

ARQUITECTURA DE SOFTWARE

LAS PIEZAS DE UNA APP DE VERDAD



UNA APP ES UN SISTEMA DE PIEZAS

Si le pides a Claude «hazme una app», te entrega código que funciona en tu computador. Pero el problema aparece con el primer cliente real, cuando algo se cae y no sabes dónde mirar, porque una app es un sistema de piezas que conversan entre sí.

Piénsalo como un restaurante. Cada parte tiene un trabajo distinto y todas tienen que coordinarse.

EN EL RESTAURANTE: EL SALÓN

INTERFAZ

Lo que el usuario ve y toca: pantallas, formularios, botones. Ahí se hace el pedido.

EN EL RESTAURANTE: LA COCINA

SERVIDOR

Recibe los pedidos, aplica las reglas del negocio y decide qué responder. Nunca confía ciegamente en lo que llega de la interfaz.

EN EL RESTAURANTE: LA DESPENSA

BASE DE DATOS

Donde se guarda todo lo que tiene que sobrevivir: usuarios, pedidos, pagos, historial.

EN EL RESTAURANTE: LOS REPARTIDORES

SERVICIOS EXTERNOS

Lo que habla con el mundo de afuera: correos, pagos, mensajes, archivos, inteligencia artificial.

Esta guía sale de la base de conocimiento de devground, donde tengo escritas las decisiones de arquitectura que uso en mis proyectos:
github.com/elkisdm/devground/tree/main/knowledge

TU PROYECTO DECIDE, NO LA MODA

Ninguna arquitectura es buena o mala por sí sola. Lo que decide cuál te conviene son tres cosas de tu proyecto.

1

CUÁNTAS PERSONAS LA VAN A MANTENER

Si eres tú solo, o tú y la IA, todo lo que agregues lo vas a operar tú.

2

QUÉ TAN ENREDADAS ESTÁN SUS PARTES

Si un pago, un pedido y el inventario tienen que cambiar juntos en la misma operación, separarlos te complica la vida.

3

CÓMO LLEGA EL TRÁFICO

Parejo todo el día, en picos de vez en cuando, o mucho más en una parte que en el resto.

POR DÓNDE EMPEZAR: UN MONOLITO MODULAR

Una sola aplicación que se despliega de una vez, pero ordenada por dentro en módulos que no se mezclan (usuarios, pagos, notificaciones). Martin Fowler recomienda justamente empezar así y separar después.

Dentro de cada módulo, tres capas: la que **recibe** las peticiones, la que tiene **las reglas del negocio** y la que **guarda y lee** los datos. Las reglas no deberían saber si los datos vienen de Postgres o de otro lado. Con eso ya tienes la mayor parte del orden que dan las arquitecturas más sofisticadas, por bastante menos trabajo.

CUÁNDO SEPARAR UNA PARTE

Separa un módulo en un servicio aparte solo cuando tengas una razón que puedas medir.

- ☐ recibe mucho más tráfico que el resto (la referencia que uso es unas diez veces más);
- ☐ lo trabaja un equipo distinto que necesita desplegar sin coordinarse;
- ☐ cambia a un ritmo muy distinto del resto;
- ☐ necesita una tecnología que no encaja con lo demás.

LO QUE TODAVÍA NO NECESITAS

PATRÓN	PARA QUÉ SIRVE	CUÁNDO TODAVÍA NO
MICROSERVICIOS	Que equipos grandes desplieguen partes por separado	Menos de cinco personas, o partes que dependen mucho entre sí
CLEAN O HEXAGONAL	Aislar las reglas del negocio de la base de datos y del framework	Un MVP, o requisitos que cambian cada semana
CQRS	Separar cómo se escribe de cómo se lee, para volúmenes enormes	Una app que guarda y muestra datos de forma normal, que es casi todas

Si la IA te propone alguno para tu primera versión, pregúntale qué problema concreto resuelve en tu caso. Si no te da uno con números, quédate con el monolito modular.

¿SERVERLESS O SERVIDOR PROPIO?

Ojo, serverless no es una arquitectura, es una forma de desplegar. Puedes tener un monolito en serverless o microservicios en un servidor.

Serverless (funciones que se encienden solo cuando llega una petición) conviene cuando el tráfico llega en picos o es esporádico: pagas por uso y no administras máquinas. A cambio, cada arranque en frío agrega una espera, y con tráfico constante sale caro. El equipo de Amazon Prime Video contó en 2023 que movió un sistema de monitoreo de funciones serverless a un servicio en contenedores y que el costo de esa parte bajó en un 90 %.

Servidor propio conviene cuando el tráfico es constante y alto, cuando la latencia es crítica o cuando necesitas conexiones persistentes, como un chat en vivo. Lo normal es combinar: servidor para el grueso y funciones para webhooks y tareas sueltas.

CADA FUNCIÓN SE DISEÑA POR CÓMO SE USA

La pregunta es cómo van a usar las personas los datos que guardas, y casi nunca es la misma para toda la app. En Netflix, ver un capítulo es casi pura lectura, pero ir guardando en qué minuto quedaste es casi pura escritura.

Por cada función importante, anota si se lee mucho o se escribe mucho y cuáles son las consultas que más se repiten. Con eso se elige la base de datos, y no al revés. Para pagos, pedidos y todo lo que tiene que cuadrar, Postgres, que al principio alcanza para casi todo. El detalle de caché, réplicas y colas está en la guía «Arquitectura pro con Claude Code».

ALGO VA A FALLAR

Los servicios externos casi nunca se caen de golpe: se ponen lentos. Una llamada que espera 30 segundos es peor que una que falla en uno, porque mientras espera ocupa recursos, y si llegan muchas así se cae tu app entera.

1

TODA LLAMADA EXTERNA TIENE UN TIEMPO LÍMITE

Si el usuario está esperando, entre uno y cinco segundos. En tareas de fondo, más. Lo que no puede pasar es que espere para siempre, que es lo que hacen muchas librerías si no les dices nada.

2

LOS REINTENTOS ESPERAN CADA VEZ MÁS

Un segundo, después dos, después cuatro, y como mucho tres intentos si hay alguien esperando. Reintentar sin pausa contra un servicio caído solo lo hunde más.

3

SI TARDA MÁS DE UN SEGUNDO, VA A UNA COLA

Correos, PDF o llamadas a una IA no deberían pasar mientras el usuario espera. La app responde al instante y un proceso aparte resuelve la tarea, con reintentos.

ASÍ FUNCIONA EN ELKIS.DEV

Cuando confirmas tu suscripción al boletín, la confirmación se guarda primero y el correo de bienvenida se envía después. Si el servicio de correo falla, tu suscripción queda confirmada igual, porque una falla de afuera no debería deshacer lo que ya hiciste.

DÓNDE VIVE CADA VALOR

La pregunta no es «¿lo quiero cambiar sin deploy?», porque casi todo es más cómodo sin deploy. Es quién debería poder cambiarlo y cuánto daño hace si cambia mal.

DÓNDE VIVE	QUÉ GUARDA	EJEMPLOS
CÓDIGO	Cómo funciona el sistema	Las reglas del negocio, la integración con un proveedor
VARIABLES DE ENTORNO	Con qué infraestructura corre	Claves de API, la dirección de la base de datos, el proveedor de pagos
FEATURE FLAGS	Qué función nueva está encendida y para quién	Una pantalla nueva solo para el 10 % de los usuarios
BASE DE DATOS	Cómo opera el negocio hoy	Precios, horarios, límites, textos de los mensajes
DATOS	Lo que pasó	Usuarios, pedidos, pagos

Lo que más daño puede hacer, como cambiar el proveedor de pagos, pasa por un deploy, nunca por un botón. Lo que cambia seguido y hace poco daño, como un texto, va en la base de datos. Los secretos nunca van en el código: si una clave se filtra, la rotas, porque borrar el commit no alcanza.

A mí me pasa con este mismo sitio: para publicar una guía tengo que hacer un deploy, y una guía es contenido, no estructura. Por eso la tengo anotada para pasarla a la base de datos.

UN STACK PARA EMPEZAR

No es el único posible, pero cubre las cuatro piezas con poca configuración y deja crecer sin rehacer todo.

HERRAMIENTA	PIEZA	PARA QUÉ LA USAS
NEXT.JS	Interfaz + servidor	Pantallas y lógica del servidor en una sola base de código.
SUPABASE	Base de datos + usuarios	Postgres con registro, inicio de sesión y permisos por fila incluidos.
RESEND	Servicio externo	Correos transaccionales: bienvenida, recuperar contraseña, avisos.
RAILWAY	Procesos	Lo que tiene que correr solo: tareas programadas, colas y procesos largos.

ESCRIBE LO QUE DECIDISTE

Cada decisión que no sea obvia (qué base de datos, por qué un monolito, dónde va un valor) se escribe en un archivo corto dentro del proyecto, con el contexto, lo que decidiste, lo que descartaste y lo que te cuesta. Se llama registro de decisión de arquitectura, o ADR.

Parece burocracia, pero es lo único que evita que en seis meses, tú o la IA, deshagan una decisión sin saber por qué se tomó. La IA no se acuerda de la conversación de hoy, pero sí lee los archivos del proyecto.

EL PLANO ANTES QUE EL CÓDIGO

Quiero construir esta app: [qué hace, para quién y qué problema resuelve]. La vamos a mantener [cuántas personas].

Antes de escribir código, ayúdame a diseñar su arquitectura. Responde:

1. Qué piezas necesita (interfaz, servidor, base de datos, servicios externos) y por qué cada una.
2. Qué módulos tiene y cuáles dependen mucho entre sí.
3. Por cada función importante: si se lee mucho o se escribe mucho, y cuáles son las consultas que más se van a repetir.
4. Qué tipo de base de datos conviene según esas consultas.
5. Cómo ordeno los módulos dentro de un monolito modular, con sus capas.
6. Qué servicios externos uso, qué tiempo límite y qué reintentos lleva cada llamada, y qué pasa si uno se cae.
7. Qué tareas pueden tardar más de un segundo y deberían ir a una cola.
8. Dónde vive cada valor configurable: código, variables de entorno, feature flags o base de datos, y por qué.
9. Dónde conviene alojar cada pieza para empezar (serverless o servidor) y por qué.
10. Qué riesgos ves y qué dejarías para una segunda etapa.

Si me propones microservicios, CQRS o arquitectura hexagonal, dime qué problema concreto resuelven en mi caso. No escribas código todavía. Muéstrame primero el plano y escribe las decisiones como registros de decisión en docs/adr/.

CHECKLIST ANTES DEL PRIMER CLIENTE

- | | |
|---|---|
| ☐ Las piezas y los módulos están escritos en un documento. | ☐ Las tareas lentas pasan por una cola con reintentos. |
| ☐ Los permisos se validan en el servidor. | ☐ Los errores quedan registrados donde los revisas. |
| ☐ Ningún secreto está en el código ni en el repositorio. | ☐ Hay respaldos y un entorno de prueba aparte. |
| ☐ Cada llamada externa tiene un tiempo límite. | ☐ Las decisiones importantes están escritas. |