keep coding

React Avanzado

con Next.js

16/02/2026

keep coding

Alex Martínez

Full-Stack Engineer

Índice

Índice

Dinámica Asíncrona

Este curso está diseñado para que lo hagas a tu ritmo, pero requiere código activo.

- El contenido teórico es denso: toma notas.
- Cuando veas el icono , detén el video obligatoriamente.

  PAUSA EL VÍDEO
- No pases al siguiente video sin resolver el reto.

Visual Studio Code

Ten tu editor abierto en todo momento.

keep coding

1. Arquitectura y Server Components

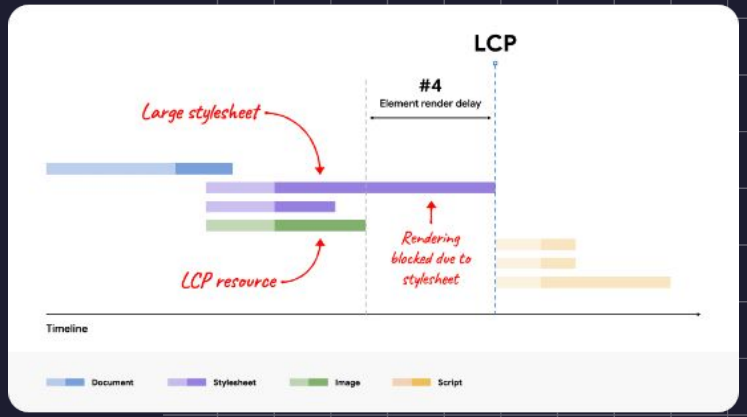
keep coding

El Problema: "Waterfalls"

El coste de la SPA clásica

En React tradicional (Client-Side), el navegador espera a descargar todo el JS antes de empezar a pedir datos.

Esto genera una **"cascada"** de peticiones secuenciales.



El Nuevo Modelo Mental

Server Components

Se ejecutan **una sola vez** en el servidor (build o request). Su código JS **nunca viaja al cliente**. Acceso directo a BBDD y secretos.

Client Components

Son los componentes React de siempre (`useState`, `useEffect`). Se envían al navegador para añadir **interactividad** (clicks, inputs).

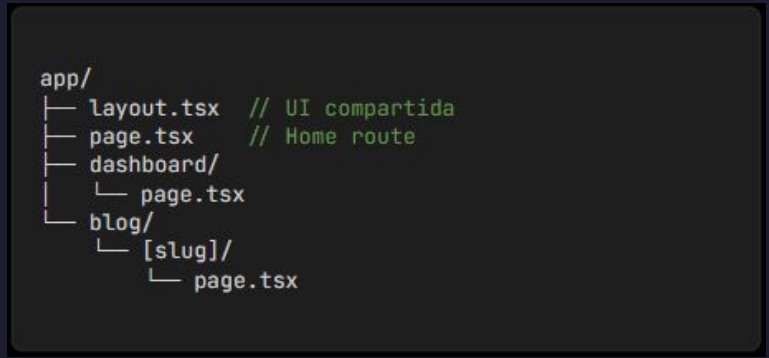
File System Routing

Las carpetas son URLs

En Next.js App Router, la estructura de carpetas define tus rutas públicas.

- app/page.tsx → / (Home)
- app/dashboard/page.tsx → /dashboard
- app/blog/[slug]/page.tsx → /blog/mi-post

Olvídate de configurar React Router. Si está en la carpeta, es una ruta.



Live Coding: Setup

Objetivos de la sesión

- Inicializar el proyecto Next.js
- Limpiar el código boilerplate
- Crear nuestra primera estructura de Layout
- Entender el archivo layout.tsx

```
# Abre tu terminal

npx create-next-app@latest mi-webapp

# Opciones:
# TypeScript: Yes
# Tailwind: Yes
# App Router: YES (Crucial)
```

keep coding

Reto 1: Arquitectura Base

PAUSA EL VÍDEO (20 MIN)

Es hora de crear los cimientos. No avances hasta tener esto funcionando:

- **Sidebar Component:** Crea un componente visual a la izquierda.
- **Global Layout:** Importa el Sidebar en app/layout.tsx.
- **Layout CSS:** Usa Flexbox/Grid para que el Sidebar esté fijo y el contenido (`children`) ocupe el resto.
- **Server Data:** Imprime `new Date().getTime()` en el Sidebar.

```
export default function RootLayout({ children }) {
  return (
    <html>
      <body className="flex">
        <Sidebar />
        <main>{children}</main>
      </body>
    </html>
  )
}
```

keep coding

La Frontera: ¿Dónde se ejecuta?

Por Defecto

Todos los componentes en `app/` son Server Components. No pueden usar hooks (`useState`) ni eventos (`onClick`).

La Red

El código de servidor se queda en el servidor. Al navegador solo llega el HTML resultante. ¡Seguridad y velocidad!

"use client"

La directiva mágica para cruzar la frontera. Transforma un archivo y sus importaciones en código de navegador.

La Directiva "use client"

Cuándo usarla

Solo añade "use client" en las hojas del árbol de componentes (botones, inputs, carruseles).

⚠️PELIGRO:
Nunca importes variables sensibles (secrets) de BBDD en un archivo marcado con "use client". Se filtrarán al navegador.

```
'use client'; // 👈 Primera línea obligatoria

import { useState } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(c => c + 1)}>
      Clicks: {count}
    </button>
  );
}
```

keep coding

Patrón de Composición

❌ Importar Directamente

Si importas un Server Component dentro de un archivo "use client", se convierte en cliente (y falla si tiene Node.js code).

🧩 Usar children

Pásalo como prop `children`. El Client Component solo renderiza el "marco", y Next.js rellena el hueco con el HTML del servidor.

✅ Resultado

Mantienes la interactividad en el padre y el renderizado de servidor en el hijo. Lo mejor de ambos mundos.

Reto 2: Cruzando la Frontera

PAUSA EL VÍDEO (25 MIN)

Objetivo: Interactive Sidebar

1. Crea un componente ThemeToggle que sea Client Component.
2. Añade estado (`useState`) para simular cambio de tema.
3. Intenta usarlo en tu `Sidebar` (que es Server Component).
4. **Bonus:** Crea un componente `UserProfile` (Server) que lea una cookie y pásalo dentro de un Wrapper Interactivo usando `children`.

```
// Pista para el Bonus:

<ClientWrapper>
  <ServerComponent />
</ClientWrapper>

// React renderiza el ClientWrapper en el
navegador,
// pero el ServerComponent viene ya "cocinado"
// desde el servidor.
```

El Portero de Next.js

Interceptando peticiones

El archivo middleware.ts (ahora proxy.ts) se ejecuta antes de que la petición llegue a tu página o API.

- Se ejecuta en el Edge (súper rápido).
- Ideal para: Autenticación, Redirecciones, A/B Testing.
- No tiene acceso al DOM, solo a Request/Response.

Live Coding: Auth Básica

Implementación

Vamos a proteger la ruta /dashboard

```
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  const token = request.cookies.get('token')

  if (!token &&
request.nextUrl.pathname.startsWith('/dashboard')) {
    return NextResponse.redirect(new URL('/login',
request.url))
  }
}
```

Pasos a seguir:

- Crear proxy.ts en la raíz.
- Definir la lógica de redirección.
- Crear una ruta /login para setear la cookie.
- Verificar que no podemos entrar a dashboard sin cookie.

keep coding

Resumen

Arquitectura

App Router usa el sistema de archivos para rutas y Layouts anidados.

RSC

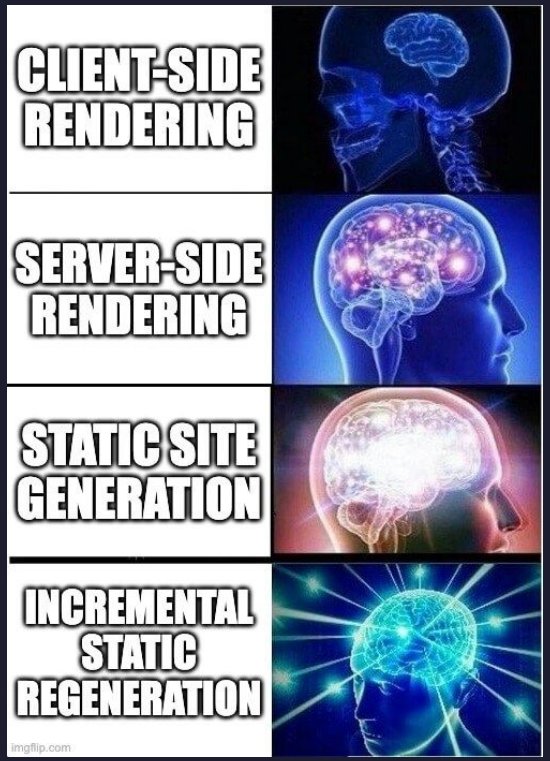
Server Components por defecto. "use client" solo para interactividad específica.

Middleware/Proxy

La primera línea de defensa para proteger nuestras rutas antes del renderizado.

keep coding

Estrategias de Renderizado



keep coding

El Menú del Renderizado

Static (SSG)

"**Cocinar antes de abrir**". Se genera en el build. Máxima velocidad, datos congelados. Ideal para Blogs o Marketing.

Server (SSR)

"**Cocinar al momento**". Se genera en cada petición. Datos siempre frescos, mayor carga de servidor. Como Express.js clásico.

Incremental (ISR)

"**Recalentar y decorar**". El usuario ve una copia rápida (caché) mientras el servidor regenera la nueva versión en segundo plano.

Comparativa Node.js

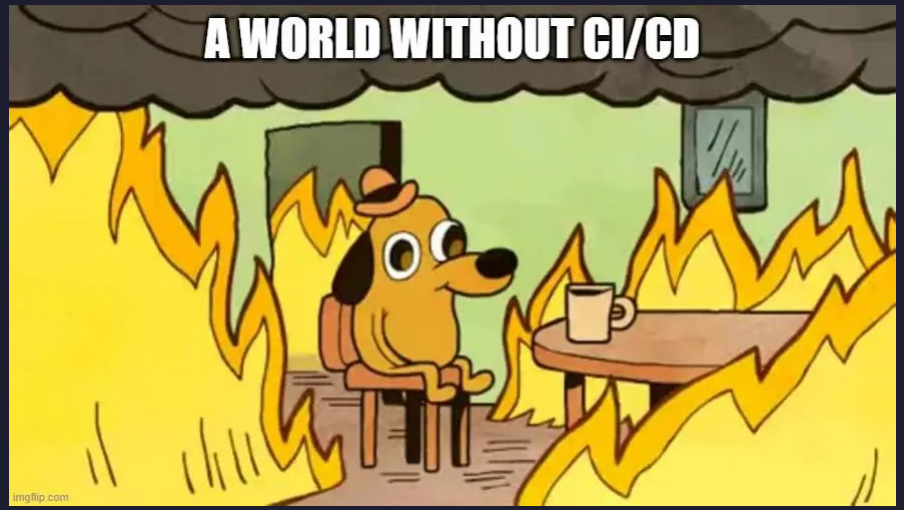
Static Site Generation (SSG)

Es como usar fs.writeFileSync para crear un .html y subirlo a un CDN.

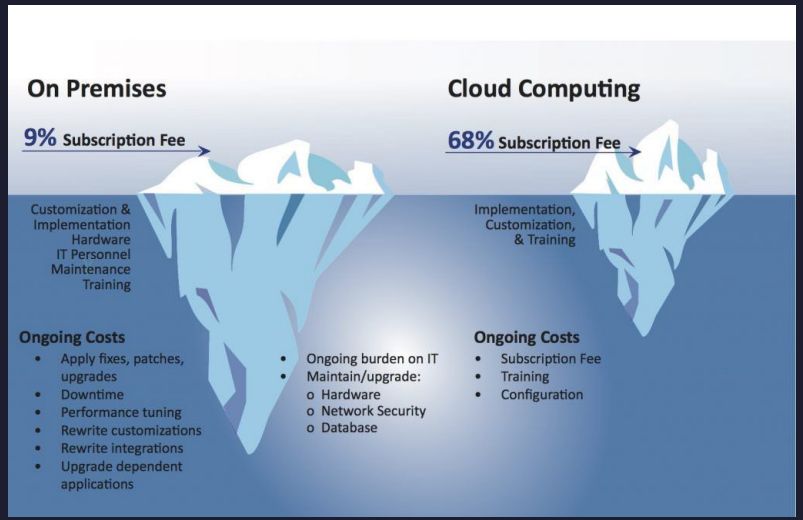
Server Side Rendering (SSR)

Es el Express.js de toda la vida. res.render() en cada visita.

Despliegue y CI/CD



¿Qué es Vercel?



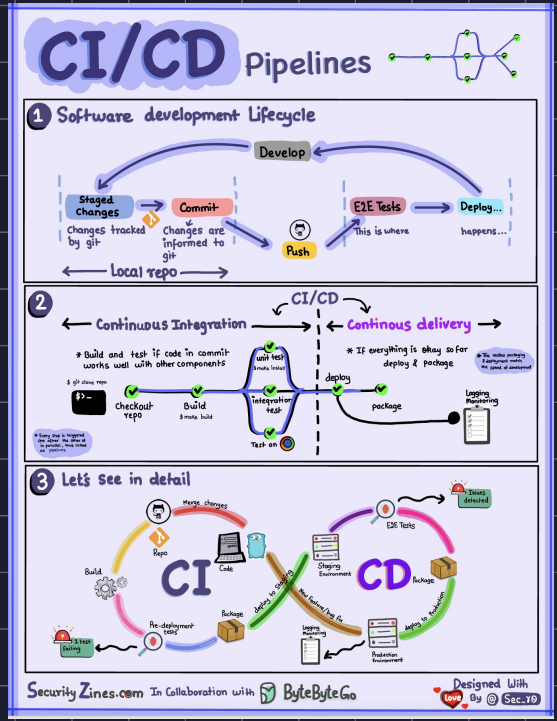
Frontend Cloud

Vercel no es solo un hosting. Es una infraestructura **Serverless** que envuelve a AWS para que tú no tengas que configurar servidores Linux ni Nginx.

- **Edge Network:** Tu web se replica en todo el mundo.
- **Serverless Functions:** Tu API escala de 0 a millones automáticamente.
- **Optimización:** Imágenes y Assets automáticos.

keep coding

El Flujo CI/CD



keep coding

El Flujo CI/CD

keep coding

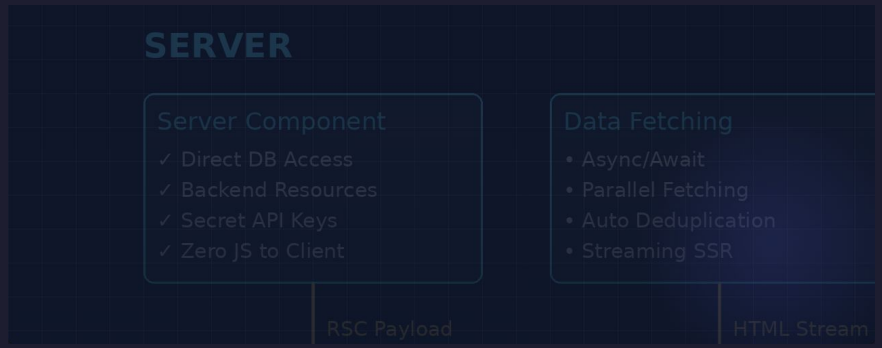
Configurando Vercel

keep coding

Próxima Clase

DATA FETCHING

Conectaremos nuestra estructura a una BBDD real y aprenderemos por qué `fetch` en Next.js es mágico.



keep coding

2. Datos y Velocidad

keep coding

Flash Quiz: Repaso Clase 1

Antes de avanzar, verifica si tienes claros los conceptos de arquitectura.

1. ¿Dónde se ejecuta por defecto un componente en `app/`?

En el Servidor (Server Component).

2. ¿Qué directiva usamos para añadir interactividad?

"use client"

3. ¿Cómo pasamos un Server Component dentro de un Client Component?

Usando la prop `children` (Composición).

keep coding

El Pasado: Client-Side Fetching

La "Fatiga del useEffect"

En React tradicional, traer datos es un ritual complejo:

1. Crear estado para datos (`data`).
2. Crear estado de carga (`loading`).
3. Crear estado de error (`error`).
4. Usar `useEffect` con array de dependencias vacío.
5. Gestionar el "flasheo" de contenido.

```
const [users, setUsers] = useState([]);
const [loading, setLoading] = useState(true);

useEffect(() => {
  fetch('/api/users')
    .then(res => res.json())
    .then(data => {
      setUsers(data);
      setLoading(false);
    });
}, []);

if (loading) return <Spinner />;
```

El Presente: Server Fetching

Simplicidad Asíncrona

En Next.js App Router, los componentes pueden ser async.

- Acceso directo a la BBDD (sin API intermedia).
- Sin gestión de estado manual.
- El componente "espera" en el servidor y envía el HTML con los datos ya pintados.

```
import { db } from '@/lib/db';

// Componente ASYNC!! 🤯
export default async function Page() {
  // Llamada directa a BBDD
  const users = await db.user.findMany();

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Comparativa Visual

❌ React Clásico (20 líneas)

```
useEffect(() ⇒ {
  setIsLoading(true);
  fetch('/api/data')
    .then(res ⇒ res.json())
    .then(data ⇒ {
      setData(data);
      setIsLoading(false);
    });
}, []);

if (isLoading) return ;
return ;
```

✅ Server Component (5 líneas)

```
export default async function Page() {
  const data = await db.query();

  return ;
}
```

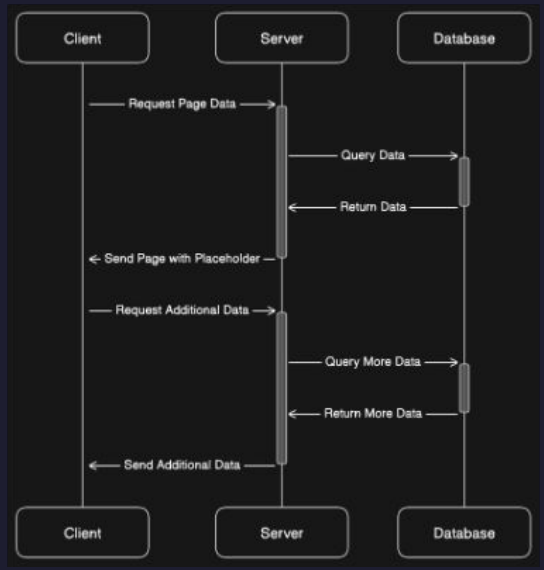
keep coding

Deduplicación de Requests

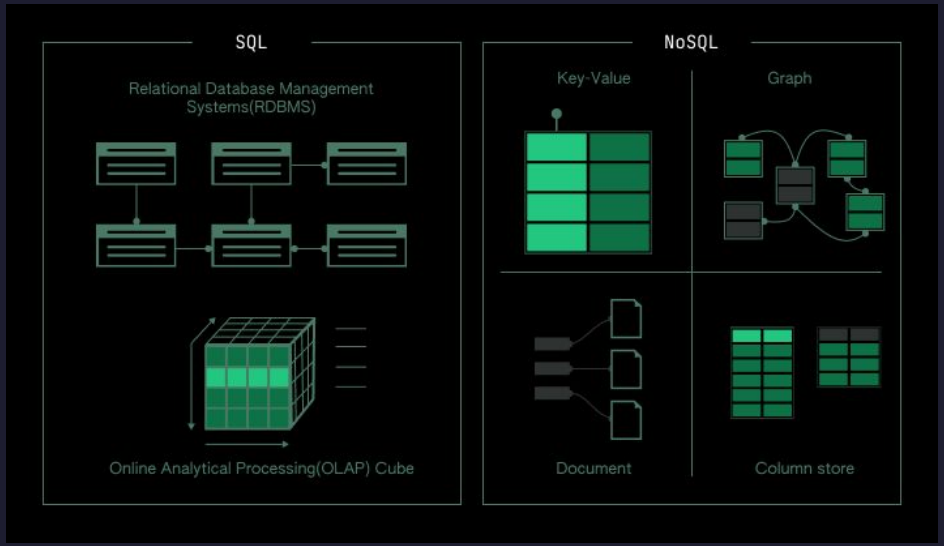
No necesitas Context ni Redux

Si necesitas el "Usuario Actual" en el Layout, en la Página y en el Sidebar... simplemente pídelo 3 veces.

Next.js extiende fetch (y cachea funciones de DB si usas `cache` de React) para que solo se haga **1 consulta real** a la base de datos.



El Nuevo Paradigma: NoSQL vs SQL



- **MongoDB (NoSQL):** "Caja de juguetes". Guardas documentos JSON flexibles. Si metes un dato extra, no pasa nada.
- **PostgreSQL (SQL):** "Hoja de Excel". Columnas estrictas. Si la columna dice 'Edad', no puedes escribir 'Veinte'.
- **¿Por qué cambiar?** Esa disciplina nos da seguridad de tipos e integridad de datos.
- **Prisma:** Es nuestro traductor. Nos deja hablar con SQL usando JavaScript.

Piedra Rosetta: De Mongoose a Prisma

Acción	Mongoose (Lo que sabéis)	Prisma (Lo nuevo)
Modelo	`new Schema({ name: String })`	`model User { name String }`
Buscar	`User.find({ age: 18 })`	`prisma.user.findMany({ where: { age: 18 } })`
Crear	`User.create({ name: 'Pepe' })`	`prisma.user.create({ data: { name: 'Pepe' } })`
ID	`_id (ObjectId raro)`	`id (String/Int limpio)`

¡Fijaos en el autocompletado! Prisma genera los tipos por nosotros.

Adiós al .populate()

❌ Antes (Mongoose)

```
User.find()
 .populate('posts')
 .exec((err, users) => {
   // Magia negra...
 })
```

✅ Ahora (Prisma)

```
const user = await prisma.user.findMany({
 include: {
   posts: true // ¡Trae todo!
 }
})
```

Los "Lookups" y agregaciones en NoSQL son costosos y complejos de escribir.

Las bases de datos relacionales (SQL) están *diseñadas* para unir datos. Es nativo y eficiente.

schema.prisma

```
model Profile {
  id     Int    @default(autoincrement()) @id
  bio    String
  user   User   @relation(fields: [userId], references: [id])
  userId Int
}

model Post {
  id         Int        @default(autoincrement()) @id
  createdAt  DateTime   @default(now())
  title      String
  published  Boolean    @default(false)
  categories Category[] @relation(references: [id])
  author     User       @relation(fields: [authorId], references: [id])
  authorId   Int
}

model Category {
  id    Int    @default(autoincrement()) @id
  name  String
  posts Post[] @relation(references: [id])
}

enum Role {
  USER
  ADMIN
}
```

- **Contrato Único:** Todo el modelo de datos vive en un solo archivo.
- **Legible:** Sintaxis propia diseñada para ser leída por humanos.
- **Generador:** Al guardar, Prisma lee esto y genera el cliente TypeScript (`node_modules`).
- **Migraciones:** También genera los comandos SQL para crear las tablas en Postgres.

```
>_ npx prisma db push
```

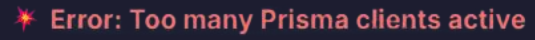
El comando mágico para sincronizar Schema → BBDD.

Setup: Prisma Client

¿Por qué tanto código?

En desarrollo, Next.js usa **Hot Module Replacement (HMR)**.

Si simplemente haces const prisma = new PrismaClient(), cada vez que guardes un archivo se creará una nueva conexión.



Este código asegura que solo exista una instancia global.

```
// lib/prisma.ts (Patrón Singleton)

import { PrismaClient } from '@prisma/client'

const globalForPrisma = global as unknown as {
  prisma: PrismaClient
}

export const prisma =
  globalForPrisma.prisma || new PrismaClient()

if (process.env.NODE_ENV !== 'production')
  globalForPrisma.prisma = prisma
```

keep coding

Live Coding 1: Conectando BBDD

Configurar Prisma

Crearemos el archivo `lib/prisma.ts` con el patrón singleton para evitar fugas de memoria.

Seed Data

Insertamos algunos datos falsos en la BBDD para tener algo que pintar.

Async Page

Transformaremos `page.tsx` en `async function` y haremos nuestra primera query.

keep coding

Reto 1: Tu Primera Query

~~PAUSA EL VÍDEO (20 MIN)~~

Objetivo: Mostrar lista de proyectos en el Dashboard

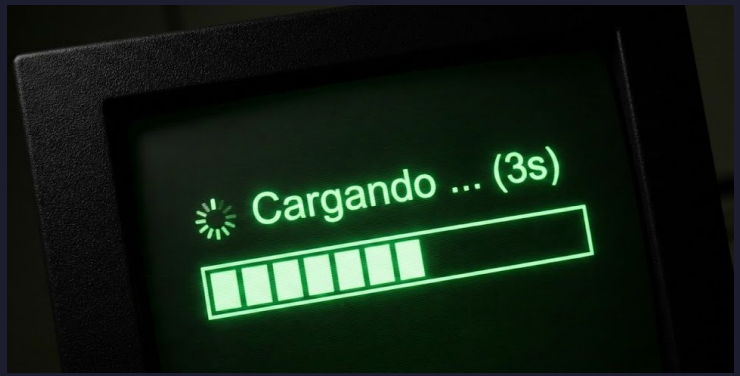
1. Instancia Prisma correctamente en `lib/prisma.ts`.
2. En `app/dashboard/page.tsx`, recupera los proyectos con `prisma.project.findMany()`.
3. Renderizarlos en una lista (`ul` > `li`).
4. **Bonus:** Añade un `await new Promise(...)` de 3 segundos para simular lentitud.

El Problema del SSR Clásico

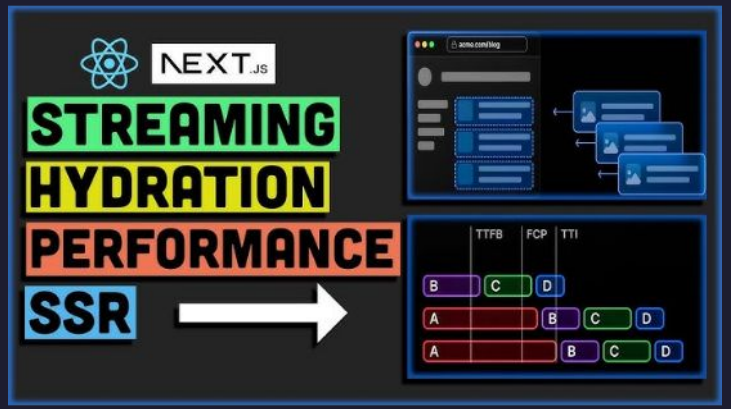
"All or Nothing"

En el SSR tradicional (Server Side Rendering), el servidor tiene que preparar **TODA** la página antes de enviar el primer byte de HTML.

Si una parte de la página (ej. "Comentarios") tarda 3 segundos, **toda la página** se queda en blanco 3 segundos. El usuario no ve ni el Header.



La Solución: Streaming



HTML por partes (Chunks)

Con Streaming, enviamos el HTML tan pronto como está listo.

1. El **Layout** es estático -> Se envía YA.
2. Los **Datos Lentos** -> Se envían como un "hueco" (Skeleton).
3. Cuando la BBDD responde -> React "inyecta" el contenido real en el hueco.

Sintaxis 1: loading.tsx

La forma fácil (Nivel Ruta)

Si creas un archivo llamado loading.tsx en tu carpeta de ruta, Next.js envolverá automáticamente tu page.tsx en un Suspense.

Mientras la página carga sus datos `async`, se mostrará lo que exportes en `loading.tsx`.

```
// app/dashboard/loading.tsx

export default function Loading() {
  return <div>Cargando dashboard...</div>;
}

// Next.js hace esto por ti:
// <Layout>
//   <Suspense fallback={<Loading />}>
//     <Page />
//   </Suspense>
// </Layout>
```

keep coding

Sintaxis 2: Granular Suspense

La forma experta (Nivel Componente)

¿Por qué bloquear toda la página si solo tarda una lista?

Podemos envolver componentes específicos en <Suspense>. Así el título carga instantáneo y solo la lista muestra el spinner.

```
import { Suspense } from 'react';

export default function Page() {
  return (
    <main>
      <h1>Mi Dashboard</h1> {/* Instantáneo */}

      <Suspense fallback={<Skeleton />}>
        <UserStats /> {/* Lento */}
      </Suspense>

      <Suspense fallback={<Skeleton />}>
        <RecentActivity /> {/* Muy Lento */}
      </Suspense>
    </main>
  );
}
```

keep coding

Patrón: Parallel Data Fetching

Evita las "Waterfalls" secuenciales

Si usas múltiples `await` seguidos, se ejecutan uno tras otro (1s + 1s = 2s).

Usa Promise.all para iniciarlos a la vez (1s + 1s = 1s).

```
// ❌ Secuencial (Lento)
const users = await getUsers();
const posts = await getPosts();

// ✅ Paralelo (Rápido)
const [users, posts] = await Promise.all([
  getUsers(),
  getPosts()
]);
```

keep coding

Live Coding 2: Streaming UX

Simular Latencia

Añadiremos retrasos artificiales para poder "ver" el streaming en local.

Skeleton UI

Diseñaremos un esqueleto de carga con Tailwind (`animate-pulse`) para reemplazar el texto "Cargando...".

Suspense Boundary

Moveremos la carga de datos a un componente hijo para desbloquear el título de la página.

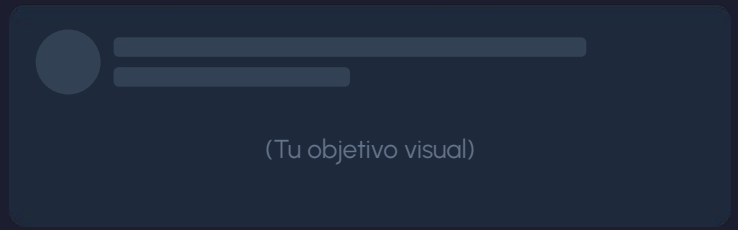
keep coding

Reto 2: Skeleton Perfection

PAUSA EL VÍDEO (20 MIN)

Objetivo: Transforma la experiencia de carga

1. Crea un componente `<Card>` que imite la forma de tus tarjetas de proyecto.
2. Usa `bg-slate-800` y `animate-pulse`.
3. Implementa Streaming usando `loading.tsx` O `<Suspense>`.
4. Verifica que el Layout no parpadea al navegar.



¡Misión Cumplida!

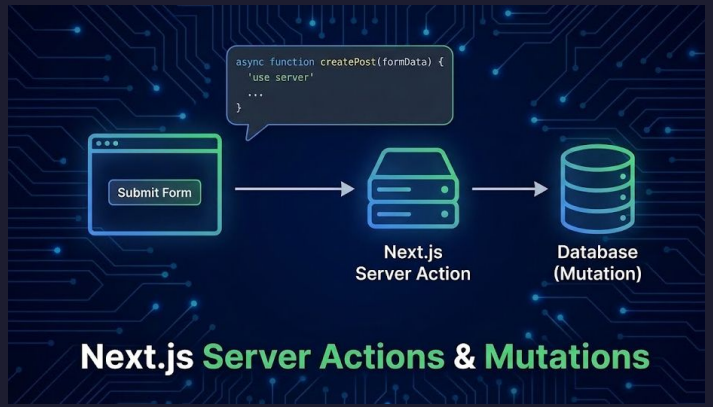
Ya sabemos leer datos a la velocidad de la luz.

keep coding

Próxima Clase

MUTACIONES

Aprenderemos a escribir en la BBDD con **Server Actions**. Prepárate para decir adiós a las API Routes.



keep coding

3. Mutaciones (Server Actions)

keep coding

Flash Quiz: Repaso Clase 2

Antes de avanzar, verifica si tienes claros los conceptos de Datos y Velocidad.

1. ¿Qué patrón evita el "Flasheo" de pantalla blanca?

Streaming SSR (con Suspense o loading.js).

2. ¿Es necesario usar `useEffect` para hacer fetch en Next.js?

No. Usamos Server Components async.

3. ¿Cómo evitamos instanciar Prisma 10 veces en desarrollo?

Usando el patrón Singleton en globalThis.

Un minuto de silencio...

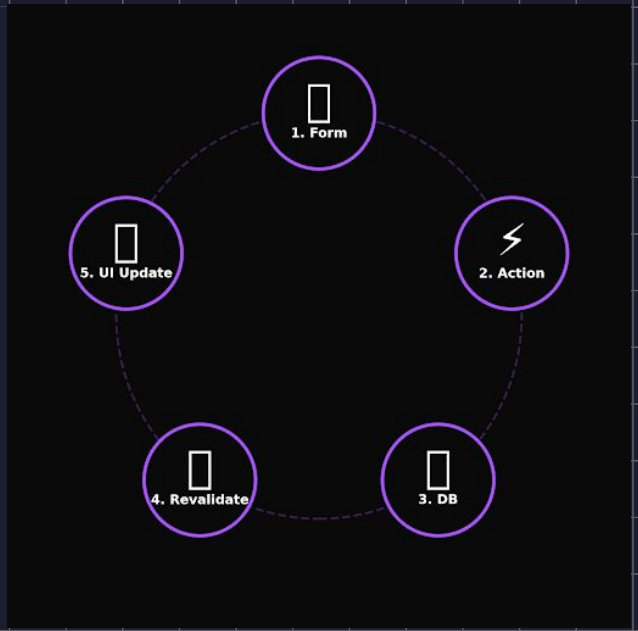
"Fuiste útil, pero ahora tenemos Server Actions."

El Nuevo Ciclo de Vida

El Círculo Virtuoso

Ya no hay separación entre "Backend" y "Frontend" en tu cabeza. Es un flujo continuo.

1. **Form**: Usuario envía datos (Progressive Enhancement).
2. **Action**: Función en servidor se ejecuta.
3. **DB**: Prisma escribe el dato.
4. **Revalidate**: Limpiamos la caché.
5. **UI**: React repinta con el dato nuevo.



RPC: El "Túnel" Mágico

Remote Procedure Call

- La Ilusión: Ejecutar una función en el servidor como si estuviera en tu archivo local.
- La Abstracción: Oculta toda la "fontanería" HTTP: fetch, headers, serialización y endpoints.
- En Next.js: Las Server Actions son RPCs tipados, seguros e integrados.

> 🪄 Tú invocas la función, Next.js cruza el "túnel" de internet por ti.

keep coding

La Directiva "use server"

¿Qué hace realmente esta línea mágica?

- Marca una función asíncrona para que pueda ser invocada desde el cliente (nuestro RPC oculto).
- Next.js crea automáticamente un endpoint HTTP POST oculto y único para esta función.

⚠ Regla de Oro: Solo funciona en Async Functions.

```
// Opción A: Server Action Inline (En Server
Component)
export default function Page() {
  async function create(formData: FormData) {
    'use server'; // 👈 MAGIA AQUÍ
    await db.post.create(...)
  }

  return (
    <form action={create}>
      <button>Guardar</button>
    </form>
  )
}
```

Mejor Práctica: Actions en Fichero

Separación de Intereses

Para mantener el código limpio y reutilizable, es mejor sacar las acciones a un archivo separado.

Además, esto permite importar la Server Action dentro de un Client Component (ej. un botón con `onClick`).

```
// app/actions.ts
'use server';

import { db } from '@/lib/db';

export async function createPost(formData: FormData)
{
   const title = formData.get('title');
   await db.post.create({ data: { title } });
}
```

```
// app/page.tsx
import { createPost } from './actions';

export default function Page() {
  return <form action={createPost}>...</form>
}
```

keep coding

La Aduana: "Network Boundary"

El túnel RPC existe, pero no todo cabe por él. Los datos deben poder serializarse a texto para cruzar.

- ✅ Primitivos (string, number, boolean)
- ✅ JSON Objects
- ✅ FormData
- ✅ Date (se convierte a ISO String)

- ❌ Funciones (callbacks…)
- ❌ DOM Events (e, e.target, e.preventDefault)
- ❌ Class Instances (se pierden los métodos)
- ❌ Componentes React (JSX)

> ⚠️ Regla de Oro: Si `JSON.stringify(dato)` falla o pierde información, tu Server Action fallará.

Peligro de seguridad

No es “Backend privado”

Aunque la función viva en el servidor, al poner 'use server' la estáis haciendo Pública a Internet.

Cualquiera puede enviar una petición POST a esa función si averigua el ID que genera Next.js.

❌ NO HAGAS ESTO:

```
export async function deleteUser(userId) {
  'use server';
  // ⚠️ Peligro: No verificamos quién llama
  await db.user.delete({ where: { id: userId } });
}
```

✅ SIEMPRE VERIFICA:

```
export async function deleteUser() {
  'use server';
  const session = await getSession(); // Auth check
  if (!session) throw new Error('Unauthorized');

  await db.user.delete({ where: { id: session.userId
} });
}
```

keep coding

Live Coding 1: Hello Action

Crear Formulario

Haremos un input simple para "Añadir Tarea". Sin useState. HTML nativo.

Server Action

Crearemos `actions.ts`. La función recibirá `FormData`. Haremos un `console.log` para ver que sale en la terminal del servidor.

Conexión

Usaremos la prop `action={myAction}` en el formulario.

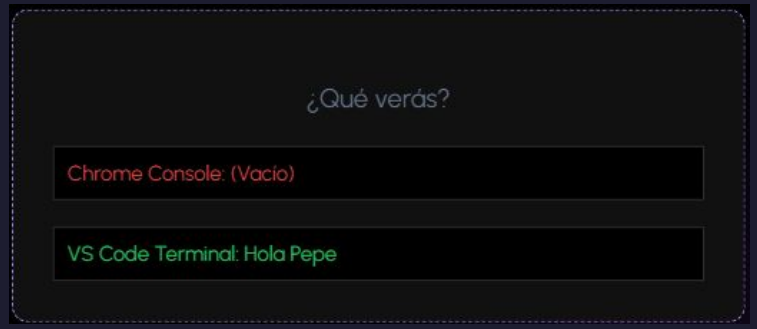
keep coding

Reto 1: El Log Fantasma

PAUSA EL VÍDEO (15 MIN)

Objetivo: Entender dónde se ejecuta tu código.

1. Crea un formulario con un input `name="username"`.
2. Crea una Server Action que haga `console.log("Hola " + formData.get("username"))`.
3. Envía el formulario.
4. **Misión:** Encuentra dónde ha salido el log. (Pista: No mires la consola de Chrome).



keep coding

La Trampa de En local funciona

¿Por qué tu app falla al subirla a Vercel si funciona perfectamente en local?

keep coding

Dos Realidades Opuestas

DEVELOPMENT

- **Caché Débil:** Next.js sabe que editas. Cachea poco para mostrar cambios rápido.
- **Refrescar Funciona:** F5 suele regenerar la página y traer datos nuevos.
- **Sensación Falsa:** "¡Mira, he guardado en BD y se ve al instante!"

PRODUCTION

- **Static by Default:** Al hacer build, Next.js saca una "foto" de tus datos.
- **Inmutable:** La página mostrará los datos del momento del build para siempre.
- **El Bug:** Guardas en BD, refrescas... y sigues viendo el dato viejo.

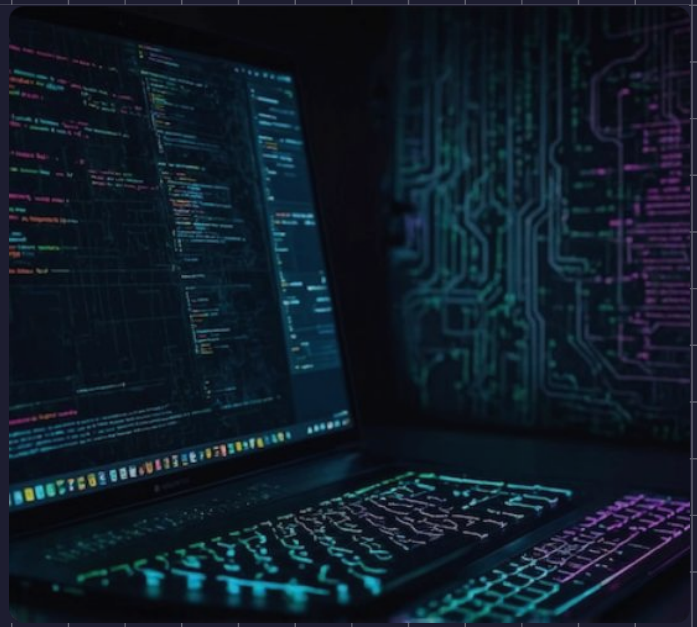
El Remedio

EL PROBLEMA

Next.js en Producción no sabe que has cambiado la base de datos, tienes que decírselo explícitamente.

La solución es invocar la revalidación bajo demanda:

```
revalidatePath '/dashboard')
(                          ;
  // "¡Oye Next.js, tira la foto vieja y saca una nueva!"
```



keep coding

Live Coding 2: Escribiendo en DB

El Flujo Completo

Ahora vamos a conectar las piezas:

1. Importar `prisma` en la Server Action.
2. `await prisma.task.create(...)`
3. `revalidatePath('/')`
4. Ver cómo la tarea aparece mágicamente en la lista sin recargar la página completa.

```
import { revalidatePath } from 'next/cache';

export async function createTask(formData: FormData) {
  'use server';

  const title = formData.get('title');

  await prisma.task.create({
    data: { title }
  });

  // 👇 El secreto para actualizar la UI
  revalidatePath('/dashboard');
}
```

UX Básica: Feedback

¿Está cargando?

Como no usamos `useState` ni `fetch`, no tenemos un `isLoading` manual.

Usamos el hook useFormStatus. **Pero ojo:** Solo funciona dentro de un componente que esté dentro del <form>

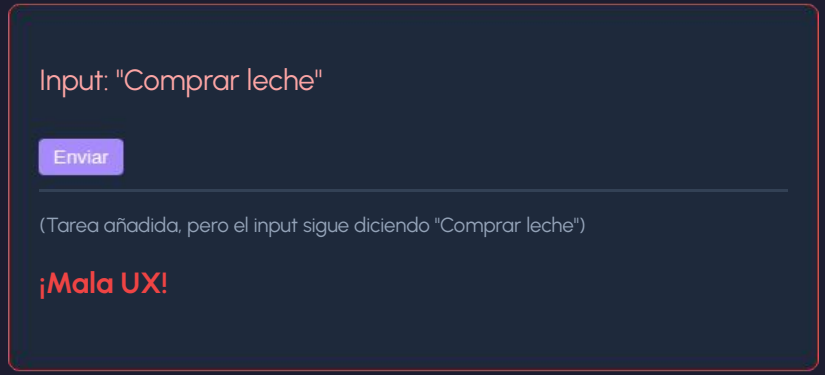
```
'use client'; // Hook de cliente
import { useFormStatus } from 'react-dom';

export function SubmitButton() {
  const { pending } = useFormStatus();

  return (
    <button disabled={pending}>
      {pending ? 'Guardando...' : 'Guardar'}
    </button>
  );
}
```

El Problema del Reset

El formulario no se limpia

En una app SPA como Next.js, cuando envías el form, la página no se recarga completamente. Por tanto, el texto que escribiste en el input **se queda ahí**.

¿Cómo lo limpiamos si no queremos usar `useEffect` ni manipular el DOM manualmente?



keep coding

Reto 2: Limpieza sin JS Cliente

PAUSA EL VÍDEO (20 MIN)

Objetivo:

Limpia el input tras enviar, usando trucos de React Server Components.

Pista: React identifica los componentes por su `key`. Si la `key` cambia, React destruye el componente y crea uno nuevo (limpio).

```
// app/page.tsx

export default async function Page() {
  // Truco: Usar una key aleatoria o basada en timestamp
  // que cambie cada vez que revalidamos.
  const key = new Date().getTime();

  return (
    <div>
      <NewTaskForm key={key} />
      <TaskList />
    </div>
  )
}
```

keep coding

Progressive Enhancement

Sin JavaScript

Lo creas o no, este formulario funciona si el usuario deshabilita JS en el navegador. Next.js usa el envío nativo HTML POST.

Conexiones Lentas

En móviles 3G, el formulario funciona antes de que el bundle de React termine de hidratarse.

Accesibilidad

Al usar estándares web (`form`, `input`, `button type="submit"`), ganamos accesibilidad gratis.

keep coding

Control de Errores Básico

Try / Catch

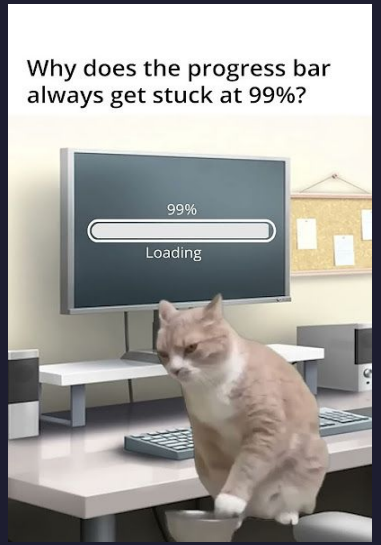
Las Server Actions son funciones normales. Si algo falla (BBDD caída), lanzan una excepción.

Debemos envolver nuestra lógica en `try/catch` y devolver un objeto serializable (JSON) con el error para pintarlo en la UI.

```
export async function create(formData: FormData) {
  'use server';
  try {
    await db.create(...);
    return { success: true };
  } catch (e) {
    return { error: 'Error al guardar' };
  }
}
```

¡Misión Cumplida!

Hoy hemos aprendido a mutar datos. Pero... ¿y si tarda 3 segundos?

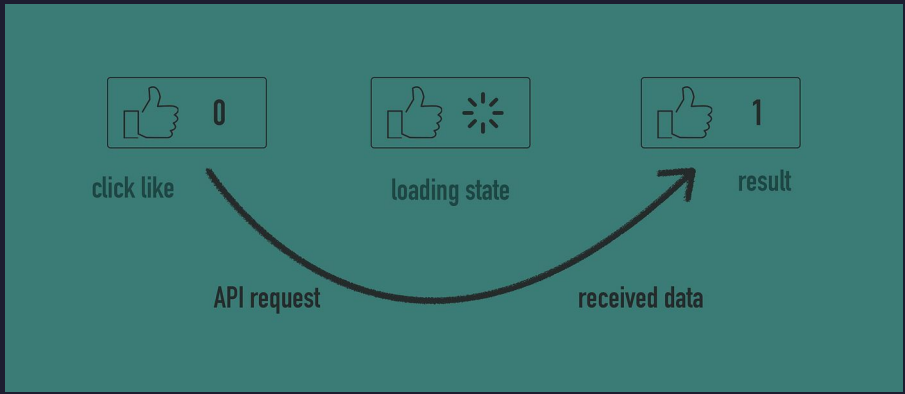


keep coding

Próxima Clase

UX AVANZADA

Estado Optimista (Zero-latency UI) y validación robusta con Zod.



keep coding

4. Formularios y UX Avanzada

Flash Quiz: Repaso Clase 3

Antes de avanzar, verifica si tienes claros los conceptos de Mutaciones y Server Actions.

1. ¿Dónde se ejecuta una función marcada con "use server"?

Siempre en el servidor (Endpoint POST oculto).

2. ¿Podemos importar una Server Action en un Client Component?

Sí. Es la forma estándar de añadir interactividad.

3. ¿Qué función usamos para limpiar la caché tras mutar?

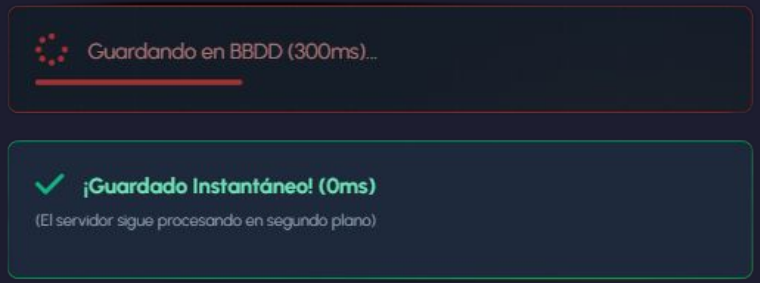
revalidatePath() o revalidateTag().

La Psicología de la Espera

El Valle de la Desesperación

Cualquier interacción que tarde más de **100ms** rompe la sensación de "instantaneidad".

Si tu usuario hace clic en "Guardar" y espera 1 segundo a que el servidor responda, siente que la app está "rota" o es "pesada".



keep coding

Neuro-UX: Percepción vs Realidad

Dopamina

El feedback instantáneo libera pequeñas dosis de satisfacción. La espera genera cortisol (estrés).

Velocidad Percibida

Tu app puede ser lenta (server), pero si la UI responde rápido (optimistic), el usuario la sentirá rápida.

Confianza

Una UI robusta (validaciones claras) e instantánea genera confianza en la marca.

keep coding

La Herramienta: useActionState

Gestión de Estado Robusta

Olvídate de crear `useState` manuales para `error`, `success`, `data`.

React 19 (y Next.js) nos trae el hook definitivo para conectar formularios con Server Actions y manejar su ciclo de vida.

```
'use client';

import { useActionState } from 'react';
import { createPost } from './actions';

const initialState = { message: null, errors: {} };

export function Form() {
  // 👇 La magia ocurre aquí
  const [state, dispatch, isPending] =
      useActionState(createPost, initialState);

  return (
    <form action={dispatch}>...</form>
  );
}
```

🪄 Anteriormente llamado `useFormState`.

keep coding

Diagrama de Estados

Validación: “Don’t Trust Users”

Zod: TypeScript First Validation

Nunca confíes en lo que llega en `FormData`. El usuario puede manipular el HTML.

Usamos **Zod** para definir un esquema estricto. Si los datos no encajan, Zod nos devuelve errores formateados listos para la UI.

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

// Definimos el "contrato"
const schema = z.object({
  email: z.email({
    message: "Email inválido"
  }),
  password: z.string().min(6, {
    message: "Mínimo 6 caracteres"
  })
});

// Tipado automático (Inferencia) 🤯
type SchemaType = z.infer<typeof schema>;
```

Patrón: Action + Zod

```
// app/actions.ts
'use server';

export async function login(prevState: any, formData: FormData) {

  // 1. Validar los datos crudos
  const validatedFields = schema.safeParse({
    email: formData.get('email'),
    password: formData.get('password'),
  });

  // 2. Si falla, devolver errores a useActionState
  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
    }
  }

  // 3. Si éxito, mutar BBDD
  await db.user.create(validatedFields.data);
}
```

keep coding

Live Coding 1: Formulario Robusto

Setup Zod

Instalar `zod` y definir el esquema para crear una Tarea (título mínimo 3 letras).

Action refactor

Modificar nuestra Server Action para que devuelva `{ errors }` en lugar de lanzar excepciones.

UI Feedback

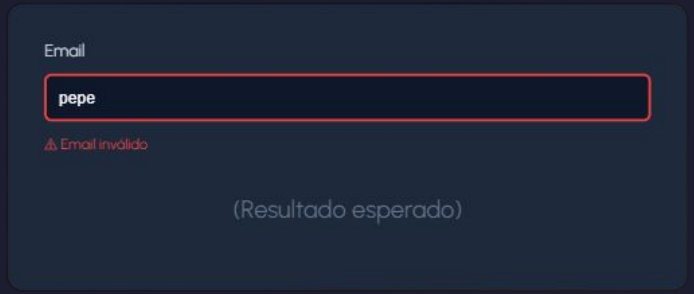
Conectar `useActionState` para mostrar el mensaje de error debajo del input.

Reto 1: Validación Visual

PAUSA EL VÍDEO (20 MIN)

Objetivo: Mejorar la UX de los errores.

1. Añade validación de "Email" a tu formulario.
2. Si `state.errors.email` existe, el input debe tener **borde rojo**.
3. Muestra el mensaje de error en texto rojo pequeño debajo.
4. Prueba a enviar "pepe" (inválido) y "pepe@gmail.com" (válido).



keep coding

Optimistic UI: La Mentira Piadosa

Zero Latency UI

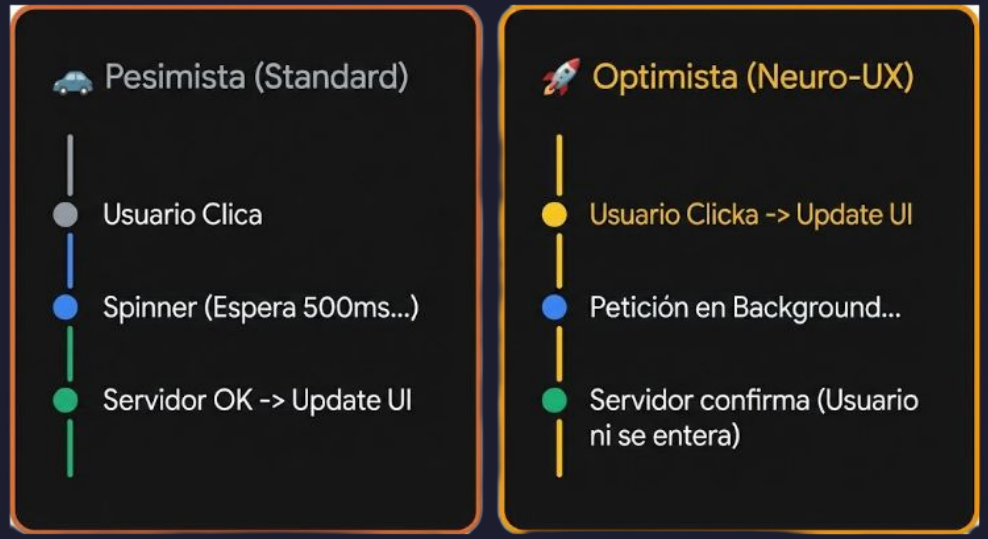
¿Por qué esperar al servidor si sabemos (al 99%) que va a funcionar?

Optimistic UI consiste en actualizar la interfaz **inmediatamente** al hacer click, asumiendo que el servidor responderá OK.

Si el servidor falla, hacemos "Rollback" (volvemos atrás) silenciosamente.

keep coding

Flujo: Pesimista vs Optimista



keep coding

Herramienta: useOptimistic

Un hook para mentir

Recibe el estado real (del servidor) y devuelve una versión "trampeada" que puedes mutar instantáneamente.

En cuanto el servidor devuelve el dato nuevo real, el estado optimista se descarta.

```
import { useOptimistic } from 'react';

export function LikeButton({ likes }) {
  // [estadoVisual, mutarVisualmente]
  const [optimisticLikes, addOptimisticLike] =
      useOptimistic(
        likes, // Estado Real (Source of Truth)
        (state, newLike) => state + 1 // Reducer
      );

  return (
    <button onClick={async () => {
        addOptimisticLike(1); // ⚡ Instantáneo
        await saveLikeInServer(); // 🐢 Lento
    }}>
      ❤️ {optimisticLikes}
    </button>
  );
}
```

Live Coding 2: Instant Like

Instagram Feel

Vamos a implementar el clásico botón de "Me Gusta".

1. Crear Server Action `toggleLike`.
2. Añadir un delay artificial de 2 segundos para notar la lentitud.
3. Implementar `useOptimistic` para que el corazón se rellene al instante.
4. Ver la discrepancia entre la UI (rápida) y el Network (lento).

keep coding

Reto 2: Comentario Instantáneo

PAUSA EL VÍDEO (25 MIN)

Nivel Avanzado

Un contador es fácil. Ahora hazlo con una lista.

1. Tienes una lista de comentarios.
2. Al enviar el form, usa `addOptimisticComment` para inyectar el comentario en la lista visualmente.
3. El comentario debe aparecer **antes** de que el servidor responda.
4. **Bonus:** Ponle opacidad al 50% mientras sea optimista ("Pending State visual").

```
// Pista para el Reducer:

useOptimistic(comments, (state, newComment) => {
  return [...state, {
    id: Math.random(),
    text: newComment,
    pending: true
  }];
})
```

¡Misión Cumplida!

Ya tenemos una app que gestiona datos, valida errores y se siente instantánea.

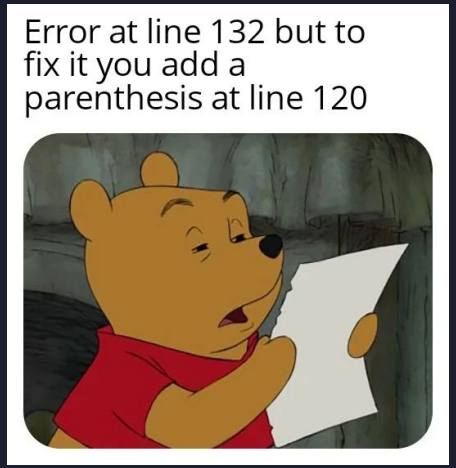
keep coding

Próxima Clase

Control de Errores y Seguridad

¿Qué pasa cuando el servidor explota?

`error.js`, `global-error.js` y recuperación de desastres.



keep coding

5. Control de Errores y Seguridad

keep coding

Flash Quiz: Repaso UX

Verifica si has interiorizado los conceptos de la clase anterior.

1. ¿Qué hook usamos para conectar forms con Server Actions?

useActionState (antes useFormState).

2. ¿Qué librería usamos para validar datos en el servidor?

Zod (TypeScript Schema Validation).

3. ¿Qué es la "Optimistic UI"?

Actualizar la interfaz antes de que el servidor confirme.

keep coding

El Problema: Fallos no Capturados

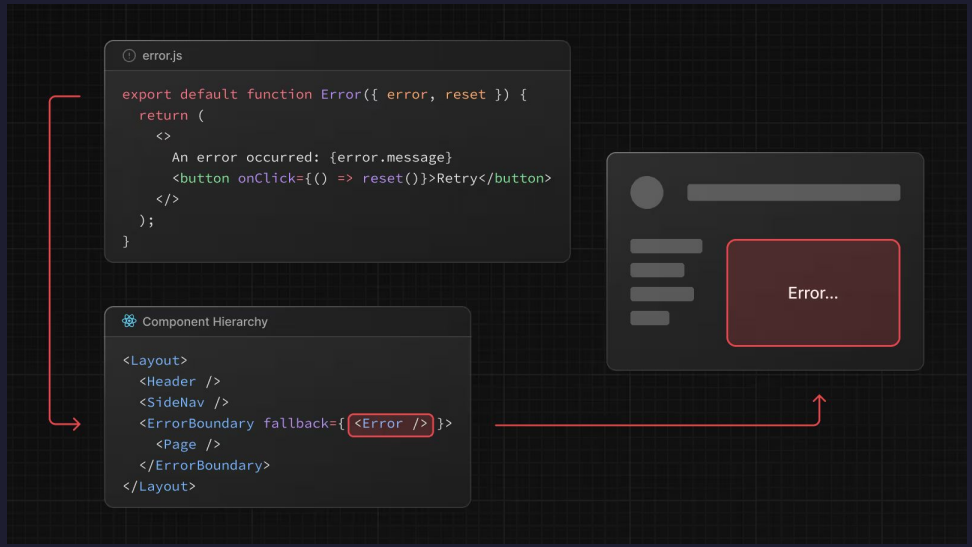
Pánico Nuclear

En React, si un componente lanza un error durante el renderizado y nadie lo captura, **todo el árbol se desmonta.**

El usuario ve una pantalla blanca. La aplicación muere. Se pierde toda la navegación.

La Solución: Contención

Next.js usa React Error Boundaries automáticos. Piensa en muñecas rusas (Matrioshkas).



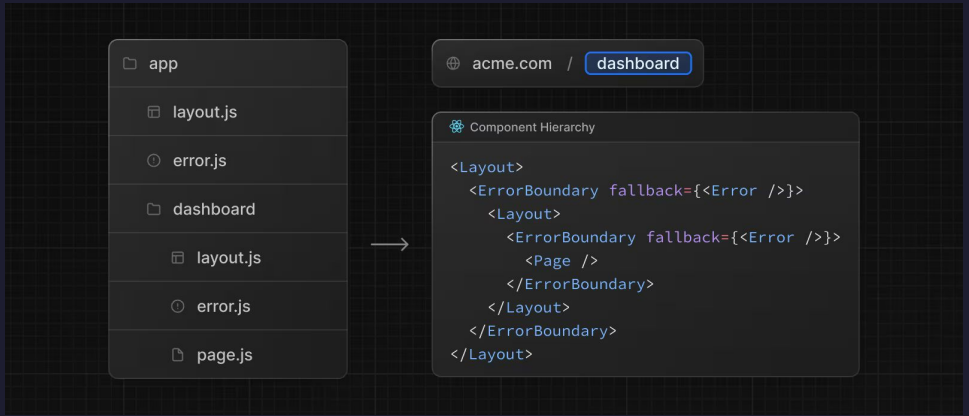
keep coding

Jerarquía de Archivos

El Orden Importa

Next.js envuelve tu página en este orden específico. Si `page.tsx` falla, `error.tsx` se activa.

Si `layout.tsx` falla, `error.tsx` **NO** se activa (porque el error está por encima de él).



Sintaxis: error.tsx

Reglas de Oro

- **Debe ser Client Component:** Usa 'use client'. Los Error Boundaries de React no funcionan en el servidor.
- **Props:** Recibe error (detalles) y reset (función para reintentar).

```
'use client'; // 👈 Obligatorio

import { useEffect } from 'react';

export default function Error({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  useEffect(() => {
    console.error(error); // Log to service (Sentry)
  }, [error]);

  return (
    <div>
      <h2>¡Algo salió mal!</h2>
      <button onClick={() => reset()}>
        Reintentar
      </button>
    </div>
  );
}
```

keep coding

Live Coding 1: Chaos Engineering

Romper la App

Lanzaremos un `throw new Error("BBDD caída")` intencionado en `page.tsx`.

Crear error.tsx

Veremos cómo el Layout se mantiene vivo mientras el contenido principal muestra el mensaje de error amigable.

Reset Button

Implementaremos el botón de reintento para recuperar la app sin recargar la página completa.

keep coding

Reto 1: Granularidad

PAUSA EL VÍDEO (15 MIN)

Objetivo: Define errores específicos por ruta.

1. En `/dashboard`, crea un `error.tsx` elegante.
2. En una sub-ruta `/dashboard/settings`, fuerza un error diferente.
3. Comprueba que el error de Settings NO reemplaza el Sidebar del Dashboard.
4. Bonus: Usa `global-error.tsx` para capturar fallos en el Root Layout.

Tip:

`global-error.tsx` reemplaza TODA la app, incluyendo `<html>` y `<body>`. Debes volver a escribir esas etiquetas dentro de él.

404: Not Found

Cuando el dato no existe

Además de errores técnicos (500), tenemos errores lógicos (404).

Usa la función notFound() dentro de un Server Component si el ID no existe en la BBDD.

Next.js renderizará el archivo not-found.tsx más cercano.

```
import { notFound } from 'next/navigation';

export default async function Profile({ params }) {
  const user = await db.user.find(params.id);

  if (!user) {
    notFound(); // 👈 Detiene ejecución y muestra UI 404
  }

  return <div>Hola {user.name}</div>;
}
```

Seguridad

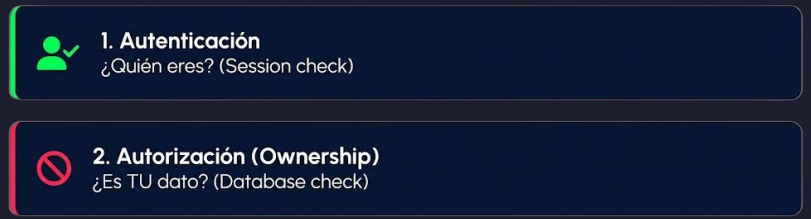
No todos los errores son accidentales.

¿Qué pasa si un usuario malintencionado intenta borrar un post que no es suyo?

keep coding

Checklist de Seguridad (Acciones)

En Server Actions, tú eres el responsable de validar **Autenticación** y **Autorización.**




```
export async function deletePost(id: string) {
  'use server';

  // 1. Auth Check
  const session = await getSession();
  if (!session) throw new Error("401");

  // 2. Ownership Check (CRUCIAL)
  const post = await db.post.findUnique({ where: { id } });

  if (post.authorId !== session.user.id) {
     throw new Error("403 - No toques lo que no es tuyo");
  }

  // 3. Execute
  await db.post.delete({ where: { id } });
}
```

Live Coding 2: Hacking Ético

Crear Acción Vulnerable

Haremos una función `deletePost` que solo pide el ID. Cualquiera puede invocarla.

Explotar la Vulnerabilidad

Demostraré cómo borrar el post de otro usuario enviando una petición POST manual.

Parchear (Secure)

Añadiremos la verificación de propiedad (Author ID check) antes de ejecutar el `delete`.

keep coding

Reto 2: El Guardián

PAUSA EL VÍDEO (20 MIN)

Objetivo: Protege la privacidad de tus usuarios.

1. Implementa una función simulada `getSession()` que devuelva `{ userId: 'admin' }`.
2. Crea una **Server Action** `updateProfile(userId, newData)`.
3. **Misión:** Asegúrate de que si intento actualizar el perfil de 'pepe', la acción falle. Solo debo poder actualizar 'admin'.

keep coding

Resumen: Aplicación Robusta

Jerarquía

Usamos `error.tsx` anidados para que un fallo en un componente no rompa toda la página.

Recuperación

Ofrecemos botones `reset()` para que el usuario pueda reintentar sin recargar.

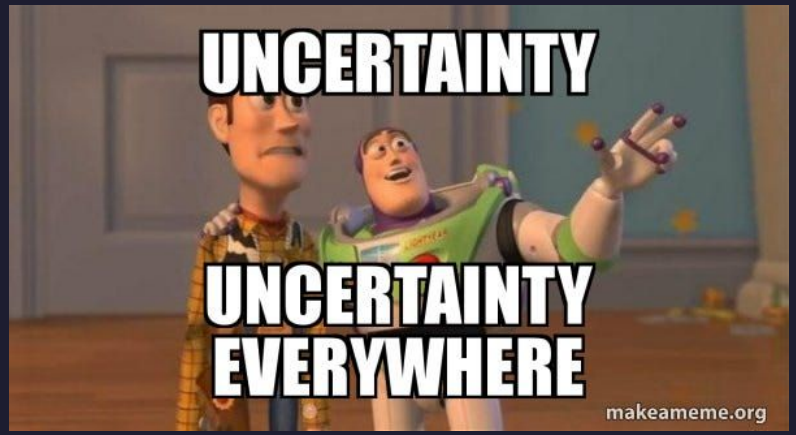
Autorización

Validamos siempre la propiedad del dato (`authorId === userId`) en cada Server Action.

¡Misión Cumplida!

Tu app es segura y resiliente.

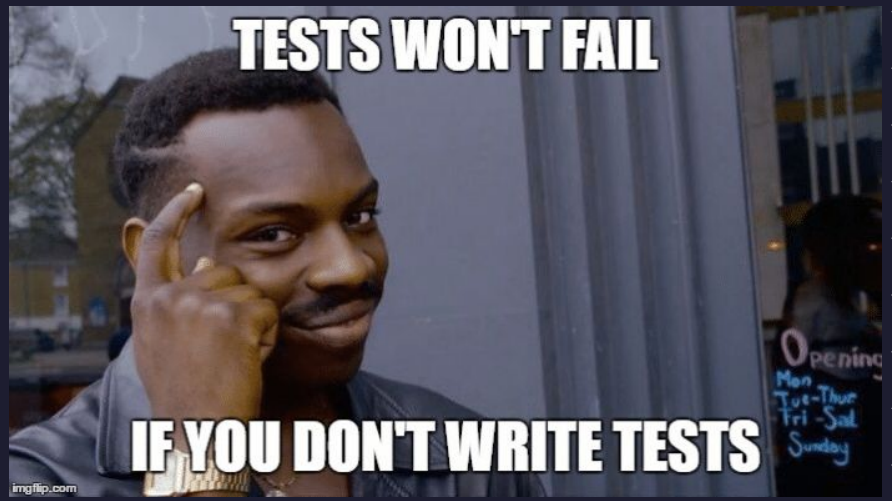
Pero, ¿cómo sabes que seguirá siéndolo mañana?



Próxima Clase

Testing

Unit Testing de Server Actions, E2E con Playwright y puesta en producción.



keep coding

6. Testing y Calidad en Producción

Flash Quiz: Repaso Resiliencia

Antes de certificar la calidad, recordemos cómo manejar errores.

1. ¿Dónde se renderiza el archivo `error.tsx`?

En el Cliente ('use client' obligatorio).

2. ¿Qué función usamos para un 404 manual?

notFound() desde 'next/navigation'.

3. ¿Quién debe validar que un usuario es dueño de un dato?

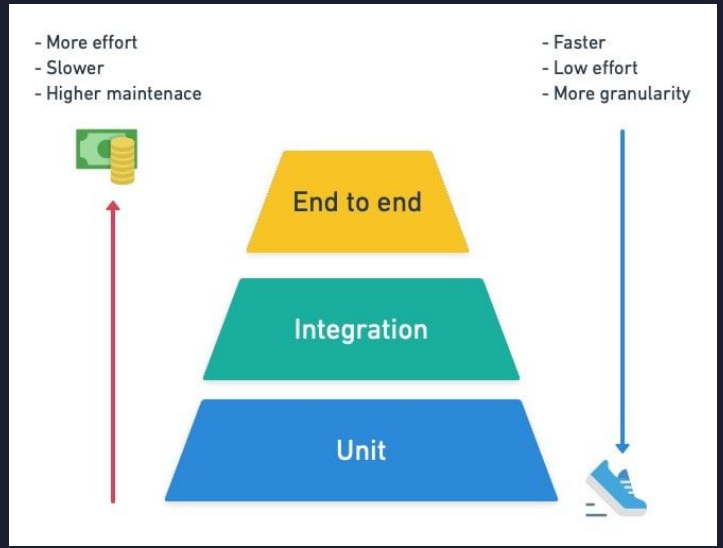
La Server Action (Authorization Check).

keep coding

El Problema: Fallos no Capturados

¿Qué testeamos?

- E2E (Pocos): Flujos críticos. "Login -> Comprar -> Pagar". Si esto falla, perdemos dinero.
- Integration: ¿Renderiza el componente con los datos?
- Unit (Muchos): Server Actions puras. Utilidades. Lógica de negocio aislada.



keep coding

Unit Testing: Server Actions

¡Son solo funciones!

La gran ventaja de las Server Actions es que son funciones asíncronas puras (`async function`).

No necesitas montar componentes, ni navegador, ni React Testing Library para testear la lógica de negocio.

Simplemente impórtala, mockea la BBDD y ejecútala.

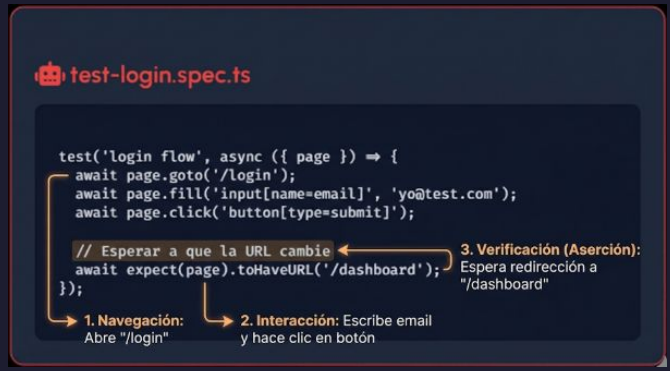
```
// actions.test.ts
import { createPost } from './actions';
import { db } from '@/lib/db';

// Mockeamos la BBDD
jest.mock('@/lib/db');

test('createPost guarda en DB', async () => {
  const formData = new FormData();
  formData.append('title', 'Hola Mundo');

  await createPost(formData);

  expect(db.post.create).toHaveBeenCalledWith({
    data: { title: 'Hola Mundo' }
  });
});
```

E2E: Playwright

El Robot Usuario

Playwright levanta un navegador real (Chrome/Firefox) y usa tu web como un humano.

- Click en botones.
- Relleno de formularios.
- Espera a que aparezcan elementos.
- Capturas de pantalla y videos de fallos.



keep coding

Live Coding 1: Playwright Setup

Init

Ejecutaremos `npm init playwright@latest`. Veremos la configuración de `playwright.config.ts`.

Write Test

Crearemos un test que navegue a nuestro Formulario de Tareas y añada una.

Run & UI Mode

Ejecutaremos el test en modo UI (`npx playwright test --ui`) para ver "la magia" en vivo.

Reto 1: El Robot Crítico

PAUSA EL VÍDEO (20 MIN)

Objetivo: Automatiza el flujo principal de tu app.

1. Instala Playwright.
2. Crea un test `tasks.spec.ts`.
3. Pasos: Ir a la home -> Escribir "Aprender Testing" -> Click Enviar.
4. **Assert:** Verifica que el texto "Aprender Testing" aparece en la lista de tareas.

keep coding

CI/CD: El Guardián del Deploy

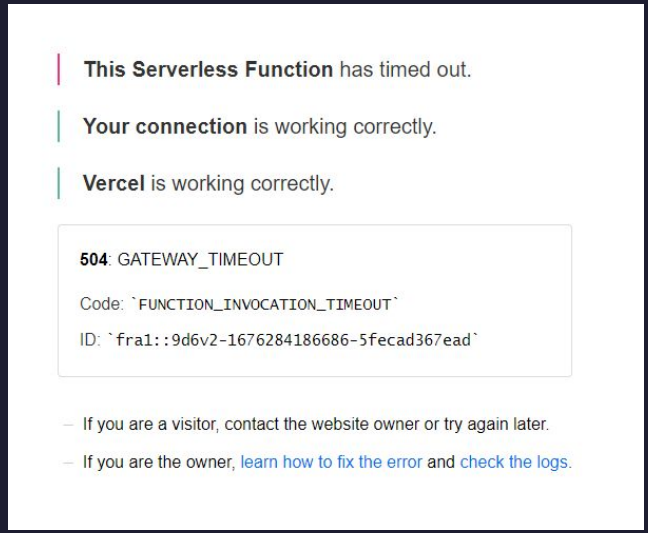
Recurso: más info para integrarlo en vuestra pipeline CI/CD →
https://vercel.com/kb/guide/how-can-i-run-end-to-end-tests-after-my-vercel-preview-deployment

La Realidad Serverless

El límite de los 10 segundos

En Vercel (Plan Hobby), una Server Action o API Route no puede durar más de **10 segundos** (o 60s pagando 20 €/mes).

Si intentas procesar un video, enviar 1000 emails o generar un PDF pesado... tu usuario verá un error 504 Gateway Timeout.



keep coding

Solución: Background Jobs

El Problema

El usuario no puede esperar. El navegador cierra la conexión.

La Arquitectura (Colas)

En lugar de procesar, la Server Action solo **encola** el trabajo ("Envia este email").
Usamos servicios propios como un servidor **Express.js**. O bien utilizar un proveedor como **Inngest**, **Trigger.dev** o **Upstash QStash**.

keep coding

Build Outputs: ¿Dónde despliego?

Default (Vercel)

Ideal para Serverless. Next.js divide tu app en pequeñas funciones lambda. Cero configuración.

Standalone

output: 'standalone'. Genera una carpeta mínima con solo lo necesario para correr en un contenedor Docker (AWS ECS, Hetzner, Digital Ocean).

Static Export

output: 'export'. Genera solo HTML/CSS/JS. Sin servidor. Pierdes Server Actions y SSR dinámico. (Solo para Landing Pages).

keep coding

El Camino Recorrido

De SPA a Full Stack

- **Clase 1:** RSC & Arquitectura
- **Clase 2:** Data & Streaming
- **Clase 3:** Server Actions
- **Clase 4:** Formularios y UX Avanzada
- **Clase 5:** Errores & Seguridad
- **Clase 6:** Testing & Producción

www.keepcoding.io cursos@keepcoding.io (+34) 916 33 1779