Universidad Icesi

Sesiones y manejo de errores en Next.js

Computación en Internet 3 · Semana 14

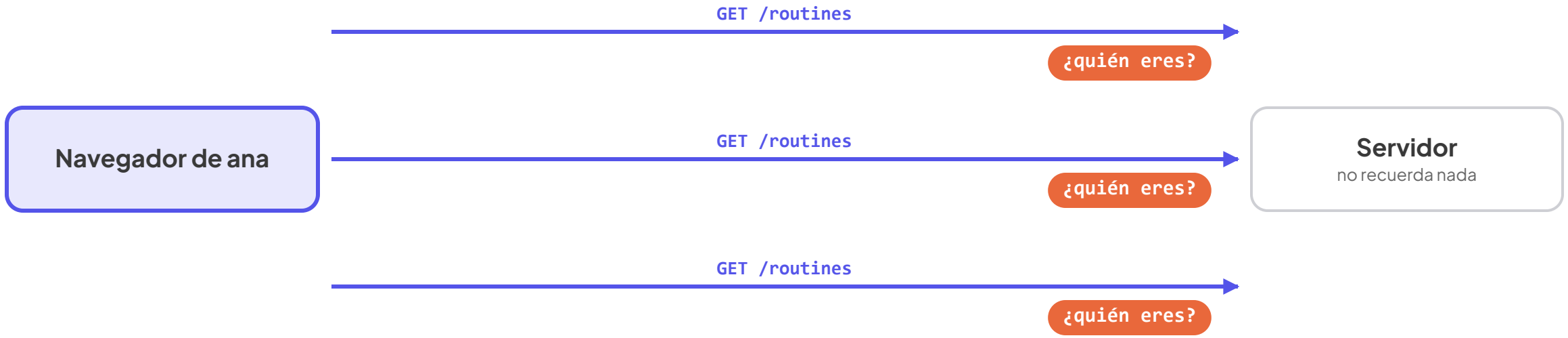
Universidad Icesi

Universidad ICESI

1. ¿Qué es una sesión?

Y dónde se guarda

HTTP no tiene memoria



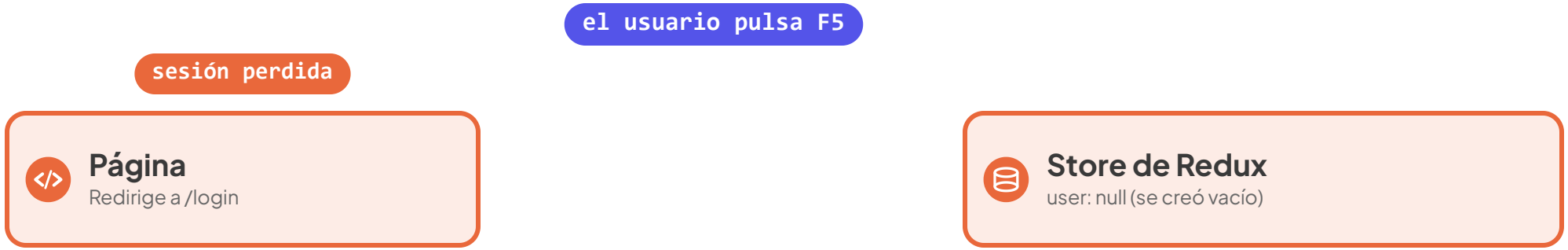
Cada petición llega sola. Una **sesión** es lo que se guarda después del login y viaja con **cada petición** para que el servidor sepa quién eres.

El login de la semana 13: la sesión vive en Redux



Mientras no recargues, la página sabe quién eres: el login guardó al usuario en el store.

Al recargar, la sesión desaparece



Recargar crea un store nuevo desde cero. **La memoria de la página no es una sesión**: hay que guardarla en otro lado.

Guardar la sesión en memoria (Redux)



Memoria (Redux)

Sobrevive a recargar

No

El servidor la puede leer

No

JavaScript puede leerla

Difícil, pero se pierde

Viaja sola en cada petición

No

Sirve para el estado de la interfaz, **no como sesión**: al recargar o abrir otra pestaña se pierde.

Guardar la sesión en localStorage



localStorage

Sobrevive a recargar

Sí

El servidor la puede leer

No: solo en el navegador

JavaScript puede leerla

Sí: un XSS la roba

Viaja sola en cada petición

No: la agregas tú

Cómodo, pero el servidor no lo ve (el primer render no sabe quién eres) y un script malicioso puede leerlo con una línea.

Guardar la sesión en cookie común



Cookie común

Sobrevive a recargar

Sí, hasta que vence

El servidor la puede leer

Sí

JavaScript puede leerla

Sí: un XSS la roba

Viaja sola en cada petición

Sí

El servidor ya la ve, pero si la crea JavaScript, también la puede leer cualquier script de la página.

Guardar la sesión en cookie httpOnly



Cookie httpOnly

Sobrevive a recargar

Sí, hasta que vence

El servidor la puede leer

Sí

JavaScript puede leerla

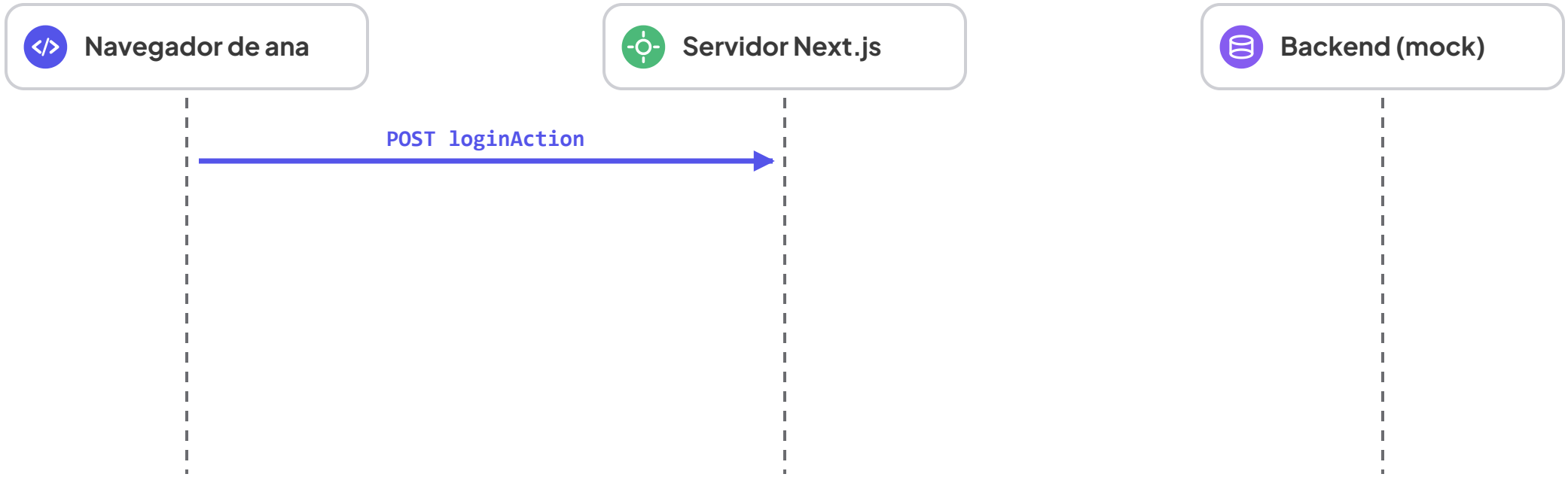
No

Viaja sola en cada petición

Sí

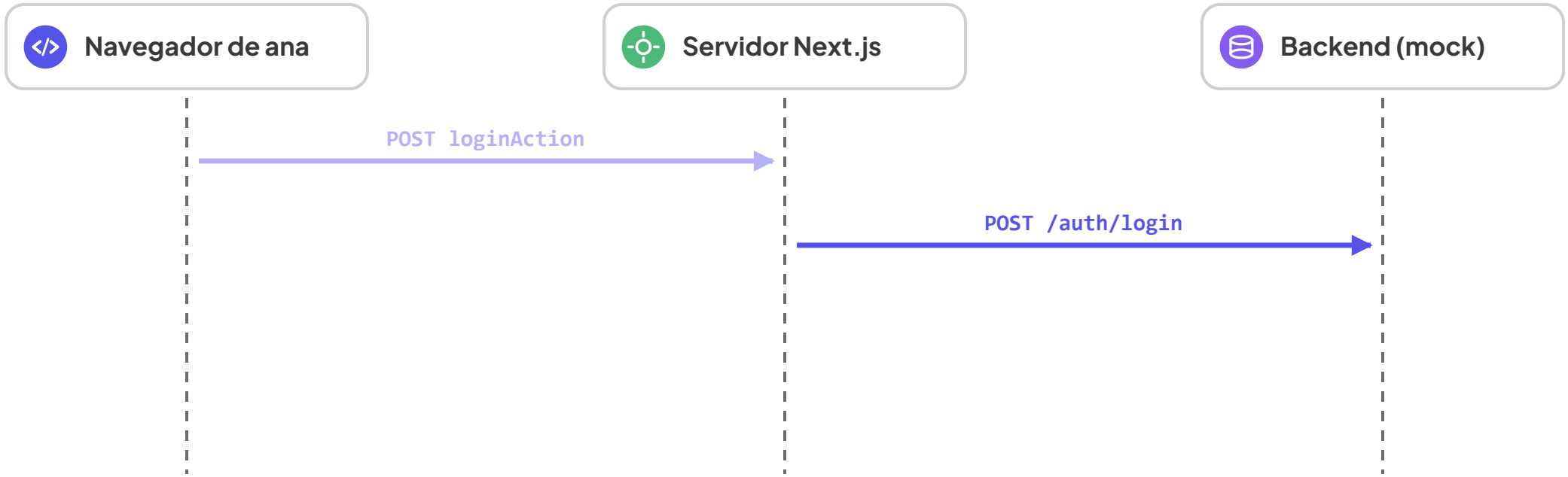
La opción recomendada para una sesión: la crea el servidor, persiste, viaja sola y JavaScript no puede leerla.

Iniciar sesión (1 de 4)



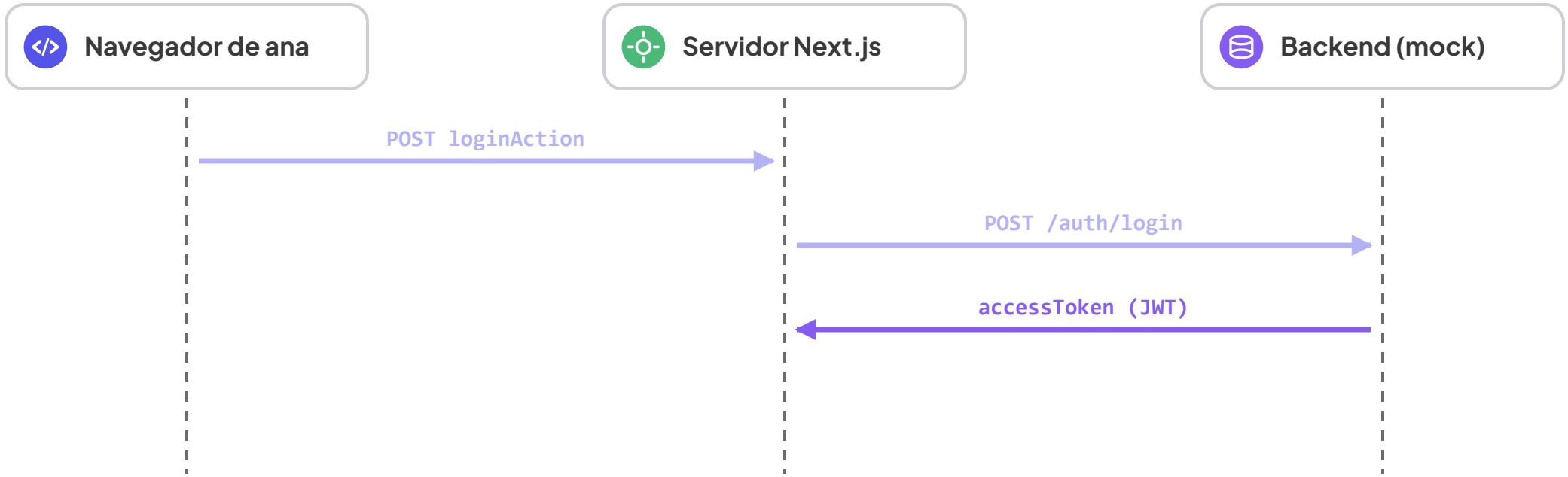
LoginForm llama a useAuth().login(), el hook despacha el thunk y el thunk invoca **loginAction()**: un POST al servidor. No escribiste ningún endpoint.

Iniciar sesión (2 de 4)



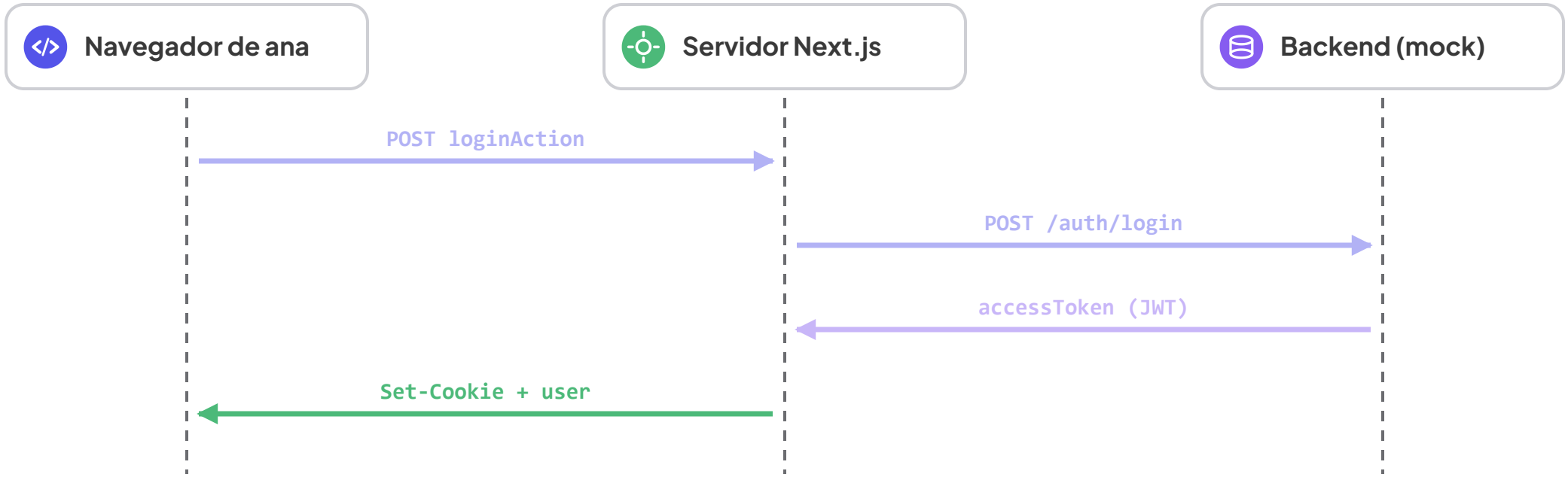
La acción valida los datos y llama a LoginUseCase, que le pide el token al **backend** (aquí, un mock).

Iniciar sesión (3 de 4)



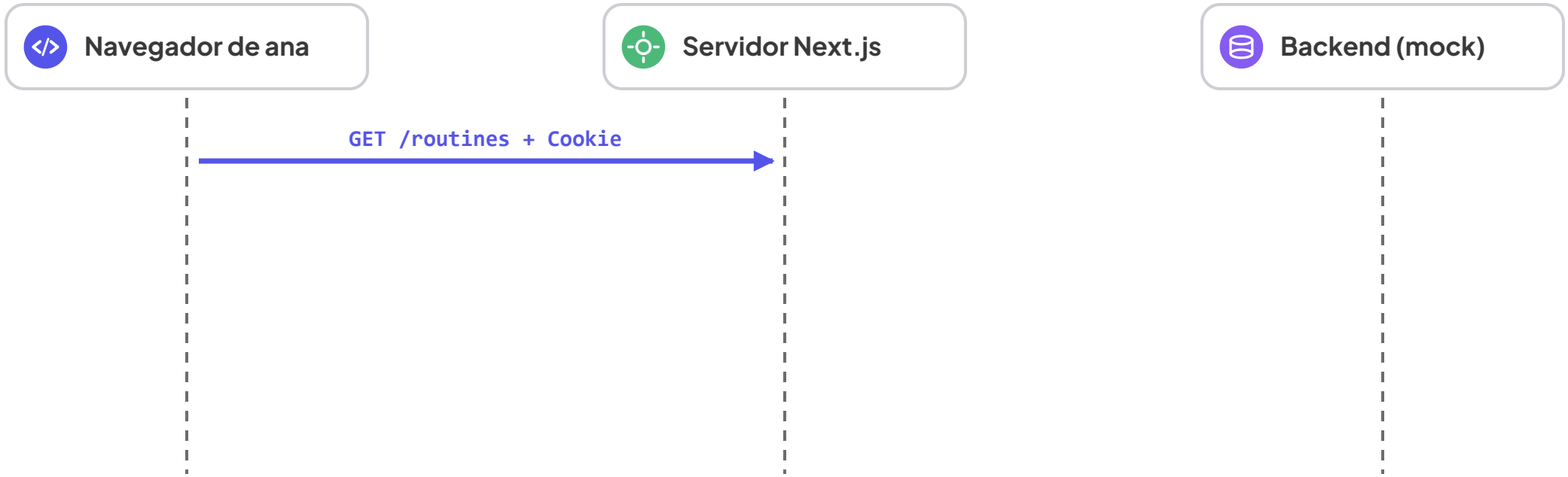
Si las credenciales son correctas, el backend **firma un JWT** con el id, el nombre y los roles de ana, y lo devuelve.

Iniciar sesión (4 de 4)



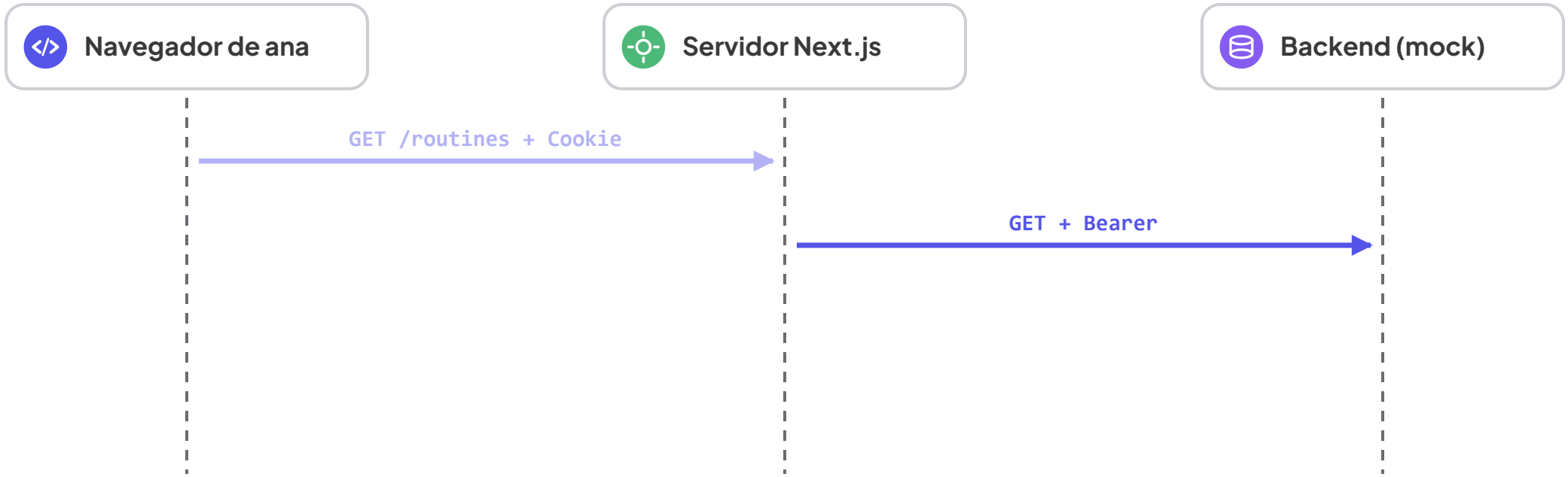
La acción **decodifica** el token, lo guarda tal cual en una cookie httpOnly y devuelve el usuario. Después el cliente navega con router.replace.

Pedir una ruta protegida (1 de 4)



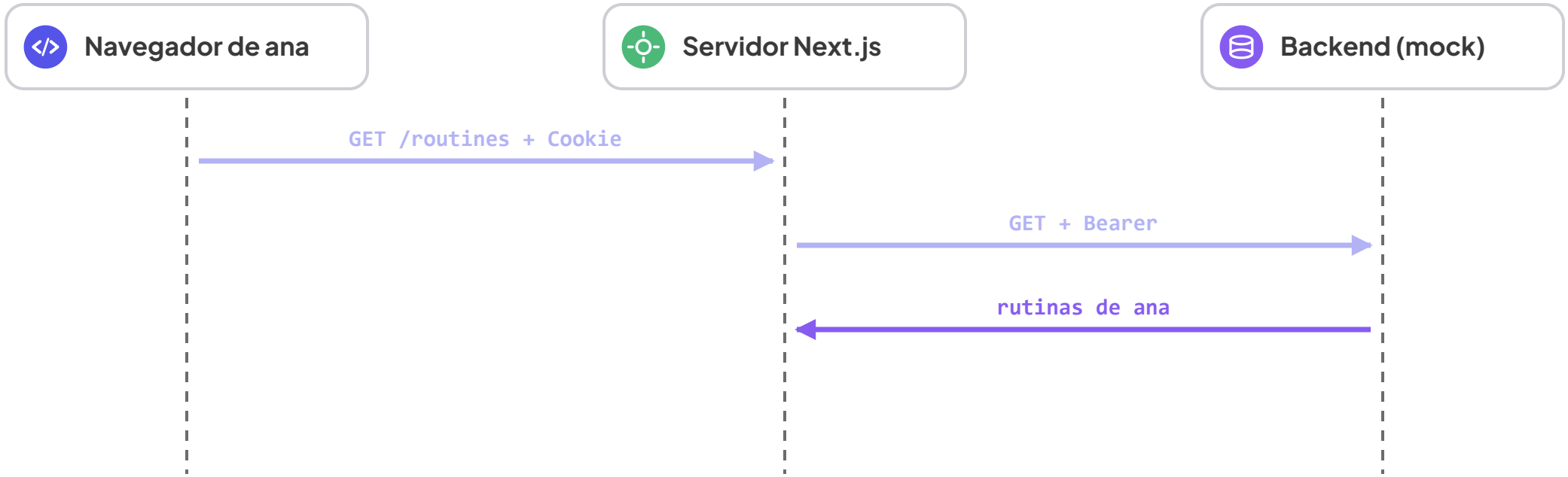
El navegador envía la cookie **solo**. El proxy decodifica el token, comprueba que no haya vencido (sin llamar al backend) y deja pasar.

Pedir una ruta protegida (2 de 4)



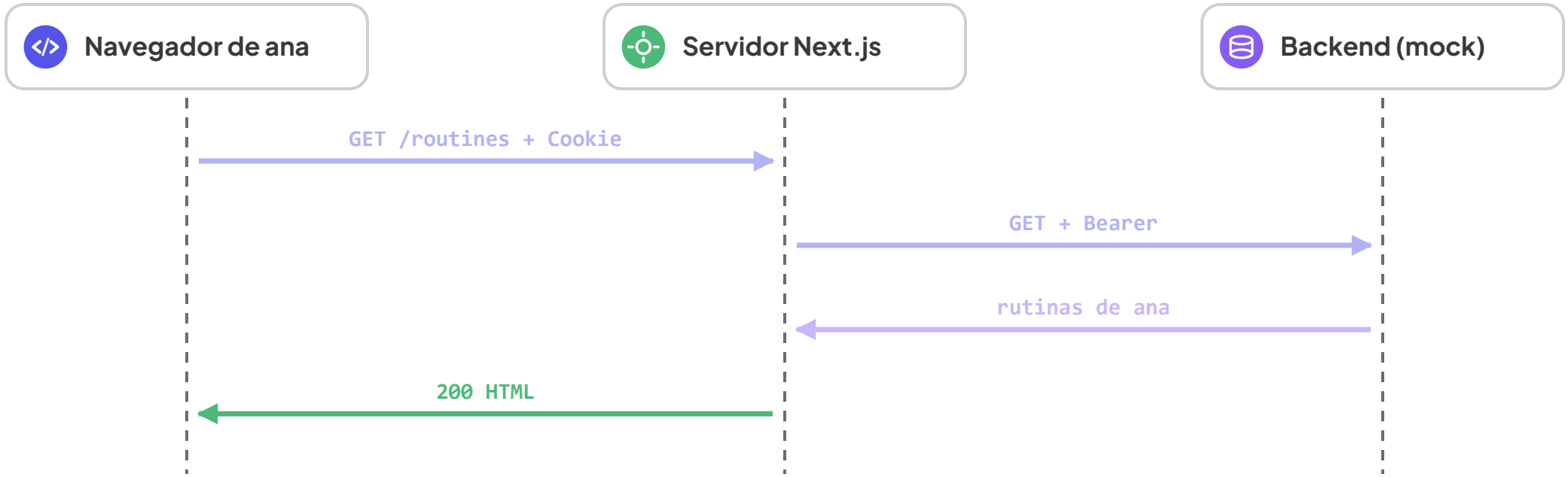
requireSession() decodifica el token y obtiene el id de ana. La página le pide las rutinas al backend con `Authorization: Bearer`.

Pedir una ruta protegida (3 de 4)



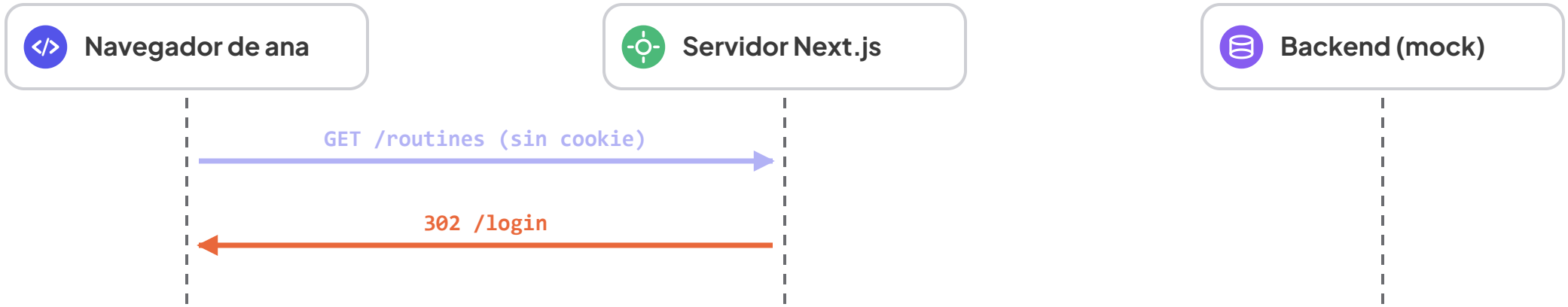
El backend **verifica la firma** del token: él es quien lo valida. Solo devuelve las rutinas de ese usuario.

Pedir una ruta protegida (4 de 4)



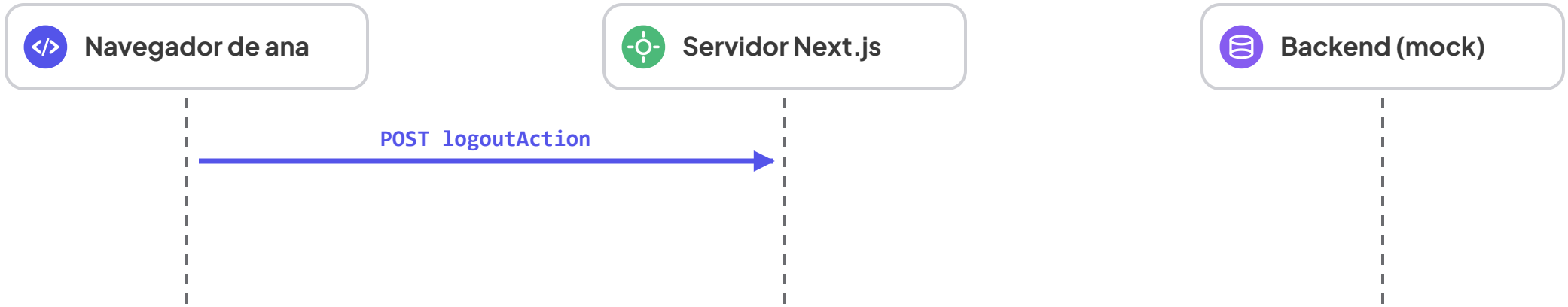
El usuario ve /routines. Todo ocurrió en el servidor: **el navegador nunca vio el token**.

Y si no hay cookie



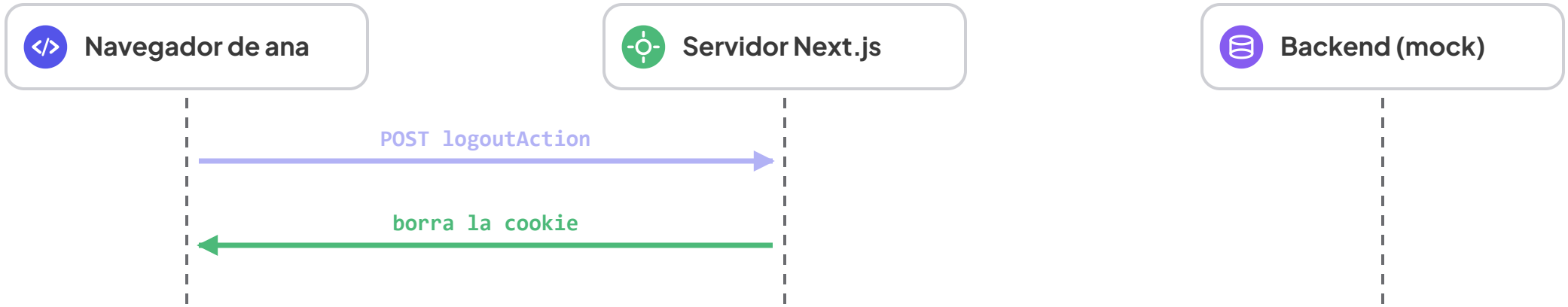
Otro navegador o una ventana de incógnito: la petición llega sin sesión. El proxy redirige a /login **sin renderizar la página ni llamar al backend**.

Cerrar sesión (1 de 2)



El botón «Cerrar sesión» llama a useAuth().logout(): el thunk invoca la Server Action **logoutAction()**.

Cerrar sesión (2 de 2)



Solo el servidor puede borrar una cookie httpOnly: responde con `Set-Cookie` y `Max-Age=0`. El cliente navega a /login.

2. Configurar la sesión

Guardar y decodificar el token

Universidad ICESI

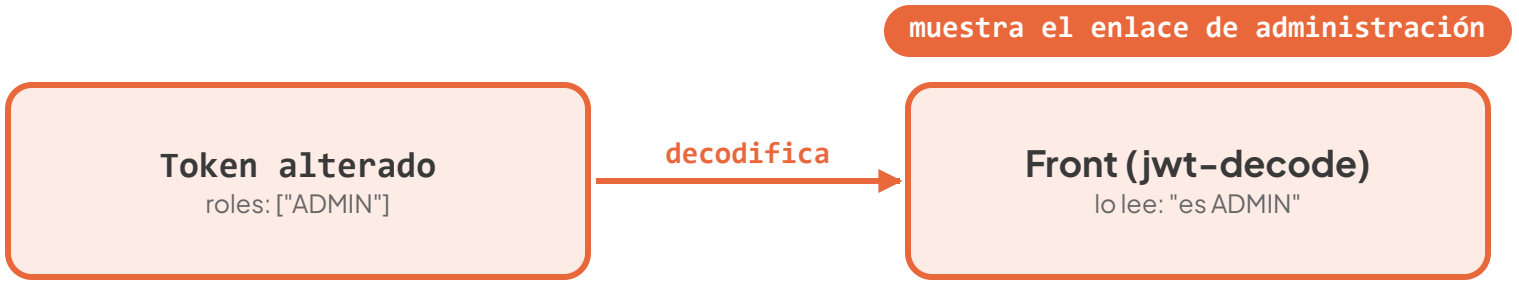
Tres trabajos del front con el token

1. Guardarlo
En una cookie httpOnly: sobrevive a las recargas y JavaScript no la lee.

2. Decodificarlo
Para saber quién es el usuario y qué roles tiene, sin ninguna petición. Esa información va al estado de la app.

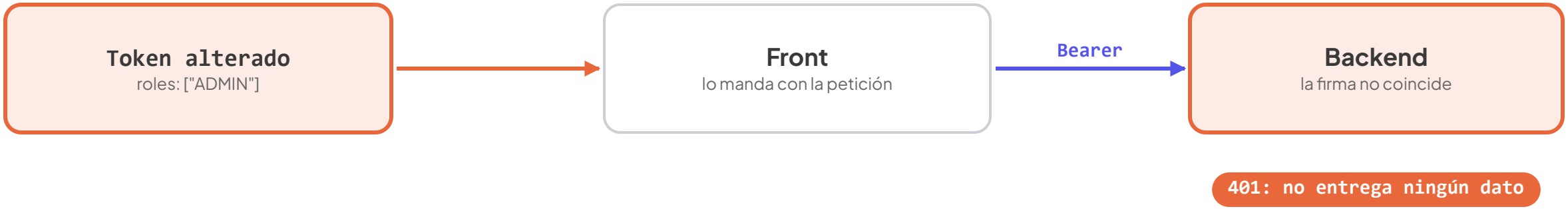
3. Reenviarlo
Al backend en cada llamada (Authorization: Bearer). El backend verifica la firma y decide.

Lo que decodifica el front solo sirve para la interfaz



Alguien modificó el contenido del token. **El front lo decodifica igual**: no comprueba la firma, así que lo creería.

El backend es quien verifica



Lo que lee el front sirve para **mostrar u ocultar**, nunca para proteger datos. Un token falso no consigue ni un dato: lo verifica el backend.

¿Qué librería para decodificar?

Opción	Qué hace	Veredicto
jwt-decode	Decodifica; no necesita clave; funciona en navegador y servidor	**La elegida**
jose	Verifica y firma además de decodificar	Más de lo que necesitamos
atob a mano	Decodifica el base64 y el UTF-8 tú mismo	Funciona, pero es trabajo manual

No hace falta ninguna variable de entorno: **el front no usa ninguna clave**. El mock firma con un secreto de prueba dentro de su propio archivo.

Instalar

```terminal
npm install jwt-decode zod server-only
```

jwt-decode
Decodifica el token. No lo verifica ni lo firma.

zod
Valida los datos que llegan del formulario.

server-only
Hace fallar el build si un componente de cliente importa un archivo del servidor.

session.ts: decodificar el token

src/lib/session.ts

```ts
import { jwtDecode } from 'jwt-decode'

export function decodeToken(token?: string) {
  if (!token) return null
  try {
    const payload = jwtDecode<TokenPayload>(token)
    if (payload.exp * 1000 < Date.now()) return null
    return payload
  } catch {
    return null
  }
}
```

jwtDecode abre el token
Devuelve sub, name, roles y exp. No necesita ninguna clave: un JWT está en base64.

null en vez de un error
Sin token o con un token mal formado: «no hay sesión» es un resultado normal.

Y la firma, ¿quién la comprueba?

src/lib/session.ts

```ts
import { jwtDecode } from 'jwt-decode'

export function decodeToken(token?: string) {
  if (!token) return null
  try {
    const payload = jwtDecode<TokenPayload>(token)
    if (payload.exp * 1000 < Date.now()) return null
    return payload
  } catch {
    return null
  }
}
```

No el front. Decodificar no comprueba que nadie haya modificado el token: un token alterado se decodifica igual de bien. La firma la comprueba el backend cada vez que recibe el token.

Vigencia: el campo exp

src/lib/session.ts

```ts
import { jwtDecode } from 'jwt-decode'

export function decodeToken(token?: string) {
  if (!token) return null
  try {
    const payload = jwtDecode<TokenPayload>(token)
    if (payload.exp * 1000 < Date.now()) return null
    return payload
  } catch {
    return null
  }
}
```

exp son segundos desde 1970
Se multiplica por 1000 para compararlo con Date.now().

Comprobar `exp` no es verificar la firma: solo evita tratar como sesión un token que ya no sirve.

createSession: guarda el token del backend

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

El token lo emite el backend
El front no lo crea ni lo firma: lo decodifica para leer el vencimiento.

Token ilegible, sin sesión
Si no se puede decodificar, no se crea la cookie.

cookies() es asíncrona

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

await cookies()
Desde Next.js 15 hay que esperarla.

Solo en una acción o un Route Handler
set y delete no se pueden usar mientras se renderiza un Server Component.

set(): la respuesta lleva Set-Cookie

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

El navegador la guarda
Y la reenvía sola en cada petición a este sitio.

```
Set-Cookie: session=eyJhbG…; Path=/; Expires=…; HttpOnly; Secure; SameSite=Lax
```

httpOnly: JavaScript no la puede leer

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

document.cookie no la muestra
Un script inyectado (XSS) no puede copiar el token.

Reduce el daño, no lo elimina
Ese script aún puede hacer peticiones como el usuario.

Por eso el cliente no puede decodificar el token: lo decodifica **el servidor** y entrega a la interfaz solo el usuario y sus roles.

secure: solo por HTTPS

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

Nadie en la red la copia
El navegador solo la envía en conexiones cifradas.

En localhost
Los navegadores modernos suelen aceptarla igual; Safari no.

sameSite: 'lax' frena el CSRF

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

No se manda en peticiones de otro sitio
Por ejemplo, un formulario POST de otra página.

Sí al llegar con un enlace
strict no la manda ni así; none siempre (exige secure).

expires: la cookie vence con el token

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

Se toma del exp del token
Cookie y token caducan a la vez.

Al vencer, el proxy redirige al login
Con un refresh token se podría renovar sin pedir la contraseña (no lo vemos aquí).

path: '/' = todo el sitio

src/lib/session.ts

```ts
export async function createSession(token: string) {
  const payload = decodeToken(token)
  if (!payload) throw new Error('Token inválido')

  const store = await cookies()
  store.set('session', token, {
    httpOnly: true,
    secure: true,
    sameSite: 'lax',
    expires: new Date(payload.exp * 1000),
    path: '/',
  })
  return payload
}
```

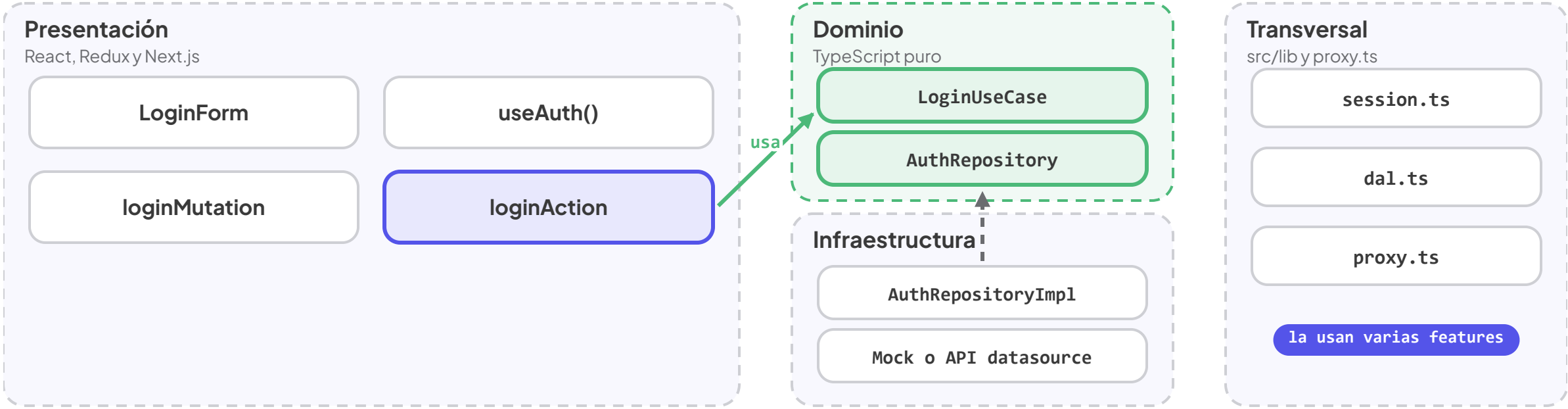
La cookie acompaña a todas las rutas
El proxy y las páginas de cualquier ruta necesitan leerla.

Universidad ICESI

3. Integrarla en tu arquitectura

Dónde va cada archivo

El dominio no sabe de cookies ni de JWT



La regla de dependencia de la semana 12 no cambia: una cookie es un detalle de HTTP y un JWT es un detalle del backend. **El dominio solo pregunta: ¿estas credenciales me dan un token?**

Dominio: ¿estas credenciales dan un token?

estructura de la feature auth

```
features/auth/
├── domain/
│   ├── entities/auth-tokens.ts
│   ├── repositories/auth.repository.ts
│   └── usecases/login.usecase.ts
├── infrastructure/
│   ├── auth.mock.datasource.ts
│   ├── auth.api.datasource.ts
│   └── auth.repository.impl.ts
├── di/auth.container.ts
└── presentation/
    ├── actions/auth.actions.ts
    ├── store/auth.slice.ts
    ├── hooks/use-auth.ts
    └── components/ LoginForm, AuthHydrator
lib/session.ts   lib/dal.ts
store/store.ts   (+ authReducer)
proxy.ts         (en la raíz)
```

Un tipo, un contrato y un caso de uso
AuthTokens, AuthRepository y LoginUseCase.

Nada de cookies ni de JWT
La sesión no es una regla de negocio.

Infraestructura: el backend simulado

estructura de la feature auth

```
features/auth/
├── domain/
│   ├── entities/auth-tokens.ts
│   ├── repositories/auth.repository.ts
│   └── usecases/login.usecase.ts
├── infrastructure/
│   ├── auth.mock.datasource.ts
│   ├── auth.api.datasource.ts
│   └── auth.repository.impl.ts
├── di/auth.container.ts
└── presentation/
    ├── actions/auth.actions.ts
    ├── store/auth.slice.ts
    ├── hooks/use-auth.ts
    └── components/ LoginForm, AuthHydrator
lib/session.ts   lib/dal.ts
store/store.ts   (+ authReducer)
proxy.ts         (en la raíz)
```

El mock simula al backend
Devuelve un JWT firmado con un secreto de prueba.

Con backend real, otro datasource
auth.api.datasource.ts ocupa su lugar y el dominio no se entera.

DI: se arma la feature

estructura de la feature auth

```
features/auth/
├── domain/
│   ├── entities/auth-tokens.ts
│   ├── repositories/auth.repository.ts
│   └── usecases/login.usecase.ts
├── infrastructure/
│   ├── auth.mock.datasource.ts
│   ├── auth.api.datasource.ts
│   └── auth.repository.impl.ts
├── di/auth.container.ts
└── presentation/
    ├── actions/auth.actions.ts
    ├── store/auth.slice.ts
    ├── hooks/use-auth.ts
    └── components/ LoginForm, AuthHydrator
lib/session.ts   lib/dal.ts
store/store.ts   (+ authReducer)
proxy.ts         (en la raíz)
```

El Composition Root
Crea el caso de uso con su repositorio y datasource.

Un cambio de una línea
Pasar del mock al backend real se hace solo aquí.

Presentación: formulario, hook, thunk y acción

estructura de la feature auth

```
features/auth/
├── domain/
│   ├── entities/auth-tokens.ts
│   ├── repositories/auth.repository.ts
│   └── usecases/login.usecase.ts
├── infrastructure/
│   ├── auth.mock.datasource.ts
│   ├── auth.api.datasource.ts
│   └── auth.repository.impl.ts
├── di/auth.container.ts
└── presentation/
    ├── actions/auth.actions.ts
    ├── store/auth.slice.ts
    ├── hooks/use-auth.ts
    └── components/ LoginForm, AuthHydrator
lib/session.ts   lib/dal.ts
store/store.ts   (+ authReducer)
proxy.ts         (en la raíz)
```

La cadena de la semana 12

El formulario llama al hook, el hook despacha el thunk y el thunk invoca la acción.

La acción corre en el servidor

Pide el token, lo guarda en la cookie y devuelve el usuario.

lib/: la sesión es transversal

estructura de la feature auth

```
features/auth/
├── domain/
│   ├── entities/auth-tokens.ts
│   ├── repositories/auth.repository.ts
│   └── usecases/login.usecase.ts
├── infrastructure/
│   ├── auth.mock.datasource.ts
│   ├── auth.api.datasource.ts
│   └── auth.repository.impl.ts
├── di/auth.container.ts
└── presentation/
    ├── actions/auth.actions.ts
    ├── store/auth.slice.ts
    ├── hooks/use-auth.ts
    └── components/ LoginForm, AuthHydrator
lib/session.ts   lib/dal.ts
store/store.ts   (+ authReducer)
proxy.ts         (en la raíz)
```

session.ts
Decodifica el token y escribe la cookie.

dal.ts
Entrega la sesión junto al dato. Como src/store/, lo usan todas las features.

proxy.ts y el reducer

estructura de la feature auth

```
features/auth/
├── domain/
│   ├── entities/auth-tokens.ts
│   ├── repositories/auth.repository.ts
│   └── usecases/login.usecase.ts
├── infrastructure/
│   ├── auth.mock.datasource.ts
│   ├── auth.api.datasource.ts
│   └── auth.repository.impl.ts
├── di/auth.container.ts
└── presentation/
    ├── actions/auth.actions.ts
    ├── store/auth.slice.ts
    ├── hooks/use-auth.ts
    └── components/ LoginForm, AuthHydrator
lib/session.ts   lib/dal.ts
store/store.ts   (+ authReducer)
proxy.ts         (en la raíz)
```

proxy.ts: la primera barrera
Redirige a quien no tiene sesión, antes de renderizar.

store.ts
Solo se registra el reducer de auth.

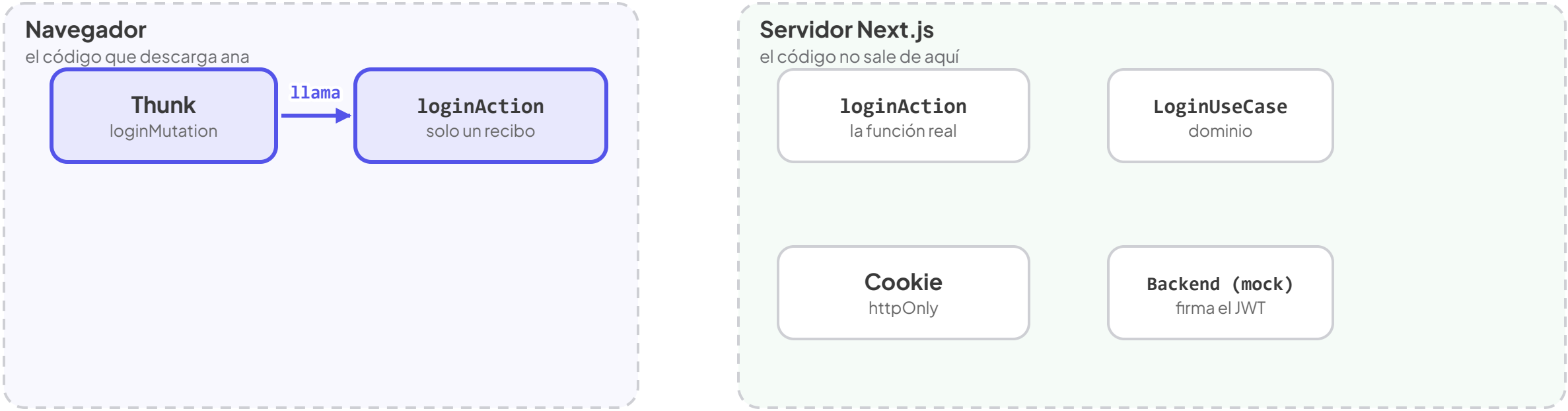
Universidad ICESI

4. use server

Server Actions: qué son y cómo se usan

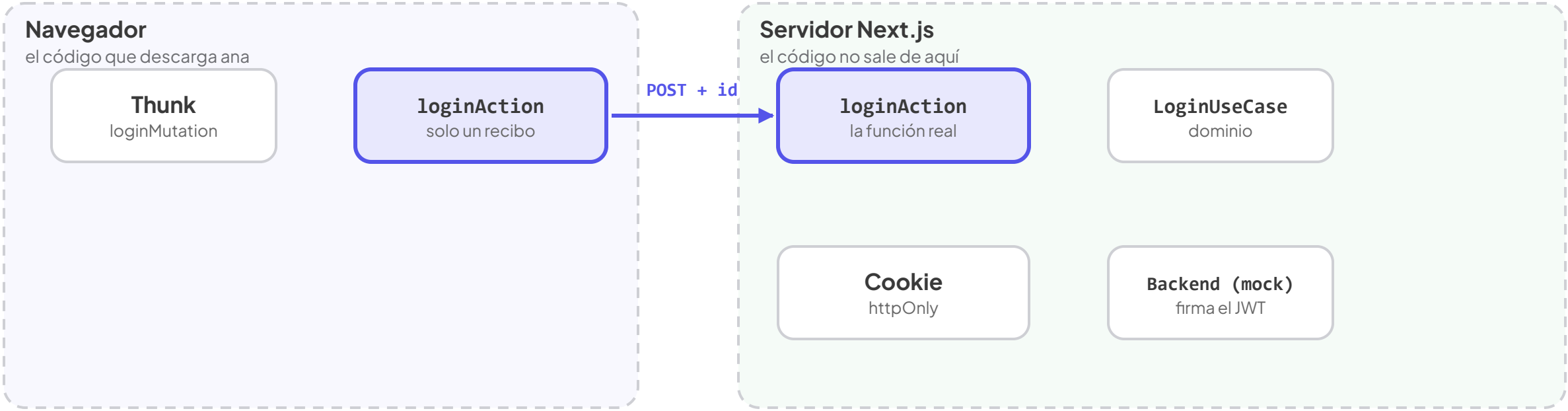
Universidad icesi

El thunk llama a loginAction



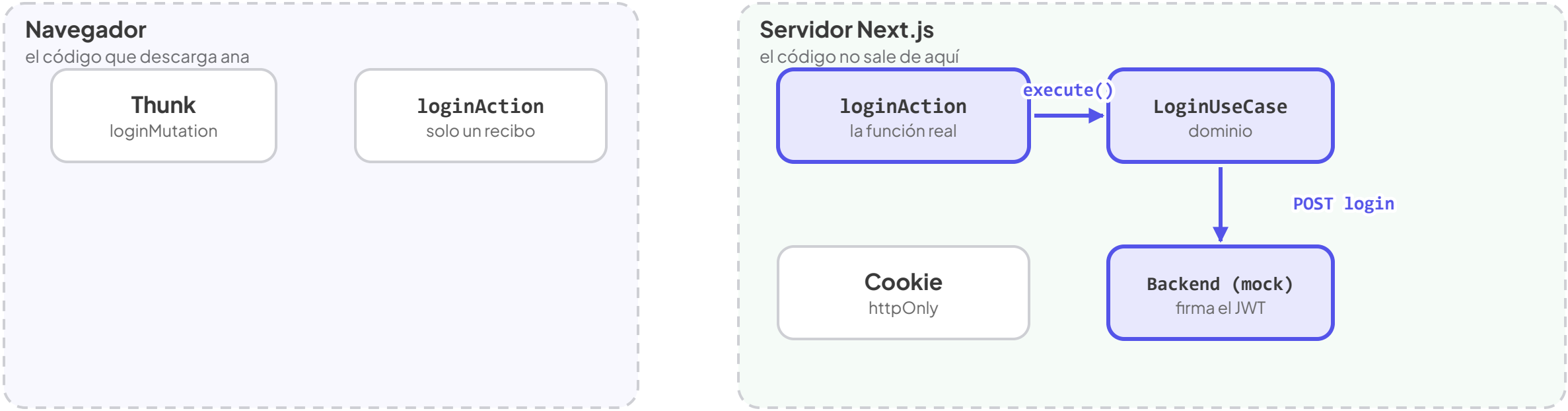
Para el thunk es una llamada normal: `await loginAction(email, password)`. Pero en el navegador esa función es un **recibo con su id**: el código real no se descargó.

El recibo hace un POST



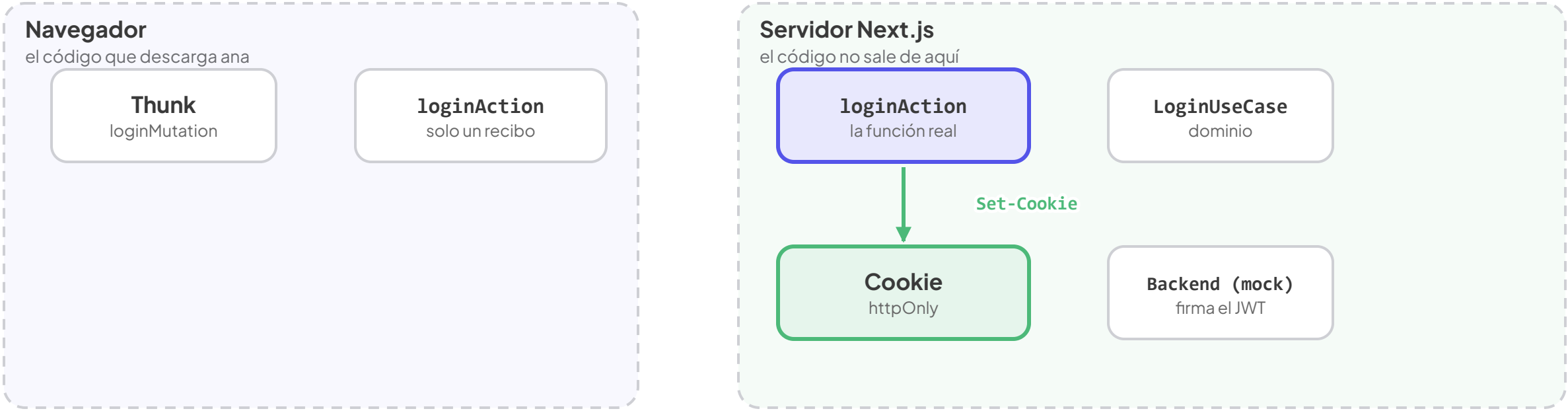
Llamar al recibo envía un **POST** con el encabezado `Next-Action` y los argumentos. Es lo que escribe use server: tú no creas el endpoint ni el fetch.

En el servidor corre la función real



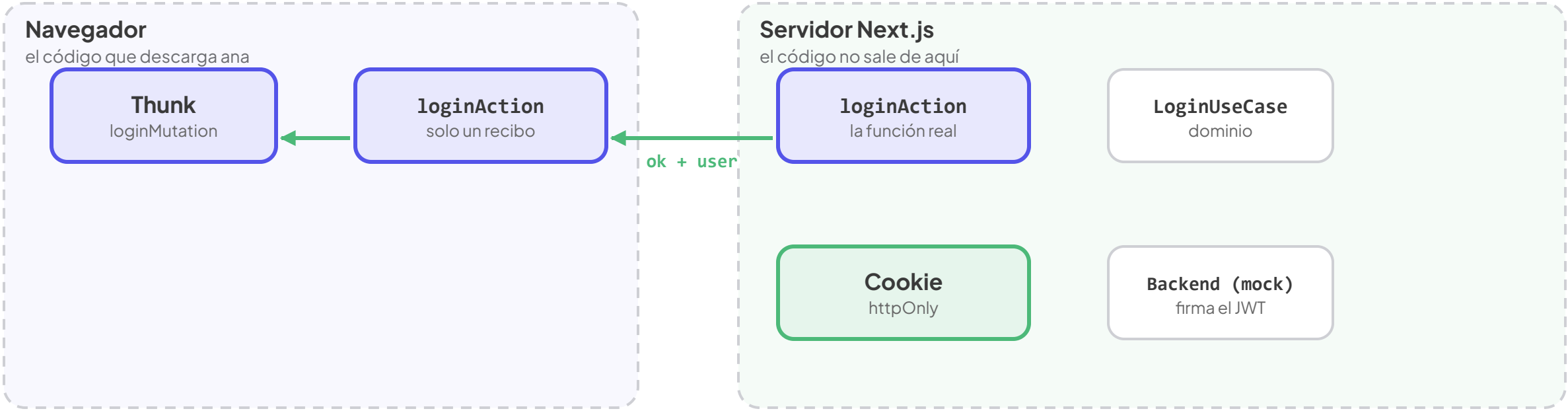
Next.js encuentra la función por su id y la ejecuta. Allí sí existen el caso de uso y el datasource mock, con los usuarios y la clave con que se firma el JWT.

Solo el servidor escribe la cookie



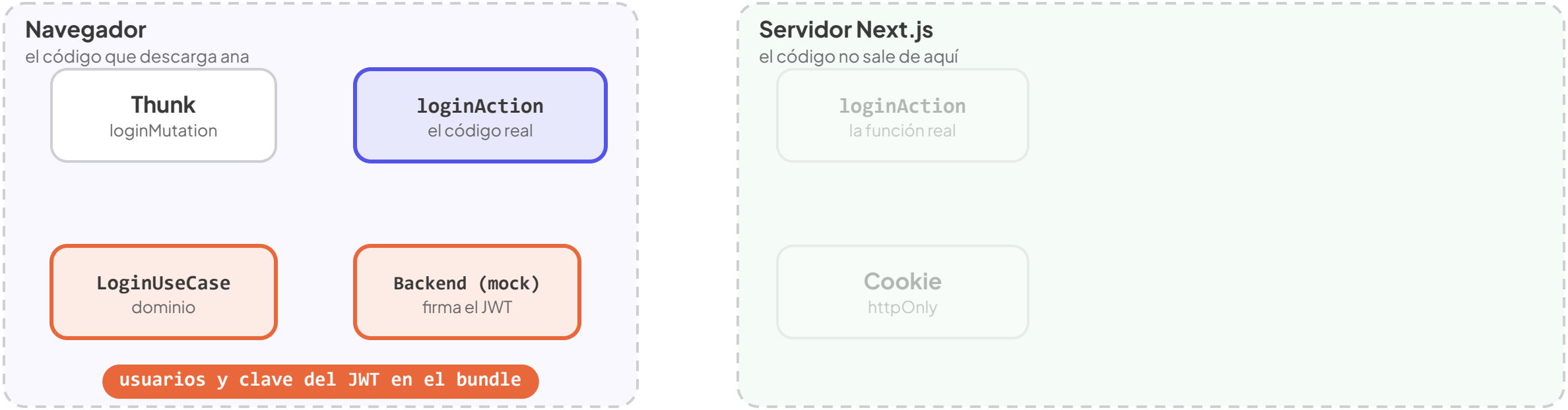
Una cookie **httpOnly** solo se puede crear desde el servidor, con el encabezado `Set-Cookie` en la respuesta. Por eso la acción corre allí.

La respuesta vuelve al thunk



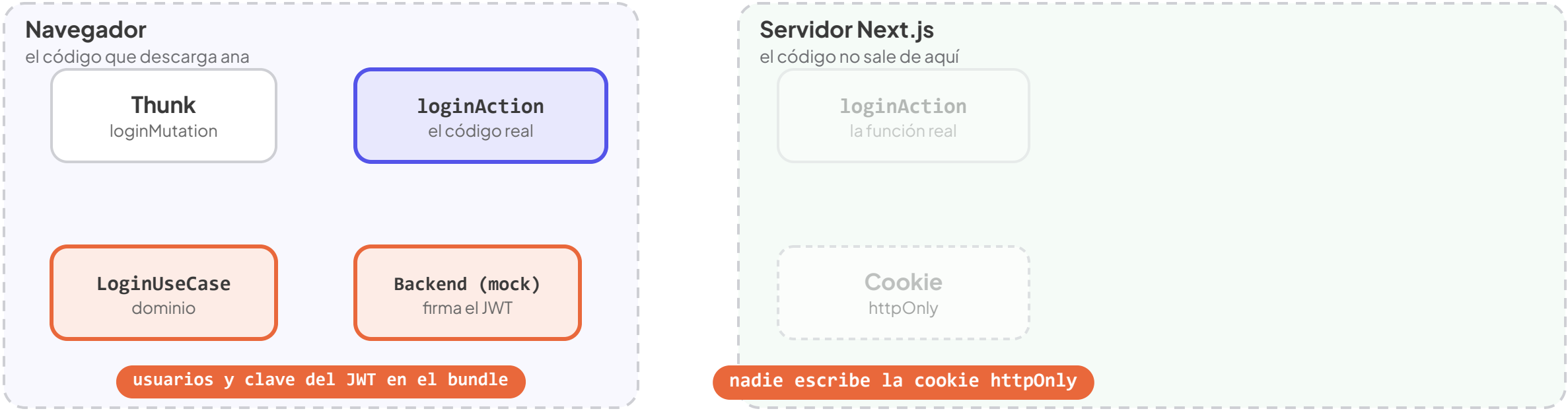
El navegador guarda la cookie por su cuenta y el recibo termina su promesa. El thunk sigue **como si hubiera llamado una función local**.

Sin use server, el código viaja al navegador



Sin la directiva, el archivo es un módulo normal: el bundler lo incluye en el JavaScript del navegador con todo lo que importa. **El caso de uso y el mock se descargan** (y server-only haría fallar el build).

Y nadie puede escribir la cookie



JavaScript del navegador no puede crear una cookie httpOnly: la sesión no se podría guardar de forma segura. **Por eso la acción corre en el servidor.**

Cómo se escribe: al inicio del archivo

app/actions.ts

```ts
'use server'

export async function saludar(nombre: string) {
  return `Hola, ${nombre}`
}
```

Todas las funciones exportadas
Pasan a ser Server Actions.

Un componente de cliente no puede definirlas
Las importa desde un archivo con use server.

Cómo se escribe: dentro de una función

app/page.tsx

```tsx
export default function Page() {
  async function crear(formData: FormData) {
    'use server'
    // se ejecuta en el servidor
  }

  return <form action={crear}>{/* campos */}</form>
}
```

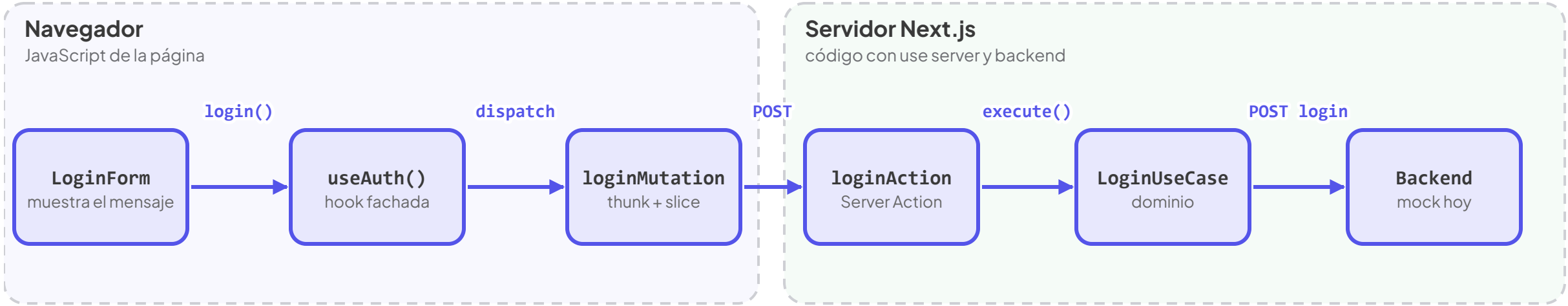
Solo en un Server Component
La forma normal en este curso es al inicio del archivo.

Universidad ICESI

5. Implementar el login

De la vista al caso de uso

La cadena del login



La cadena de la semana 12 (vista, hook, thunk) más una **frontera con el servidor**. La vista no conoce Redux ni la acción: el hook es la fachada, igual que useRoutines.

La directiva 'use server'

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

Funciones del servidor
Todas las funciones exportadas del archivo pasan a ser Server Actions.

Quién la llama
LoginForm → useAuth() → loginMutation → loginAction.

Lo que recibe: argumentos simples

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

No un FormData
La llama el thunk, no un formulario: recibe el correo y la contraseña.

No es de confianza
Todo lo que recibe viene del cliente.

Validar en el servidor, siempre

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

safeParse no lanza excepciones
Devuelve { success, data | error }: no hay try/catch.

La validación del formulario es comodidad
Una acción es accesible con un POST directo.

Error esperado: datos inválidos

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

Se devuelve, no se lanza
La acción responde { ok: false, message }.

El camino hasta la pantalla
El thunk lo convierte en rejectWithValue y el slice lo guarda en errorMessage.

Delegar en el caso de uso

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

La acción es un adaptador

No decide si las credenciales son válidas: LoginUseCase le pide el token al backend.

Error esperado: credenciales inválidas

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

Un mensaje genérico
Sin decir si falló el correo o la contraseña.

Por qué
Decir «ese correo no existe» le dice a un atacante qué correos están registrados.

Éxito: el token a la cookie y el usuario al cliente

auth.actions.ts

```ts
'use server'
const schema = z.object({
  email: z.string().email(),
  password: z.string().min(6),
})
export async function loginAction(email: string, password: string) {
  const parsed = schema.safeParse({ email, password })
  if (!parsed.success) {
    return { ok: false, message: parsed.error.issues[0].message }
  }
  const tokens = await authContainer.login.execute(email, password)
  if (!tokens) {
    return { ok: false, message: 'Credenciales inválidas' }
  }
  const payload = await createSession(tokens.accessToken)
  return {
    ok: true,
    user: { id: payload.sub, name: payload.name, roles: payload.roles },
  }
}
```

createSession decodifica y guarda
El token se queda en la cookie httpOnly.

El cliente recibe id, nombre y roles
Nunca el token ni una contraseña.

El slice: el estado de la sesión

auth.slice.ts

```ts
interface AuthState {
  user: AuthUser | null    //{id, name, roles}
  isLoading: boolean
  errorMessage: string | null
}

export const loginMutation = createAsyncThunk(
  'auth/login',
  async (p: { email: string; password: string }, { rejectWithValue }) => {
    try {
      const result = await loginAction(p.email, p.password)
      if (!result.ok) return rejectWithValue(result.message)
      return result.user
    } catch {
      return rejectWithValue('No se pudo iniciar sesión.')
    }
  }
)
```

user guarda id, nombre y roles
Lo que la interfaz necesita saber de la sesión.

isLoading y errorMessage
El estado del login, igual que routines.slice.ts.

El thunk: el puente al servidor

auth.slice.ts

```ts
interface AuthState {
  user: AuthUser | null    //{id, name, roles}
  isLoading: boolean
  errorMessage: string | null
}

export const loginMutation = createAsyncThunk(
  'auth/login',
  async (p: { email: string; password: string }, { rejectWithValue }) => {
    try {
      const result = await loginAction(p.email, p.password)
      if (!result.ok) return rejectWithValue(result.message)
      return result.user
    } catch {
      return rejectWithValue('No se pudo iniciar sesión.')
    }
  }
)
```

Esperado: ok es false
rejectWithValue(result.message): el mensaje exacto de la acción.

Inesperado: catch
La red o el servidor fallaron: un mensaje genérico propio.

Los reducers del slice

auth.slice.ts

```ts
reducers: {
  setUser: (state, action: PayloadAction<AuthUser | null>) => {
    state.user = action.payload
  },
},
extraReducers: (builder) => {
  builder
    .addCase(loginMutation.pending, (s) => {
      s.isLoading = true
      s.errorMessage = null
    })
    .addCase(loginMutation.fulfilled, (s, a) => {
      s.isLoading = false
      s.user = a.payload
    })
    .addCase(loginMutation.rejected, (s, a) => {
      s.isLoading = false
      s.errorMessage = a.payload as string
    })
},
```

fulfilled: guarda el usuario
Tras un login correcto, state.user queda con sus roles.

setUser: para la recarga
Lo usa AuthHydrator al cargar la página.

Registra `auth: authReducer` en makeStore, junto a `routines`.

El hook fachada: useAuth()

use-auth.ts

```ts
export function useAuth() {
  const dispatch = useAppDispatch()
  const { user, isLoading, errorMessage } =
    useAppSelector((state) => state.auth)

  const login = async (email: string, password: string) => {
    const action = await dispatch(loginMutation({ email, password }))
    return loginMutation.fulfilled.match(action)
  }

  const hasRole = (role: string) => !!user?.roles.includes(role)

  return { user, isLoading, errorMessage, login, hasRole }
}
```

login() devuelve true o false
Así el componente decide si navegar, sin conocer Redux.

hasRole(role)
Responde si el usuario tiene ese rol: para la interfaz.

Lo único que conocen los componentes
Si cambia el estado o la acción, no se enteran.

LoginForm: conectado al hook

LoginForm.tsx

```tsx
export function LoginForm() {
  const router = useRouter()
  const { login, isLoading, errorMessage } = useAuth()
  const [email, setEmail] = useState('')
  const [password, setPassword] = useState('')

  async function onSubmit(e: FormEvent) {
    e.preventDefault()
    const ok = await login(email, password)
    if (ok) router.replace('/routines')
  }

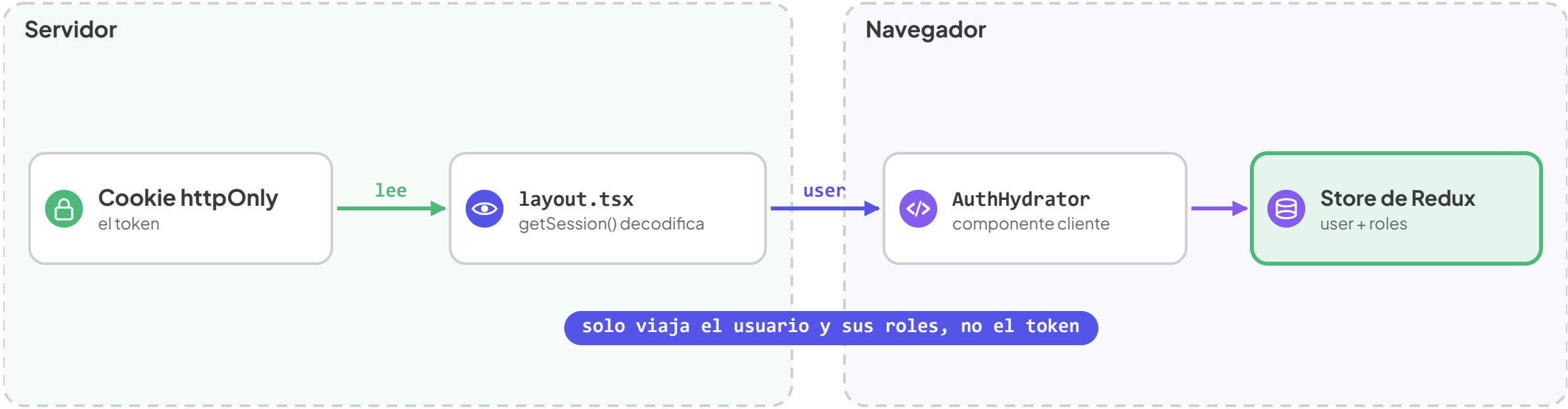
  //...
  {errorMessage && <p role="alert">{errorMessage}</p>}
  <button disabled={isLoading}>Iniciar sesión</button>
}
```

Solo conoce useAuth
No sabe que existe Redux, una Server Action ni una cookie.

La redirección la hace el cliente
router.replace cuando login() devuelve true.

Campos con label
Accesibles, y las pruebas los encuentran por su etiqueta.

Al recargar, el store nace vacío



El cliente no puede decodificar el token: está en una cookie **httpOnly**. Lo hace el servidor (en el layout) y le entrega a la interfaz solo el usuario y sus roles.

AuthHydrator y el layout raíz

app/layout.tsx

```tsx
export default async function RootLayout({children}) {
  const session = await getSession()
  const user = session && {
    id: session.userId,
    name: session.name,
    roles: session.roles,
  }
  return (
    <StoreProvider>
      <AuthHydrator user={user}/>
      {children}
    </StoreProvider>
  )
}
```

AuthHydrator.tsx

```tsx
'use client'

export function AuthHydrator({user}) {
  const dispatch = useAppDispatch()

  useEffect(() => {
    dispatch(setUser(user))
  }, [dispatch, user])

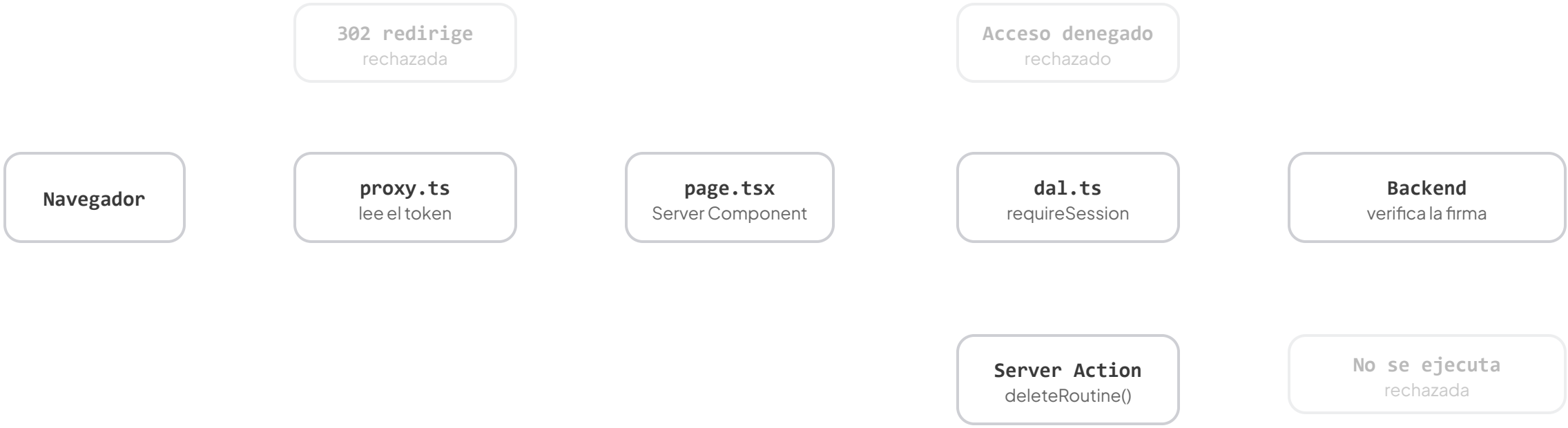
  return null
}
```

> Esto **no protege nada**: solo llena la interfaz.

6. Proteger las rutas

Tres puertas y los roles en la interfaz

Una sola puerta no alcanza



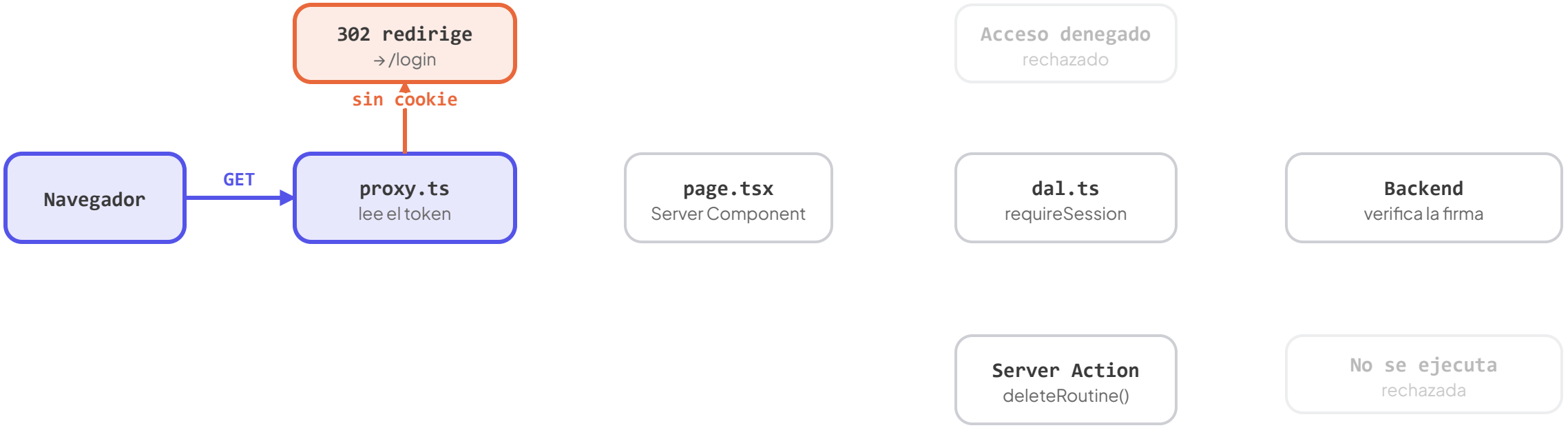
Proteger una ruta es decidir, **en cada petición**, si el usuario puede ver ese contenido. Hay tres puertas: el **proxy**, el **DAL** junto al dato y cada **Server Action**.

Qué comprueba cada capa, y qué no puede hacer

Capa	Qué comprueba	Qué NO puede hacer
proxy.ts	Token decodificable y vigente; puede mirar los roles. Redirige rutas enteras	No verifica la firma ni sabe si el dato es del usuario
El DAL (requireSession)	Lo mismo, junto al dato, y entrega el token para el backend	No decide permisos: lo hace el backend
Cada Server Action	Que haya una sesión, por su cuenta	No hereda lo que comprueba la página

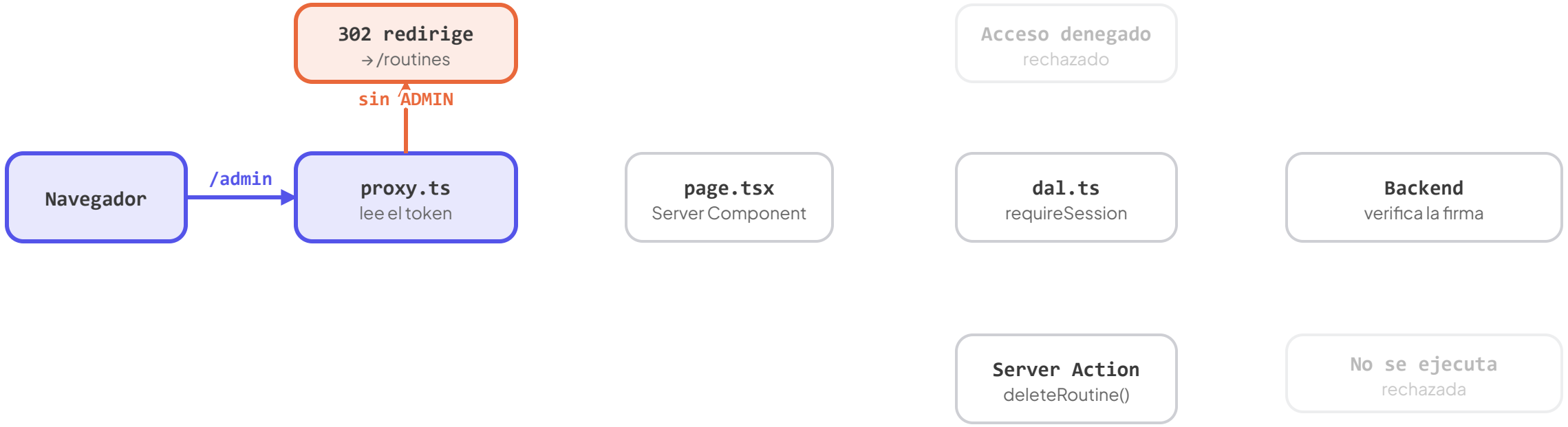
> El front decodifica el token y decide rápido, sin peticiones. **El backend verifica la firma y decide los permisos.**

Sin cookie: la detiene el proxy



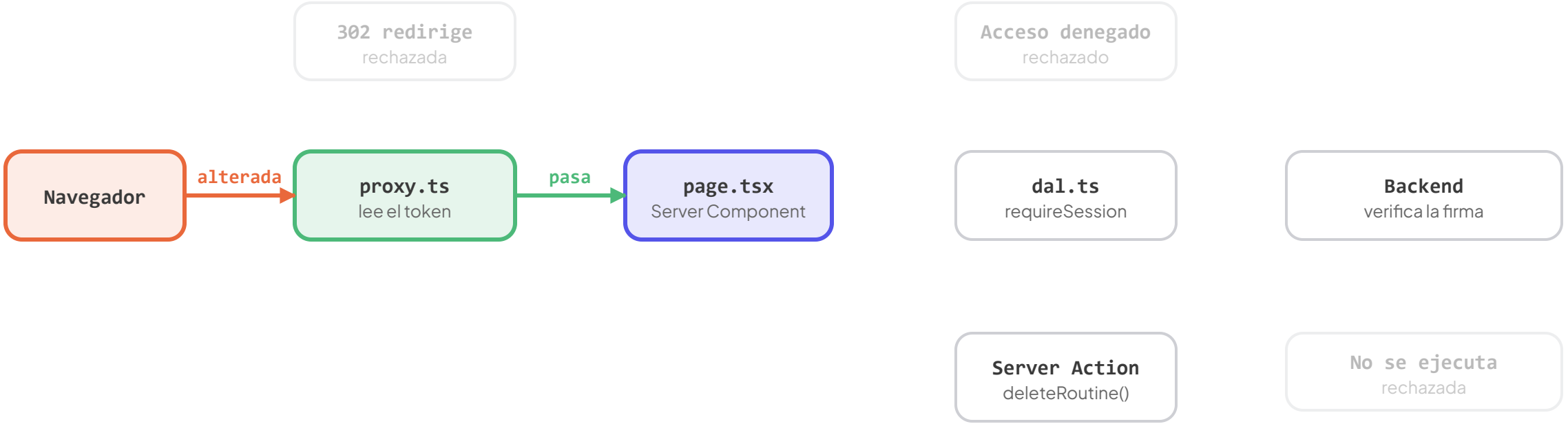
Otra persona escribe la URL sin sesión. El proxy redirige a /login **sin renderizar la página ni llamar al backend**. Es barato y rápido.

Rol insuficiente: ana pide /admin



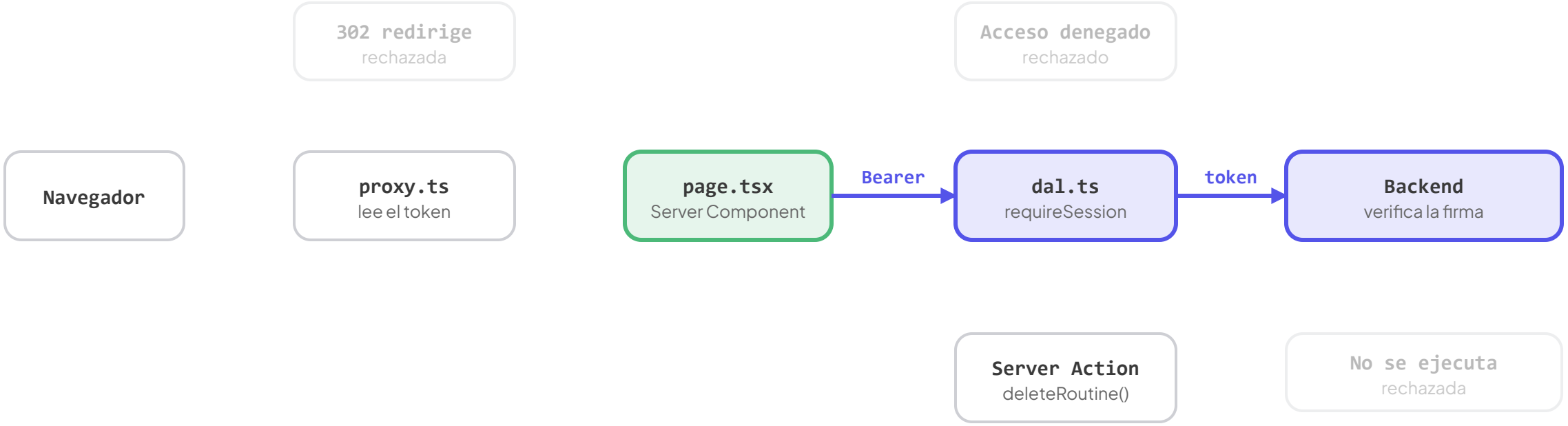
El proxy decodifica el token, ve `roles: [USER]` y redirige. Es una mejora de experiencia: **no verifica nada**, y el backend volverá a comprobar el rol.

Cookie alterada (1 de 3): llega al proxy



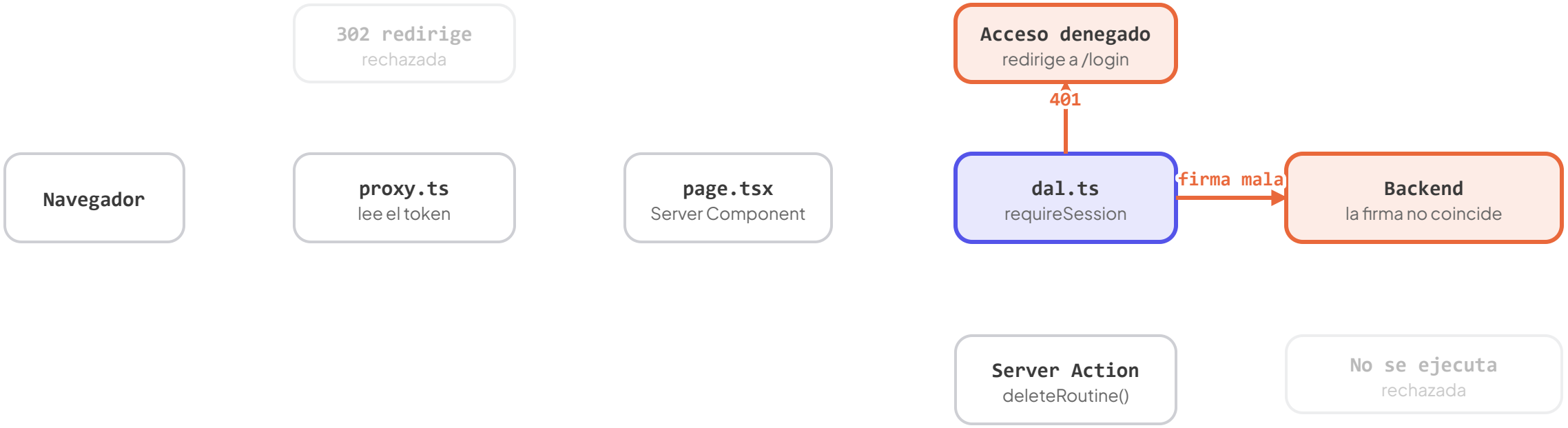
Alguien cambió el contenido del token (por ejemplo, `roles: [ADMIN]`). El front **no verifica la firma**: ve un token bien formado y vigente, y lo deja pasar.

Cookie alterada (2 de 3): la página llama al backend



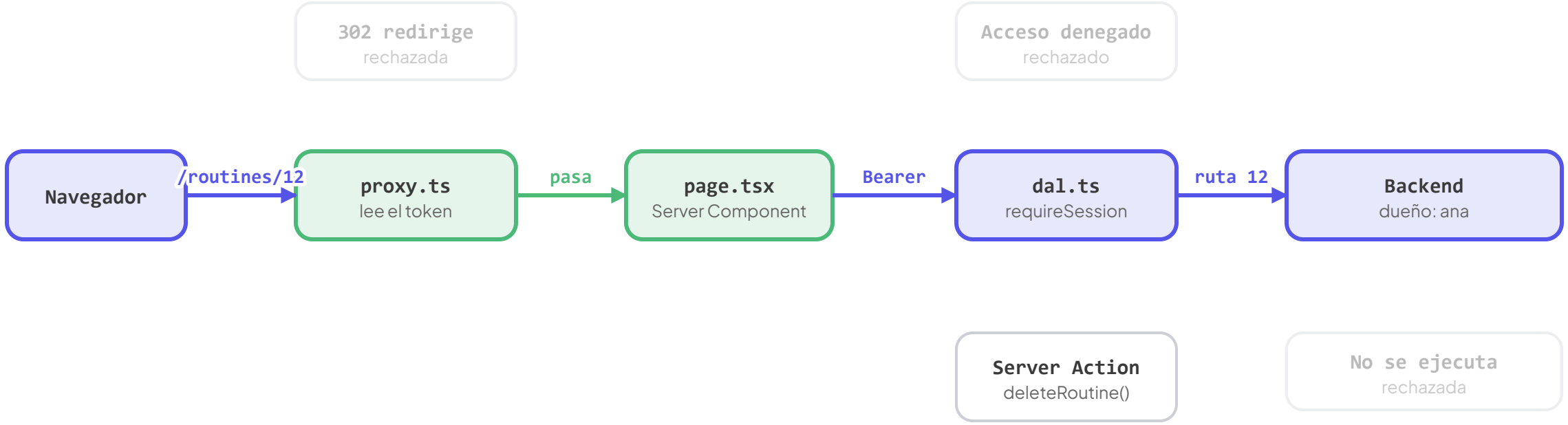
La página llama a requireSession() y le pide los datos al backend **con el token**.

Cookie alterada (3 de 3): el backend la rechaza



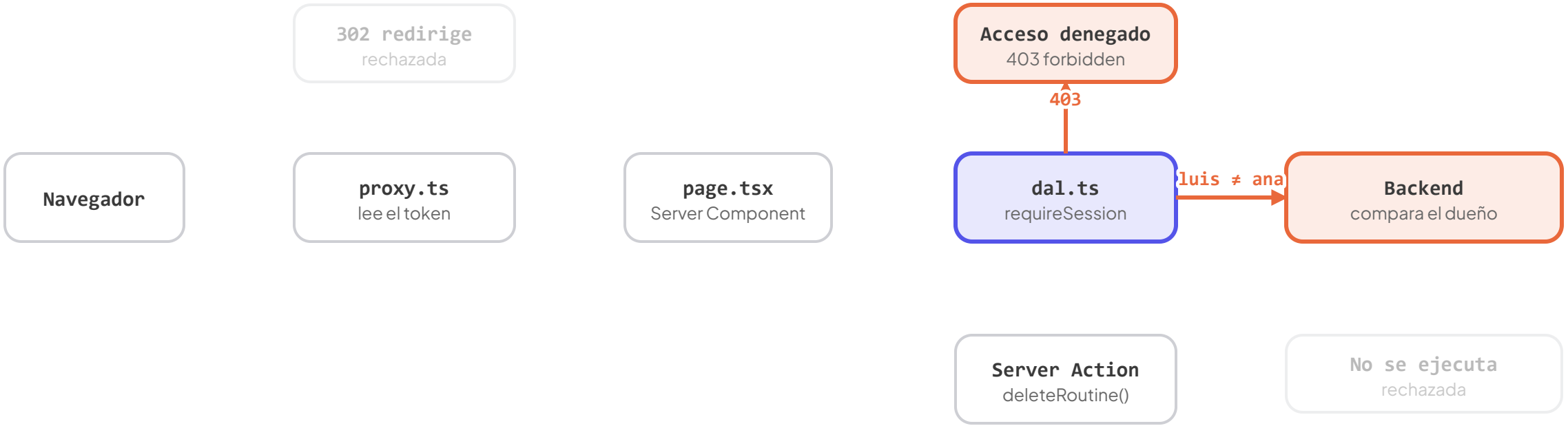
Aquí sí se verifica: la firma ya no coincide con el contenido y el backend rechaza el token. El token falso no consiguió ni un dato.

Rutina de otro usuario (1 de 2)



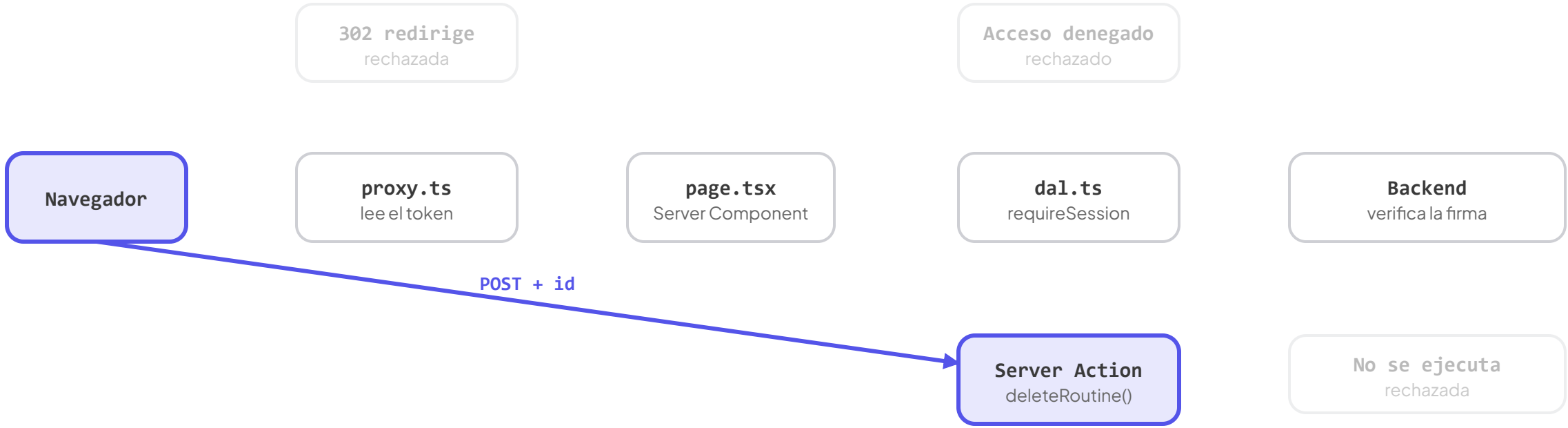
Luis tiene una sesión válida y pasa el proxy. Cambia el id de la URL a la rutina de ana: la autenticación está bien, **la autorización no**.

Rutina de otro usuario (2 de 2)



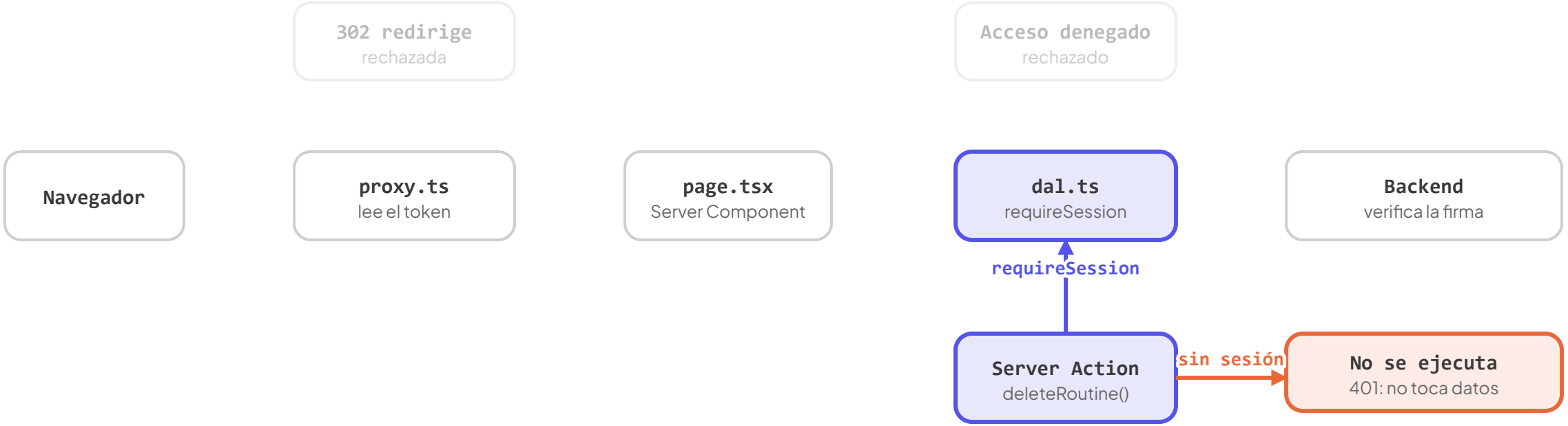
El backend compara el dueño de la rutina con el id del token y responde 403. Esa regla vive **junto al dato**, no en el proxy.

POST directo a la acción (1 de 2)



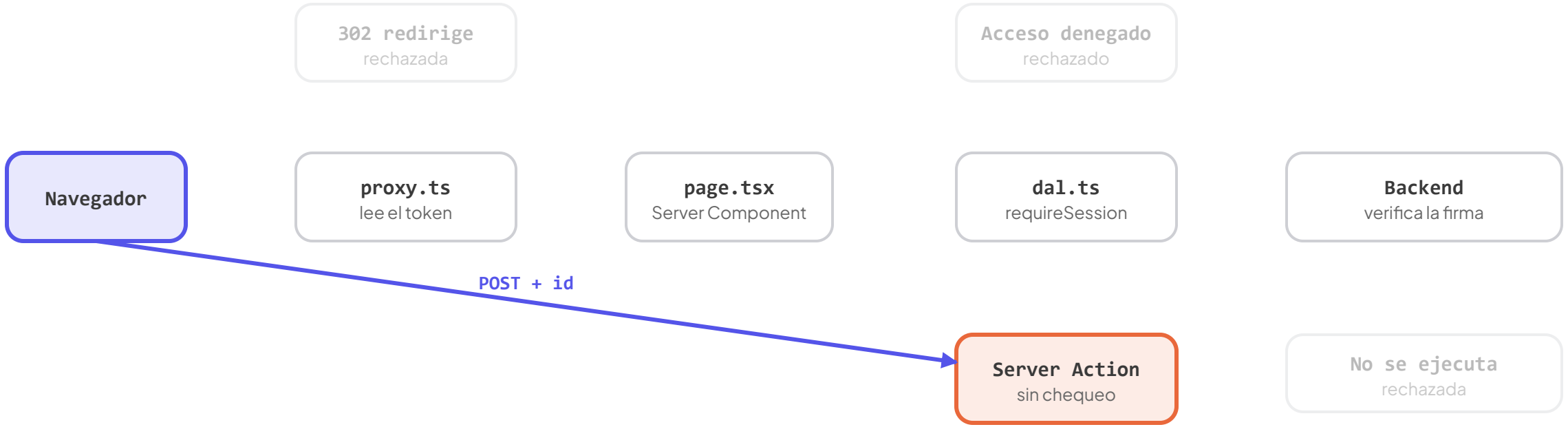
Alguien con curl envía el id de la acción desde la consola. La página no se renderiza: **nada de lo que comprobaste en ella corre**.

POST directo a la acción (2 de 2)



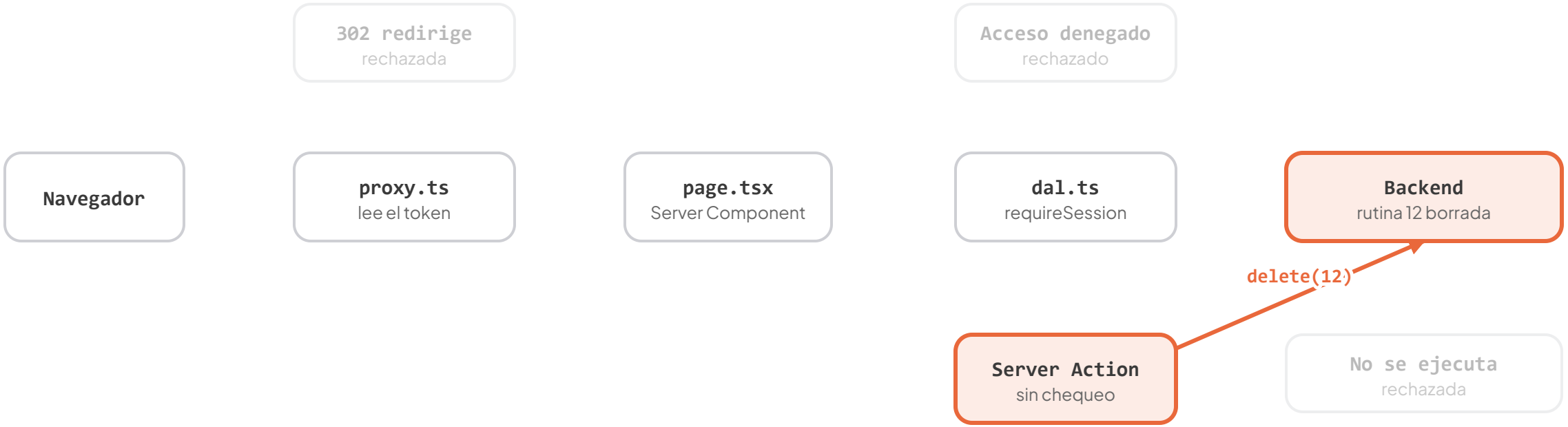
La primera línea de la acción exige una sesión. Cada acción es su propia puerta: **se detiene antes de tocar los datos**.

Acción sin chequeo (1 de 2)



Alguien pensó «la página ya está protegida». Pero la acción es un endpoint propio, y el proxy solo cubre las rutas de su matcher.

Acción sin chequeo (2 de 2)



Sin sesión ni comprobación del dueño, la acción ejecuta. Es **la brecha más común** al empezar con Server Actions.

El DAL: getSession

src/lib/dal.ts

```ts
export async function getSession() {
  const token = (await cookies()).get('session')?.value
  const payload = decodeToken(token)
  if (!payload) return null

  return {
    userId: payload.sub,
    name: payload.name,
    roles: payload.roles,
    accessToken: token!,
  }
}

export async function requireSession() {
  const session = await getSession()
  if (!session) redirect('/login')
  return session
}
```

Decodifica la cookie
Devuelve id, nombre, roles y el token, o null si no hay sesión vigente.

Entrega el token
Para llamar al backend con Authorization: Bearer.

El DAL: requireSession

src/lib/dal.ts

```ts
export async function getSession() {
  const token = (await cookies()).get('session')?.value
  const payload = decodeToken(token)
  if (!payload) return null

  return {
    userId: payload.sub,
    name: payload.name,
    roles: payload.roles,
    accessToken: token!,
  }
}

export async function requireSession() {
  const session = await getSession()
  if (!session) redirect('/login')
  return session
}
```

Exige una sesión
Si no hay, redirige a /login.

Se llama antes de leer o cambiar datos
En las páginas y en cada Server Action.

Cada Server Action exige su sesión

routines.actions.ts

```ts
export async function deleteRoutineAction(routineId: string) {
  // 1. ¿Hay sesión? (si no, redirige a /login)
  const { accessToken } = await requireSession()

  // 2. Autorización: la decide el backend
  const result = await routineContainer.deleteRoutine.execute(
    routineId,
    accessToken,
  )
  if (!result.ok) {
    return { ok: false, message: 'No tienes permiso' }
  }
  return { ok: true }
}
```

Autenticación: ¿quién eres?
El front lo supone leyendo el token.

Es un endpoint público
Nadie debe poder borrar una rutina sin sesión.

La autorización la decide el backend

routines.actions.ts

```ts
export async function deleteRoutineAction(routineId: string) {
  // 1. ¿Hay sesión? (si no, redirige a /login)
  const { accessToken } = await requireSession()

  // 2. Autorización: la decide el backend
  const result = await routineContainer.deleteRoutine.execute(
    routineId,
    accessToken,
  )
  if (!result.ok) {
    return { ok: false, message: 'No tienes permiso' }
  }
  return { ok: true }
}
```

Recibe el token
Verifica la firma y compara el dueño de la rutina con el id del token.

403 si no es suya
El datasource lo traduce en { ok: false }.

proxy.ts: sin sesión, al login

proxy.ts

```ts
export function proxy(req: NextRequest) {
  const path = req.nextUrl.pathname
  const payload = decodeToken(req.cookies.get('session')?.value)
  const isProtected = ['/routines', '/admin'].some((r) => path.startsWith(r))

  if (isProtected && !payload) {
    return NextResponse.redirect(new URL('/login', req.nextUrl))
  }
  // Solo experiencia de usuario: el backend vuelve a comprobar el rol
  if (path.startsWith('/admin') && !payload?.roles.includes('ADMIN')) {
    return NextResponse.redirect(new URL('/routines', req.nextUrl))
  }
  if (path === '/login' && payload) {
    return NextResponse.redirect(new URL('/routines', req.nextUrl))
  }
  return NextResponse.next()
}
```

Decodifica la cookie
Sin llamar al backend y sin verificar la firma.

Ruta protegida sin sesión
Redirige a /login antes de renderizar.

proxy.ts: los roles

proxy.ts

```ts
export function proxy(req: NextRequest) {
  const path = req.nextUrl.pathname
  const payload = decodeToken(req.cookies.get('session')?.value)
  const isProtected = ['/routines', '/admin'].some((r) => path.startsWith(r))

  if (isProtected && !payload) {
    return NextResponse.redirect(new URL('/login', req.nextUrl))
  }
  // Solo experiencia de usuario: el backend vuelve a comprobar el rol
  if (path.startsWith('/admin') && !payload?.roles.includes('ADMIN')) {
    return NextResponse.redirect(new URL('/routines', req.nextUrl))
  }
  if (path === '/login' && payload) {
    return NextResponse.redirect(new URL('/routines', req.nextUrl))
  }
  return NextResponse.next()
}
```

/admin pide el rol ADMIN
Si el token no lo trae, redirige a /routines.

Solo experiencia de usuario
Un token alterado lo pasaría: el backend vuelve a comprobar.

proxy.ts: ya tienes sesión

proxy.ts

```ts
export function proxy(req: NextRequest) {
  const path = req.nextUrl.pathname
  const payload = decodeToken(req.cookies.get('session')?.value)
  const isProtected = ['/routines', '/admin'].some((r) => path.startsWith(r))

  if (isProtected && !payload) {
    return NextResponse.redirect(new URL('/login', req.nextUrl))
  }
  // Solo experiencia de usuario: el backend vuelve a comprobar el rol
  if (path.startsWith('/admin') && !payload?.roles.includes('ADMIN')) {
    return NextResponse.redirect(new URL('/routines', req.nextUrl))
  }
  if (path === '/login' && payload) {
    return NextResponse.redirect(new URL('/routines', req.nextUrl))
  }
  return NextResponse.next()
}
```

No vuelvas a /login
Con sesión, /login redirige a /routines.

Next.js 16 renombró `middleware.ts` a `proxy.ts`. En Next.js 15 o anterior el archivo se llama middleware.ts.

Usar los roles en la interfaz

Header.tsx

```tsx
export function Header() {
  const { user, hasRole } = useAuth()

  return (
    <nav>
      <Link href="/routines">Mis rutinas</Link>
      {hasRole('ADMIN') && <Link href="/admin">Administración</Link>}
      {user && <span>Hola, {user.name}</span>}
    </nav>
  )
}
```

Sin pedir nada al backend
El usuario y sus roles ya están en el estado.

Ocultar no es proteger
Cualquiera puede escribir /admin: el backend comprueba el rol.

No decidas la sesión solo en el layout

Por el renderizado parcial, un layout **no se vuelve a renderizar** en cada navegación entre rutas hijas: no vería que la sesión cambió.

El layout solo **hidrata la interfaz**. La sesión se exige en cada página, o mejor, junto al dato, con `requireSession()`.

Los datos del token pueden estar desactualizados hasta que venza. Si necesitas el dato **fresco**, pregúntale al backend (GET /auth/me).

Resumen: sesiones y rutas protegidas

Sesión

Cookie httpOnly con el token del backend: persiste y JavaScript no la lee.

Token

El backend firma y verifica; el front **solo decodifica** con jwt-decode.

Arquitectura

El dominio no sabe de cookies; session.ts y dal.ts van en lib.

use server

La acción corre en el servidor: escribe la cookie y es un endpoint público.

Estado

Usuario y roles en Redux, hidratados desde el layout al recargar.

Proteger

Proxy, DAL y cada acción; el **backend decide** los permisos.

Universidad ICESI

Manejo de errores

Qué hacer cuando algo sale mal

Dos clases de error, y una tercera

ESPERADOS

Fallas **previstas**: validación de un formulario, credenciales incorrectas, un recurso que no existe.

Se **devuelven como valor** y se muestran en la interfaz.

INESPERADOS

Bugs o fallas externas: una excepción, un backend caído, la red.

Se **lanzan** y los atrapa un error boundary o un try/catch.

SE LANZAN A PROPÓSITO

redirect, notFound, unauthorized y forbidden: **control de flujo**.

Next.js los convierte en una redirección o en una página especial.

Qué mecanismo corresponde a cada situación

Situación	Mecanismo	Resultado
Credenciales incorrectas	return { ok: false, message }	rejectWithValue y errorMessage en el formulario
Un bug al renderizar una página	throw	error.tsx del segmento, con retry()
Falla de una acción desde un thunk	try/catch en el thunk	errorMessage en el store
Sin sesión	redirect('/login')	Navega al login
El recurso no existe	notFound()	not-found.tsx, estado 404
Falla en un onClick	try/catch y useState	Un boundary no la atrapa

Errores esperados: devuélvelos como valor

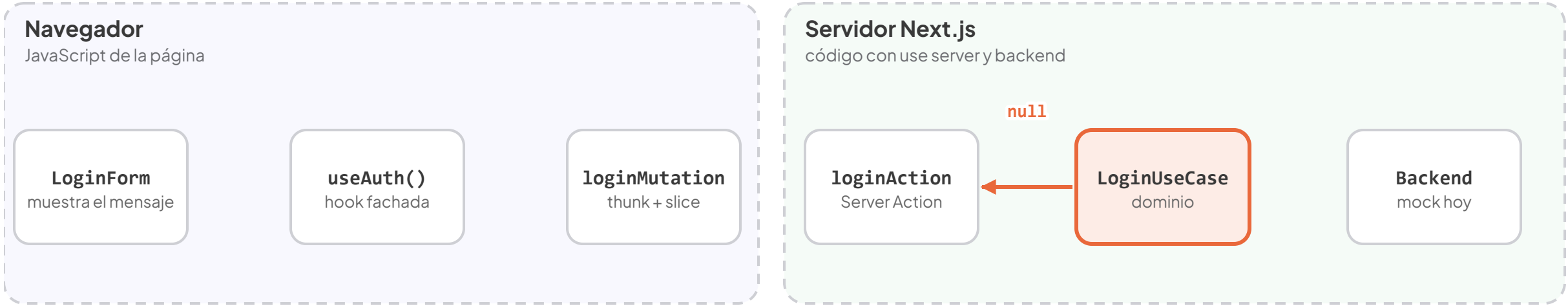
auth.actions.ts (fragmento)

```ts
const tokens = await authContainer.login.execute(email, password)
if (!tokens) {
  return { ok: false, message: 'Credenciales inválidas' }
}
```

Un throw perdería el mensaje
En producción Next.js oculta el texto de un error lanzado en el servidor.

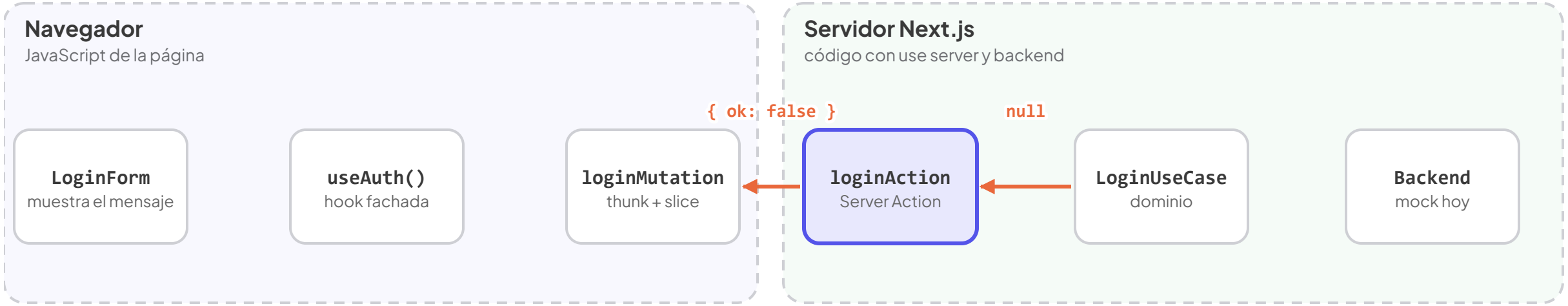
Es el flujo normal
El usuario se equivocó de contraseña: ve el mensaje y reintenta.

Credenciales inválidas (1 de 3): el caso de uso responde null



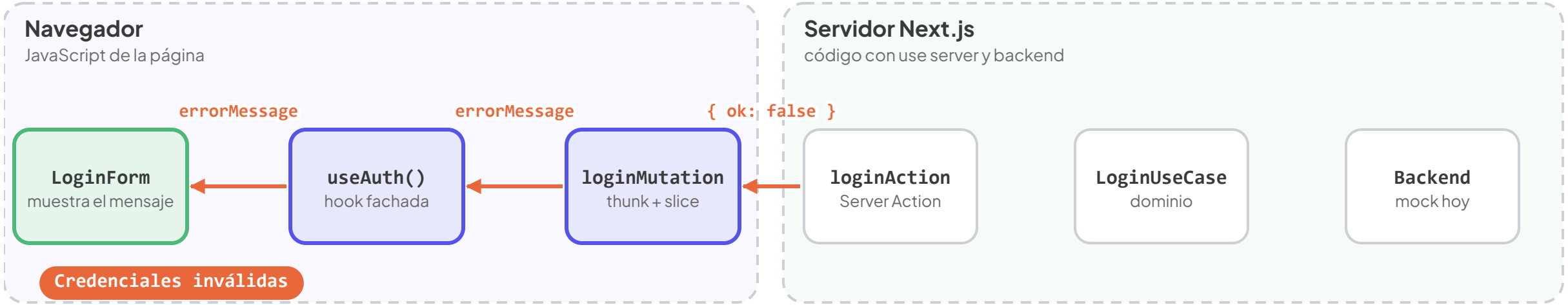
El backend responde 401 y el datasource lo traduce en **null**. No es un fallo del programa: es una respuesta prevista. Nadie lanza nada.

Credenciales inválidas (2 de 3): la acción lo devuelve como valor



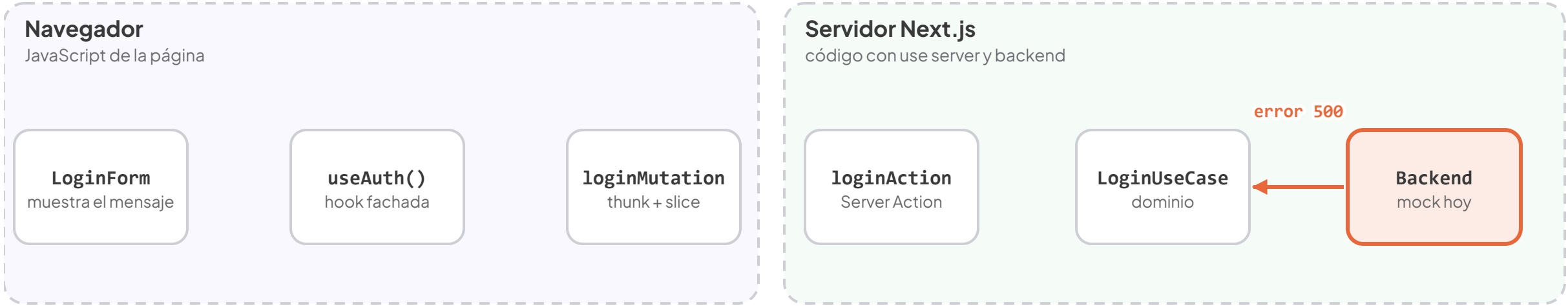
La acción devuelve `{ ok: false, message }`. Es un **valor**: viaja intacto por la red, con su texto.

Credenciales inválidas (3 de 3): llega al formulario



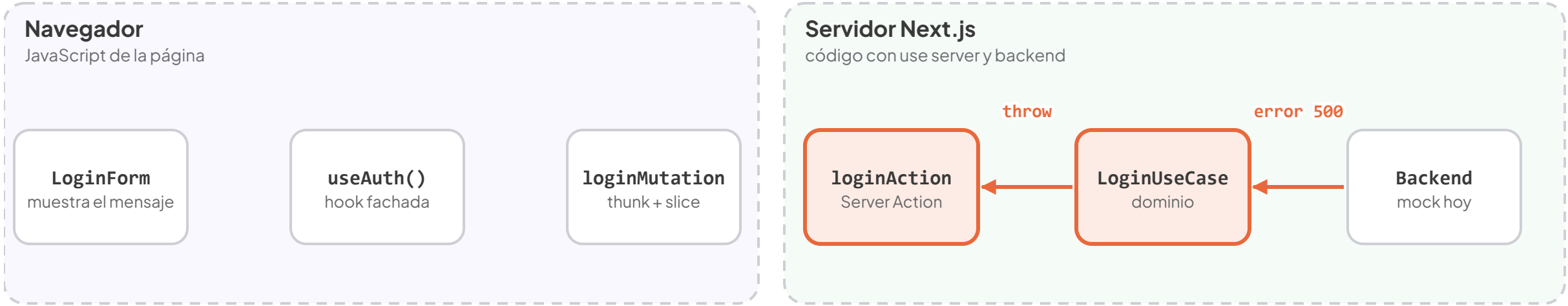
El thunk hace `rejectWithValue(message)`, el slice lo guarda en errorMessage y LoginForm lo muestra. **Un error esperado llega con su mensaje exacto.**

El backend se cae (1 de 4): responde 500



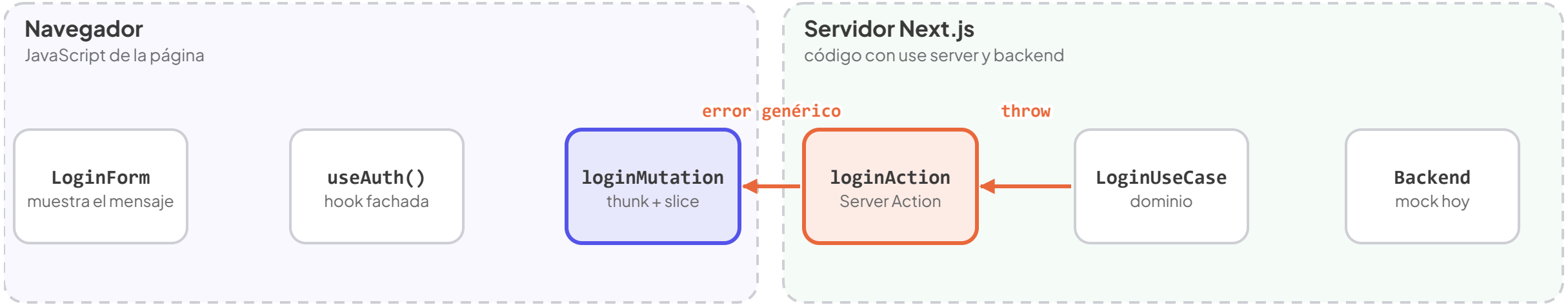
Es una falla externa que nadie previó: el backend responde con un error y el datasource lanza una excepción dentro del servidor.

El backend se cae (2 de 4): la acción no la atrapa



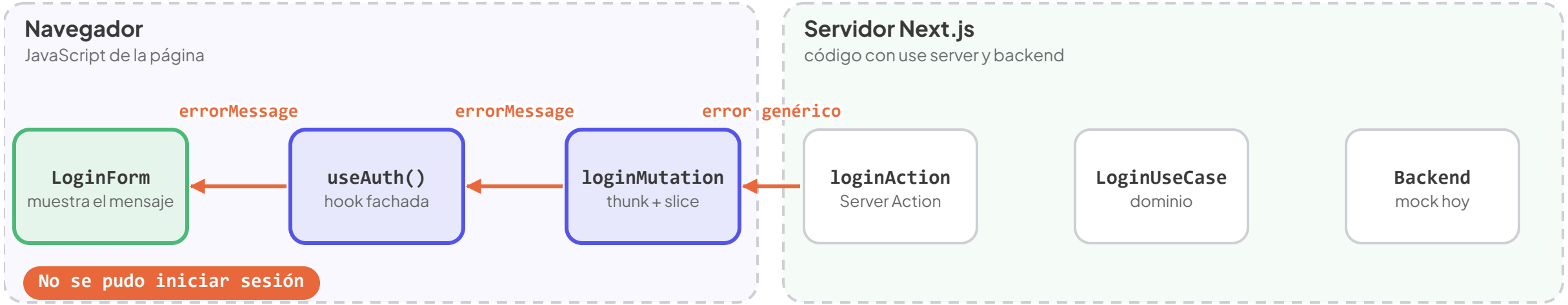
La acción no atrapa la excepción, así que **sube**. Next.js responde con un error: en producción el texto original se oculta (solo viaja un identificador, el digest).

El backend se cae (3 de 4): el thunk la atrapa



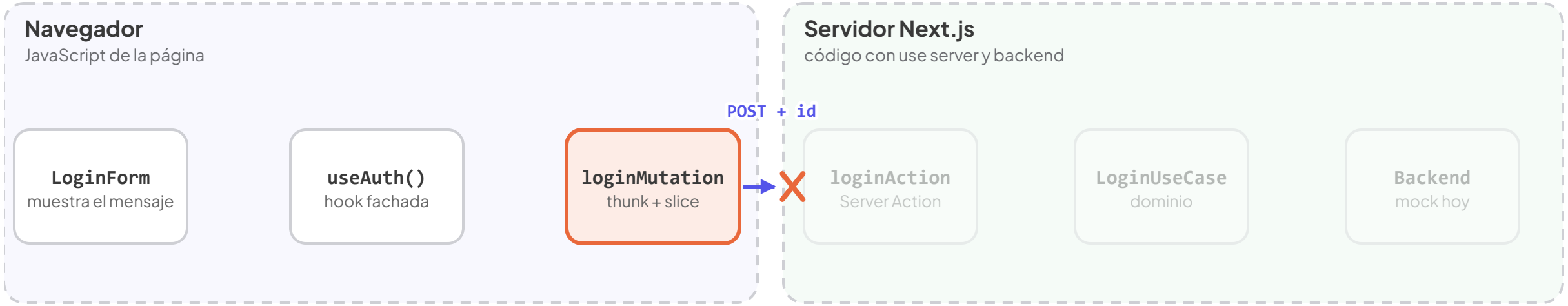
La llamada del thunk falla y su `try/catch` la atrapa. Aunque no lo tuviera, createAsyncThunk atrapa las excepciones: el boundary nunca la ve.

El backend se cae (4 de 4): un mensaje genérico



ana ve el mensaje genérico que escribiste. **Un error inesperado llega sin detalles, a propósito**: el detalle real queda en los registros del servidor.

Se cae la red (1 de 2): el POST no llega



La llamada a la acción nunca llega al servidor: ni el caso de uso ni el backend se ejecutan. La promesa del thunk se rechaza.

Se cae la red (2 de 2): el mismo camino



Dos causas distintas, **un mismo camino**: por eso el thunk es el lugar de atrapar lo inesperado. En ningún caso se llega a error.tsx.

El thunk atrapa lo inesperado

auth.slice.ts (fragmento)

```ts
async (p, { rejectWithValue }) => {
  try {
    const result = await loginAction(p.email, p.password)
    if (!result.ok) return rejectWithValue(result.message)
    return result.user
  } catch {
    return rejectWithValue('No se pudo iniciar sesión.')
  }
}
```

Esperado: ok es false
Llegó como valor: se pasa tal cual.

Inesperado: el catch
Un mensaje propio, no el texto técnico de la excepción.

Los error boundaries solo atrapan errores durante el **renderizado**: el código asíncrono (un thunk, un onClick) no lo cubre ninguno. Usa try/catch y guarda el error en el estado.

Errores al renderizar: error.tsx

app/routines/error.tsx

```tsx
'use client'

export default function Error({
 error,
 retry,
}: {
 error: Error & { digest?: string }
 retry: () => void
}) {
  useEffect(() => { console.error(error) }, [error])

  return (
    <div role="alert">
      <h2>No pudimos cargar tus rutinas</h2>
      <button onClick={() => retry()}>Reintentar</button>
    </div>
  )
}
```

Un error boundary
Next.js lo muestra en lugar de la parte de la pantalla que falló.

Debe ser un componente de cliente
Los layouts de arriba siguen visibles.

error y retry()

app/routines/error.tsx

```tsx
'use client'

export default function Error({
 error,
 retry,
}: {
 error: Error & { digest?: string }
 retry: () => void
}) {
  useEffect(() => { console.error(error) }, [error])

  return (
    <div role="alert">
      <h2>No pudimos cargar tus rutinas</h2>
      <button onClick={() => retry()}>Reintentar</button>
    </div>
  )
}
```

Recibe el error y retry()
retry() intenta renderizar de nuevo el segmento.

Antes se llamaba reset
La documentación actual usa retry.

Registrar el error

app/routines/error.tsx

```tsx
'use client'

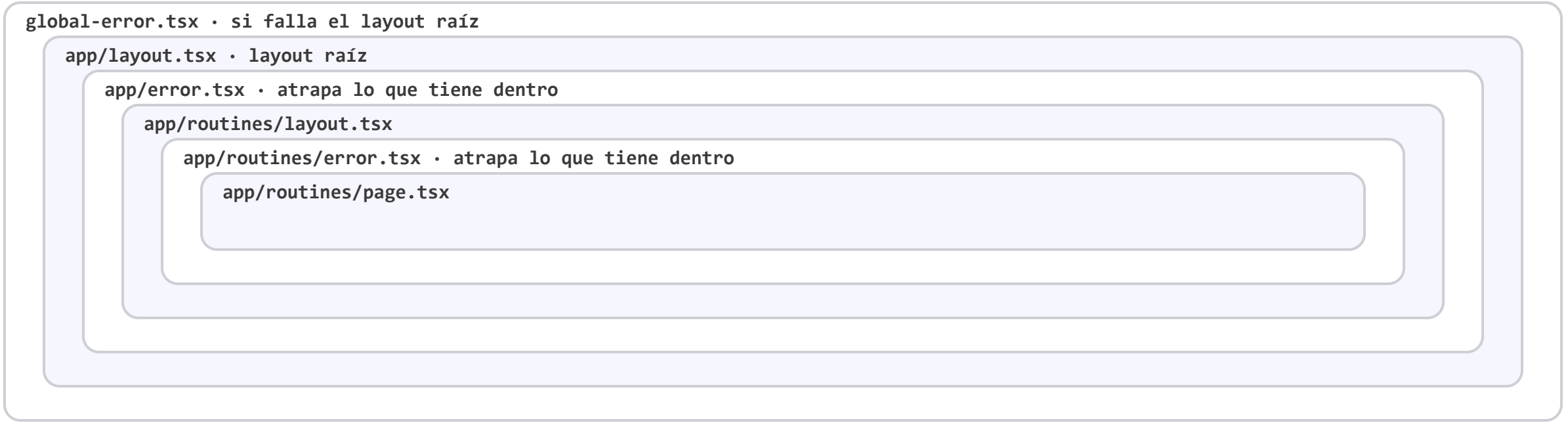
export default function Error({
 error,
 retry,
}: {
 error: Error & { digest?: string }
 retry: () => void
}) {
  useEffect(() => { console.error(error) }, [error])

  return (
    <div role="alert">
      <h2>No pudimos cargar tus rutinas</h2>
      <button onClick={() => retry()}>Reintentar</button>
    </div>
  )
}
```

Aquí va el registro

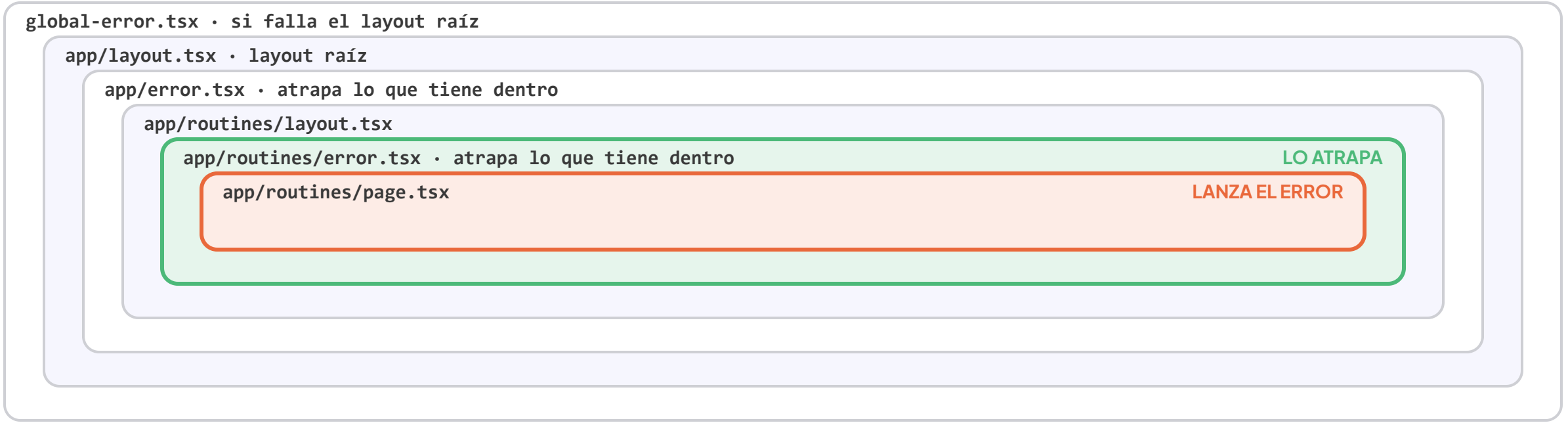
En producción recibes error.digest, el mismo identificador de los registros del servidor.

Las capas de una ruta, de afuera hacia adentro



Cada carpeta de app/ puede tener su layout, su error.tsx y su página. **Un error.tsx atrapa lo que tiene dentro**, pero no el layout.tsx de su propia carpeta: el layout está por fuera.

Un error en la página: lo atrapa routines/error.tsx



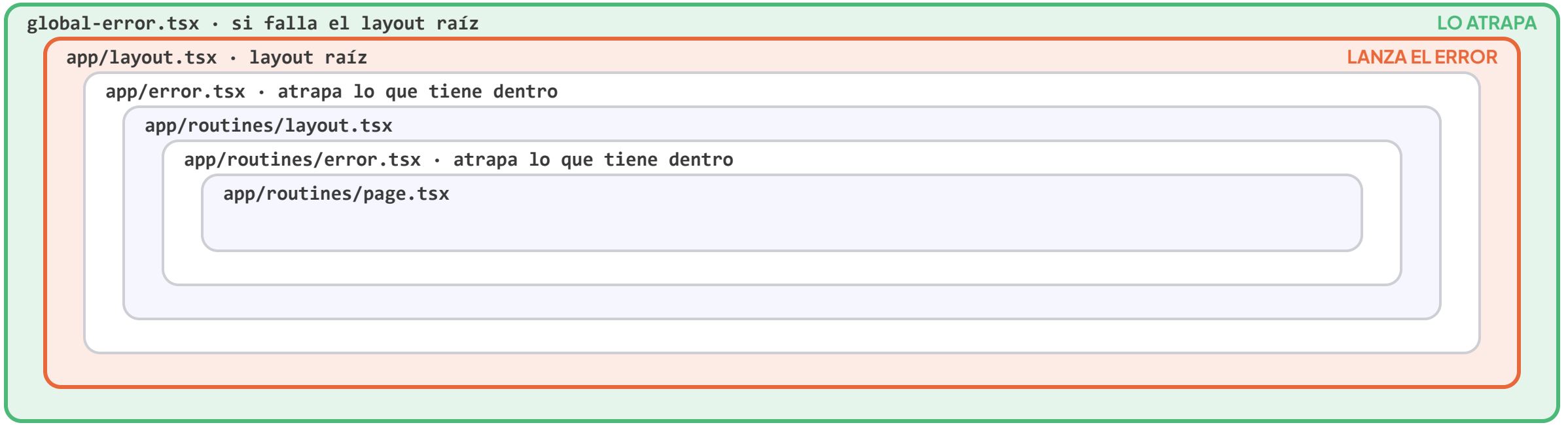
El error.tsx del segmento envuelve a la página: es el más cercano hacia afuera. **Los layouts siguen en pantalla**; solo se reemplaza la página.

Un error en el layout de routines: sube un nivel



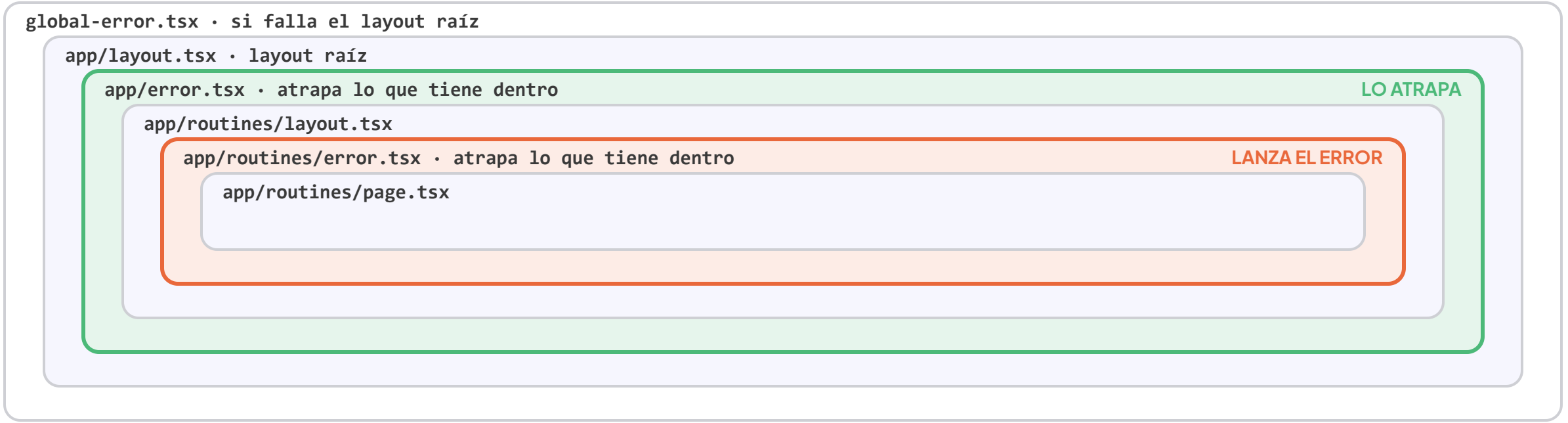
routines/error.tsx no cubre el layout de su propia carpeta. Lo atrapa **app/error.tsx**: se reemplaza el layout de routines y todo lo que contiene.

Un error en el layout raíz: global-error.tsx



app/error.tsx tampoco cubre el layout de su carpeta. Para el layout raíz existe **global-error.tsx**: reemplaza toda la página.

Si el propio error.tsx falla



Un error boundary no se atrapa a sí mismo: el error sube al error.tsx del nivel superior.

Sin error.tsx en el segmento, el error burbujea



Si no existe routines/error.tsx, el error sube hasta el más cercano hacia arriba: app/error.tsx. **Un error.tsx por segmento** te deja decidir qué parte de la pantalla se reemplaza.

global-error.tsx: lleva sus propias etiquetas

app/global-error.tsx

```tsx
'use client'

export default function GlobalError({retry}) {
  return (
    <html>
      <body>
        <h2>Algo salió mal</h2>
        <button onClick={() => retry()}>Reintentar</button>
      </body>
    </html>
  )
}
```

Reemplaza toda la página
Por eso debe incluir html y body.

Un fallback dentro de un componente
Para proteger solo una parte, Next.js ofrece catchError (de next/error).

Errores que se lanzan a propósito

Función	Cuándo	Resultado	Archivo
redirect('/login')	No hay sesión	Navega a otra ruta	—
notFound()	El recurso no existe	Estado 404	not-found.tsx
unauthorized()	Sin sesión, con página propia	Estado 401	unauthorized.tsx
forbidden()	Hay sesión pero no permiso	Estado 403	forbidden.tsx

> `unauthorized()` y `forbidden()` son **experimentales**: se activan con `experimental: { authInterrupts: true }` en next.config.ts.

notFound y forbidden en una página

app/routines/[id]/page.tsx

```tsx
export default async function RoutinePage({params}) {
  const { userId } = await requireSession()
  const { id } = await params

  const routine = await getRoutine(id)
  if (!routine) notFound()
  if (routine.ownerId !== userId) forbidden()

  return <RoutineView routine={routine}/>
}
```

notFound: 404
El recurso no existe: se muestra not-found.tsx.

forbidden: 403
Hay sesión, pero la rutina es de otro usuario.

Sin sesión
requireSession hace redirect a /login.

Universidad ICESI

No las metas en un try/catch

`redirect()`, `notFound()`, `unauthorized()` y `forbidden()` funcionan **lanzando una excepción**.

Dentro de un `try/catch`, el catch se la traga: no hay redirección ni página de error. Llámalas **fuera del try**, o vuelve a lanzar con `unstable_rethrow(error)`.

Tampoco escribas `return notFound()`: la función nunca devuelve nada, detiene la ejecución.

Qué ve el usuario y qué ves tú

	Mensaje al usuario	Registro para ti
Error esperado	Específico y accionable: «Credenciales inválidas»	Normalmente no hace falta
Error inesperado	Genérico: «No se pudo iniciar sesión». Nunca el texto de una excepción	El detalle completo: qué falló y dónde
Seguridad	Que no revele información: no digas «ese correo no existe»	Sí puedes registrar el intento

> En producción, Next.js reemplaza el mensaje de un error del servidor por uno genérico y agrega un `digest`: el mismo identificador aparece en los registros.

Resumen: manejo de errores

Esperado

`return { ok: false, message }` y rejectWithValue: llega con su mensaje.

Inesperado

`try/catch` en el thunk: mensaje genérico, detalle en el servidor.

Al renderizar

`error.tsx` por segmento y `global-error.tsx` para el layout raíz.

A propósito

redirect, notFound, unauthorized y forbidden: nunca en un try/catch.

El código asíncrono

Un boundary no lo atrapa: thunks y onClick usan try/catch.

El usuario

Mensajes claros; sin revelar qué correos existen.

Próxima semana

Pruebas end-to-end con Playwright

Universidad ICESI