

Ingeniería de Software

APIs y patrones de diseño

Recursos, idempotencia y evolución · OpenAPI · Patrones que siguen vigentes

Gerson Garrido

Semana 6

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: APIs

1. Un contrato con quien no está en la sala
2. Diseño orientado a recursos
3. Seguro e idempotente
4. La respuesta perdida y la clave de idempotencia
5. Códigos de estado y errores
6. Evolución, versiones y deprecación
7. OpenAPI

Sesión B: patrones

8. Qué sigue vigente
9. Strategy y dependency injection
10. Adapter y repository
11. Decorator y observer
12. Cuándo un patrón estorba

Lectura

AIP-121, 136, 158 y 180 · Azure REST API Guidelines · RFC 9110, sec. 9.2 · Refactoring.Guru (Strategy, Adapter, Decorator, Observer, Factory Method) · Fowler, *Dependency Injection* (2004)

SECCIÓN 01

Una API es un contrato

01



Una interfaz que no se puede corregir después

Dentro del repositorio

- Renombran un método
- `tsc` marca cada llamada
- Se corrigen en el mismo pull request

En una API HTTP

- Renombran un campo
- Nadie marca nada
- La app se rompe en el celular de alguien

Contrato ***(contract)***

Lo que cada operación pide y lo que promete. Quien llama no está en su repositorio: no se le puede revisar ni actualizar a la vez que el servidor.

La ley de Hyrum en una API

Con suficientes usuarios, todo lo observable termina siendo parte del contrato.

- El **orden de una lista**: la primera cancha es “la principal”.
- El **texto de un error**: buscar “reservada” dentro de `message`.
- Un **campo que falta**: `teamName` solo aparece si la hora está ocupada.

> **Idea clave**
>
> Una especificación no elimina el problema, pero dice qué se promete. Lo que queda fuera puede romper a alguien, pero no es una promesa rota.

SECCIÓN 02

Diseño orientado a recursos

02

Sustantivos en las rutas, verbos en el método

AIP-121: primero los recursos, luego sus relaciones, al final las operaciones.

rutas de Kallpa

```
GET  /api/courts                  la colección de canchas
GET  /api/courts/1/availability   algo que le pertenece a la cancha 1
GET  /api/members/2/bookings      las reservas del vecino 2
POST /api/bookings                agrega una reserva a la colección
```

Error común

Verbos en la ruta: `/api/getCourts`, `/api/deleteBooking?id=12`. El método ya es el verbo, y lo que está en el medio entiende `DELETE`, no `deleteBooking`.

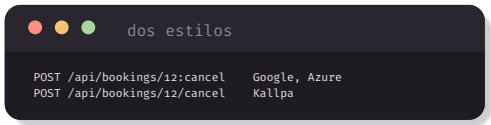
Métodos estándar

Método	HTTP	Sobre	En Kallpa
List	`GET`	colección	`GET /api/courts`
Get	`GET`	recurso	—
Create	`POST`	colección	`POST /api/bookings`
Update	`PATCH`	recurso	—
Delete	`DELETE`	recurso	—

- `PATCH` y no `PUT`: `PUT` reemplaza todo, y un cliente HTTP viejo borraría los campos que no conoce (AIP-134).
- Disponibilidad y caja del día: cálculos que solo leen. `GET` les queda bien.

Métodos propios: cancelar

- `DELETE` sugiere que deja de existir; la reserva se queda como `cancelled`.
- `PATCH` con `status` sugiere que se puede poner cualquier valor.
- AIP-136: método propio, *solo* si los estándar no alcanzan. `POST` si hay efectos.



Lo que importa: una sola forma en toda la API.

Error común

Un `GET` que cambia datos. Navegadores, proxies, caches y robots lo repiten o lo adelantan sin preguntar.

SECCIÓN 03

Seguro e idempotente

03

Las definiciones de RFC 9110

Seguro ***(safe)***

Quien llama *"does not request, and does not expect, any state change"*.

Idempotente ***(idempotent)***

"the intended effect on the server of multiple identical requests [...] is the same as the effect for a single such request".

Método	Seguro	Definido como idempotente
`GET`, `HEAD`, `OPTIONS`, `TRACE`	sí	sí
`PUT`, `DELETE`	no	sí
`POST`, `PATCH`	no	no

Tres precisiones que se olvidan

1. **Es sobre el efecto, no sobre la respuesta.**
   “though the response might differ”: un segundo `DELETE` puede dar `404`.
2. **Es sobre lo que se pidió.**
   Un `GET` que escribe una línea de log sigue siendo seguro.
3. **“No definido como idempotente” no es “nunca lo es”.**
   Un programa genérico solo ve el método. Una operación concreta sobre `POST` puede serlo, si se diseña y se documenta así.

Kallpa: reintentar y cancelar dos veces

make recorrido (líneas cortadas)

```
--- reservar y reintentar
$ curl -X POST localhost:3100/api/bookings -H content-type: application/json -d {"courtId":2,"memberId":2,"date":"2026-11-10","startHour":10,"t
HTTP/1.1 200 OK
{"booking":{"id":16,"courtId":2,"memberId":2,"date":"2026-11-10","startHour":10,"hours":1,"teamName":"Los Halcones","priceCents":8100,"status":

$ curl -X POST localhost:3100/api/bookings -H content-type: application/json -d {"courtId":2,"memberId":2,"date":"2026-11-10","startHour":10,"t
HTTP/1.1 409 Conflict
{"error":{"code":"SLOT_TAKEN","message":"La cancha ya está reservada en ese horario."}}

--- cancelar dos veces
$ curl -X POST localhost:3100/api/bookings/16/cancel
HTTP/1.1 200 OK
{"booking":{"id":16,"courtId":2,"memberId":2,"date":"2026-11-10","startHour":10,"hours":1,"teamName":"Los Halcones","priceCents":8100,"status":

$ curl -X POST localhost:3100/api/bookings/16/cancel
HTTP/1.1 409 Conflict
{"error":{"code":"ALREADY_CANCELLED","message":"La reserva ya estaba cancelada."}}
```

¿Son idempotentes?

Cancelar

- Como método: `POST`, no definido.
- Como operación: la reserva queda igual, con el mismo `cancelledAt`. El efecto sí es idempotente.
- La respuesta cambia (`200`, luego `409`): la definición lo permite.
- Falta decirlo en el contrato.

Reservar

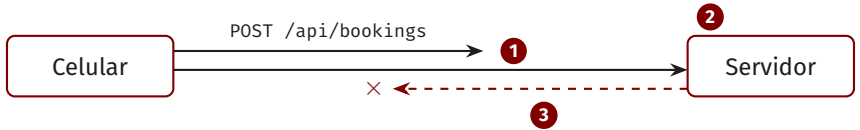
- El segundo intento no creó nada…
- …pero si la reserva se canceló en el medio, el reintento crea otra.
- El `409` viene de una regla del negocio, no de la idempotencia.

SECCIÓN 04

La respuesta perdida

04

Lo único que sabe quien llama



1. Nunca llegó. No hay reserva.
2. Llegó y se está procesando. Puede que haya.
3. Se procesó y se perdió la respuesta. Hay reserva.

> **Desde el celular, las tres se ven igual**
>
> Si la operación es idempotente, se reintenta sin pensar. Si no, reintentar es una apuesta.

El caso 3, de verdad: `make retry`

```
make retry

Kallpa, POST /api/bookings
  intento 1: se cortó la conexión (fetch failed)
  intento 2: 409 {"error":{"code":"SLOT_TAKEN","message":"La cancha ya está reservada en ese horario."}}
  reservas del vecino 2 en la base de datos: 1

Pagos sin clave, POST /charges
  intento 1: se cortó la conexión (fetch failed)
  intento 2: 201 {"charge":{"id":2,"memberId":2,"cents":8100}}
  cobros en la base de datos: 2
```

- **Kallpa** no duplica, pero informa mal: el vecino ve un error y tiene una reserva.
- **Pagos** duplica: nada frena el segundo cobro.

La clave de idempotencia

la solicitud

```
POST /api/bookings
Idempotency-Key: "8e03978e-40d5-43e8-..."
```

Es un borrador

IETF, versión -08 (octubre de 2026).
No es un RFC y casi todo es *SHOULD*.

- Un valor único **por intención**, no por envío.
- El servidor guarda la clave con la respuesta.

Situación	Respuesta
Clave nueva	procesa normalmente
Reintento, la primera terminó	la guardada (el borrador pide una “consistente”)
Reintento, la primera sigue en proceso	`409`
Misma clave, otro cuerpo	`422`
Falta la clave, y se exige	`400`

Como decorator de una operación

idempotency.ts

```ts
const existing = store.get(key);
if (existing && now - existing.storedAt < options.ttlMs) {
  if (existing.fingerprint !== print) return errorResponse(422, 'IDEMPOTENCY_KEY_REUSED', ...);
  if (existing.state === 'in-progress') return errorResponse(409, 'REQUEST_IN_PROGRESS', ...);
  return { ...existing.response, replayed: true };
}
store.set(key, { state: 'in-progress', fingerprint: print, storedAt: now });
try {
  const response = await operation(request);
  store.set(key, { state: 'done', fingerprint: print, storedAt: now, response });
  return { ...response, replayed: false };
} catch (error) {
  store.delete(key); // no se sabe qué pasó: un reintento puede volver a correrla
  throw error;
}
```

La **huella** (fingerprint) es un hash del cuerpo. `operation` no sabe nada de claves.

Con clave: un solo cobro

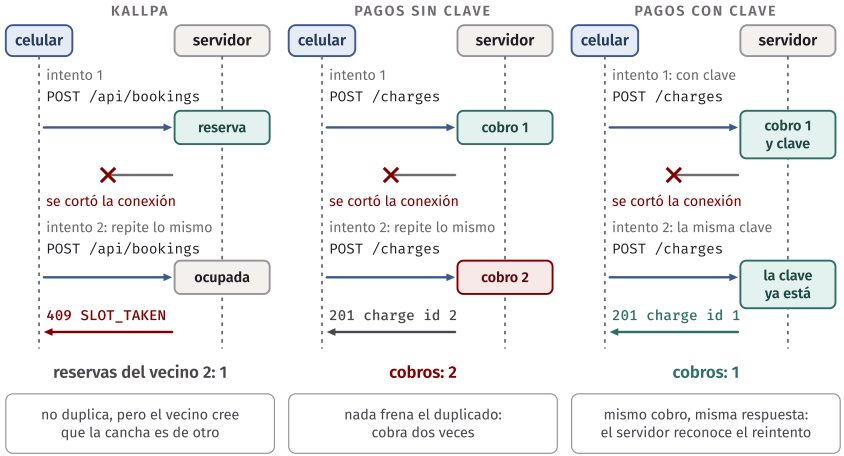
make retry

```
Pagos con clave, POST /charges
  intento 1: se cortó la conexión (fetch failed)
  intento 2: 201 {"charge":{"id":1,"memberId":2,"cents":8100}}
  cobros en la base de datos: 1
```

node --test idempotency.test.ts (sin tiempos)

```
✓ sin clave no se ejecuta nada: 400
✓ un reintento con la misma clave y el mismo cuerpo devuelve la misma respuesta sin repetir el efecto
✓ el orden de los campos no cambia la huella
✓ la misma clave con otro cuerpo: 422 y sin efecto
✓ un reintento mientras la primera todavía corre: 409
✓ cuando vence la clave, el mismo pedido vuelve a ejecutarse
✓ las respuestas de error de la operación también se guardan y se repiten
✓ el límite de este diseño: si el proceso falla después del efecto, el reintento lo repite
```

La misma respuesta perdida, tres resultados



Las decisiones que el código no toma

- **¿Dónde y cuándo se guarda?** La clave y el efecto, en la misma transacción. Si no, una caída en el medio repite el efecto.
- **¿Por cuánto tiempo?** Lo que cubra los reintentos reales. Se publica.
- **¿Efectos fuera de la base de datos?** Un aviso por WhatsApp no entra en una transacción.
- **¿Se guardan los rechazos?** Un `409` guardado seguiría diciendo “ocupada”.
- **¿De quién es la clave?** Si es global, adivinarla da la respuesta de otro.
- **¿Es obligatoria?** En una API que ya existe, volverla obligatoria rompe a todos.

En la industria

Stripe y Adyen aparecen en la sección *Implementation Status* del borrador. Azure exige que todo sea idempotente y, para `POST`, usa `Repeatability-Request-ID` y `Repeatability-First-Sent` (OASIS).

Con IA: reintentos

“Agrega reintentos a este cliente HTTP”:

- Lo típico: un bucle con backoff.
- Pasa todas las pruebas: en una prueba normal no se pierde ninguna respuesta.

Qué revisar

- ¿Es idempotente lo que se reintenta?
- ¿La clave se genera fuera del bucle?

Cómo se comprueba

Una prueba con una red que corta y un contador de efectos: `lost-response.test.ts`.

SECCIÓN 05

Códigos de estado y errores

05

Códigos de estado con significado

`200` OK	`201` Created + `Location`	`204` No Content
`400` Bad Request: mal formada	`401`: no sé quién eres	`403`: sé quién eres, y no
`404` Not Found	`405` + `Allow`	`409` Conflict: choca con el estado
`413` Content Too Large	`422` Unprocessable Content	`429` Too Many Requests
`500` Internal Server Error	`503` Service Unavailable	

- `4xx`: “cambia tu solicitud”. `5xx`: “no es tu culpa”.
- Todo lo que está en el medio los lee: caches, balanceadores, bibliotecas de reintento.

Error común

`200` con `{ "success": false }`: el panel de errores queda en cero mientras nadie puede reservar.

La frase de estado la pone Node

make recorrido (líneas cortadas)

```
$ curl -X POST localhost:3100/api/bookings -H content-type: application/json -d {"courtId":2,"memberId":2,"date":"2026-11-10","startHour":10,"h
HTTP/1.1 422 Unprocessable Entity
{"error":{"code":"TOO_MANY_HOURS","message":"Se pueden reservar como máximo 3 horas seguidas."}}

$ curl -X POST localhost:3100/api/bookings -H content-type: application/json --data @/tmp/kallpa-grande.json
HTTP/1.1 413 Payload Too Large
{"error":{"code":"BODY_TOO_LARGE","message":"El cuerpo de la solicitud es demasiado grande."}}
```

- Esos son los nombres antiguos. RFC 9110: *Unprocessable Content* y *Content Too Large*.
- No es un error de Kallpa: los programas comparan el número. RFC 9112: *“A client SHOULD ignore the reason-phrase content”*.

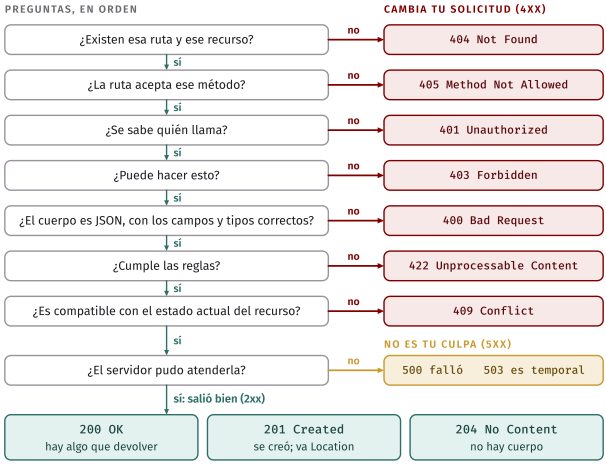
La frontera entre `400` y `422`

`hours: 4` tiene la forma correcta y no cumple una regla. ¿Qué código le toca?

- **Azure:** `400` para todo lo que falle la validación.
- **Google:** no usa `422`. `INVALID_ARGUMENT` y `FAILED_PRECONDITION` van a `400`.
- **Kallpa** (README): `400` si falta un campo o tiene el tipo equivocado; `422` para las reglas.

> **Idea clave**
>
> Más de una respuesta es razonable. Lo que no se defiende es decidirlo campo por campo: escríbanla, y revisen que todos los endpoints la respeten. En el laboratorio miran si Kallpa lo hace.

Una forma de escoger el código, no una norma



Errores que un programa pueda leer

el error de Kallpa, en todas partes

```
{ "error": { "code": "SLOT_TAKEN", "message": "La cancha ya está reservada en ese horario." } }
```

Quién	Forma	Para programas
RFC 9457	`problem+json`: `type`, `title`, `status`, `detail`, `instance`	`type`, una URI
Azure	`error`: `code`, `message`, `target`, `details`	`code`, un texto, y la cabecera `x-ms-error-code`
Google	`error`: `code`, `message`, `status`, `details`	`ErrorInfo.reason`

Ojo: en Google, `error.code` es el número HTTP; en Azure y en Kallpa, un texto.

SECCIÓN 06

Evolución y versiones

06

Compatible e incompatible

Cambio	¿Compatible?
Agregar un endpoint	Sí
Campo opcional en la solicitud	Sí, si omitirlo hace lo mismo que antes
Campo en la respuesta	Sí, si quien llama ignora lo que no conoce
Quitar o renombrar un campo	No
Cambiar su tipo o su significado	No; el significado no lo detecta ninguna herramienta
Volver obligatorio algo opcional	No
Cambiar un código de estado	No
Cambiar un valor por omisión	No (AIP-180)
Aumentar el largo máximo de un texto	AIP-180 pide tratarlo como incompatible

Qué rompe cada cambio

```
node --test evolution.test.ts (sin tiempos)

➤ qué rompe cada cambio
  ✓ agregar un campo en la respuesta: el cliente tolerante sigue igual
  ✓ agregar un campo en la respuesta: el cliente que exige campos exactos se cae
  ✓ renombrar hour a startHour: el cliente tolerante no se cae, pero devuelve basura
  ✓ envolver el arreglo suelto en un objeto: el cliente se cae
  ✓ pasar un error de 400 a 422: el cliente que comparaba el código ya no lo reconoce
```

- **Línea 2:** “agregar es compatible” tiene una condición. Pónganla por escrito.
- **Línea 3:** la peor falla es la que no lanza nada.

Error común

Renombrar un campo “porque se ve mejor” en una API que ya tiene clientes HTTP.

Un arreglo suelto no deja crecer

```
GET /api/courts
{ "courts": [...] }
```

```
GET /api/members/:id/bookings
[...]
```

A un objeto se le agrega un campo. A un arreglo, no.

Pero AIP-158 agrega

“Adding pagination to an existing RPC is a backwards-incompatible change”.

Quien lee una respuesta recibe la primera página y no nota que faltan otras.

Idea clave

El objeto hace **posible** agregar `nextPageToken` sin cambiar la forma. No lo vuelve gratis: AIP-158 pide paginar desde el inicio.

Convivir: las versiones

Forma	Ejemplo	Quién
En la ruta	`/api/v2/members/2/bookings`	Google (versión mayor)
En un parámetro	`?api-version=2021-06-04`	Azure, en toda operación
En una cabecera	`X-Api-Version: 2`	Algunas API

- Azure prohíbe la primera: *"DO NOT include a version number segment in any operation path"*.
- Lo que comparten es la promesa: una versión publicada no se rompe.

Deprecar

Anunciar que algo dejará de existir, mientras sigue funcionando: convivir, anunciar con fecha, medir y recién entonces apagar.

Deprecation y Sunset

make recorrido-v2 (líneas cortadas)

```
$ curl -s -i localhost:3101/api/members/5/bookings
HTTP/1.1 200 OK
Deprecation: @1798761600
Sunset: Sat, 01 Jan 2028 00:00:00 GMT
Link: </api/v2/members/5/bookings>; rel="successor-version"
[{"id":9,"courtId":3,"memberId":5,"date":"2026-11-03","startHour":9,"hours":2,"teamName":"

$ curl -s -i localhost:3101/api/v2/members/5/bookings
HTTP/1.1 200 OK
{"bookings":[{"id":9,"courtId":3,"memberId":5,"date":"2026-11-03","startHour":9,"hours":2,

GET /api/members/5/bookings 200
GET /api/v2/members/5/bookings 200
```

- RFC 9745: *“The act of deprecation does not change any behavior of the resource”*.
- El log dice **cuánto** se usa, no **quién**: Kallpa no tiene autenticación.

Con IA: “ordena esta API”

- El cambio más limpio suele ser incompatible: renombra, unifica, pasa de `200` a `201`.
- Actualiza en el mismo commit los clientes HTTP y las pruebas del repositorio.
- Las pruebas pasan. Los clientes HTTP de fuera del repositorio, no.

La pregunta en la revisión

¿Este diff cambió alguna prueba de la API que ya existía? Si una prueba de contrato cambió, el contrato cambió.

SECCIÓN 07

OpenAPI

07



El contrato por escrito

openapi.yaml (extracto)

```yaml
openapi: 3.1.0
info:
  title: Kallpa API
  version: 1.0.0
security: []
paths:
  /api/courts:
    get:
      summary: Lista las canchas
      responses:
        '200':
          description: Las canchas
          content:
            application/json:
              schema: # ... { courts: [Court] }
```

- `paths`: una operación por ruta y método
- `responses`: una entrada por código
- `components/schemas` y `$ref`: DRY
- `security: []`: no hay autenticación, y se dice en voz alta (semana 15)

`redocly lint`: el esqueleto

redocly lint openapi.yaml (extracto)

```
[3] openapi.yaml:14:7 at #/paths/~1api~1courts/get/responses

Operation must have at least one ‘4XX‘ response.

14 | responses:
   | ^^^^^^^^^

Warning was generated by the operation-4xx-response rule.
```

- Cuatro advertencias, cero errores. Una advertencia es una pregunta, no una orden.
- `GET /api/courts` no responde ningún `4xx`. Si agregan uno para callar la regla, la especificación miente: se justifica con un comentario.

Lo que lint no revisa

`wrong-but-valid.yaml` dice que reservar responde `201` con `{ id }`. Kallpa responde otra cosa.

```
redocly lint lint-examples/wrong-but-valid.yaml –format=summary

validating lint-examples/wrong-but-valid.yaml...
warning info-license: 1
warning no-server-example.com: 1

lint-examples/wrong-but-valid.yaml: validated in 13ms

Woohoo! Your API description is valid.
You have 2 warnings.
```

Valid quiere decir “documento bien escrito”, no “es verdad”. Para saberlo hay que comparar con el servidor: `contract.ts` lo hace.

Con IA: un cliente contra el contrato

client-example.ts: ilustración escrita para la clase

```ts
const data: any = await response.json();
return data.courts.map((court: any) => `${court.courtName} (${court.sportType})`);
```

- Un cliente como los que suele proponer un agente sin la especificación a la vista.
- Compila, corre y muestra `undefined (undefined)` en las cuatro canchas.

node –test contract.test.ts (sin tiempos)

```
✓ el cliente de ejemplo corre sin errores y muestra basura
✓ la especificación detecta los campos inventados sin correr el cliente
```

Denle la especificación como contexto, y después comprueben.

El laboratorio: la API como es hoy

En parejas, en una rama nueva de su copia de Kallpa.

1. `make seed` y `make start`; recorrer con `curl -s -i` y anotar cada respuesta.
2. Repetir la reserva y cancelar dos veces: ¿qué sabe quien llama?
3. Copiar `code/week-06/openapi.yaml` a `docs/openapi.yaml`.
4. Describir los siete endpoints, con sus errores en un `enum`.
5. Cero errores de `redocly lint`; `redocly build-docs`.
6. `docs/api-revision.md`: al menos cuatro inconsistencias. ¿Corregirlas rompe? ¿Cómo convivirían?
7. Diseñar una clave de idempotencia para `POST /api/bookings`.

La regla

La especificación describe lo que la API hace hoy. Las mejoras se proponen aparte.

SECCIÓN 08

Patrones de diseño

08

Qué sigue vigente

Patrón de diseño ***(design pattern)***

Una solución con nombre a un problema que se repite, con sus ventajas y sus costos. No es código para copiar.

- Catálogo clásico: Gamma, Helm, Johnson y Vlissides (1994). 23 patrones, en C++ y Smalltalk.
- **Sigue:** el problema y el vocabulario. “Aquí va un adapter” ahorra un párrafo.
- **Cambia:** la forma. Algunos patrones le daban la vuelta a límites de los lenguajes de 1994.

Lo que dijo Norvig en 1996

"16 of 23 patterns are either invisible or simpler" en Dylan o Lisp, *"for at least some uses of each pattern"*.

Por	Patrones
Tipos de primera clase (6)	Abstract Factory, Flyweight, Factory Method, State, Proxy, Chain of Resp.
Funciones de primera clase (4)	Command, **Strategy**, Template Method, Visitor
Macros (2)	Interpreter, Iterator
Combinación de métodos (2)	Mediator, **Observer**
Multimétodos (1), módulos (1)	Builder; Facade

Lean qué midió

La revisión de un experto, sobre Dylan y Lisp, no un experimento. Adapter y Decorator no están entre los 16.

SECCIÓN 09

Strategy y dependency injection

09



Strategy: cambiar un algoritmo

como en el libro

```ts
export interface HourRanker {
  compare(a: number, b: number): number;
}

export class EarliestFirst
    implements HourRanker {
  compare(a: number, b: number): number {
    return a - b;
  }
}
// ... y una clase por cada algoritmo
```

en TypeScript

```ts
export type Rank =
  (a: number, b: number) => number;

export const earliestFirst: Rank =
  (a, b) => a - b;

export const closestTo =
  (target: number): Rank => (a, b) =>
    Math.abs(a - target)
      - Math.abs(b - target) || a - b;
```

- `sort` ya recibe un comparador: el patrón sigue ahí, sin la ceremonia.
- **En Kallpa:** `Clock`. `systemClock` en producción, `fixedClock` en las pruebas.
- **Costo:** para saber qué código corre, hay que mirar quién lo pasó.

Dependency injection: recibir, no crear

test/helpers.ts de Kallpa

```
1  const db = openDatabase(':memory:');
2  const sent: BookingConfirmation[] = [];
3  const notifier: Notifier = { bookingConfirmed: (confirmation) => sent.push(confirmation) };
4  const server = createServer({ db, clock: fixedClock(options.now ?? MONDAY_10AM), notifier, log: ()
       => {} });
```

- Fowler (2004): por **constructor**, **setter** o **interfaz**. Lo central: separar la configuración del uso.
- Kallpa: por constructor, y sin clases.
- Error común: creer que hace falta un contenedor. Pasar parámetros al construir el objeto **es** dependency injection.

SECCIÓN 10

Adapter y repository

10



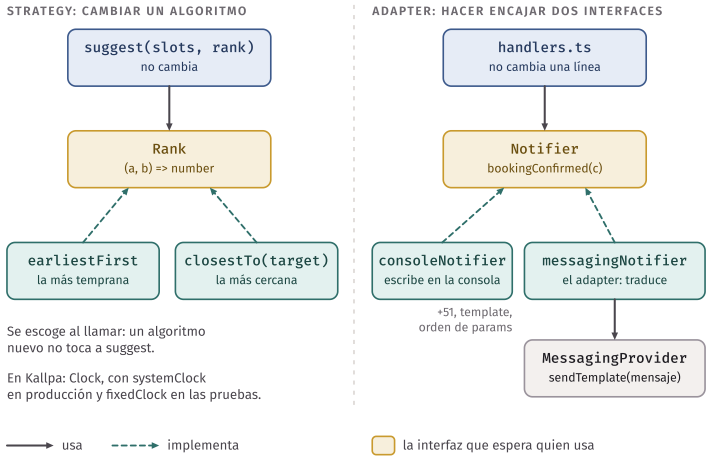
Adapter: hacer encajar dos interfaces

patterns/adapter.ts: el proveedor es inventado para la clase

```
export function messagingNotifier(provider: MessagingProvider): Notifier {
  return {
    bookingConfirmed(c: BookingConfirmation) {
      provider.sendTemplate({
        to: `+51${c.phone}`,
        template: 'reserva_confirmada',
        params: [c.memberName, c.courtName, c.date, `${c.startHour}:00`, String(c.hours), String(c
      .priceCents)],
      });
    },
  };
}
```

- Toda la traducción, en un solo lugar: `handlers.ts` no cambia.
- **Cuándo no:** para un proveedor que todavía no existe.

Strategy y adapter: la misma forma, otro fin



Repository: esconder cómo se guarda

Repository (Fowler)

"Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects."

`CourtRepository`
Devuelve `Court`, un tipo del dominio. Se ajusta a la definición.

`BookingRepository`
Devuelve `BookingRow`, con `start_hour` y `court_id`. Tiene el nombre, pero no esconde lo que promete.

Idea clave

Poner `Repository` en el nombre de una clase no la vuelve un repositorio.

SECCIÓN 11

Decorator y observer

11



Decorator: envolver sin cambiar

decorator-server.diff

```
type RawHandler = (req: IncomingMessage, res: ServerResponse) => Promise<void>;

function withLog(log: (line: string) => void, next: RawHandler): RawHandler {
  return async (req, res) => {
    try {
      await next(req, res);
    } finally {
      log(`${req.method} ${req.url} ${res.statusCode}`);
    }
  };
}

return http.createServer(withLog(log, withErrors(dispatcher(router))));
```

- Misma firma por fuera y por dentro. Los frameworks web le llaman **middleware**.
- `make decorator`: 40 de 40 pruebas, a cambio de 44 líneas nuevas y 21 borradas.

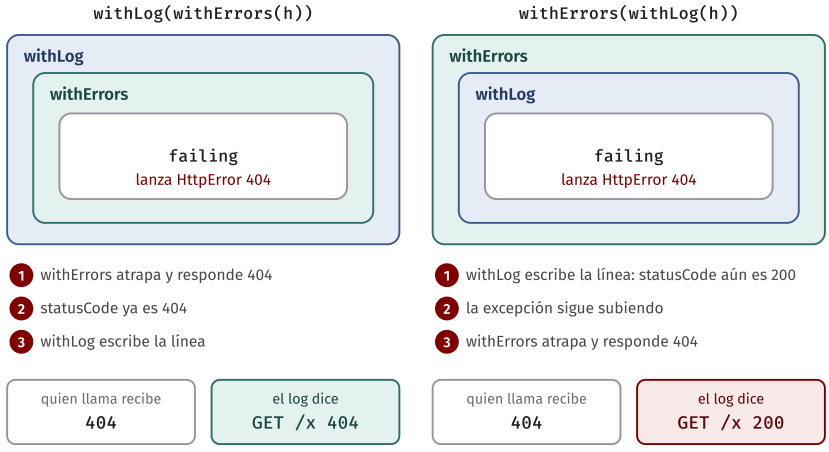
El orden importa

```
node --test patterns.test.ts (sin tiempos)

✓ decorator: withLog(withErrors(h)) registra el estado que recibió el cliente HTTP
✓ decorator: al revés, withErrors(withLog(h)) registra un 200 que nunca existió
```

Cuando `withLog` escribe su línea, nadie convirtió todavía la excepción en una respuesta. El tipo es el mismo y `tsc` no se queja: solo una prueba protege el orden.

El orden importa: las dos composiciones



Observer: quien produce no conoce a quien reacciona

patterns/observer.ts

```
saved.push(booking);
events.emit('booking-confirmed', booking);

// En otro lugar:
events.on('booking-confirmed', (e) => log.push(`whatsapp ${e.bookingId}`));
events.on('booking-confirmed', (e) => log.push(`caja ${e.priceCents}`));
```

- Se agregan reacciones sin tocar a quien produce.
- **Costo 1:** el flujo no se lee de corrido; hay que buscar quién escucha.
- **Costo 2:** el orden es implícito.
- **Cuándo no:** Kallpa hoy tiene un solo reaccionante. Un `EventEmitter` ahí sería prematuro.

Costo 3: un listener que falla afecta a todos

patterns.test.ts

```ts
events.on('booking-confirmed', () => {
  throw new Error('WhatsApp no responde');
});
events.on('booking-confirmed', () => log.push('caja'));
assert.throws(() => confirmBooking(events, saved, event), /WhatsApp no responde/);
assert.equal(saved.length, 1); // la reserva ya está guardada...
assert.deepEqual(log, []); // ...y la caja nunca se enteró
```

- `emit` es **síncrono**: llama a los listeners uno por uno, en la misma llamada.
- Quien llamó a la API recibe un `500` por una reserva que sí se hizo.

Idea clave: el desacoplamiento era del código, no de las fallas.

SECCIÓN 12

Cuándo un patrón estorba

12

Más interfaces sin esconder nada

patterns/overkill.ts

```ts
export interface NotifierFactory {
  create(): Notifier;
}
export class ConsoleNotifierFactory
    implements NotifierFactory {
  create(): Notifier {
    return consoleNotifier;
  }
}
export class NotifierFactoryProvider {
  getFactory(): NotifierFactory {
    return new ConsoleNotifierFactory();
  }
}
```

La prueba lo resume:

patterns.test.ts

```ts
assert.equal(
  new NotifierFactoryProvider()
    .getFactory().create(),
  consoleNotifier,
);
```

Módulos superficiales (semana 5). No abaratan ningún cambio.

Factory Method: una superclase con un método que las subclases redefinen. En TypeScript suele bastar una función que elige.

El Singleton

patterns/overkill.ts

```ts
// El código que depende del Singleton no lo dice en su firma.
export function isMorningWithSingleton(): boolean {
  return GlobalClock.getInstance().now().getUTCHours() < 17; // 12:00 en Lima
}
// El mismo código con el reloj como parámetro.
export function isMorning(clock: Clock): boolean {
  return clock.now().getUTCHours() < 17;
}
```

- Estado global con otro nombre: lo que una prueba le hace le llega a la siguiente.
- ¿Una sola instancia? Se crea una, en un solo lugar, y se pasa. Kallpa: `main.ts` abre la base de datos y `createServer` la recibe.

La pregunta

¿Qué cambio futuro abarata esto, y qué tan probable es?

Patrón	Abarata	En Kallpa
Strategy	cambiar un algoritmo	`Clock`
Dependency injection	reemplazar una dependencia	`createServer`
Adapter	cambiar de proveedor	`Notifier`, el día en que haya uno
Repository	cambiar cómo se guarda	`src/store/`, con una filtración
Decorator	agregar cosas alrededor	el `try/catch/finally` de `server.ts`
Observer	agregar reacciones	todavía no

Si no se puede nombrar el cambio, el patrón probablemente sobra.

Con IA: patrones de más

- Los modelos de lenguaje reproducen los patrones que más aparecen en el código con el que se entrenaron, a veces con más ceremonia de la necesaria.
- Es una observación de la práctica, no un resultado medido, y varía según la herramienta y el pedido.
- Un `NotifierFactoryProvider` pasa `tsc`, ESLint y todas las pruebas. Ninguna herramienta lo marca.

En la revisión

Para cada abstracción nueva del diff: ¿qué cambio facilita? ¿Cuántas implementaciones tiene hoy?

En resumen

- Una API es un contrato con quien no está en la sala.
- Idempotente es sobre el efecto buscado, no sobre la respuesta ni sobre el método solamente.
- Una respuesta perdida deja a quien llama sin saber. Una clave de idempotencia reconoce el reintento; la garantía exige la misma transacción.
- Compatible es lo que no rompe a quien ya llama. Versionar, anunciar, medir y recién apagar.
- OpenAPI escribe el contrato. Lint revisa el documento, no el servidor.
- Un patrón es una respuesta: siempre con la pregunta de qué cambio abarata.

Para la próxima clase

- Terminar el laboratorio: `docs/openapi.yaml` y `docs/api-revision.md`.
- P2 pide una especificación como esta para su proyecto (`docs/api/openapi.yaml`); se entrega en la semana 8.
- Semana 7: arquitectura y ADRs. Leer Fowler y Lewis, *Microservices*, y Fowler, *MonolithFirst*.
- Apuntes: `notes/week-06.md`. Código: `code/week-06`.

> **Pregunta**
>
> Si hoy tuvieran que cambiar `GET /api/members/:id/bookings` para que devuelva un objeto, ¿a quién le avisarían? ¿Cómo sabrían si ya pueden apagar la versión vieja?