

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Arquitectura y decisiones de arquitectura

Capas, hexagonal, monolito y microservicios · Atributos de calidad · ADR · C4

Gerson Garrido

Semana 7

Hoy

Sesión A

1. Qué es arquitectura
2. Capas
3. Puertos y adaptadores
4. Monolito y microservicios

Sesión B

5. Atributos de calidad
6. El caso de las dos instancias
7. ADR y deuda técnica
8. Modelo C4

Lecturas

A: Fowler y Lewis, *Microservices*; Fowler, *MonolithFirst*; Cockburn, *Hexagonal Architecture*.

B: Nygard, *Documenting Architecture Decisions*; MADR; Brown, *The C4 model*.

Complementarias: Fowler, *PresentationDomainDataLayering* y *TechnicalDebtQuadrant*; SQLite, *Appropriate Uses For SQLite*.

Un pedido que parece de configuración

La administración del complejo

“Queremos correr dos copias del servidor sobre el mismo archivo, para que una caída no deje a los vecinos sin reservar.”

- Parece una línea en un script.
- Medido en Kallpa: la mitad de las peticiones simultáneas termina en 500.
- El arreglo obvio quita los 500 y deja canchas reservadas dos veces.

¿Cómo se decide algo así? ¿Y cómo se deja escrito?

SECCIÓN 01

Qué es arquitectura

01

Una definición para decidir

Arquitectura

Las decisiones de un sistema que son caras de cambiar después: cómo se divide en partes, cómo se comunican, dónde viven los datos y cómo se despliega.

- Lo que cuenta es el costo de cambiarlo, no el tema ni el tamaño del dibujo.
- Nygard mira dónde buscarlas: estructura, características no funcionales, dependencias, interfaces, técnicas de construcción.
- Diseño y arquitectura son lo mismo a distinta escala: cambia el costo de equivocarse.

Una prueba con Kallpa

Decisión	¿Qué cuesta cambiarla?	¿Arquitectura?
Puerto 3000 por omisión	Una variable de entorno	No
`quotePriceCents` se llama así	Un renombre automático	No
Página sin frameworks	Reescribir `public/`	Poco
Montos en céntimos enteros	Columnas, cálculos, pruebas, datos	Sí
Fecha en texto, hora entera	Esquema, datos, API	Sí
SQLite en el mismo proceso	Mucho más que `store`	Sí
Un solo programa desplegado	Datos, redes, despliegues, fallas	Sí

¿No lo esconde `store`?

- El SQL está en `store`. ¿Cambiar a PostgreSQL es reescribir esa carpeta?
- No: `node:sqlite` es síncrono; un cliente de PostgreSQL, asíncrono.
- Todos los métodos devuelven `Promise`. Todos los handlers usan `await`.
- Y aparece un punto entre revisar y guardar donde corren otras peticiones.

Límite del ocultamiento

Un módulo esconde sus decisiones hasta donde su interfaz lo permite. “Síncrono o tarda lo que tarda la red” es parte de la interfaz.

SECCIÓN 02

Capas

02



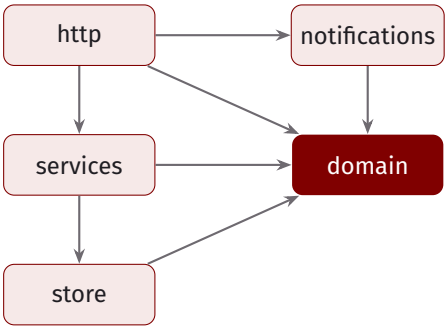
Arquitectura en capas

Capas

Cada capa usa solo a las de abajo y no sabe nada de las de arriba.

Fowler: al separar presentación, dominio y datos, cuando se trabaja en una se puede ignorar casi todo de las otras.

Kallpa es una versión relajada: `domain` va al costado y casi todos lo usan.



Medir en vez de creer

```
make dependencias

== src/http
      6 from '../domain/
      2 from '../notifications/
      2 from '../services/
      1 from '../store/
== src/services
      2 from '../domain/
      3 from '../store/
== src/store
      6 from '../domain/
== src/domain
== src/notifications
      1 from '../domain/
```

- `domain` no importa a nadie: las reglas se prueban sin base de datos.
- Y esa línea: `http` importa algo de `store`, saltándose a `services`.
- ¿Qué es? ¿Importa?

“Es solo un tipo”

src/http/handlers.ts

```ts
import type { BookingRow } from '../store/booking-repository.ts';

export function toBookingJson(row: BookingRow): Booking {
  return {
    id: row.id,
    courtId: row.court_id,
    memberId: row.member_id,
```

- En ejecución esa línea no existe: Node quita los tipos.
- Pero si alguien renombra `court_id`, tiene que cambiar `handlers.ts`.
- Acoplarse es necesitar saber algo del otro. Un tipo es conocimiento.

La regla como revisión automática

eslint.config.js

```
{
  files: ['src/domain/**/*.ts'],
  rules: { 'no-restricted-imports': ['error', {
    patterns: ['../store/*', '../http/*', '../services/*', 'node:sqlite', 'node:http'] }] },
},
{
  files: ['src/http/**/*.ts'],
  rules: { 'no-restricted-imports': ['error', { patterns: ['../store/*'] }] },
},
```

make capas

```
/work/code/week-07/build/kallpa/src/http/handlers.ts
  18:1  error  '../store/booking-repository.ts' import is restricted from being used by a pattern  no-restricted-imports
✗ 1 problem (1 error, 0 warnings)
```

Lo que la regla no mira

- **Revisa lo que se le dijo.** `http/server.ts` importa `DatabaseSync` de `node:sqlite`: no se reporta.
- **Declaraciones `import`, no todo.** Un `import('../store/database.ts')` dinámico en `domain` pasa sin aviso (hay una prueba).
- **Imports, no SQL.** Un handler que recibe la conexión puede llamar a `db.prepare(...)` sin importar nada de `store`.

Error común

Tratar la arquitectura como un dibujo. La pregunta útil es: ¿qué pasa si alguien lo viola? Si la respuesta es “nada”, el diagrama no protege a nadie.

SECCIÓN 03

Puertos y adaptadores

03



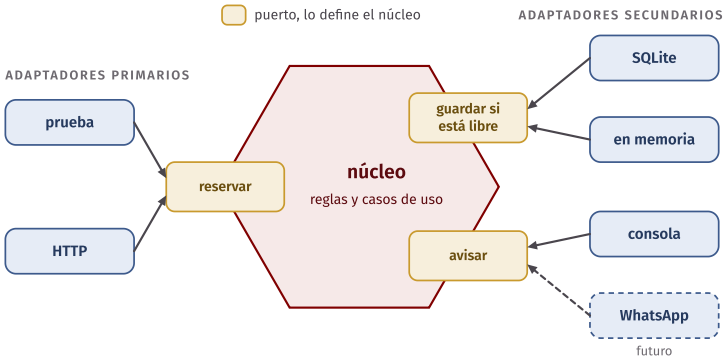
El problema que resuelve Cockburn

Intención del patrón

Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

- Una fuga arriba: reglas en los controladores de la interfaz.
- Una fuga abajo: reglas mezcladas con el SQL.
- El hexágono no es por el número seis: es para tener espacio y salir del dibujo de una sola dimensión.

Puertos y adaptadores



Las flechas dicen quién conoce a quién: todas apuntan hacia el núcleo.

Al ejecutar, el núcleo llama a los adaptadores secundarios, pero solo los conoce por el puerto que él definió.

Las dependencias apuntan hacia el núcleo: el adaptador de SQLite conoce al núcleo; el núcleo no conoce a SQLite.

Definiciones

Puerto

Una conversación con un propósito, definida por el núcleo en sus términos. En código, una interfaz. *“Guardar esta reserva si la hora está libre”* es un puerto; *“ejecutar este INSERT”*, no.

Adaptador

El código que conecta un puerto con una tecnología concreta. Por cada puerto suele haber varios: el de producción, el de pruebas.

Kallpa como hexágono: la versión honesta

Lo que sí

- `Clock`: `systemClock`, `fixedClock`.
- `Notifier`: `consoleNotifier` y el doble de prueba.
- Por eso las pruebas fijan la hora y capturan los avisos sin tocar el código de reservas.

Lo que no

- Los datos no tienen puerto: `BookingService` arma repositorios de SQLite.
- El caso de uso vive en un adaptador: `createBooking` está en `src/http`.
- `Notifier` se define junto a su adaptador, fuera del núcleo.

Un sistema real casi nunca es un estilo puro. Describirlo bien sirve para decidir qué fuga vale la pena arreglar.

Un puerto para los datos

hexagon/ports.ts

```ts
export type Reservations = {
  // Promete las dos cosas juntas: revisar que la hora esté libre y guardar. Quien
  // implementa el puerto decide cómo; si devuelve null, la cancha estaba ocupada.
  bookIfFree(booking: NewBooking): SavedBooking | null;
};
```

- Dos adaptadores: en memoria y SQLite con transacción.
- Las mismas pruebas corren contra los dos: el puerto es un contrato.
- No ofrece `list` e `insert` por separado: “revisar y guardar juntos” no queda en manos de quien llama.

Capas o hexágono

	En capas (Kallpa)	Hexagonal
¿Quién define la interfaz de datos?	`store`, la capa de abajo	El núcleo, según lo que necesita
La lógica depende de…	Indirectamente, la base de datos	Nadie
Probar un caso de uso	Con base de datos, aunque en memoria	Con cualquier adaptador
Costo	Menos piezas	Una interfaz más por cosa externa

Error común

Una interfaz por clase “para que sea hexagonal”. Con los mismos métodos de su única implementación, no esconde nada.

SECCIÓN 04

Monolito y microservicios

04



Monolito

Monolito

Un solo programa que se despliega entero. Dice cómo se ejecuta, no qué tan bueno es su código.

Fowler y Lewis: también escala a lo ancho, *"running many instances behind a load-balancer"*. Lo que no puede es escalar solo una parte.

Ventajas reales

- Una sola base de código
- Llamadas locales
- Transacciones simples
- Un solo despliegue

Costos

- Cambios atados: publicar todo
- Con los años cuesta conservar los módulos

Microservicios (Fowler y Lewis, 2014)

1. Componentes como servicios
2. Organizados por capacidades del negocio
3. Productos, no proyectos
4. *Smart endpoints and dumb pipes*
5. Gobierno descentralizado
6. Datos descentralizados
7. Automatización de la infraestructura
8. Diseño para las fallas
9. Diseño evolutivo

Lo central

Servicios que se despliegan por separado, cada uno con sus datos, comunicándose por la red.

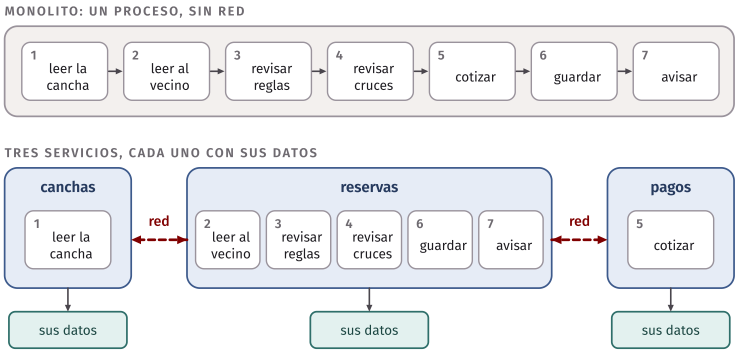
No es una definición formal: son características comunes.

Lo que se paga

Con un monolito	Con microservicios
Una llamada a función siempre llega	Puede fallar, tardar o llegar dos veces
Una transacción abarca todo	Sin transacción común: consistencia eventual y compensar errores
Un log, un proceso	Trazar una petición por varios servicios
Una prueba de punta a punta	Levantar varios servicios o simular contratos
Una versión	Varias conviviendo: APIs compatibles hacia atrás
Un despliegue	Un pipeline por servicio y alguien que opere todo

Se gana: despliegues independientes, escalar solo una parte, fronteras que el lenguaje no deja cruzar, aislamiento de algunas fallas.

Kallpa en tres servicios



Cada llamada por la red puede fallar, tardar o llegar dos veces.

¿Se puede reservar con canchas caído? Si reservas guarda y el cobro en pagos falla, ¿quién deshace la reserva?
Sin una transacción común, hay que aceptar que la consistencia sea eventual.

Son preguntas que hoy no existen. Y Kallpa recibe unas decenas de reservas al día.

Monolito modular

Monolito modular

Se despliega como un solo programa, con módulos de fronteras claras que se hacen cumplir. Cada módulo expone una interfaz y esconde sus tablas.

- Tiene las transacciones y el despliegue único del monolito.
- Ninguna red impide cruzar la frontera: la disciplina viene de una herramienta, como la regla de ESLint.

Dos casos, contados por sus equipos (experiencia, no comparación controlada)

Shopify (2019): Rails, más de mil desarrolladores; eligen monolito modular.
Segment (2018): junta más de 140 servicios en uno por el costo de operarlos, y pierde aislamiento de fallas.

La ley de Conway

> **Melvin Conway, 1968, citado por Fowler y Lewis**
>
> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*

- Equipos de interfaz, lógica y base de datos: un cambio pequeño es un proyecto entre tres equipos.
- Es una observación sobre organizaciones, no una ley física.
- Por eso un equipo de cinco, con un cliente, no tiene la organización que justifica diez servicios: la red no refleja ninguna frontera de comunicación, la inventa.

Monolith first

> **Fowler, 2015**
>
> *Almost all the successful microservice stories have started with a monolith that got too big and was broken up. Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.*

- **Razón 1:** no se sabe aún si el producto sirve, y el costo de los microservicios frena.
- **Razón 2:** las fronteras buenas se aprenden. Moverlas entre servicios es mucho más difícil que entre módulos.
- **Sus reservas:** son anécdotas; no hay unanimidad; "cualquier consejo debe verse como tentativo". Y no cualquier monolito se puede partir.

Los errores de cada lado

Error común

Elegir microservicios “porque escalan” sin un requisito de escala. Kallpa recibe unas decenas de reservas al día.

SQLite: un sitio con menos de 100 mil visitas al día debería funcionar bien (estimado conservador, no un límite).

Y al revés

“Es un monolito” no es excusa para que todo dependa de todo. Lo contrario de un microservicio no es un desorden.

En su proyecto suele ser un monolito con buenos módulos. En P2 importa poder explicar por qué.

Con IA: el camino más corto cruza capas

Un cambio escrito para la clase, como los que suele proponer un agente: el handler recibe la conexión y crea un `BookingRepository`.

```
make cruce

ℹ tests 40
ℹ suites 6
ℹ pass 40
ℹ fail 0
...
cd build/cruce && eslint -c capas-eslint.js . ; echo "eslint terminó con código $?"

/work/code/week-07/build/cruce/src/http/handlers.ts
  19:1  error  '../store/booking-repository.ts' import is restricted from being used by a pattern  no-restricted-imports

✗ 1 problem (1 error, 0 warnings)

eslint terminó con código 1
```

- El `make check` de Kallpa pasa. Solo la regla lo detecta.
- Los ADR y el diagrama dan contexto; la regla detecta el cruce. Y tiene huecos: el `node:sqlite` nuevo no se reporta. Sigue haciendo falta la revisión.

SECCIÓN 05

Atributos de calidad

05

Atributos de calidad

Atributo de calidad ***(quality attribute)***

Qué tan bien hace el sistema lo que hace. Los requisitos no funcionales de la semana 3, vistos desde la arquitectura.

ISO/IEC 25010 los clasifica. Una lista no decide nada: lo que importa es que cada uno se pueda medir.

- Disponibilidad
- Rendimiento
- Seguridad
- Mantenibilidad
- Facilidad de prueba
- Facilidad de despliegue
- Costo

Un atributo se escribe como escenario

Estímulo	Entorno	Respuesta	Medida
20 vecinos reservan a la vez	operación normal, lunes 8:00	confirmación u hora ocupada	p99 $<$ 500 ms; ninguna doble reserva

“Kallpa debe ser rápida” no se puede verificar. Esto sí.

Bass, Clements y Kazman agregan la fuente del estímulo y la parte afectada; si hacen falta, se usan.

Percentil

Que el p99 sea 500 ms: el 99 % tarda eso o menos. Con 20 muestras, el p99 es la más lenta: hacen falta suficientes.

De vago a medible

Vago	Medible
Siempre disponible	Con el proceso caído, vuelve en menos de 1 minuto; en un mes, no más de 43 minutos fuera de servicio (99.9 %)
Seguro	Un vecino que manipula la petición no puede cancelar la reserva de otro: 403 en el 100 % de los casos
Fácil de mantener	Agregar un deporte con su tarifa toca a lo sumo 2 archivos y se publica en menos de un día
Fácil de desplegar	Alguien nuevo levanta el sistema con el README en menos de 15 minutos

El número sale de lo que le duele al cliente, no de lo que suena bien.

Medir el escenario con dos instancias

```
make medir

Una instancia:
  100 rondas, 2000 peticiones: 100×200  1900×409
  latencia: p50 3.7 ms, p99 7.5 ms, máximo 11.1 ms
  horas reservadas dos veces: 0
Dos instancias, sobre la Kallpa publicada:
  100 rondas, 2000 peticiones: 100×200  893×409  1007×500
  latencia: p50 2.4 ms, p99 7.3 ms, máximo 11.2 ms
  horas reservadas dos veces: 0
```

- Dos instancias: el escenario se cumple al pie de la letra, con la mitad de las peticiones en 500.
- La latencia hasta baja: fallar es rápido. La medida estaba incompleta.
- Mejor: … menos de 500 ms, ninguna responde 5xx y ninguna cancha reservada dos veces.

Los atributos compiten

Se mejora	Se empeora	En Kallpa
Disponibilidad con dos instancias	Consistencia, simplicidad	Coordinar los datos y repartir peticiones
Seguridad con autenticación	Facilidad de uso	Un vecino ya no reserva en dos clics
Rendimiento con cache	Consistencia	La página muestra libre una hora tomada
Facilidad de prueba con puertos	Simplicidad	Una interfaz más por cosa externa

Decidir es elegir qué atributo gana, para este cliente, y dejar escrito qué se pierde.

SECCIÓN 06

El caso de las dos instancias

06

Revisar y después guardar

src/http/handlers.ts

```
1  const request = parseBookingRequest(await readJson(req));            // línea 79
2  const court = deps.service.findCourt(request.courtId);               // línea 81
3  const sameDay = deps.service.bookingsForCourt(court.id, request.date)// línea 92: revisa
4    .filter((b) => b.status === 'confirmed');
5  if (sameDay.some((b) => overlaps(/* ... */))) { throw /* 409 */; }
6  const id = deps.service.saveBooking({ /* ... */ });                  // línea 104: guarda
```

Condición de carrera *(race condition)*

El resultado depende del orden en que se intercalan las operaciones.
Check-then-act: si otro actúa entre revisar y actuar, lo revisado ya no es cierto.

En un proceso no pasa… hasta que pasa

- Node corre una función hasta el siguiente `await`. El único está en la línea 79.
- De la línea 81 en adelante todo es síncrono: `DatabaseSync`.
- 20 reservas simultáneas: **un 200 y diecinueve 409**.

await-intermedio.diff, entre revisar y guardar

```
1  await new Promise((resolve) => setTimeout(resolve, 10));
```

- Veinte reservas confirmadas para la misma hora, en un solo proceso.
- Y las 40 pruebas de Kallpa pasan: ninguna manda peticiones simultáneas.

Con dos procesos: el entrelazado

Instancia A	Instancia B	En la base de datos
Revisa: 9:00 libre		Sin reservas
	Revisa: 9:00 libre	Sin reservas
Guarda “Equipo A”		1 reserva
	Guarda “Equipo B”	2 reservas a la misma hora

No hace falta esperar a tener suerte: `doble-reserva.test.ts` fuerza ese orden con dos conexiones y el `BookingService` real. Forzar el orden da siempre el mismo resultado.

Lo que pasa cuando se corre de verdad

```
make dos-instancias
```

```
Respuestas (cantidad y código HTTP):
      1 200
     13 409
      6 500
Reservas confirmadas de la cancha 3 el 2026-11-05 a las 9:00:
16|Prueba 2
```

```
/tmp/b.log
```

```
Error: database is locked
    at CourtRepository.findById (file:///work/code/week-07/build/kallpa/src/store/court-repository.ts:16:80)
    at BookingService.findCourt (file:///work/code/week-07/build/kallpa/src/services/booking-service.ts:25:25)
    at Object.handler (file:///work/code/week-07/build/kallpa/src/http/handlers.ts:81:32)
```

- Seis corridas seguidas: 3, 8, 2, 6, 10 y 3 errores. Cambia cada vez.
- Falló un SELECT: una lectura, ni siquiera una escritura. ¿Por qué?

Los locks de SQLite

Lock	Quién lo tiene	Qué permite a los demás
SHARED	Cualquiera que lea; pueden ser muchos	Leer
RESERVED	Uno solo, que se prepara para escribir	Leer, pero no escribir
PENDING y EXCLUSIVE	Uno solo, mientras escribe en el archivo	Nada nuevo

- Con `busy_timeout` en 0, como en Kallpa, SQLite no espera: `database is locked`.
- En la medición, en 99 de 100 rondas una instancia respondió 500 a todo lo que recibió: mientras una confirma su `INSERT`, la otra atiende sus 10 y todas chocan.
- Con `curl` son menos. El número depende de cómo llegan las peticiones.

Una casi doble reserva

otra traza del mismo script

```
Error: database is locked
    at BookingRepository.insert (file:///work/code/week-07/build/kallpa/src/store/booking-repository.ts:54:8)
    at BookingService.saveBooking (file:///work/code/week-07/build/kallpa/src/services/booking-service.ts:53:27)
    at Object.handler (file:///work/code/week-07/build/kallpa/src/http/handlers.ts:104:29)
```

- Línea 104: esta petición ya pasó la revisión, y falló al guardar.
- Lo único que evitó una segunda reserva fue el lock.
- Lo que nos protege es un error que nadie pidió, no una decisión de diseño.
- En más de 40 corridas (más de 4000 rondas) no apareció ninguna doble reserva. Eso no quiere decir que no pueda pasar: la prueba forzada lo muestra.

El arreglo obvio

solo-espera.diff

```
1  const db = new DatabaseSync(path, { timeout: 5000 });
```

make medir

```
Dos instancias, con solo-espera.diff:
  100 rondas, 2000 peticiones: 172×200  1828×409
  latencia: p50 5.6 ms, p99 15.1 ms, máximo 17.4 ms
  horas reservadas dos veces: 72
```

- Sin errores 500, y las pruebas de Kallpa pasan.
- Y 72 de 100 horas reservadas dos veces: 172 respuestas 200, con su aviso. En veinte corridas, de 1 a 79, nunca 0.

La idea central del caso

Un invariante entre procesos

Revisar y guardar tienen que ser una sola operación indivisible para la base de datos.

- Esperar el lock no los une: el segundo espera y guarda con una revisión vencida.
- El error 500 molesta y se ve. La doble reserva es silenciosa: la descubre un equipo al llegar a la cancha.
- El escenario solo lo detectó porque medía las dobles reservas.

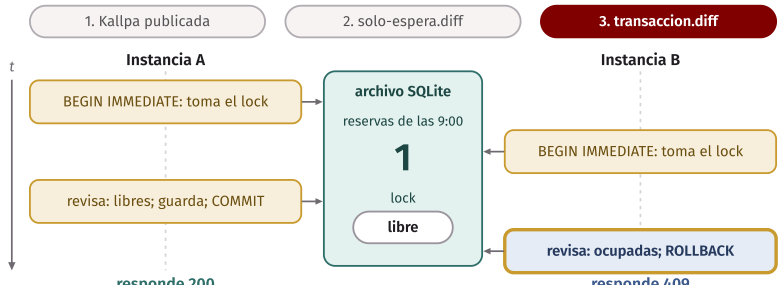
Alternativa: una transacción con `BEGIN IMMEDIATE`

```
make medir

Dos instancias, con transaccion.diff:
  100 rondas, 2000 peticiones: 100×200  1900×409
  latencia: p50 4.8 ms, p99 12.4 ms, máximo 13.9 ms
  horas reservadas dos veces: 0
```

- Toma el lock antes de revisar. Con `BEGIN` a secas, el segundo en escribir falla (hay una prueba).
- Mientras SQLite espera, Node está bloqueado: `DatabaseSync` es síncrono.
- El diff la pone en el handler para que se lea en pocas líneas. Según la semana 5, su lugar es un módulo profundo.

El mismo entrelazado, tres versiones



Revisar y guardar son una sola operación para la base de datos: B ve lo que A guardó.

UNA CORRIDA DE MAKE MEDIR: 2000 PETICIONES EN 100 RONDAS

	200	409	500	horas reservadas dos veces
Kallpa publicada	100	893	1007	0
con solo-espera.diff	172	1828	0	72
con transaccion.diff	100	1900	0	0

Alternativa: una restricción, y una trampa

```sql
CREATE UNIQUE INDEX one_booking_per_start ON bookings (court_id, date, start_hour)
  WHERE status = 'confirmed';
```

- Detiene dos reservas con la misma hora de inicio.
- No detiene un cruce: de 9:00 a 11:00 y de 10:00 a 11:00 tienen distinta `start_hour`.
- Para expresar la regla: una fila por hora ocupada, `booking_hours (court_id, date, hour)`. Exige migración y mantener las filas al cancelar.
- A cambio, la regla vale para todo el que escriba.

Y las demás

- **Una sola instancia, y aceptar el riesgo.** Hoy no tiene 500 ni dobles reservas (salvo que entre un `await`). Medido: mediana de 101 ms para volver a responder si algo la reinicia. No incluye detectar la caída ni sirve si cae la máquina.
- **PostgreSQL.** Ordena las peticiones, admite varias máquinas, tiene *exclusion constraints*. Y un servidor más, y todas las capas se vuelven asíncronas.
- **Modo WAL.** Los lectores no se bloquean. Sigue habiendo un solo escritor. No une revisar con guardar.

Dos preguntas sobre el pedido mismo

¿De qué caída protege? Dos procesos en una máquina no protegen de que caiga la máquina o el disco, y SQLite desaconseja compartir el archivo por red.

¿Quién reparte a los vecinos? Hace falta un load balancer: otra pieza que puede caerse.

SECCIÓN 07

ADR y deuda técnica

07

Un año después, alguien nuevo

Ve que solo corre una instancia. Sin saber por qué, Nygard le deja dos opciones:

Aceptar a ciegas

Quizá la razón ya no vale. El equipo le termina teniendo miedo a cambiar cualquier cosa.

Cambiar a ciegas

Quizá sostenía algo que nadie está probando, como la regla de los cruces.

Los documentos largos no ayudan: nadie los lee ni los mantiene. Nygard (2011): un documento corto, por cada decisión importante.

ADR

ADR *(architecture decision record)*

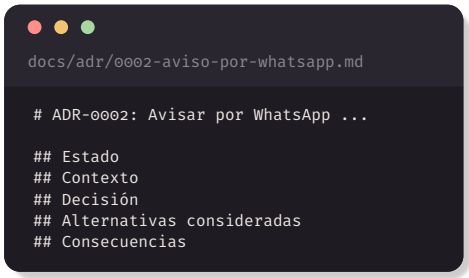
Un documento corto con **título**, **estado**, **contexto**, **decisión** y **consecuencias**, también las malas.

MADR agrega las alternativas consideradas, con pros y contras.

Reglas

- En el repositorio y revisado en un PR
- Numerado, sin reutilizar números
- Una o dos páginas
- Una conversación con quien lea después
- Si cambia, no se reescribe: uno nuevo lo reemplaza; en el viejo solo cambia el estado

El formato del curso (P2)



- `Estado`: Propuesta, Aceptada, Rechazada, Reemplazada u Obsoleta.
- Lo revisa `P2-ADR`: al menos 3, con todas las secciones.
- La idea es la de MADR, con títulos en español.

Ejemplo completo en `code/week-07/adr-ejemplo/`, sobre otra decisión de Kallpa: cómo mandar los avisos de verdad.

El contexto, con sus fuerzas

- El aviso es un complemento: un aviso caído no debe impedir reservar.
- Un `await` a un proveedor entre revisar y guardar rompería los cruces.
- Dos personas a tiempo parcial; unas decenas de reservas al día; no se paga otro servidor.

Escenario de calidad

Con el proveedor de WhatsApp caído durante 10 minutos, en operación normal, el 99 % de las reservas responde en menos de 500 ms y, cuando el proveedor vuelve, cada aviso sale en menos de 2 minutos.

Alternativas con pros y contras

Opción	A favor	En contra
Llamar dentro de la petición	Un adaptador y nada más	Si tarda, tarda la reserva. No cumple el escenario
Tabla de pendientes y temporizador (elegida)	Cumple. Se guarda junto con la reserva. Sin piezas nuevas	Hasta 30 s de retraso. Puede llegar repetido
Cola externa y proceso aparte	Habitual con mucho volumen; escala aparte	Dos piezas más. Ya no hay una sola transacción. Nadie la ha operado

Consecuencias: un vecino puede recibir dos veces un mensaje; el temporizador supone una sola instancia. Cuándo revisarla: si pasa de unos cientos de avisos al día.

Dos errores comunes

Un ADR sin alternativas

"Decidimos usar SQLite." Quien llegue en un año no sabe si la decisión sigue vigente cuando cambie el contexto.

Escribirlo después

Para justificar lo que ya se hizo. El valor está en decidir con el contexto a la vista.

ADR o design doc: el design doc describe cómo se construye algo, y envejece. El ADR registra una decisión y su motivo, y se conserva aunque cambie.

La deuda técnica como decisión

	Imprudente	**Prudente**
Deliberada	“No hay tiempo para diseñar”	“Publiquemos ahora y asumamos las consecuencias”
Inadvertida	“¿Qué son las capas?”	“Ahora sabemos cómo debimos hacerlo”

Fowler, *TechnicalDebtQuadrant*

Con un ADR

“El invariante depende de que el tramo entre revisar y guardar no tenga `await`. Si hace falta una segunda instancia o una llamada externa, se paga con una transacción.”

Deja de ser un defecto escondido: es una apuesta con su condición de pago.

Con IA: ADR y diagramas

- Son contexto para el agente, para que no “rediseñe” sin querer.
- Un modelo puede redactar el borrador. Las alternativas y consecuencias salen del equipo y del cliente: no sabe cuántos mantendrán el sistema ni cuánto puede pagar el complejo.
- Para comprobar un borrador:
  - ¿Cada número del contexto tiene de dónde salir?
  - ¿Alguien del equipo conoce cada alternativa?
  - ¿Las consecuencias son de este proyecto o servirían para cualquiera?
- Un borrador que proponga `timeout` para resolver `database is locked` se ve razonable. Solo la medición muestra que es peor.

SECCIÓN 08

Modelo C4

08

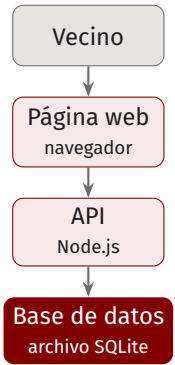
Cuatro niveles de zoom

1. Contexto	El sistema como una caja, sus usuarios y sistemas externos: vecinos, administración, WhatsApp.
2. Contenedores	Lo que se ejecuta o guarda datos por separado: página, API en Node.js, archivo SQLite.
3. Componentes	Las partes de un contenedor: `http`, `services`, `store`, `domain`, `notifications`.
4. Código	Clases y funciones. Casi nunca se dibuja a mano.

Contenedor no es Docker

Error común

La página de C4 lo dice: *Not Docker!* Un contenedor es una aplicación o un almacén de datos: algo que tiene que estar corriendo o existir para que el sistema funcione.



La página corre en otro lugar. El archivo no tiene proceso, pero sin él nada funciona. Dos instancias no agregan un contenedor: se ven en el diagrama de despliegue.

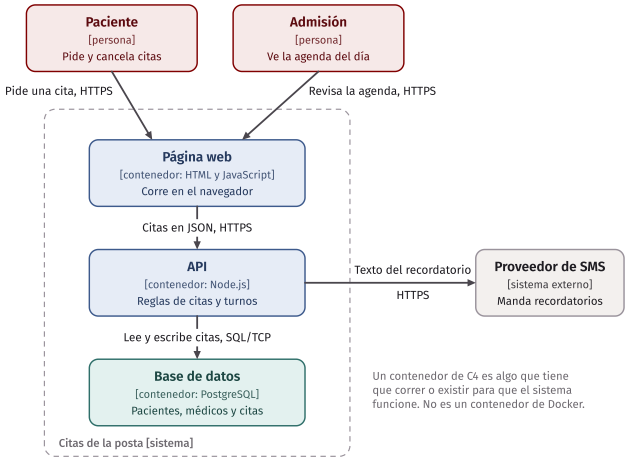
Diagramas como texto: Mermaid en GitHub

docs/arquitectura.md

```
C4Context
  title Kallpa: diagrama de contexto
  Person(vecino, "Vecino", "Reserva y cancela canchas")
  Person(admin, "Administración", "Consulta la caja del día")
  System(kallpa, "Kallpa", "Reservas de canchas del complejo")
  System_Ext(whatsapp, "WhatsApp", "Por ahora, la consola del servidor")
  Rel(vecino, kallpa, "Reserva y cancela", "navegador")
  Rel(admin, kallpa, "Consulta la caja", "navegador")
  Rel(kallpa, whatsapp, "Avisa la reserva confirmada")
```

- En un bloque `mermaid`, GitHub lo dibuja. Vive en el repo y se revisa en un PR.
- Lo más útil está en las flechas: qué viaja y por qué protocolo. “Usa” no dice nada.
- Si queda amontonado, un `flowchart` también vale: C4 es una forma de pensar los niveles.

Otro sistema: los contenedores de las citas de una posta



SECCIÓN 09

Laboratorio

09

Laboratorio: un ADR y un diagrama de contenedores

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

1. Mapear las dependencias reales y compararlas con el README.
2. `no-restricted-imports` en `eslint.config.js`. `make lint`. ¿Se corrige el código o la regla?
3. Contexto, contenedores y componentes en Mermaid, en `docs/arquitectura.md`.
4. `bash /work/code/week-07/dos-instancias.sh`: respuestas, base de datos, logs.
5. Explicar: ¿dónde se revisa, dónde se guarda, qué tendría que pasar en el medio?
6. `docs/adr/0001-instancias-y-cruces-de-reservas.md`: escenario medible, al menos tres alternativas, decisión, consecuencias y deuda.
7. Pull request. La revisión responde: ¿alguien que no estuvo entendería por qué se decidió así?

> **Se evalúa el razonamiento**
>
> Se puede defender más de una respuesta. Si detuvieron `make start`, el script corre; si no, se niega.

Para la próxima clase

- Terminar el laboratorio: el pull request con el diagrama, el ADR y el cambio de ESLint, y la revisión de otra pareja.
- **P2**: se entrega la semana 8, con la demo del sprint 1. Al menos 3 ADR, con el formato de hoy.
- Semana 8: repaso y examen parcial. Repasen las semanas 1 a 7 y sus preguntas de repaso. Para el examen pesa, entre otros, el ADR de Nygard.

Pregunta

En su proyecto, ¿cuál es la decisión más cara de cambiar? ¿Dónde está escrito por qué se tomó y qué haría cambiarla?