: es una máquina (a veces una VM alquilada) corriendo tu programa, con la diferencia de que está siempre encendida y conectada.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: Docker Compose

Docker Compose describe un sistema de varios contenedores en un YAML: la app + Postgres + Redis, sus redes y volúmenes. `docker compose up` levanta el entorno completo de desarrollo con un comando.

Explicación de: rollback

Un **rollback** es volver atrás: la migración se revierte, el deploy regresa a la versión anterior. Todo cambio serio tiene plan de rollback *antes* de ejecutarse — si no, el plan es rezar.

Explicación de: health check

Un **health check** es el endpoint `/health` donde la app dice "estoy viva". El orquestador/load balancer lo consulta cada pocos segundos: si falla, reinicia la instancia o deja de mandarle tráfico.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega

es el resultado del build, no tu código crudo.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

E.8. Lo que deberías saber hacer ahora

- ☐ Dibujar el árbol de tu stack de memoria y decir qué vive en cada carpeta
- ☐ Explicar por qué el SQL tiene casa propia en los tres
- ☐ Reconocer cuándo una estructura “por tipo técnico” está fallando

F F. Checklists del profesional

Antes de exponer un endpoint / una migración / un deploy / decir que está listo

> **En una frase**
>
> Las listas de verificación completas que el libro fue construyendo capítulo a capítulo — reunidas para uso diario: antes de mergear, antes de migrar, antes de desplegar, antes de decir "está listo".

F.1. Antes de exponer un endpoint nuevo

```
CONTRATO
[ ] Schema de entrada validado (tipos, required, límites) - cap. 4
[ ] Schema de salida: solo campos del contrato, nada interno - cap. 5
[ ] Status code correcto: 200/201/204, nunca 200-con-error - cap. 6

AUTH
[ ] ¿Requiere sesión? → middleware/Depends aplicado - cap. 16
[ ] ¿Requiere rol? → requireRole o verificación en la query - cap. 17
[ ] ¿Toca un recurso ajeno? → ownership EN el WHERE + 404 - cap. 17

ERRORES
[ ] Errores de dominio mapeados al contrato {error:{code}} - cap. 25
[ ] Nada de stack traces, nada de mensajes internos - cap. 25
[ ] 422 para input inválido con detalle por campo - cap. 4

DATOS
[ ] Query parametrizada - cero concatenación - cap. 9
[ ] ¿Más de una escritura? → transacción - cap. 13
[ ] ¿Lista? → ORDER BY + LIMIT 21 (paginación) - cap. 12
[ ] Índice que cubre el patrón de acceso - EXPLAIN - cap. 12

TEST
[ ] Camino feliz - cap. 28
[ ] 401 sin sesión, 403 rol débil, 404 recurso ajeno - cap. 28
[ ] 422 input inválido - cap. 28
[ ] Fuga de campos internos (password_hash, team_id…) - cap. 28
```

F.2. Antes de correr una migración

```
[ ] Tiene up Y down probados - cap. 11
[ ] No edita una migración ya aplicada en ningún entorno - cap. 11
[ ] Compatible con la versión VIEJA del código (deploy escalonado) - cap. 35
[ ] CREATE INDEX CONCURRENTLY en tablas grandes - cap. 12/35
[ ] NOT NULL solo con DEFAULT o en dos pasos - cap. 35
[ ] Corre en CI antes del deploy (migrate-gate) - cap. 34
[ ] Tiempo estimado medido en staging con datos reales - cap. 39
```

F.3. Antes de un deploy

```
[ ] Tests verdes en CI - cap. 34
[ ] Imagen construida una vez, la MISMA para staging y prod - cap. 32
[ ] Secretos en env vars / secret manager, nunca en la imagen - cap. 24/32
[ ] La app corre como non-root, multi-stage, imagen mínima - cap. 32
[ ] /health responde y /health/deep toca la BD - cap. 33
[ ] Graceful shutdown: deja de aceptar, drena, cierra pool - cap. 33
[ ] La migración ya corrió (o es expand-phase) - cap. 34/35
[ ] Logs JSON + request_id funcionando - cap. 23
[ ] Plan de rollback conocido ANTES de apretar el botón - cap. 35
```

F.4. Antes de decir “está listo”

```
SEGURIDAD (cap. 26 auditada por endpoint)
[ ] Cada endpoint: auth   rol   ownership   exposición
[ ] Login/register con rate limit - cap. 18
[ ] Cookies httpOnly+Secure+SameSite / CORS con orígenes exactos - cap. 15/18
[ ] Webhooks verifican firma sobre body crudo + dedupe - cap. 22
[ ] WebSocket autenticado y autorizado igual que REST - cap. 38

ROBUSTEZ
[ ] Errores → handler global único, formato estable - cap. 25
[ ] Config validada al boot (fail-fast, sin defaults secretos) - cap. 24
[ ] Llamadas externas: timeout + retry + degradación - cap. 19
[ ] Lo que no espera el usuario va a un job - cap. 21
[ ] Logs reconstruyen una request sin reproducirla - cap. 23

DATOS
[ ] Migraciones versionadas y aplicadas - cap. 11
[ ] Índices verificados con EXPLAIN + volumen real - cap. 12/39
```

```
[ ] Operaciones multi-paso dentro de tx - cap. 13
[ ] Lo versionado es append-only con puntero - cap. 37

PRUEBAS Y DOCS
[ ] Pirámide: servicios > API > e2e - cap. 27
[ ] Suite verde de corrido en máquina limpia - cap. 28
[ ] README levanta el proyecto sin ti - cap. 39
[ ] openapi.json publicado + runbook de 5 líneas - cap. 39
[ ] ADR escrito para cada decisión que costó - cap. 36
```

F.5. Mini reto

Pasar la checklist completa a un proyecto tuyo anterior y contar los hallazgos.

Soluciones

La experiencia universal: el primer pase marca 10–20 casillas en rojo — endpoints sin ownership en la query, secretos con default, listas sin paginar, errores que filtran formato interno. No es que tu proyecto fuera malo: es que "funciona" y "está listo" son estándares distintos y esta lista es la distancia entre ambos. La segunda vez que escribes un endpoint, la lista ya corre en tu cabeza — para eso era.

F.6. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: imagen (Docker)

Una **imagen** es la plantilla inmutable de la que nacen contenedores: la foto del disco + cómo arrancar. Se construye en capas (Dockerfile), se versiona con tags, se publica en un registry.

Explicación de: WebSocket

WebSocket es una conexión que queda *viva*: en vez de preguntar- responder-cerrar como HTTP, ambos pueden hablar cuando quieran. Es como el servidor puede *empujar* — por eso sirve para chat y colaboración en vivo.

Explicación de: background job / cola

Un **job** es trabajo que se hace sin que el usuario espere: mandar el email, generar el PDF. Va a una **cola** y un *worker* lo procesa en segundo plano — la respuesta HTTP sale inmediata.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: staging

Staging es el ensayo general: una copia de producción (misma config, datos falsos o anonimizados) donde pruebas el deploy antes de hacerlo de verdad. Si falla ahí, falló gratis.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: middleware

Un **middleware** es un filtro en la cadena del request: pasa por él antes de llegar a tu handler. Auth es middleware — "verifica el token" vive una vez y protege todas las rutas, no se copia en cada una.

Explicación de: variable de entorno

Una **variable de entorno** es configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`. El mismo binario corre en dev y prod con distintas vars — los secretos nunca se escriben en el código.

F.7. Lo que deberías saber hacer ahora

- ☐ Auditar un endpoint nuevo contra la lista completa en minutos
- ☐ Saber qué verifica un deploy seguro y en qué orden
- ☐ Tener "está listo" definido por controles, no por intuición

G G. Un día de trabajo: 50 tickets

Del «conecta a esta API» al «escribe el test primero»

> **En una frase**
>
> 50 tareas como las daría un lead en tu primer trabajo — de "haz un fetch a esta API" hasta "aquí está el test, escribe la implementación". Cada ticket tiene su solución colapsada: **intenta resolverlo primero**, luego compara. Si uno te cuesta, la referencia `→ cap. X` te dice qué releer.

Cómo usarlo: un nivel por sesión, en orden — están graduados como una semana real de onboarding. Tu AI CLI es el lead: si una solución no te queda clara, pégale el ticket y pídele que te lo explique distinto.

> **Explicación de: CLI**
>
> Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

G.1. Nivel 1 · Primeros tickets

Los que te asignan la primera semana: leer, conectar, no romper.

> **Ticket 1 — Conecta a esta API**
>
> **Te llega:** "Necesito saber si esta API pública responde. Trae la lista de usuarios de `https://jsonplaceholder.typicode.com/users` e imprime sus nombres. Cualquier stack."
> **Solución.**

```
const res = await fetch("https://jsonplaceholder.typicode.com/users");
const users = await res.json();
users.forEach(u => console.log(u.name));
```

`fetch` devuelve una Promise del *response*; `.json()` devuelve otra Promise del body parseado — por eso dos `await`. → *fe-05, cap-00*

Ticket 2 — El mismo, pero que falle bien

Te llega: "El ticket 1 explota si la API responde 500 — agrega manejo de error real: status no-ok y JSON malformado."
Solución.

```
const res = await fetch(url);
if (!res.ok) throw new Error(`HTTP ${res.status}`);  // 404/500 are NOT exceptions
const users = await res.json();                      // throws if body isn't JSON
```

`fetch` solo rechaza por falla de *red* — un 500 es un response "exitoso" para fetch. `res.ok` es tu frontera. → *fe-05, cap-06*

Ticket 3 — Inspecciona el endpoint sin código

Te llega: "Antes de escribir nada, mira qué devuelve `GET /tasks/7` del staging. Quiero status, headers y body."
Solución.

```
curl -i https://staging.taskflow.dev/tasks/7
# -i includes response headers - status line + headers + body
```

`-i` muestra el intercambio completo: status code, `content-type`, el JSON. Inspeccionar antes de codificar es el hábito — el endpoint te dice su contrato. → *cap-00*

Ticket 4 — Lee este stack trace

Te llega: `TypeError: Cannot read properties of undefined (reading 'map') at listTasks (src/services/tasks.ts:14) at handler (src/routes/tasks.ts:8).` ¿Qué pasó y dónde miras primero?
Solución. Se lee de arriba abajo: *qué* (`undefined.map` — algo que creíste array vino vacío), *dónde* (`tasks.ts:14` — la primera línea tuya en el stack), *desde quién* (el handler de `routes/tasks.ts:8`). La BD devolvió `undefined` donde esperabas `rows[]` — típico cuando la query falla silenciosamente o se desestructura mal. → *cap-06*

Ticket 5 — Saca la URL a variable de entorno

Te llega: "Hay `https://api.stripe.com` quemada en el código. En staging apunta a otro host. Corrígelo."
Solución.

```
API_URL = os.environ["API_URL"]   # fails fast if missing - better at boot
# requests.get(f"{API_URL}/charges")
```

Config por entorno, no por código: el mismo artefacto corre en dev, staging y prod. El `[]` que lanza error si falta es a propósito — mejor explotar al arrancar que a las 3am. → *cap-02, cap-24*

Ticket 6 — Filtra y ordena esta lista

Te llega: "Del array de tareas, dame las pendientes ordenadas por prioridad (1 primero), solo id y título."
Solución.

```
const open = tasks
  .filter(t => !t.done)
  .sort((a, b) => a.priority - b.priority)
  .map(t => ({ id: t.public_id, title: t.title }));
```

`filter` elige filas, `sort` ordena, `map` proyecta columnas — es `WHERE`/`ORDER BY`/`SELECT` en memoria. El pipeline de colecciones ES SQL mental. → *cap-00, cap-09*

Ticket 7 — Convierte este callback a async/await

Te llega: código legacy con `fetch(url).then(r => r.json()).then(data => ...)`. Reescríbelo legible.
Solución.

```
const res = await fetch(url);
const data = await res.json();
// same Promise chain - await is syntax for .then(), not new machinery
```

`.then()` y `await` son la *misma* asincronía con distinta sintaxis. `await` se lee como pasos; `.then()` se encadena. Saber ambas = leer código viejo y escribir nuevo. → *fe-05*

Ticket 8 — Cancela este fetch si el usuario navega

Te llega: "En la pantalla de detalle, si el usuario sale antes de que responda la API, el setState sobre un componente muerto ensucia la consola."
Solución.

```
const ctrl = new AbortController();
fetch(url, { signal: ctrl.signal }).then(/* ... */);
return () => ctrl.abort();   // cleanup del useEffect
```

`AbortController` es el "cancelar" de fetch — en React va en el cleanup del `useEffect`. → *fe-05*

Ticket 9 — POST con JSON

Te llega: "Crea la tarea llamando `POST /tasks` con `{title: '...'}` — el servidor espera JSON."
Solución.

```
const res = await fetch("/tasks", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title }),      // serialize - don't interpolate
});
```

Tres piezas obligatorias: `method`, el header que declara el formato, y el body *serializado* (string, no objeto). → *fe-05, cap-03*

Ticket 10 — Distingue estos errores

Te llega: el endpoint devuelve a veces 400, 401, 403, 404, 422, 500. Para cada uno: ¿quién tiene la culpa y qué revisas?
Solución. 400/422: el *request* está mal (tu body/schema) → revisa lo que envías. 401: no autenticado → el token. 403: autenticado sin permiso → roles. 404: no existe (o no debes saber que existe). 500: *ellos* — revisa logs del servidor, no tu cliente. La familia 4xx es "arregla tu pedido"; 5xx es "avisar al backend". → *cap-00, cap-06*

G.2. Nivel 2 · El día a día: endpoints

Tu primera semana construyendo: un CRUD que no avergüence.

Ticket 11 — POST que crea de verdad

Te llega: "`POST /tasks` debe crear la tarea, devolver 201 con el objeto creado, y el `public_id` (no el id interno)."
Solución.

```
@app.post("/tasks", status_code=201)
def create_task(body: TaskIn, user=Depends(get_current_user)):
    row = create_task_service(user.id, body.title)
    return TaskPublic(public_id=row["public_id"], title=row["title"],
                      done=row["done"], created_at=row["created_at"])
```

201 + el recurso creado con su forma *pública* — el `id` interno no sale. → *cap-03, cap-05*

Ticket 12 — Valida o devuelve 422

Te llega: "Están entrando tareas sin título y con prioridad 99. Rechaza en la entrada con errores por campo."
Solución.

```ts
const TaskIn = z.object({
  title: z.string().min(1).max(200),
  priority: z.number().int().min(1).max(5).default(3),
});
const parsed = TaskIn.safeParse(req.body);
if (!parsed.success) throw new ValidationError(parsed.error.issues);
// → 422 {"error":{"code":"VALIDATION","fields":[{"field":"priority",...}]}}
```

Schema declarativo una vez, validación antes del handler — el body malo nunca toca la BD. → *cap-04*

Ticket 13 — El 404 correcto

Te llega: "`GET /tasks/{id}` devuelve el objeto aunque sea de otro usuario. Y cuando no existe, devuelve un stack trace."
Solución.

```go
task, err := getTask(ctx, user.ID, publicID)   // WHERE public_id AND user_id
if errors.Is(err, pgx.ErrNoRows) {
    writeError(w, 404, `{"error":{"code":"TASK_NOT_FOUND"}}`)
    return
}
```

Dos fixes en uno: la query filtra por dueño (cap. 17) y el error pasa por el mapper — el cliente ve el formato `{error:{code}}`, el stack queda en logs. → *cap-05, cap-06, cap-17*

Ticket 14 — DELETE: 204 o 404, nunca 200 con basura

Te llega: "DELETE responde 200 con la tarea borrada. Y si no existe, 200 igual con `null`. Estándar, porfa."
Solución.

```sql
DELETE FROM tasks WHERE public_id = $1 AND user_id = $2;
-- rows affected: 1 → 204 No Content · 0 → 404
```

204 sin body = "borrado, no hay nada que devolver". 0 filas = 404 (no existe *o no es tuya* — indistinguibles a propósito). → *cap-03, cap-17*

Ticket 15 — Pagina esta lista

Te llega: "La lista de tareas ya son 40.000. Pagina — y no con `?page=` que se degrada."
Solución.

```sql
SELECT public_id, title, created_at FROM tasks
WHERE user_id = $1 AND (created_at, id) < ($2, $3)
ORDER BY created_at DESC, id DESC LIMIT 21;   -- 21: one extra to know if there's a next page
```

Respuesta: `{items, next_cursor}` — el cursor es `created_at,id` de la última fila. El índice salta directo; OFFSET cuenta y descarta. → *cap-12*

Ticket 16 — El contrato de salida

Te llega: "`GET /me` está devolviendo `password_hash` en el JSON. Literalmente."
Solución.

```python
class UserPublic(BaseModel):          # the output contract
    public_id: str
    email: str
    created_at: datetime
    # password_hash doesn't exist here - can't leak
```

Whitelist, no blacklist: defines lo que *sí* sale. `SELECT *` + devolver la fila es cómo se filtran secretos. → *cap-05, cap-14*

Ticket 17 — PATCH parcial

Te llega: "El toggle de 'hecha' manda la tarea entera. Quiero `PATCH /tasks/{id}` que solo acepte `{done}`."
Solución.

```javascript
const PatchTask = z.object({ done: z.boolean() });
// PATCH = partial update: only declared fields, only changed ones
```

PATCH declara *solo* los campos mutables — `title` no viaja si no cambia. Más seguro que PUT-por-costumbre: menos superficie para pisar datos. → *cap-03, cap-04*

Ticket 18 — Errores consistentes

Te llega: "Unos endpoints devuelven `{message}`, otros `{errors}`, otros un string. Un solo formato, todos."
Solución.

```json
{"error": {"code": "TASK_NOT_FOUND", "message": "task not found"}}
```

Un mapper central (cap. 6): toda excepción de dominio se traduce a ese formato — `code` estable para el cliente programático, `message` para humanos. El frontend escribe UN parser de errores. → *cap-06*

Ticket 19 — Filtros por query param

Te llega: "`GET /tasks?done=false&priority=1` — dos filtros combinables."
Solución.

```
SELECT ... FROM tasks
WHERE user_id = $1
  AND ($2::boolean IS NULL OR done = $2)
  AND ($3::int IS NULL OR priority = $3)
ORDER BY created_at DESC;
```

Parámetros opcionales: si no vienen, el filtro se desactiva (`IS NULL OR`). La validación del query param también es schema — `done=banana` → 422. → *cap-03, cap-09*

Ticket 20 — Health checks reales

Te llega: "El orquestador necesita saber si estamos vivos, y yo necesito saber si la BD responde — sin que uno dispare alarmas del otro."
Solución.

```
@app.get("/health")          # liveness: process up - cheap, for the orchestrator
def health(): return {"status": "ok"}

@app.get("/health/deep")     # readiness: dependencies reachable - for you
def health_deep():
    conn.execute("SELECT 1")
    return {"status": "ok", "db": "up"}
```

`/health` barato para el load balancer; `/health/deep` toca la BD para diagnóstico real. Separarlos evita que un flap de BD tumbe instancias sanas. → *cap-33*

G.3. Nivel 3 · Base de datos

Te dejan tocar la BD. Cada ticket es una query o una migración.

Ticket 21 — La query del reporte

Te llega: "Necesito: tareas abiertas de cada usuario, cuántas son — usuarios con cero también cuentan."
Solución.

```sql
SELECT u.email, COUNT(t.id) AS open_tasks
FROM users u
LEFT JOIN tasks t ON t.user_id = u.id AND t.done = FALSE
GROUP BY u.id, u.email
ORDER BY open_tasks DESC;
```

`LEFT JOIN` porque "con cero cuentan" — `INNER` los borraría. El `COUNT(t.id)` cuenta filas emparejadas: NULL (sin tareas) cuenta 0. → *cap-09*

Ticket 22 — Tareas con sus tags en UNA query

Te llega: "El endpoint de detalle hace 1 query por tarea más 1 por sus tags. Una sola query."
Solución.

```sql
SELECT t.public_id, t.title, tg.name AS tag
FROM tasks t
LEFT JOIN task_tags tt ON tt.task_id = t.id
LEFT JOIN tags tg ON tg.id = tt.tag_id
WHERE t.public_id = $1;
-- 3 rows if 3 tags - group in code: one task, tag list
```

Una tarea con 3 tags devuelve 3 filas — agrupas en código. Una ida a la BD, cero N+1. → *cap-09, cap-12*

Ticket 23 — Índice que falta

Te llega: "`WHERE user_id=? AND done=? ORDER BY created_at DESC` va a 400ms con 2M de filas. El EXPLAIN dice Seq Scan."
Solución.

```sql
CREATE INDEX CONCURRENTLY idx_tasks_user_done_created
  ON tasks (user_id, done, created_at DESC);
-- re-EXPLAIN: Index Scan, ~0.5ms
```

Igualdad primero (`user_id`, `done`), orden al final (`created_at`). `CONCURRENTLY` en prod: no bloquea escrituras mientras se construye. → *cap-12*

Ticket 24 — Migración aditiva

Te llega: "Hay que agregar `tasks.completed_at TIMESTAMPTZ`. Prod tiene 500k filas. Sin downtime."
Solución.

```sql
-- 000003_add_completed_at.up.sql
ALTER TABLE tasks ADD COLUMN completed_at TIMESTAMPTZ;
-- down: ALTER TABLE tasks DROP COLUMN completed_at;
```

Aditivo puro: filas viejas quedan NULL, nada se reescribe, la migración viaja con el código que la usa en el mismo PR. → *cap-11*

Ticket 25 — Todo o nada

Te llega: "Al completar una tarea hay que: marcarla, registrar el evento y dar el punto. Si el paso 2 falla, hoy queda a medias."
Solución.

```python
with pool.connection() as conn:
    with conn.transaction():                    # BEGIN
        conn.execute("UPDATE tasks SET done=TRUE WHERE public_id=%s", (tid,))
        conn.execute("INSERT INTO events (type, ref) VALUES ('task.done', %s)", (tid,))
        conn.execute("UPDATE users SET points=points+1 WHERE id=%s", (uid,))
    # exit → COMMIT; exception → ROLLBACK, all or nothing
```

Una conexión, un BEGIN, los tres writes juntos. El `with` garantiza el ROLLBACK si algo lanza. → *cap-13*

Ticket 26 — Caza el N+1

Te llega: "En logs veo que `GET /projects` ejecuta 61 queries. El código tiene un loop con tres `await` dentro."
Solución. Tres N+1 anidados: $1 + 20{\times}3 = 61$. La cura — una query con agregación:

```sql
SELECT p.id, u.email AS owner, COUNT(DISTINCT t.id) AS task_count,
       array_agg(DISTINCT tg.name) AS tags
FROM projects p
JOIN users u ON u.id = p.owner_id
LEFT JOIN tasks t ON t.project_id = p.id
LEFT JOIN project_tags pt ON pt.project_id = p.id
LEFT JOIN tags tg ON tg.id = pt.tag_id
GROUP BY p.id, u.email;
```

→ *cap-12*

Ticket 27 — Mata la concatenación

Te llega: (code review) `cursor.execute("SELECT * FROM users WHERE email = '" + email + "'")` — el junior dice "funciona".
Solución.

```
cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
```

El dato viaja por canal separado — `' OR '1'='1` se trata como texto literal, no como SQL. En code review: "funciona" no es el criterio; "es segura" sí. → *cap-09*

Ticket 28 — Diseña el FK

Te llega: "`comments` cuelga de `tasks` y de `users`. ¿Qué ON DELETE y por qué?"
Solución.

```
task_id BIGINT NOT NULL REFERENCES tasks(id) ON DELETE CASCADE,
user_id BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
```

CASCADE en ambos: comentario sin tarea ni autor no tiene sentido. Alternativa defendible: `SET NULL` en `user_id` si los comentarios deben sobrevivir al usuario borrado (columna nullable entonces). Lo que no se vale: que el framework lo elija sin que tú lo decidieras. → *cap-08*

Ticket 29 — El UPDATE condicional

Te llega: "Dos usuarios completan la misma tarea a la vez y ambos ganan el punto. Solo el primero debe ganar."
Solución.

```
UPDATE tasks SET done = TRUE
WHERE public_id = $1 AND done = FALSE
RETURNING id;
-- first tx: 1 row → award point · second: 0 rows → "already completed"
```

Compare-and-set: el `WHERE` convierte check-then-act (con carrera) en un acto atómico. No necesitas `FOR UPDATE` cuando el UPDATE *es* la condición. → *cap-13*

Ticket 30 — Replica esta query en tu stack

Te llega: "Esta query anda en `psql` — pásala al servicio con el driver, parametrizada, mapeada al struct."
Solución (Go).

```
rows, err := pool.Query(ctx,
    `SELECT public_id, title, done FROM tasks
     WHERE user_id = $1 AND done = FALSE
     ORDER BY priority`, userID)
if err != nil { return nil, err }
defer rows.Close()
return pgx.CollectRows(rows, pgx.RowToStructByName[Task])
```

Pool prestado, `$1` parametrizado, `defer rows.Close()` devuelve la conexión, `CollectRows` mapea a structs. → *cap-10*

G.4. Nivel 4 · Auth y seguridad

Te confían la puerta. Cada error aquí es un incidente.

Ticket 31 — Registro que no traiciona

Te llega: "POST /auth/register — y obvio, la contraseña no puede quedar legible."
Solución.

```
const passwordHash = await bcrypt.hash(password, 10);
await pool.query(
  "INSERT INTO users (email, password_hash) VALUES ($1, $2)",
  [email, passwordHash],
);
```

bcrypt con salt por usuario y costo 10. La clave en claro vive solo durante el request — nunca en la tabla, nunca en logs. → *cap-14*

Ticket 32 — Login + cookie

Te llega: "Login que devuelva el JWT — en cookie, no en el body. Y el mismo error si falla email o password."
Solución.

```
token = jwt.encode({"sub": str(user["id"]), "exp": now()+timedelta(minutes=15)},
                   SECRET, algorithm="HS256")
response.set_cookie("token", token, httponly=True, secure=True,
                    samesite="lax", max_age=900)
# wrong email OR wrong password → same 401 "invalid credentials"
```

httpOnly contra XSS, Secure para HTTPS, SameSite contra CSRF, `exp` corto. Error genérico = no enumerar usuarios. → *cap-15*

Ticket 33 — Protege el grupo, una línea

Te llega: "Las 8 rutas de /tasks y /projects necesitan auth. No copies la verificación en cada una."
Solución.

```
app.use("/tasks", auth);
app.use("/projects", auth);
// or in FastAPI: user = Depends(get_current_user) per route
// or in Go: mux.HandleFunc("GET /tasks", auth(listTasks))
```

El middleware verifica una vez, propaga el usuario, corta con 401. Proteger = una palabra, no un copipega. → *cap-16*

Ticket 34 — Lo ajeno no existe

Te llega: "`GET /tasks/42` devuelve 403 si es de otro. Se puede enumerar qué tareas existen probando ids."
Solución.

```
SELECT ... WHERE public_id = $1 AND user_id = $2
-- 0 rows → 404, identical to "doesn't exist"
```

404 para lo ajeno: "no existe" y "no es tuya" producen respuestas byte a byte iguales — no hay nada que enumerar. El 403 se reserva para miembros con rol insuficiente (cap. 17). → *cap-17*

Ticket 35 — Roles por proyecto

Te llega: "Los proyectos se comparten: viewer solo lee, editor edita, admin todo. El rol es *de ese* proyecto, no global."
Solución.

```
CREATE TABLE project_members (
  project_id BIGINT REFERENCES projects(id) ON DELETE CASCADE,
  user_id BIGINT REFERENCES users(id) ON DELETE CASCADE,
  role TEXT NOT NULL CHECK (role IN ('viewer','editor','admin')),
  PRIMARY KEY (project_id, user_id)
);
```

- un factory `requireRole("editor")` que consulta la membresía: no-miembro → 404, rol insuficiente → 403. → *cap-17*

Ticket 36 — Rate limit al login

Te llega: "Están probando contraseñas contra /auth/login a máquina."
Solución.

```
POST /auth/login → rate_limit(5 req/min per IP AND per email)
  attempt #6 → 429 {"error":{"code":"RATE_LIMITED"}} + Retry-After: 42
```

Por IP *y* por cuenta (el botnet tiene IPs, no tu cuenta). Antes de bcrypt — no gastes el hash caro en intentos que rechazarás. → *cap-18*

Ticket 37 — Arregla el CORS

Te llega: "El frontend en :5173 no puede llamar la API en :8000 — 'blocked by CORS policy' y con cookies encima."
Solución.

```
Access-Control-Allow-Origin: http://localhost:5173   ← exact, never * with credentials
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Content-Type
```

Con cookies, `*` está prohibido por el estándar — se nombra el origen exacto. Y recuerda: CORS lo enforcea el navegador; curl no pide permiso. → *cap-18*

Ticket 38 — Refresh con rotación

Te llega: "Los refresh tokens viven 30 días y se reusan — si uno se filtra, es un mes de sesión regalada."
Solución.

```
if session.used_at or session.revoked_at:
    revoke_family(session.user_id)      # used token came back = theft
    raise UnauthorizedError("session reuse detected")
mark_used(session.id)
return issue_access(), issue_refresh()  # new pair every time
```

Cada canje mata el viejo y emite uno nuevo. El viejo reapareciendo = robo detectado → sesión entera revocada. → *cap-18*

Ticket 39 — El refresh también es secreto

Te llega: "La tabla sessions guarda el refresh token en texto plano."
Solución.

```
refresh_hash TEXT NOT NULL UNIQUE,   -- store the hash, send the token
```

Misma lección que contraseñas: si `sessions` se filtra, tokens en claro = sesiones regaladas. Se guarda `sha256(token)`; el opaco viaja al cliente. → *cap-14, cap-18*

Ticket 40 — 'Cerrar sesión en todos lados'

Te llega: "El usuario perdió su laptop. Quiere matar todas sus sesiones ya."
Solución.

```
UPDATE sessions SET revoked_at = now()
WHERE user_id = $1 AND revoked_at IS NULL;
```

Por eso el refresh es *persistido*: revocar es un UPDATE. Los access viven hasta su `exp` (~15 min — el precio conocido del híbrido). Re-pedir contraseña antes de ejecutar esto: es operación sensible. → *cap-18*

G.5. Nivel 5 · Senior

Donde el ticket es un problema, no una instrucción. Aquí se gana el título.

Ticket 41 — El test ya está escrito (TDD)

Te llega: "Aquí está el test. Hazlo pasar con lo mínimo — no escribas una línea que el test no pida."

```
def test_create_task_rejects_empty_title():
    with pytest.raises(ValidationError):
        create_task(user_id=1, title="")
```

Solución.

```
def create_task(user_id: int, title: str) -> Task:
    if not title.strip():
        raise ValidationError("title is required")
    return insert_task(user_id, title.strip())
```

Esto es TDD: el test *es* la especificación. Red (falla) → Green (lo mínimo que pasa) → Refactor (mejora sin romper). Programar "al revés" — primero qué debe ser verdad, luego el código. → *cap-27*

Ticket 42 — TDD otra vez: contrato de paginación

Te llega: "El test dice: `GET /tasks` devuelve `{items: [...], next_cursor: str|null}` y con `?cursor=` continúa la lista. Hazlo pasar."
Solución.

```python
def list_tasks(user_id: int, cursor: str | None):
    where = "AND (created_at, id) < (%s, %s)" if cursor else ""
    params = (user_id, *decode(cursor)) if cursor else (user_id,)
    rows = fetch(f"SELECT ... WHERE user_id=%s {where} LIMIT 21", params)
    next_cursor = encode(rows[20]) if len(rows) == 21 else None
    return {"items": rows[:20], "next_cursor": next_cursor}
```

El test definió el contrato *antes* que el código — por eso TDD produce APIs pensadas desde el consumidor. → *cap-12, cap-28*

Ticket 43 — Refactor: el handler gordo

Te llega: "Este handler valida, consulta la BD, decide reglas y serializa — 80 líneas. Sácale capas."
Solución.

```
handler:   parse → call service → status code          (5 líneas)
service:   business rules, domain errors               (decisiones)
storage:   SQL + row mapping                           (persistencia)
schemas:   TaskIn / TaskPublic                         (contratos)
```

El handler queda *tonto*: traduce HTTP y delega. La regla de negocio se puede testear sin HTTP ni request falso — es la paga real del refactor. → *cap-07*

Ticket 44 — POST idempotente

Te llega: "El cliente reintenta `POST /payments` tras timeout y cobra dos veces."
Solución.

```python
key = request.headers["Idempotency-Key"]
if existing := find_by_idempotency_key(key):
    return existing, 200                          # same request → same answer
result = process_payment(...)
store_idempotency(key, result)
return result, 201
```

El cliente manda una llave única por *operación* (no por intento); el servidor recuerda el resultado y lo repite en reintentos en vez de re-ejecutar. Así funcionan Stripe y toda API de pagos seria. → *cap-21*

Ticket 45 — 'Se puso lento, arregla'

Te llega: "El listado pasó de 200ms a 2s después del release de ayer. Sin más info."
Solución. El método, en orden: (1) ¿qué cambió? — el diff del release. (2) Medir: logs/APM → qué endpoint; `EXPLAIN ANALYZE` → qué hace la query. (3) Sospechosos usuales: Seq Scan nuevo (¿índice perdido?), N+1 (¿un loop nuevo?), lock contention (¿una transacción larga?). (4) Fix +

misma medición para confirmar. Lo que nunca: más máquina a un problema no medido. → *cap-12, cap-23*

Ticket 46 — Elige la cimentación

Te llega: "Un webhook de facturación llega ~50 veces al día con picos. ¿VPS, PaaS o Lambda? Defiende tu respuesta."
Solución. Lambda/FaaS: eventos irregulares, pagas por ejecución — un proceso 24/7 por 50 eventos/día es un taxi corriendo en la puerta. Si los picos fueran sostenidos y altos → PaaS/contenedor. La defensa no es la elección, es el *criterio*: forma del tráfico + estado + operación disponible. → *cap-01, nu-05*

Ticket 47 — Escribe el ADR

Te llega: "En 6 meses nadie va a saber por qué elegiste Postgres con JSONB para los documentos. Documéntalo."
Solución.

```
# ADR-03: Postgres + JSONB para documentos
## Contexto: contenido variable + miembros/permisos relacionales
## Decisión: JSONB en documents.content, relacional para el resto
## Alternativas: MongoDB (pierde FKs), columnas fijas (rígido)
## Consecuencias: +flexibilidad con índices GIN; -validación en app
```

Media página: contexto, decisión, alternativas descartadas, precio aceptado. El código dice qué; el ADR dice por qué. → *cap-36*

Ticket 48 — Webhook receptor

Te llega: "Vamos a recibir webhooks de un proveedor. Que nadie pueda falsificarlos y que si fallamos reintente bien."
Solución.

```
signature = request.headers["X-Signature"]
expected = hmac_sha256(WEBHOOK_SECRET, request.body)
if not hmac.compare_digest(signature, expected):   # verify BEFORE parsing
    return 401
# respond 200 fast → process async (job queue): provider retries on failure
```

Firma HMAC con secreto compartido — verificas *antes* de procesar. Y respondes rápido: el procesamiento va a una cola — si tardas, el proveedor reintenta y duplicas trabajo. → *cap-22*

Ticket 49 — El email que no bloquea

Te llega: "El registro tarda 3s porque el email de bienvenida se envía dentro del request."
Solución.

```
# in the register service - respond first, work after
insert_user(...)
enqueue("send_welcome_email", {"user_id": user.id})   # job queue
return user                                            # 201 in ~50ms
```

El response no espera al email: `enqueue` lo deja para un worker (BullMQ / Celery / asynq). "Responder ya, terminar después" — el patrón de toda tarea que excede la paciencia HTTP. → *cap-21*

Ticket 50 — Code review de un junior

Te llega: este endpoint en review. Lista qué cambiarías antes de aprobar:

```
@app.get("/users/{id}")
def get_user(id: str):
    row = conn.execute(f"SELECT * FROM users WHERE id = {id}").fetchone()
    return row
```

Solución. Cinco hallazgos: (1) `f-string` = inyección SQL → `%s` parametrizado. (2) `SELECT *` + devolver la fila → `password_hash` se filtra: contrato `UserPublic`. (3) Sin autenticación ni ownership → `WHERE id =%s AND id =%s` con el usuario del token, 404 a lo ajeno. (4) Sin manejo de `None` → 404 estructurado, no 500. (5) `id` como str → tipo en la firma para validación. Un endpoint de 3 líneas enseña cinco capítulos. → *caps. 5, 9, 16, 17*

G.6. Cierre del apéndice

50 tickets después, el patrón es visible: el trabajo real no es "saber sintaxis" — es **reconocer qué tipo de problema es cada ticket** y saber qué capítulo lo resuelve. Para seguir: convierte cualquiera de estos en un prompt para tu AI (el formato de los "Prompt listo" de cada capítulo) y compara su respuesta con la solución de aquí. Donde difieran, aprendes dos veces.

G.7. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: string

Un **string** es texto entre comillas: `"hola"`. El nombre viene de "cadena de caracteres" — una secuencia de letras. Todo lo que llega de un formulario o una URL llega como string, aunque parezca número.

Explicación de: booleano

Un **booleano** es un valor de dos estados: `true` o `false`. Es el resultado de toda comparación (`age > 18`) y lo que los `if` evalúan. Nombrado por George Boole, el matemático de la lógica.

Explicación de: null / None / nil

null (TS), `None` (Py), `nil` (Go) = "aquí no hay valor". Es la respuesta a "¿qué devuelvo cuando no hay nada?" — y la fuente del bug más famoso de la historia (su inventor lo llamó "el error del billón de dólares"). Por eso el código revisa `if x is not None`.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Explicación de: localhost / 127.0.0.1

localhost significa "esta misma máquina" — es la dirección que tu computadora usa para hablarse a sí misma. Cuando desarrollas, el "servidor" y el "cliente" viven en tu laptop: por eso todo es `localhost:3000`.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

H H. Preguntas de entrevista: diseño, arquitectura y bases de datos

Las que separan 'sé hacer endpoints' de 'sé diseñar sistemas'

> **Cómo usar esto**
>
> Repaso activo: lee la pregunta, respóndela **en voz alta**, *después* despliega la respuesta modelo. Las respuestas son de largo "hablado" — en una entrevista nadie quiere un ensayo, quieren oír cómo piensas. Si una te cuesta, la referencia te dice qué capítulo releer.

H.1. Diseño y arquitectura

> P: Walk me through building a fullstack app from scratch.
>
> **R:** "Empiezo por el dominio: entidades y relaciones en un diagrama ER antes de cualquier código. Luego el contrato — rutas, schemas de entrada/salida, códigos de error — porque el frontend y el backend pueden construirse en paralelo contra ese contrato. Arquitectura por capas: rutas traducen HTTP, servicios deciden, storage persiste. Primero lo que demuestra el riesgo: en Nexus fue el versionado de documentos, no el login." *El entrevistador busca: orden (dominio→contrato→capas), no "instalo React".* → *cap-36*

> P: What is a REST API and why is it used?
>
> **R:** "Una API donde los recursos tienen URL (`/tasks/42`), los verbos HTTP llevan la acción (GET lee, POST crea, DELETE borra), el servidor es stateless — cada request trae todo lo necesario — y los status codes son parte del contrato (201, 422, 404). Se usa porque es simple, cacheable y universal: cualquier cliente HTTP lo habla." → *cap-03*

> P: How would you design an API?
>
> **R:** "Contrato primero: recursos en plural y minúscula, verbos que respetan semántica (GET nunca muta), errores estructurados con `code` estable + `message`, paginación por cursor si la lista crece, versionado en la URL o headers cuando el contrato tenga que cambiar. El schema de entrada y

de salida se definen una sola vez — y la salida filtra lo interno (`password_hash` jamás sale).” → *cap-03, 04, 05*

P: REST vs GraphQL — when would you pick each?

R: “REST cuando el dominio es recursos claros y quieres caché HTTP y simplicidad — el default. GraphQL cuando el cliente necesita *seleccionar campos* o componer muchas entidades en un request (apps con vistas muy distintas) — pagas un runtime de queries y caché más difícil. RPC/actions cuando el dominio no es recursos sino *operaciones* (`/tasks/complete`). En Nexus elegí REST: el dominio son documentos y miembros.” → *cap-03*

P: Explain MVC / layered architecture and why it matters.

R: “Separación por responsabilidad: la capa HTTP traduce requests, el servicio decide reglas de negocio sin saber de HTTP, el storage habla SQL sin saber de negocio. Las dependencias apuntan hacia adentro — el servicio no importa Express. Lo compras: cada capa se puede testear y cambiar sola. Lo pagas: más archivos para un CRUD simple.” → *cap-07*

P: What is dependency injection and why is it useful?

R: “En vez de que el componente *cree* sus dependencias, se las dan. En FastAPI `Depends(get_current_user)` declara la dependencia en la firma y el framework la resuelve. Sirve para dos cosas: el handler no sabe cómo se construye lo que usa (puedo cambiar la implementación), y en tests puedo inyectar una versión falsa — una sesión de mentira, una BD en memoria.” → *cap-16*

P: How would you design a URL shortener?

R: “El clásico de system design junior. Modelo: `links(id, slug, long_url, user_id, created_at)`. Redirect: `GET /{slug}` → 301 a la URL — un index único en slug hace el lookup O(log n). Generación del slug: base62 del id o hash corto con reintento en colisión. Escala: lecturas » escritas, así que caché caliente (Redis) delante y la BD aguanta. La pregunta de seguimiento siempre es: ¿qué pasa cuando el slug colisiona y cómo lo resuelves?”

P: Why document architecture decisions (ADRs)?

R: “Porque el código dice *qué* y el ADR dice *por qué*. En seis meses nadie recuerda por qué se eligió snapshot+diff y no CRDT — el ADR captura el contexto, las alternativas descartadas y el costo aceptado. Corta: media página por decisión. En Nexus hay tres: capas, modelo de versionado, y last-writer-wins.” → *cap-36*

H.2. Bases de datos

P: What is a database index and why can it make a query slower?

R: "Una estructura aparte (B-tree) que mantiene columnas ordenadas con punteros a las filas — como el índice de un libro: buscas sin leer todo. Puede 'enlentecer' porque cada INSERT/UPDATE/DELETE tiene que actualizar también el índice — los índices aceleran lecturas y cobran escrituras. Y si tu query devuelve media tabla, Postgres ignora el índice porque el seq scan es honestamente más barato." → *cap-12*

P: What does ACID mean?

R: "Las cuatro promesas de una transacción. **Atomicidad**: todo o nada — el INSERT y el UPDATE van juntos o ninguno. **Consistencia**: la BD nunca queda violando sus constraints. **Aislamiento**: transacciones concurrentes no se ven a medias. **Durabilidad**: lo confirmado sobrevive al crash — está en disco. El ejemplo universal: transferir dinero — debitar sin acreditar no puede pasar." → *cap-13*

P: What is a transaction and when do you need one?

R: "Un grupo de operaciones que se confirman como una unidad: `BEGIN`, las queries en UNA conexión, `COMMIT` o `ROLLBACK`. La necesitas cuando varios writes son un solo hecho de negocio — crear tarea + registrar evento + descontar crédito. La trampa: la transacción vive en una conexión del pool — si usas `pool.query()` por query, cada una viaja por cable distinto y no hay transacción." → *cap-13*

P: Explain the N+1 problem.

R: "Una query para la lista + una query por cada elemento: 20 tareas = 21 round trips. Invisible en dev, asesino en prod — y los ORMs lo disfrazan de código limpio (`task.tags` dispara una query). La cura: una sola query con JOIN, o `WHERE task_id IN (...)`, agrupando en código. Detector: loguea queries en dev — un request que dispara N queries es la alarma." → *cap-12*

P: How do you optimize a slow SQL query?

R: "Medir antes de tocar: `EXPLAIN ANALYZE` muestra el plan real — busco `Seq Scan` en tablas grandes (falta índice), `Rows Removed by Filter` altos (leyó millones para entregar veinte), y estimados que no coinciden con realidad (estadísticas viejas → `ANALYZE`). Arreglo típico: índice compuesto con las columnas de igualdad primero y la de ORDER BY al final. Luego verifico que el plan cambió — el índice que Postgres ignora no existe." → *cap-12*

P: Index scan vs sequential scan — when is each right?

R: "Seq scan lee la tabla entera — correcto cuando la tabla es pequeña o la query devuelve gran parte de ella. Index scan baja por el árbol a las filas exactas — correcto cuando filtras poco de mucho. Un seq scan en `EXPLAIN` no es automáticamente un bug; lo es cuando la tabla creció y el plan no cambió." → *cap-12*

P: What is a composite index and how do you order its columns?

R: "Un índice sobre varias columnas: `(user_id, done, created_at)`. El orden importa porque el árbol está ordenado por la primera, luego la segunda dentro de cada valor — regla práctica: **columnas de igualdad primero, rango/orden al final**. `WHERE user_id=? AND done=? ORDER BY created_at` usa el índice completo; `WHERE done=?` solo, no — el índice no 'empieza' por ahí." → *cap-12*

P: How do you change a schema in production without downtime?

R: "Expand-contract en migraciones: agregar lo nuevo (columna, tabla) sin tocar lo viejo → el código escribe en ambos → backfill de datos → deploy que solo usa lo nuevo → última migración retira lo viejo. Cada paso es seguro solo. Reglas: `ADD COLUMN` es barato (Postgres 11+: instantáneo con default), `ALTER TYPE/DROP` bloquean → se hacen por pasos; índices en tablas grandes con `CREATE INDEX CONCURRENTLY`." → *cap-11*

P: What is a connection pool?

R: "Un conjunto de conexiones abiertas a la BD que se *prestan* por request en vez de abrirse y cerrarse — abrir una conexión cuesta handshake+auth+proceso nuevo (~50ms). Con `max: 10` atiendes cientos de usuarios porque cada query dura milisegundos y devuelve su conexión. Si el pool se agota, los requests *esperan en cola* — por eso olvidar `release()`/`close()` cuelga la app a los minutos." → *cap-10*

P: How would you model document versioning (undo/redo) in a DB?

R: "El modelo de Nexus — como git interno: tabla `revisions(document_id, rev, snapshot, created_at)` y el documento guarda `current_revision` como puntero. Undo = mover el puntero atrás; redo = adelante; editar tras undo = truncar el 'futuro' — exactamente `git commit`. Snapshot completo por revisión (no solo diffs) porque el restore es O(1): lees un estado, no re-ejecutas la historia." → *cap-37*

P: SQL vs NoSQL — defend a real choice you made.

R: "En Nexus elegí PostgreSQL aunque los documentos son JSON: (a) la *mayoría* del dominio es relacional — miembros, proyectos, permisos con FKs que la BD enforza; (b) `JSONB` guarda el contenido del documento sin renunciar a lo relacional — es 'lo mejor de los dos' con índices GIN si

hace falta; (c) las revisiones necesitan transacciones fuertes. NoSQL habría sido pereza disfrazada de flexibilidad." → *cap-08, cap-37*

H.3. Vocabulario técnico del capítulo

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: pool de conexiones

Un **pool** es el staff de conexiones a la BD: abrir una conexión por request es caro, así que el pool mantiene ~10–20 abiertas y las presta. El request la usa, la devuelve, y la siguiente la reutiliza.

Explicación de: JOIN

Un **JOIN** combina tablas por su relación: "cada tarea con el nombre de su proyecto". Es la superpotencia relacional — en una query traes lo que en NoSQL serían varios viajes.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: hash / bcrypt

Un **hash** es una función de un solo sentido: `contraseña →$2b10…`. No se puede revertir — por eso las contraseñas se *hashean*, no se *cifran*. bcrypt es lento a propósito: fuerza bruta cara para el atacante.

Explicación de: firma / HMAC

Una **firma** (HMAC) prueba que un mensaje es auténtico y no fue tocado: se calcula con un secreto sobre el contenido. Los webhooks la usan para que verifiques "esto realmente vino de Stripe".

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga

el proceso” = el programa murió.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

I I. Preguntas de entrevista: frontend, tiempo real, seguridad y operación

La mitad «full» del fullstack — y lo que separa funciona de publicable

> **Cómo usar esto**
>
> Repaso activo: responde en voz alta antes de desplegar. Si una te cuesta, la referencia `→ cap. X` te dice qué releer.

I.1. Frontend y tiempo real

> **P: What happens when you type a URL and press Enter?**
>
> **R:** "El clásico — respóndelo en capas: el navegador resuelve el dominio vía DNS → abre conexión TCP → negocia TLS (certificados, el candado) → manda el HTTP request → el servidor (reverse proxy → app → BD) responde HTML → el navegador parsea el DOM, pide CSS/JS, ejecuta el JS que quizá pide más datos vía fetch a la API. Si me piden profundidad, bajo a cualquier capa — el libro entero vive en esa cadena." → *nu-01, fe-01*

> **P: WebSockets vs HTTP — when do you need each?**
>
> **R:** "HTTP es request/response: el cliente pregunta, el servidor responde, se cierra. WebSocket abre un canal *persistente y bidireccional*: el servidor puede empujar datos sin que el cliente pregunte. Necesario cuando el servidor sabe algo que el cliente quiere ya — presencia, colaboración en vivo, notificaciones. Para 'cada tanto' alcanza polling/SSE; WS se justifica cuando la latencia y el push son el producto — en Nexus, dos cursores viéndose en <300ms." → *cap-38*

> **P: How do you implement real-time collaboration?**
>
> **R:** "Conexión WebSocket autenticada (el upgrade lleva la misma cookie httpOnly — la auth aplica). El servidor mantiene un mapa documento→conexiones; cada edición hace broadcast a los demás del documento. Presencia: heartbeats para saber quién está. Conflictos: last-writer-wins es el pragmático — CRDT/OT solo cuando el merge fino por carácter es requisito, que es otro

proyecto de complejidad." → *cap-38*

P: What is the DOM?

R: "La representación del documento HTML como árbol de objetos que JavaScript puede leer y mutar — `document.querySelector`, crear nodos, escuchar eventos. El browser renderiza el DOM, no el HTML: por eso modificar el árbol cambia la pantalla sin recargar. React existe porque mutar el DOM a mano se desordena con estado complejo." → *fe-03*

P: What are React hooks — useState and useEffect?

R: "`useState` guarda estado por componente y re-renderiza al cambiarlo: la UI es función del estado, no comandos de DOM. `useEffect` corre *efectos* tras el render — fetch, suscripciones — con un array de dependencias que dice cuándo re-ejecutarse y una función de limpieza para cancelar (AbortController, unsubscribe). El error clásico: fetch en useEffect sin cleanup → race conditions cuando el usuario navega rápido." → *fe-03, fe-05*

P: How do you manage state across frontend and backend?

R: "Separando dos tipos: **server state** (las tareas viven en la BD — el frontend solo cachea y muestra: TanStack Query/SWR manejan caché, refetch, loading) y **client state** (qué modal está abierto, el tema — useState/context). El error de novato es tratar el server state como si fuera tuyo: copiarlo a un store global y mantenerlo a mano. La fuente de verdad es el servidor; tu UI es una vista cacheada." → *fe-05*

P: Explain event bubbling.

R: "Un evento sube por el árbol: click en un botón dispara handlers del botón, luego del div padre, hasta `document`. Por eso `stopPropagation()` existe — y por eso funciona la delegación: UN listener en la lista padre maneja clicks de mil hijos leyendo `event.target`. La captura es el viaje inverso (de afuera hacia adentro, antes del bubbling)." → *fe-03*

P: CSR vs SSR — what's the difference and when does it matter?

R: "CSR: el servidor manda un HTML casi vacío + JS que construye todo en el navegador — primera pintura lenta, interacción rica después. SSR: el servidor devuelve HTML completo ya renderizado — first paint rápido, SEO real, y luego el JS 'hidrata' para la interactividad. Importa para SEO y tiempo-de-primera-vista; apps detrás de login (dashboards) suelen bastar con CSR, páginas públicas piden SSR." → *fe-06*

P: Walk me through authentication end to end, frontend to backend.

R: "El form manda `POST /auth/login` → el servidor verifica bcrypt → firma un JWT y lo devuelve en cookie httpOnly+Secure+SameSite → el navegador la adjunta sola en cada request → el middleware del servidor verifica firma, carga el usuario, y decide 401 o sigue → el frontend pregunta `GET /me` para saber si hay sesión (nunca lee el token — httpOnly lo impide a propósito). Logout = cookie borrada + sesión revocada en la tabla." → *cap-14..16*

P: What is CORS and how do you handle it?

R: "Política del navegador: una página del origen A no puede leer respuestas del origen B sin permiso explícito del servidor. Para requests 'no simples' el navegador manda primero un `OPTIONS` (preflight) preguntando; el servidor responde `Access-Control-Allow-Origin: <origen exacto>` (+ `-Methods`, `-Headers`, `-Credentials` si hay cookies — que prohiben el `*`). Es el *navegador* el que enforcea: curl no pide permiso — CORS protege al usuario, no a tu API." → *cap-18*

I.2. Seguridad y operación

P: How do you secure an API?

R: "Por capas, como el libro entero: entrada — validación por schema y queries parametrizadas (cero concatenación → cero inyección SQL). Identidad — bcrypt para claves, JWT en cookie httpOnly, sesiones revocables con rotación. Permisos — ownership en la query, 404 a lo ajeno. Superficie — rate limit en /auth, CORS con orígenes explícitos, HTTPS siempre. Secretos — env vars/secret manager, nunca en el repo. Errores — el 500 genérico afuera, el stack trace solo en logs." → *caps. 4, 6, 14–18, 26*

P: JWT vs session cookies — which do you choose?

R: "No es o/es — el diseño serio es híbrido. Access JWT stateless (15 min): el servidor solo verifica firma, escala sin consultar BD — pero no se revoca. Refresh opaco persistido (30 días, tabla sessions): una consulta cada 15 min a cambio de revocación real — 'cerrar sesión en todos los dispositivos' existe. Rotación del refresh: cada uso emite uno nuevo; un viejo reutilizado = robo detectado → sesión entera fuera." → *cap-15, cap-18*

P: How do you store passwords?

R: "No se guardan — se guardan verificadores. bcrypt (o argon2) con salt aleatorio por usuario y costo ~10-12: lento a propósito, ~100ms por intento — imperceptible en login, una eternidad en fuerza bruta. Nunca cifrado (la clave que descifra también se filtra) y nunca MD5/SHA-256 (rápidos = regalo para GPUs). El login hashea lo que escribió el usuario y compara; el error es el mismo para 'no existe' y 'clave mala'." → *cap-14*

P: What is SQL injection and how do you prevent it?

R: "Concatenar datos del usuario en la query: `WHERE email = '" + email + "'` — si escribe `' OR '1'='1` la condición se vuelve verdadera para toda la tabla. Se previene con consultas parametrizadas: `$1`/`%s` envían el dato por un canal separado — la BD lo trata como texto, jamás como código. Regla sin excepciones, incluidos 'valores que yo controlo'." → *cap-09*

P: XSS vs CSRF — what are they and how do you defend?

R: "XSS: JS malicioso corriendo en tu página (un comentario que renderiza `<script>`) — roba lo que JS puede tocar. Defensa: sanitizar entrada, no `innerHTML` con datos de usuario, y cookies httpOnly (el XSS no puede leerlas). CSRF: un sitio maligno hace que el navegador *logueado* del usuario mande requests a tu API — porque la cookie viaja sola. Defensa: `SameSite=lax/strict` + origen verificado. Son ataques gemelos: uno abusa del script en tu página, el otro de la cookie automática." → *cap-15, cap-26*

P: What is rate limiting and where do you apply it?

R: "Un techo de requests por ventana — responde 429 + `Retry-After` cuando se pasa. Lo pones donde cada intento te cuesta o te revela: `/auth/login` (fuerza bruta de claves), `/auth/refresh`, `register`, `forgot-password` (enumeración de emails), códigos OTP, y endpoints caros (exports, reportes). Por IP *y* por cuenta — el botnet tiene mil IPs pero la cuenta objetivo es una." → *cap-18*

P: Your API went from 200ms to 2s. How do you troubleshoot?

R: "La pregunta real de entrevista. Orden: (1) ¿qué cambió? — deploy reciente, migración, crecimiento de datos. (2) Medir, no adivinar: logs/APM dicen qué endpoint; `EXPLAIN ANALYZE` dice qué hace la query — busco Seq Scan nuevo (¿se perdió un índice?), N+1 (¿un loop nuevo?), rows escaneadas explotando. (3) CPU alto + queries iguales → carga o lock contention. (4) Fix y verificación con la misma medición. Lo que nunca hago: tirarle más máquina a un problema que no medí." → *cap-12, cap-23*

P: What are environment variables and why use them?

R: "Configuración que vive fuera del código: `DATABASE_URL`, `JWT_SECRET`, `PORT`. Por dos razones: el mismo artefacto corre en dev, staging y prod (la diferencia es el entorno, no el binario), y los secretos no tocan el repo — el código se publica, las env vars no. En producción viven en un secret manager con rotación y auditoría, no en un `.env` del servidor." → *cap-24, nu-04*

P: Docker — image vs container, and why use it?

R: "La imagen es la receta inmutable (capas: SO base + runtime + deps + tu código); el contenedor es la ejecución viva de una imagen — como clase vs instancia. Sirve porque resuelve 'en mi máquina sí funcionaba': el mismo artefacto corre idéntico en dev, CI y prod. El Dockerfile declara la receta;

docker-compose orquesta app+BD locales para que el onboarding sea un comando." → *cap-30..32*

P: What does your testing strategy look like?

R: "Pirámide honesta: muchos tests de unidad en la capa de servicio (la regla de negocio pura — 'no borrar tarea con subtareas'), tests de integración contra BD real en los repositorios, y pocos end-to-end por los flujos críticos (login→crear→listar). Cobertura donde importa: la frontera de errores y la autorización, no getters. El test como contrato: si rompe la API pública, el test falla antes que el cliente." → *cap-27..29*

I.3. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: objeto / dict / struct

Un **objeto** es una colección de datos con nombre: `{name: "Ana", age: 30}` — cada dato es una *propiedad* (clave → valor). Python los llama `dict`, Go los arma con `struct`, TS con objetos/`interface`.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: clase / struct

Una **clase** es el molde de un objeto: define qué datos y qué métodos tiene. `class Task` es el molde; `new Task()` es una instancia concreta. Go no tiene clases — usa `struct` + métodos sueltos.

Explicación de: return

return hace dos cosas a la vez: devuelve el resultado Y termina la función — lo que esté debajo nunca corre. `return task` = "aquí está el plato, salgo de la cocina".

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

Explicación de: sesión

Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada llamada.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: DOM

El **DOM** es la página como árbol de objetos vivos: HTML es el papel, el DOM es lo que JavaScript puede tocar. `element.textContent = "hola"` cambia el DOM y el navegador repinta — el HTML original no se entera.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: repositorio

Un **repositorio** (repo) es la carpeta del proyecto con todo su historial Git adentro. GitHub/GitLab son servicios que hospedan repos para compartirlos y respaldarlos.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

J J. Preguntas de entrevista: AI en el trabajo y conductual

Las que más crecieron en entrevistas fullstack — y las que este libro entrena sin que lo notes

Cómo usar esto

Repaso activo: responde en voz alta antes de desplegar. Las preguntas de AI no son trivia aparte — cada "Prompt listo para tu AI" del libro fue una lección de prompt engineering. Aquí se convierte en respuesta de entrevista.

J.1. AI y prompt engineering

P: What is prompt engineering and why does it matter for a developer?

R: "Diseñar la entrada que guía al modelo a salidas confiables: contexto, objetivo, restricciones, formato, criterios de aceptación. Importa porque el dev de hoy *dirige* código más que lo escribe de memoria — la diferencia entre 'hazme un endpoint' y un prompt con stack, archivos, edge cases y buenas prácticas es la diferencia entre código que hay que rehacer y código que hay que revisar." → *todos los "Prompt listo" del libro*

P: Zero-shot vs few-shot vs chain-of-thought — when do you use each?

R: "Zero-shot: la instrucción sola — alcanza para tareas comunes. Few-shot: instrucción + ejemplos de entrada/salida — cuando el formato importa (quiero que devuelvas *este* JSON). Chain-of-thought: 'piensa paso a paso' — mejora razonamiento multi-paso, y en la práctica lo pides implícito al pedir 'explícame tu razonamiento antes del código'."

P: What is RAG and when is it better than a bigger prompt?

R: "Retrieval-Augmented Generation: antes de responder, se buscan los documentos relevantes y se inyectan al prompt como contexto. Mejor que meterlo todo cuando el conocimiento es grande, cambia, o es privado — el modelo responde *anclado en datos reales* en vez de alucinar. En términos de este libro: es 'SELECT relevante + prompt', no 'SELECT *' en cada request."

P: Prompt engineering vs fine-tuning — how do you decide?

R: "Prompt primero, siempre: es barato, iterativo y resuelve el 80 % — formato, tono, restricciones, ejemplos. Fine-tuning cuando el prompt no basta a escala: comportamiento consistente que necesitas en millones de llamadas, o un dominio tan específico que los ejemplos no caben en contexto. La pregunta de control: ¿puedo resolverlo con un mejor prompt? Si sí, no entreno nada."

P: What are guardrails in an LLM feature?

R: "Las fronteras que no dependen del modelo: validar la salida (schema — el mismo concepto del cap. 4), filtrar lo que no debe decir, límites de acción (el agente propone, tú apruebas — como los AI CLIs del cap. 0.5), y fallback cuando falla. Un LLM sin guardrails en prod es un handler sin validación: funciona en el demo, falla en el mundo."

P: How do you use AI tools in your daily workflow?

R: "Respuesta honesta: los CLIs de AI son herramientas de la terminal — les doy contexto (archivos, stack, contrato), pido con requisitos y criterios de aceptación (los prompts de cada capítulo son mi plantilla), **reviso el diff como si fuera un PR de un junior**, y nunca les paso secretos. Lo que me hace útil no es tener la AI — es saber verificar lo que produce. Por eso aprendí los fundamentos." → *cap-terminal*

P: How do you verify or evaluate AI-generated code?

R: "Igual que código de cualquier fuente: ¿compila/pasa tests? ¿la query está parametrizada? ¿el endpoint valida entrada? ¿los errores van al mapper? ¿el índice existe? La AI produce borradores rápidos; mi checklist de seguridad/errores/performance del libro es el filtro. La señal de junior es aceptar sin revisar; la señal de senior es saber exactamente qué revisar."

P: Describe the lifecycle of an LLM feature in production.

R: "Definir el problema y el contrato de salida → elegir modelo (costo/latencia/calidad) → estrategia de prompting (system prompt + few-shot) → guardrails y validación de salida → evaluación con casos reales → despliegue con los mismos problemas de siempre: costos, latencia, observabilidad, rate limits del proveedor. Un LLM en prod es una integración externa más — con timeouts, retries y presupuesto." → *cap-19*

J.2. Conductual y proyecto

P: Tell me about a project you built.

R: "Tu respuesta es Nexus — practícala con STAR corto: *Situación*: workspace colaborativo tipo Figma-lite. *Tarea*: backend completo solo. *Acción*: diseñé el ER, contrato OpenAPI primero, capas, versionado tipo git interno con snapshot+puntero, auth híbrida con rotación, WebSockets para presencia, auditoría completa antes de publicar. *Resultado*: API deployada con X endpoints, tests, migraciones, docs. Y lo clave: puedo defender CADA decisión — por qué Postgres, por qué LWW, por qué híbrido de sesiones." → *caps. 36–39*

P: Tell me about a challenging bug and how you resolved it.

R: "STAR con uno real del libro: 'Un endpoint devolvía datos de otros usuarios' — *S*: lista de tareas filtraba mal. *T*: encontrar la fuga. *A*: el test del endpoint pasaba porque devolvía 200 — el bug era de autorización, no de status; el `WHERE` no incluía `user_id`. *R*: fix en la query (no en el if), test nuevo que verifica *de quién* son los datos, y auditoría de los demás endpoints — el patrón se repite." → *cap-17*

P: How do you stay updated with technology?

R: "Honesta y concreta: 'Construyo — leer changelogs no pega como implementar. Cuando algo nuevo aparece lo comparo con lo que ya sé (Fastify vs Express fue un sub-tab, no un mundo nuevo). Uso las herramientas de AI para explorar rápido pero verifico con la doc oficial. Y mantengo este libro vivo: cada concepto nuevo intenta ganarse un lugar en una sección.' "

P: ¿Y si te preguntan algo que no sabes?

R: "La respuesta que más impresiona es honesta: 'Eso no lo he hecho en prod, pero te digo cómo lo atacaría' + el razonamiento en voz alta. El entrevistador evalúa cómo piensas, no tu memoria — 'no sé' seguido de 'pero por analogía con X haría Y' vale más que inventar. Memorizar no es el objetivo de este libro; *reconocer patrones* sí — y eso se defiende solo en la entrevista."

Cómo usar estas secciones

- **Repaso espaciado:** una sección por día, voz alta, antes de desplegar. Si no la respondes, la marcas y vuelves mañana.
- **Las preguntas cambian, los fundamentos no:** cada respuesta modelo apunta a un capítulo — si una te cuesta, ahí está tu lectura.
- **Tu AI CLI es tu entrevistador de práctica:** pégale cualquier pregunta de arriba y pídele que te evalúe.

J.3. Vocabulario técnico del capítulo

Explicación de: array / lista / slice

Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

Explicación de: bucle (loop)

Un **bucle** repite una acción por cada elemento o hasta cumplir una condición: `for task in tasks` hace algo con cada tarea. `map` y `filter` son bucles disfrazados de funciones — transforman/filtran colecciones sin `for` explícito.

Explicación de: sesión

Una **sesión** es la conversación continuada entre tú y el servidor: HTTP no recuerda nada entre requests, así que la sesión (vía cookie o token) es el "pulso de mano" que te identifica en cada llamada.

Explicación de: secreto

Un **secreto** es un dato que no puede publicarse: contraseñas de BD, llaves de API, el secreto que firma los JWT. Viven en `.env` (fuera de git) o en un secret manager — nunca en el código ni en el repo.

Explicación de: deploy / despliegue

Un **deploy** es poner tu código a correr en el servidor: copiar la imagen, iniciar el proceso, verificar que responde. El objetivo es que sea aburrido — automático, repetible, con rollback.

Explicación de: dominio

Un **dominio** es el nombre que compras (midominio.com) y apuntas a tu servidor por DNS. Sin él, tus usuarios tendrían que memorizar una IP. El HTTPS serio requiere dominio.

Explicación de: WebSocket

WebSocket es una conexión que queda *viva*: en vez de preguntar- responder-cerrar como HTTP, ambos pueden hablar cuando quieran. Es como el servidor puede *empujar* — por eso sirve para chat y colaboración en vivo.

Explicación de: latencia / throughput

Latencia = cuánto tarda UNA operación (ms). **Throughput** = cuántas haces por segundo. Se pelean: bajar latencia no siempre sube throughput. La latencia la siente el usuario; el throughput lo paga tu factura.

Explicación de: escalado horizontal / vertical

Vertical: máquina más gorda (más CPU/RAM) — fácil, tiene techo. **Horizontal**: más máquinas — el camino de internet, pero requiere que la app no guarde estado local (por eso todo va a BD/Redis).

Explicación de: producción

Producción (prod) es el entorno real: donde están los usuarios, los datos que importan y las consecuencias. Todo lo demás — local, staging — existe para que los errores ocurran antes de llegar ahí.

Explicación de: compilador / transpilador

Un **compilador** traduce código a otra forma: TypeScript → JavaScript (transpila), Go → binario (compila). Atrapa errores antes de ejecutar. `tsc`, `esbuild`, `go build` son compiladores.

Explicación de: terminal / consola

La **terminal** (consola, línea de comandos) es la interfaz de texto con el sistema operativo: escribes comandos, lees resultados. Es como hablarle a la computadora por cartas en vez de señalar con el mouse.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: índice (index)

Un **índice** es la tabla de contenido de una tabla: sin él, buscar un usuario es leer las 10 millones de filas (*seq scan*); con él, es ir directo a la página. Se crea según cómo se consulta, y cada uno cuesta escrituras.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: autenticación

La **autenticación** responde "¿quién eres?" — login, contraseña, token. Se confunde con autorización ("¿qué puedes hacer?"), que es la pregunta siguiente. Primero te identificas, luego te dejan o no pasar.

Explicación de: autorización / permisos

La **autorización** responde "¿qué puedes hacer?": eres usuario válido (autenticado), pero ¿puedes borrar *esta* tarea? Se decide por rol o por ownership — y se verifica en cada request, no se recuerda.

EXTRA · Lo que los roadmaps dicen sobre carrera

EXTRA Más allá del código, las guías externas coinciden en lo no-técnico —

- **Resolver problemas y comunicación**: el trabajo no termina en código correcto; hay que explicar soluciones al equipo y a otros departamentos
- **Portafolio**: perfil de GitHub activo + proyectos de código abierto
- **Networking**: LinkedIn, meetups, conferencias — "la conexión correcta puede ser la diferencia entre quedarte y conseguir el trabajo soñado"
- **Voluntariado/open source**: experiencia real visible para empleadores mientras no tienes

empleo
- **Recursos de aprendizaje**: freeCodeCamp (introducción gratuita), Khan Academy (fundamentos), MDN Web Docs (tutoriales por nivel), libros como *JavaScript Professional* (Zakas) y cursos prácticos de Node.js (Andrew Mead)

El hilo que los une: *aprendizaje basado en proyectos* — construir cosas reales retiene más que la teoría sola. Es literalmente el diseño de este libro.

K K. Preguntas decisivas: ¿X o Y?

El arte de elegir — criterios, no gustos

> **En una frase**
>
> El trabajo real son decisiones disfrazadas de preguntas: "¿fetch o axios?", "¿React o Vue?", "¿monolito o microservicios?", "¿me quedo en Postgres o me voy a Mongo?". Aquí practicas **decidir con criterio**: cada tarjeta te da la situación, tú eliges en voz alta, y la respuesta te dice qué criterio mandaba — y cuál era la trampa.

K.1. La meta-habilidad: 5 preguntas que resuelven casi cualquier "¿X o Y?"

Antes de las tarjetas, el patrón detrás de todas ellas. Cuando alguien te pregunte "¿usamos X o Y?", corre esta lista en tu cabeza:

> **Explicación de: array / lista / slice**
>
> Un **array** es una colección ordenada de elementos accedidos por posición: `["a","b","c"][0]` es `"a"` (se cuenta desde 0). Python las llama *listas*, Go *slices* — misma idea, distinto acento.

1. **¿Cuál es la *forma* del problema?** Forma del dato (relacional o documento), del tráfico (constante o picos), de la UI (dashboard o página pública). La mayoría de dilemas se resuelven aquí.
2. **¿A qué escala?** Con 10 usuarios casi todo sirve; con 10.000 cambian las respuestas. Decide para la escala *real de hoy* más un margen — no para el Netflix que no eres.
3. **¿Quién lo mantiene?** Tú solo, o un equipo que ya sabe X. La herramienta que tu equipo conoce suele ganarle a la "mejor" que nadie sabe.
4. **¿Qué cuesta cambiar de opinión?** Decisiones de *una vía* (la BD, el proveedor de nube) merecen análisis serio. Decisiones de *dos vías* (fetch→axios, una librería de UI) se cambian en una tarde — no les des semanas.
5. **¿Qué viene incluido?** Ecosistema, librerías, empleos, tutoriales. React no gana por su sintaxis: gana porque el mercado ya está ahí.

Explicación de: base de datos

Una **base de datos** es el programa que guarda datos de forma permanente y los responde rápido. Relacional (Postgres): tablas con relaciones. No relacional: documentos, clave-valor, grafos — cada una para una forma de dato distinta.

Los defaults del libro — la respuesta correcta hasta que tengas una razón para irte:

```
Base de datos → PostgreSQL    API → REST + JSON      Arquitectura → monolito
HTTP client → fetch           Frontend → React       Deploy → managed platform
Estado UI → useState local    Auth → JWT + refresh persistido
```

Regla de oro: el default gana hasta que haya una razón. "Mejor tecnología" sin razón concreta es gusto, no criterio.

K.2. Frontend

Decisión 1 — fetch, axios o TanStack Query

La situación: tu componente lista tareas. El compañero instala axios "porque es lo que usan todos". Otro dice "fetch ya viene, no instales nada". ¿Quién tiene razón?
La decisión. Los dos y ninguno — depende de *qué* estás haciendo:

- `fetch` es nativo: cero dependencias, suficiente para llamadas simples. Su precio: no rechaza en 4xx/5xx (`res.ok` es tu trabajo), no trae timeout, JSON manual.
- `axios` agrega: JSON automático, errores en 4xx/5xx, interceptors (agregar el token a *todas* las llamadas en un solo lugar), timeouts. Su precio: +13KB y una dependencia más.
- **TanStack Query / SWR** no es un cliente: es manejo de *server state* (caché, refetch, loading/error por ti). Cuando tu lista recarga datos, deduplica llamadas y cachea — es lo correcto, con fetch o axios debajo.

La señal: ¿es "una llamada suelta"? → fetch. ¿La app entera necesita headers/interceptors consistentes? → axios. ¿Los datos son *estado del servidor* que hay que cachear y refrescar? → TanStack Query encima de cualquiera. → *fe-05*

Decisión 2 — React, Vue, Angular o Svelte

La situación: empiezas un proyecto nuevo y te preguntan "¿qué framework?". Todos "hacen lo mismo" — ¿cómo se decide de verdad?
La decisión. La estructura (reactividad, componentes, estado) es la misma en los cuatro — lo que cambia es el material:

- **React**: el mercado más grande. Más empleos, más librerías, más respuestas en Google. El default cuando no hay razón para otra cosa.

- **Vue**: curva más suave, todo integrado (router + estado oficiales). Excelente para equipos pequeños o quien viene de HTML/CSS.
- **Angular**: trae *todo* decidido (DI, RxJS, estructura) — pesa, pero en equipos enterprise esa imposición es una ventaja. Su DI se parece al backend (FastAPI `Depends` te sonará familiar).
- **Svelte**: menos boilerplate, sin virtual DOM — ecosistema menor y menos empleos. Brillante, pero revisa el mercado antes de apostarle tu carrera.

El criterio honesto: mercado de empleos → qué sabe tu equipo → qué pide el proyecto. *No* la sintaxis — la aprendes en una semana; el ecosistema es lo que no puedes cambiar en una semana. → *fe-04*

Decisión 3 — ¿TypeScript o JavaScript?

La situación: el starter viene en TS y a un compañero "le estorban los tipos" — propone volver a JS puro.
La decisión. TypeScript, casi siempre — pero entiende *por qué*: los tipos son documentación que la máquina verifica. El error `task.tile` (donde era `title`) JS lo deja pasar hasta producción; TS te lo subraya mientras escribes. El costo real: configuración, curva inicial, y verbosidad. JS puro es defendible en scripts pequeños, prototipos de un día, o cuando el equipo entero es junior y el tipo extra confunde más que lo que protege. Pero en código que va a prod y lo toca más de una persona, TS se paga solo — el backend ya valida contratos (cap. 4); TS es el mismo contrato dentro de tu código. → *fe-02*

Decisión 4 — CSR, SSR o SSG

La situación: hay que decidir cómo se renderiza la app y tres voces dicen tres siglas distintas.
La decisión. Pregunta: ¿quién necesita ver qué, y cuándo?

- **CSR** (client-side): el servidor manda un HTML vacío + JS. Perfecto para apps detrás de login (dashboards, Nexus) — nadie indexa tu panel en Google, y la interactividad es lo que importa.
- **SSR** (server-side): HTML completo por request. Cuando SEO y primera-pintura-rápida importan *y el contenido cambia por usuario*: tiendas, perfiles públicos.
- **SSG** (static): HTML pre-generado en build. Blogs, docs, marketing — contenido igual para todos, servido desde CDN por centavos.

La trampa: elegir SSR "porque es moderno" para un dashboard que nadie googleará — pagas complejidad de hidratación sin comprar nada. → *fe-06*

Decisión 5 — useState, Context o librería de estado

La situación: el estado del proyecto creció y alguien grita "¡necesitamos Redux!".
La decisión. Escala del problema, no moda:

- `useState` local: el 80 % de los casos — estado que vive y muere en un componente (modales, inputs, toggles).

- `Context` / props: compartir a pocos niveles, baja frecuencia (tema, usuario actual). Su trampa: cada cambio re-renderiza todo el árbol.
- Zustand/Redux/jotai: estado complejo compartido por muchas vistas con lógica de actualización no trivial.

Y la pregunta que ahorra la mitad de los stores: **¿esto es server state?** Si los datos viven en la BD (tareas, usuarios), no van en un store — van en TanStack Query (decisión 1). Los stores son para estado *de la UI*. → *fe-03, fe-05*

K.3. Backend

Decisión 6 — Node/TypeScript, Python o Go

La situación: el proyecto nuevo puede ser en cualquiera de los tres del libro. ¿Cuál?
La decisión. Los tres hacen *el mismo backend* — el libro entero lo demuestra capítulo a capítulo. La elección real:

- **TypeScript/Node**: un solo lenguaje en front y back — la opción natural para fullstack pequeños; ecosistema npm gigante.
- **Python/FastAPI**: si el producto toca datos/AI/ML, el ecosistema ya está ahí; FastAPI es el framework más didáctico de los tres.
- **Go**: cuando la concurrencia y el rendimiento por núcleo importan (workers, proxies, alto throughput) — y un binario único de deploy hermoso.

El criterio: qué sabe el equipo → qué pide el dominio (AI→Python, I/O masivo→Go, fullstack unificado→TS) → mercado local de empleos. La trampa: elegir "el más rápido" para un CRUD — el cuello de botella será la BD, no el lenguaje. → *cap-02*

Decisión 7 — Monolito o microservicios

La situación: alguien propone "hagámoslo microservicios desde el día uno, así escala".
La decisión. Monolito hasta que un límite *real* lo pida. Un monolito es un deploy, una transacción, un log — y se puede partir después porque tus capas (cap. 7) ya separaron el dominio. Microservicios compran deploys independientes y escala por pieza — y pagas: red entre servicios, transacciones distribuidas, debugging que cruza procesos, devops por servicio. La señal real para partir: un equipo que necesita deployar sin pisar a otro, o una pieza que escala 10× distinto al resto. La trampa: microservicios con un equipo de 2 — compraste todos los costos sin tener el problema. → *cap-07*

Decisión 8 — REST, GraphQL o RPC

La situación: la app nueva podría ser "algo más moderno que REST".
La decisión. REST es el default: recursos con URL, verbos HTTP, caché gratis, cualquier cliente lo habla. **GraphQL** cuando el cliente debe *elegir campos* o componer muchas entidades en un request (una misma API para apps muy distintas) — pagas runtime de queries y caché manual. **RPC/acciones** (`/tasks/complete`) cuando el dominio son operaciones, no recursos — es REST honesto, no un pecado. La señal para salir de REST: *los clientes* necesitan flexibilidad que tú no controlas. → *cap-03*

Decisión 9 — Express, Fastify, NestJS u otro

La situación: elegiste Node — ahora ¿qué framework?
La decisión.

- **Express**: el default — ecosistema gigante, todo tutorial lo usa. Su precio: no impone nada; la estructura la pones tú (cap. 7).
- **Fastify**: la misma idea con ~2× throughput y validación integrada — cuando el rendimiento del framework *realmente* medido importa.
- **NestJS**: impone arquitectura (DI, módulos, decoradores) — para equipos grandes donde "cada quien hace su estructura" es el problema; se siente como Angular/Spring en el backend.
- **Hono**: ultraligero y corre en edge/runtimes no-Node.

La señal: ¿equipo grande y código heterogéneo? → NestJS. ¿Todo lo demás? → Express hasta medir una razón. → *cap-01, cap-07*

Decisión 10 — ¿En el request o en un job?

La situación: al registrarse hay que enviar email, generar un PDF de bienvenida y notificar a Slack. El endpoint ahora tarda 4 segundos.
La decisión. Todo lo que el usuario no necesita *para continuar* sale del request → cola de trabajos (BullMQ/Celery/asynq). El response necesita solo "usuario creado" — 201 en ~50ms, el resto lo hace un worker. La pregunta decisiva: **"¿el usuario está esperando esto para seguir?"** Sí → request. No → job. La trampa: "es solo un email" × 20 features = cada endpoint lento. → *cap-21, ticket 49 del Ap. G*

K.4. Datos

Decisión 11 — SQL o NoSQL

La situación: "¿Y si mejor Mongo? Total, devolvemos JSON."
La decisión. Relacional por defecto — tus datos son relaciones (tareas *de* usuarios) y `REFERENCES` las enforza gratis. Documento solo si la *forma* de cada registro varía de verdad. Y la respuesta del 80 %: `JSONB` dentro de Postgres cuando necesitas campos variables. La trampa es confundir el *formato* de salida (JSON) con la *estructura* del dato (relaciones). La sección completa está en cap-08: familias NoSQL, señales y local-vs-prod. → *cap-08*

Decisión 12 — ORM o SQL a mano

La situación: el ORM se siente "más profesional", pero hay queries que no sabes traducir.
La decisión. SQL primero — por eso el libro te hizo escribir a mano los caps. 8–13: quien solo sabe ORM no sabe qué está generando (el N+1 del cap. 12 nació de un ORM bonito). El ORM gana cuando: el equipo es grande y la velocidad+tipado pesan más que el control fino; o las queries son 95 % CRUD aburrido. Punto medio real: *query builders* (Kysely, SQLAlchemy Core) — SQL con tipos, sin caja negra. El criterio: usa el ORM como **generador de un SQL que tú ya sabrías escribir**, no como sustituto de entenderlo. → *cap-09, cap-12*

Decisión 13 — Postgres, MySQL o SQLite

La situación: tres opciones relacionales — ¿en qué difieren para tu caso?
La decisión.

- **Postgres**: el default del libro — features (JSONB, tipos ricos, extensibilidad), comunidad, managed en todas las nubes.
- **MySQL**: igual de capaz para lo común; se elige cuando el equipo/ hosting ya vive ahí (mucho hosting legacy y WordPress lo trae).
- **SQLite**: la BD *es un archivo* — perfecta para apps locales, embebidas, prototipos y tests rápidos. Su límite: escrituras concurrentes (un writer a la vez) — no es la BD de una API con usuarios simultáneos.

La trampa: SQLite en dev "y ya Postgres en prod" — motores distintos = bugs que solo existen en prod. Mismo motor en todos los entornos. → *cap-08*

Decisión 14 — ¿Necesito Redis?

La situación: suena a que "todo sistema serio lleva Redis".
La decisión. Redis es un *complemento*, no una BD principal — es clave-valor en memoria: rápido porque no promete durabilidad ni relaciones. Lo necesitas cuando aparece una de estas necesidades **concretas**: caché de lecturas calientes (la misma query 10.000 veces/seg), sesiones a escala multi-instancia, rate limiting distribuido, colas de trabajos, pub/sub para realtime. Si ninguna existe, no lo instales "por si acaso" — cada pieza de infra es algo que hay que operar. La

señal: un problema *medido* que Postgres no resuelve solo, no una arquitectura copiada de Netflix. → *cap-08, cap-18*

Decisión 15 — ¿Migraciones del ORM o manuales?

La situación: el ORM ofrece "sync automático del schema" — ¿tomarlo?
La decisión. Auto-sync en desarrollo temprano es cómodo; en producción es un incidente esperando turno — un `ALTER` automático sobre millones de filas bloquea la tabla. El flujo serio: migraciones como **archivos versionados con el código** (up/down, numeradas, en el mismo PR que las usa), revisables, aplicables por pasos expand-contract en prod (cap. 11). Que el ORM *genere el borrador* de la migración está bien — que la aplique solo, no. La señal: si el cambio toca prod, un humano lo revisa línea por línea. → *cap-11*

K.5. Infraestructura

Decisión 16 — VPS, PaaS, contenedor o serverless

La situación: "¿Dónde lo desplegamos?" — cuatro respuestas distintas en la sala.
La decisión. Forma del tráfico + equipo disponible:

- **VPS** (una VM tuya): control total, precio fijo — pagas con *tu tiempo*: parches, firewall, deploys, la 3am. Bien si hay alguien de ops o es aprendizaje deliberado.
- **PaaS** (Render, Railway, App Engine): `git push` y ya — para la mayoría de productos chicos/medianos es el punto dulce: pagas más por hora, menos por tus horas.
- **Contenedor administrado** (ECS, Cloud Run): el medio camino — tu imagen, ellos orquestan. Cuando PaaS se queda corto en control.
- **Serverless** (Lambda, Functions): eventos irregulares y sin estado — pagas por ejecución; un proceso 24/7 para 50 invocaciones/día es un taxi encendido en la puerta.

La señal: ¿quién opera esto a las 3am y cuánto vale su hora? → *nu-01, nu-02, nu-05*

Decisión 17 — AWS, Azure o GCP — ¿y eso decide mi lenguaje?

La situación: "Si vamos a Azure habrá que usar C#, ¿no? Y en AWS ¿Python?"
La decisión. El proveedor NO decide tu lenguaje — es el mito más común. AWS, Azure y GCP corren Node, Python, Go, Java, .NET y PHP por igual: las funciones, VMs y contenedores son agnósticos. Azure es de Microsoft *como empresa*, no "la nube de C#" — corre FastAPI igual de bien que GCP. Lo que sí se acopla al proveedor son los *servicios propietarios* (Cosmos DB, S3, BigQuery) — ahí vive el lock-in real, no en el runtime. El criterio real entre nubes: qué usa tu empresa/objetivo laboral (empleos y contratos ya firmados) → precio de los 3-4 servicios que usarás → quién en el equipo ya lo conoce. La funcionalidad es la misma: el supermercado del cap. nu-03 tiene los mismos pasillos en los tres. → *nu-03, nu-04*

Decisión 18 — ¿Docker o 'a pelo' en el servidor?

La situación: deployar es `git pull && npm start` en el VPS — ¿para qué Docker?
La decisión. Docker compra **reproducibilidad**: el mismo artefacto corre en tu laptop, CI y prod — "en mi máquina sí funcionaba" deja de existir, y el onboarding es `docker compose up`. Lo pagas: una capa más que entender (imágenes, volúmenes, redes). "A pelo" con systemd es defendible cuando: un solo servicio, un solo servidor, un solo dev — y así fue el deploy clásico de nu-01. La señal para migrar a Docker: segundo desarrollador, segundo entorno, o CI — cuando "funciona en mi máquina" cueste la primera tarde. → *nu-01, cap-30*

Decisión 19 — ¿BD administrada o la instalo yo?

La situación: "Postgres es gratis — la instalamos en la misma VM y nos ahorramos RDS."
La decisión. El costo real de la BD no es la licencia: son los **backups, las réplicas, los parches y la 3am**. Una BD auto-instalada te los cobra todos en horas y riesgo — y justo la BD es *lo último* que puede fallar. RDS/Cloud SQL cuesta más por mes y menos por incidente: backups automáticos con point-in-time, failover, parches del proveedor. Auto-instalar es defendible: presupuesto cero real, aprendizaje deliberado, o datos que *legalmente* no pueden salir. La señal: si la respuesta a "¿y si se corrompe a las 3am?" es silencio → managed. → *nu-02, nu-04, cap-08*

K.6. Forma de trabajar

Decisión 20 — git merge, rebase o squash

La situación: tu rama está lista y el PR ofrece tres botones — ¿cuál aprietas? *(Y de paso: por qué Git existe — varias personas editando el mismo código sin pisarse, con la historia completa para volver atrás.)*
La decisión.

- **Merge commit**: preserva la historia real — cuándo se integró qué. El default seguro.
- **Squash**: toda tu rama → un solo commit. Perfecto para "fix typo, fix typo2, wip, ahora sí" — la historia queda limpia de ruido.
- **Rebase**: re-escribe tu rama *sobre* main — historia lineal y legible. **Regla sagrada: solo en ramas tuyas, nunca en ramas que otro ya tiene** (reescribe historia = conflictos para los demás).

El criterio real: **la convención del equipo gana** — pregunta en tu primer día, no decidas solo. Si no hay convención: squash para features con commits ruidosos, merge para ramas grandes con historia valiosa.

Decisión 21 — ¿TDD o tests después?

La situación: "TDD o nada" vs "los tests son burocracia, codeo primero".
La decisión. TDD brilla cuando el *contrato es claro y la precisión importa*: dinero, auth, reglas de negocio — el test primero te obliga a pensar la API antes del código (ticket 41-42 del Ap. G). Tests-después es defendible en exploración (prototipos, spikes) donde aún no sabes la forma. Lo que *no* es negociable: **los tests existen antes de prod** — TDD es una herramienta de *diseño*, no una religión de cobertura. La señal para ir TDD: si puedes escribir "debería rechazar X" antes de tener X, el test ya es tu especificación gratis. → *cap-27, Ap. G*

Decisión 22 — ¿Deploy manual o CI/CD?

La situación: hoy deployas con `ssh + git pull` — funciona, ¿para qué un pipeline?
La decisión. El manual funciona hasta que: olvidas un paso (migración sin correr), te vas de vacaciones y nadie más puede deployar, o el "funciona en mi máquina" llega a prod. CI/CD (GitHub Actions y familia) compra: **el mismo proceso siempre** — tests corren solos en cada PR, el deploy es un botón o un merge, y cualquiera del equipo puede hacerlo. El criterio: segundo dev en el proyecto → ya merece pipeline. La señal clásica: "solo X sabe deployar" es un riesgo, no un honor. → *cap-33*

Decisión 23 — ¿Logs, métricas o tracing?

La situación: "¿Cómo sabemos qué pasa en prod?" y te nombran tres herramientas que no distingues.
La decisión. Son tres preguntas distintas, no tres alternativas:

- **Logs**: "¿qué pasó en *este* request?" — texto con contexto. La base; estructurados (JSON) desde el día uno para poder buscarlos.
- **Métricas**: "¿cómo va el *sistema*?" — números en el tiempo: latencia p95, errores/minuto, CPU. Las alarmas viven aquí.
- **Tracing**: "¿por dónde viajó *este* request entre servicios?" — solo cuando hay múltiples servicios/procesos que seguir.

Orden de adopción honesto: logs primero (casi gratis), métricas cuando hay usuarios reales que avisar antes que te avisen, tracing cuando "¿dónde se perdió el request?" deje de poder responderse con logs. → *cap-23*

K.7. La checklist decisiva — llévatela impresa

Cuando llegue el próximo "¿X o Y?" que no esté en este apéndice:

```
1. ¿Cuál es la FORMA del problema? (dato / tráfico / UI / equipo)
2. ¿Qué escala REAL tiene hoy? (no la aspiracional)
```

```
3. ¿Quién lo mantiene y qué ya sabe?
4. ¿Es puerta de una vía o de dos? (¿cuánto cuesta desdecidirse?)
5. ¿Qué trae el ecosistema? (empleos, librerías, respuestas)
→ Y si no hay razón concreta: el DEFAULT gana.
```

El reto final: agarra cualquier decisión de este apéndice, cuéntasela a tu AI CLI como si fuera tu lead ("estamos por elegir X o Y, contexto: …"), y compara su criterio con el tuyo. Donde difieran, ahí está tu siguiente hora de estudio.

Explicación de: CLI

Un **CLI** (Command Line Interface) es un programa que se usa escribiendo comandos en la terminal: `git`, `npm`, `docker` son CLIs. Lo contrario es una GUI (interfaz gráfica con botones).

Las decisiones no se memorizan — se reconocen. Después de estas 23, la siguiente "¿X o Y?" dejará de sentirse como adivinanza.

K.8. Vocabulario técnico del capítulo

Explicación de: variable

Una **variable** es una caja con nombre donde guardas un valor: `let total = 42` guarda el 42 bajo el nombre `total`. Puedes leerla y cambiarla después (`total = 50`). `const` = caja que no se puede reemplazar.

Explicación de: función

Una **función** es una receta reutilizable: recibe ingredientes (parámetros), hace pasos y devuelve un plato (`return`). La escribes una vez y la llamas mil veces: `add(2, 3)` → `5`.

Explicación de: memoria (RAM)

La **memoria RAM** es el escritorio de trabajo del programa: rápida, pero se borra al apagar. Los datos que deben sobrevivir van a disco o a la base de datos. Por eso `let tasks = []` pierde todo al reiniciar.

Explicación de: CPU / núcleos

El **CPU** es el cerebro que ejecuta instrucciones; cada **núcleo** puede ejecutar una cosa a la vez (por eso importan la concurrencia y los procesos paralelos). "CPU-bound" = el límite es el cálculo, no la espera.

Explicación de: query / consulta

Una **query** es la pregunta que le haces a la base de datos en SQL: `SELECT * FROM tasks WHERE done = false`. La BD traduce la pregunta a un plan de búsqueda — por eso los índices importan.

Explicación de: esquema (schema)

El **esquema** es el plano de la base de datos: qué tablas hay, qué columnas tiene cada una y de qué tipo. Es contrato: una fila que no cumple el esquema no entra. Se cambia con migraciones, no a mano.

Explicación de: migración

Una **migración** es un cambio versionado del esquema: un archivo que dice "crea esta columna" (up) y "bórrala" (down). Son el historial Git de la estructura de la BD — se aplican en orden y no se editan una vez aplicadas.

Explicación de: transacción

Una **transacción** es un grupo de operaciones que se confirman juntas o no se confirma ninguna: transferir dinero = restar de A *y* sumar a B. Si falla a la mitad sin transacción, el dinero desapareció.

Explicación de: runtime

El **runtime** es el motor que ejecuta tu código: Node.js es el runtime de JavaScript fuera del navegador; CPython es el de Python; Go compila a binario y el runtime va empaquetado dentro. Es "quien corre" lo que escribiste.

Explicación de: entorno (environment)

Un **entorno** es una instancia completa donde corre tu app con su propia config y datos: *local* (tu máquina), *staging* (réplica de prueba), *producción* (la real). Cada entorno tiene sus propias llaves y su propia base de datos.

Explicación de: DOM

El **DOM** es la página como árbol de objetos vivos: HTML es el papel, el DOM es lo que JavaScript puede tocar. `element.textContent = "hola"` cambia el DOM y el navegador repinta — el HTML original no se entera.

Explicación de: framework

Un **framework** es un esqueleto de aplicación ya decidido: te da la estructura (rutas, validación, errores) y tú llenas la lógica. Diferencia con librería: la librería la llamas tú; el framework te llama a ti.

Explicación de: dependencia

Una **dependencia** es código de terceros que tu proyecto usa: `npm install`, `pip install`, `go get` las traen. Cada una es deuda — ahora funciona, pero hay que mantenerla, actualizarla y confiar en ella.

Explicación de: build

El **build** (construcción) es el proceso que convierte tu código fuente en algo ejecutable/entregable: compilar TypeScript a JS, empaquetar el frontend, armar la imagen Docker. Lo que se despliega es el resultado del build, no tu código crudo.

Explicación de: Git

Git es el historial de tu proyecto: cada *commit* es una foto del código con mensaje. Permite volver atrás, trabajar en ramas paralelas y fusionar el trabajo de varias personas sin pisarse.

Explicación de: commit

Un **commit** es una foto guardada del código con un mensaje que dice por qué cambió. La historia del proyecto es una cadena de commits: puedes volver a cualquiera, ver qué cambió y quién lo hizo.

Explicación de: proceso

Un **proceso** es un programa en ejecución: el sistema operativo le da memoria propia y un lugar en la fila del CPU. Tu API es un proceso; Postgres es otro; el navegador es varios. Que "se caiga el proceso" = el programa murió.