Clase 13 – Next.js: Features Avanzados

IIC2182 – Interfaces y Experiencia de Usuario



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

Primer Semestre 2026

Conectando con la clase pasada

La clase 12 terminó en data fetching desde el cliente con SWR y TanStack Query.

El problema del client-side fetching

Si el cliente hace `fetch` directo a la DB o a APIs con secrets...

Esos secrets quedan expuestos en el bundle que llega al browser.

Cualquiera con DevTools puede leer tu API key, conectarse a tu DB, abusar del rate limit.

La solución: una capa API en el server

El cliente nunca toca la DB ni APIs externas con secrets.

Habla con un endpoint tuyo, en tu dominio.

Ese endpoint corre en el server, donde los secrets viven seguros.

Client (SWR / TanStack) → Route Handler → DB / API privada

La respuesta de Next.js al "necesito una API que corra en el server" se llama Route Handlers.

Route Handlers

Tu propia API dentro del proyecto Next

¿Qué es un Route Handler?

Route Handler

Un archivo `route.ts` dentro de `app/` que exporta funciones `GET`, `POST`, `PUT`, `DELETE`. Next lo convierte en un endpoint HTTP.

¿Para qué sirve?

APIs internas (webhooks, formularios, integraciones).

Proxying a APIs externas escondiendo secrets.

Endpoints consumidos por Client Components (SWR, TanStack).

vs page.tsx

`page.tsx` devuelve HTML.

`route.ts` devuelve **Response** (JSON, texto, etc.).

No pueden coexistir en la misma carpeta.

```ts
// app/api/items/route.ts
import { NextResponse } from 'next/server'

const items = [{ id: 1, name: 'Uno' }]

export async function GET() {
  return NextResponse.json(items)
}

export async function POST(req: Request) {
  const body = await req.json()
  const newItem = { id: items.length + 1, ...body }
  items.push(newItem)
  return NextResponse.json(newItem, { status: 201 })
}
```

Route Handlers — métodos y params

Dynamic segments

```ts
// app/api/items/[id]/route.ts
export async function GET(
  req: Request,
  { params }: { params: Promise<{ id: string }> }
) {
  const { id } = await params
  const item = await db.item.findUnique({
    where: { id },
  })
  if (!item) {
    return new Response('Not found', { status: 404 })
  }
  return Response.json(item)
}

export async function DELETE(
  req: Request,
  { params }: { params: Promise<{ id: string }> }
) {
  const { id } = await params
```

Request helpers

`req.json()` — body JSON

`req.formData()` — forms

`req.headers` — headers del request

`new URL(req.url).searchParams` — query

Response helpers

`Response.json(data)`

`NextResponse.redirect(url)`

`new Response('...', { status })`

Por default son dynamic

Se ejecutan en cada request. Podés cacheación con `export const revalidate` si el handler es idempotente.

Backend sin servidor backend

Los Route Handlers no son solo "una API JSON". Son la puerta para que Next sea full-stack — tareas tradicionales del backend, dentro del mismo proyecto.

```ts
// app/api/export/route.ts
import { db } from '@/lib/db'

export async function GET() {
  const orders = await db.order.findMany({
    include: { customer: true },
  })

  // Transformar y agregar
  const rows = orders.map(o => [
    o.id,
    o.customer.name,
    o.total,
    o.createdAt.toISOString(),
  ])

  const csv = [
    'id,customer,total,created_at',
    ...rows.map(r => r.join(',')),
  ].join('\n')
```

Generar archivos descargables

CSV de reportes (ej: ventas, usuarios).

PDFs facturados (con `pdfkit`, `react-pdf`).

ZIP con varios archivos.

Transformar / agregar datos

Sumar, agrupar, joinear varias fuentes.

Consultar 3 APIs y devolver una sola respuesta unificada.

Enriquecer con datos pre-computados.

Lo que antes requería un Express aparte

Sin spinear un servidor Node, sin AWS Lambda, sin gateway. Todo vive dentro del mismo repo y mismo deploy.

Streaming — UI progresiva con Suspense

Si tu Route Handler o page necesita esperar varias fuentes, no hace falta bloquear toda la respuesta.

```tsx
// app/dashboard/page.tsx
import { Suspense } from 'react'

export default function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>

      <Suspense fallback={<p>Cargando stats…</p>}>
        <Stats />
      </Suspense>

      <Suspense fallback={<p>Cargando actividad…</p>}>
        <Activity />
      </Suspense>
    </div>
  )
}

async function Stats() {
```

¿Qué cambia?

Sin Suspense: la página espera 4s (el fetch más lento).

Con Suspense: el shell aparece inmediato, cada sección se revela cuando su fetch termina.

Cómo funciona técnicamente

Next mantiene la conexión HTTP abierta.

Manda chunks de HTML a medida que cada `Suspense` resuelve.

El browser los pinta en orden, sin recargar.

loading.tsx = Suspense del segmento

Poner `loading.tsx` en una carpeta envuelve su `page.tsx` en un `<Suspense>` con ese fallback.

Server Actions

Mutaciones desde forms, sin escribir un endpoint

El problema: forms en React tradicional

Enviar un form a una API en React puro es sorprendentemente verboso.

Flujo tradicional

1. Crear endpoint `POST /api/tasks`.
2. En el client: `useState` para cada campo.
3. `onSubmit` con `preventDefault`.
4. `fetch` con JSON.stringify.
5. `useState` para errores y loading.
6. Invalidar cache manualmente.

Mucho código repetido para algo que en HTML clásico era solo `<form action="...">`.

Con Server Actions

1. Función con `'use server'`.
2. `<form action={createTask}>`.
3. Listo.

La función corre en el servidor. Next maneja el transporte, la serialización y la invalidación.

Server Action básica

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

import { revalidatePath } from 'next/cache'
import { db } from '@/lib/db'

export async function createTask(formData: FormData) {
  const title = formData.get('title') as string

  await db.task.create({
    data: { title, completed: false },
  })

  revalidatePath('/tasks')
}
```

```tsx
// app/tasks/new/page.tsx
import { createTask } from '@/app/actions'

export default function NewTaskPage() {
  return (
    <form action={createTask}>
      <input
        name="title"
        placeholder="Nueva tarea"
        required
      />
      <button type="submit">Crear</button>
    </form>
  )
}
```

El form ni siquiera necesita `"use client"`. El action se ejecuta en el servidor cuando se envía.

Detrás de escena, Next crea un endpoint, serializa el FormData, y te da la data tipada. Sin `fetch`, sin JSON, sin endpoint manual.

Demo: `nextjs-demo/app/server-actions/form/page.tsx`

Server Functions desde un Client Component

Una Server Action no está limitada a `<form action>`. Puede invocarse como cualquier función desde un Client Component — botones, hooks, efectos.

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

import { revalidatePath } from 'next/cache'
import { db } from '@/lib/db'

export async function toggleLike(postId: string) {
  const existing = await db.like.findFirst({
    where: { postId, userId: getCurrentUserId() },
  })

  if (existing) {
    await db.like.delete({ where: { id: existing.id } })
  } else {
    await db.like.create({
      data: { postId, userId: getCurrentUserId() },
    })
  }

  revalidatePath(`/posts/${postId}`)
```

```tsx
// app/posts/LikeButton.tsx
'use client'

import { useState, useTransition } from 'react'
import { toggleLike } from '@/app/actions'

export function LikeButton({ postId, initialLiked }) {
  const [liked, setLiked] = useState(initialLiked)
  const [isPending, startTransition] = useTransition()

  return (
    <button
      disabled={isPending}
      onClick={() => {
        startTransition(async () => {
          // Llamar la Server Action como una función nor
          const result = await toggleLike(postId)
          setLiked(result.liked)
        })
      }}
```

¿Qué pasa al importarla?

useTransition

Cuándo usarlo así

Para forms reales: react-hook-form + zod

Los hooks nativos resuelven lo básico. Pero un form de producción tiene mucho más.

Lo que NO trae React solo

Validación campo a campo, en tiempo real.

Reglas complejas: depende de otro campo, async checks.

Mensajes de error tipados, en un solo lugar.

Manejo de re-renders eficiente con muchos inputs.

Reset, dirty state, touched, default values.

La combinación estándar de la industria

Zod — define el schema de validación.

react-hook-form — maneja el estado del form sin re-renders innecesarios.

Ambas cero dependencias, TypeScript-first, ~25kb total.

Zod schema (qué es válido) → **react-hook-form** (estado + validación) → **Server Action** (re-valida + guarda)

La regla: el mismo schema valida en el cliente y en el server. Una sola fuente de verdad.

Zod: schemas como única fuente de verdad

Zod te deja describir un objeto y te da el tipo TypeScript gratis + el validador.

```ts
// app/tasks/schema.ts
import { z } from 'zod'

export const taskSchema = z.object({
  title: z
    .string()
    .min(3, 'Mínimo 3 caracteres')
    .max(80, 'Máximo 80 caracteres'),

  priority: z.enum(['low', 'medium', 'high']),

  dueDate: z.coerce.date()
    .refine((d) => d > new Date(), {
      message: 'La fecha debe ser futura',
    }),

  estimatedHours: z
    .number()
    .positive()
    .max(40)
```

Validar en runtime

```ts
const result = taskSchema.safeParse(data)
if (!result.success) {
  // result.error.issues → array de errores
}
const task = result.data  // ya tipado
```

Operadores potentes

`.refine()` — reglas custom.

`.transform()` — convertir el dato saliente.

`z.union` / `.discriminatedUnion` — sum types.

react-hook-form: useForm + register

```tsx
'use client'
import { useForm } from 'react-hook-form'
import { zodResolver } from '@hookform/resolvers/zod'
import { taskSchema, type Task } from './schema'

export function TaskForm() {
  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting },
  } = useForm<Task>({
    resolver: zodResolver(taskSchema),
    defaultValues: { priority: 'medium' },
  })

  const onSubmit = async (data: Task) => {
    await createTaskAction(data)  // Server Action
  }

  return (
```

register(name)

Conecta un input al form. Devuelve `{...register, onBlur, ref, name}` — cero boilerplate.

handleSubmit(fn)

Valida con el resolver, y solo si pasa llama tu `onSubmit` con el objeto tipado.

formState

`errors`, `isSubmitting`, `isDirty`, `touchedFields` — todo lo que necesitás para la UI.

Re-renders mínimos

Usa `refs` internamente. Tipear en un input no re-renderiza el form entero.

Cliente + server validan con el mismo schema

El cliente da feedback rápido. El server es la autoridad — nunca confiar en validación de cliente.

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

import { revalidatePath } from 'next/cache'
import { taskSchema, type Task } from './schema'
import { db } from '@/lib/db'

export async function createTaskAction(input: Task) {
  // Re-validar SIEMPRE en el server
  const result = taskSchema.safeParse(input)
  if (!result.success) {
    return {
      ok: false as const,
      errors: result.error.flatten().fieldErrors,
    }
  }

  await db.task.create({ data: result.data })
  revalidatePath('/tasks')
  return { ok: true as const }
```

```ts
'use client'
// Mostrar errores del server en el form
const onSubmit = async (data: Task) => {
  const res = await createTaskAction(data)

  if (!res.ok) {
    // Inyectar errores del server en RHF
    Object.entries(res.errors).forEach(
      ([field, messages]) => {
        setError(field as keyof Task, {
          message: messages?.[0],
        })
      }
    )
    return
  }

  toast.success('Tarea creada')
  reset()
}
```

Por qué doble validación

El cliente da UX. El server es seguridad — un atacante puede saltarse la validación del bundle JS y mandar lo que quiera.

Middleware

Código que corre antes de cada request

¿Qué es un Middleware?

> **Middleware**
>
> *Una función que intercepta cada request entrante antes de que Next.js decida qué* `page.tsx` *o* `route.ts` *renderizar. Puede inspeccionar, modificar, redirigir o bloquear la request.*

El ciclo de vida de una request en Next.js

1. El navegador envía un HTTP request a tu app.
2. **middleware.ts** corre — decide si pasar, redirigir o reescribir.
3. Next resuelve la ruta — `page.tsx` o `route.ts`.
4. Se devuelve la respuesta al cliente.

Tres acciones posibles

`NextResponse.next()` — dejar pasar.

`NextResponse.redirect(url)` — mandar a otro lado.

`NextResponse.rewrite(url)` — servir otra ruta sin cambiar la URL.

¿Por qué es útil?

Lógica que aplica a muchas rutas (auth, i18n, redirects) vive en un solo archivo, no duplicada en cada page.

Es el guardián de entrada de tu app — todo lo que entra pasa primero por acá.

Middleware: qué y cuándo

```ts
// middleware.ts vive en la RAÍZ del proyecto
// (hermano de package.json, NO dentro de app/)
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

// ① Export nombrado obligatorio: 'middleware'
//    Next lo encuentra por convención y lo corre.
export function middleware(req: NextRequest) {
  // ② Acceso tipado a cookies, headers, URL, geo, etc.
  const session = req.cookies.get('session')

  // ③ Decidir según la ruta entrante
  if (req.nextUrl.pathname.startsWith('/dashboard')) {
    if (!session) {
      // ④ redirect() corta el flujo: no llega al page.ts
      return NextResponse.redirect(
        new URL('/login', req.url),
      )
    }
  }
```

Anatomía del código

① El archivo se llama `middleware.ts` y vive en la raíz — Next lo descubre por convención.

② `NextRequest` extiende `Request` con `cookies`, `nextUrl`, `geo`.

③ `nextUrl.pathname` es la ruta sin host ni query — más fácil que parsear `req.url`.

④ `NextResponse.redirect` termina el ciclo acá: el page nunca se ejecuta.

⑤ `NextResponse.next()` es el "pasa": Next sigue resolviendo la ruta normal.

⑥ `matcher` usa la sintaxis de `path-to-regexp` — `:path*` matchea cualquier subruta.

Middleware — casos y restricciones

Casos típicos

Auth — proteger rutas privadas

i18n — redirect según idioma del browser

A/B testing — variante por cookie

Rewrites — URLs bonitas para backends

Rate limiting básico por IP

Restricciones

Corre en **Edge Runtime** — no APIs de Node (fs, net, etc.).

No accedés a la DB directamente — usá un endpoint o token.

Debe ser **rápido** — corre en cada request matched.

Regla práctica: si tu lógica necesita la DB o tarda más de 50ms, sacala del middleware y ponela en un Server Component o Route Handler.

Auth Libraries

No implementes auth desde cero

El problema de auth a mano

Implementar auth correctamente requiere resolver muchos detalles sutiles — todos con impacto de seguridad.

⚠ Lo que tenés que resolver

Hashing de passwords (bcrypt, argon2)
Gestión de sesiones (JWT vs cookies)
CSRF protection
Refresh tokens y rotación
OAuth flows (Google, GitHub, etc.)
Email verification, password reset
2FA / MFA
Protección de rutas (middleware)

La solución: una librería

Delega esto a librerías que ya lo resolvieron, auditaron y mantienen.

Tres opciones dominantes en el ecosistema Next:

- Auth.js (ex-NextAuth) — OSS, más usado
- Clerk — comercial, UI premade
- Supabase Auth — bundled con DB

Comparación de opciones

Dimensión	Auth.js	Clerk	Supabase
Modelo	Open source	SaaS (free tier)	OSS + SaaS
UI incluida	Construís vos	Componentes listos	Construís vos
Base de datos	La tuya (adapters)	De Clerk	De Supabase (Postgres)
Setup inicial	Medio	Muy rápido	Medio
Costo	$0	$0 → $25+/mo	$0 → $25+/mo

Auth.js — setup básico

```ts
// auth.ts
import NextAuth from 'next-auth'
import GitHub from 'next-auth/providers/github'

export const { handlers, signIn, signOut, auth } =
  NextAuth({
    providers: [GitHub],
  })
```

```ts
// app/api/auth/[...nextauth]/route.ts
export { GET, POST } from '@/auth'
```

```ts
// middleware.ts
export { auth as middleware } from '@/auth'
```

Usar la sesión en un Server Component:

```tsx
// app/dashboard/page.tsx
import { auth } from '@/auth'

export default async function Dashboard() {
  const session = await auth()

  if (!session) {
    return <p>No autenticado</p>
  }

  return (
    <div>
      <p>Hola, {session.user?.name}</p>
      <img src={session.user?.image} />
    </div>
  )
}
```

El middleware protege las rutas, `auth()` lee la sesión, y los providers (GitHub, Google, credentials, etc.) manejan el flujo.

Cuándo usar Next.js — y cuándo no

Next brilla cuando...

- Necesitás SEO (blogs, e-commerce, marketing sites)
- Mezclás contenido estático y dinámico
- Querés convenciones para un equipo grande
- Full-stack en un solo repo (frontend + API)
- Deploy rápido sin configurar infra

Quizás no lo necesites si...

- Es una app detrás de login, sin SEO (Vite + React suele bastar)
- Ya tenés un backend separado robusto y solo querés UI
- Necesitás ultra-control del bundle (framework-less)
- Es un prototipo de 10 líneas (CodePen, v0, StackBlitz)

Alternativas modernas

- Astro — contenido estático + islands
- Remix — forms + nested routing (web standards first)
- SvelteKit — mismas ideas pero con Svelte
- TanStack Start — SSR con TanStack Router

Resumen de las dos clases

File-based Routing Carpetas son URLs, page.tsx es la ruta, layout.tsx persiste

Server Components (default) async + fetch directo, cero JS al cliente, seguro y rápido

Client Components "use client" cuando hay interactividad, hooks o browser APIs

Rendering Strategies SSG / SSR / ISR / CSR — Next decide, vos ajustás si hace falta

Data Fetching Server por default; SWR/TanStack cuando el cliente lo requiere

Route Handlers API layer en el server: protege secrets, recibe webhooks, exporta archivos

Streaming + Suspense UI progresiva: el shell en milisegundos, las secciones lentas después

Server Actions Forms y client-button mutations sin endpoints manuales — 'use server' + revalidate

Middleware + Auth Edge logic antes de cada request; auth libraries (Auth.js, Clerk)

Deploy Vercel por 1 clic; funciona en Netlify, AWS, self-hosted

¿Preguntas?

Clase 13 - Next.js: Features Avanzados



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

IIC2182 - Primer Semestre 2026