

Navegar en Next.js

Del clic a la nueva ruta

Computación en Internet 3 • Semana 13 • Universidad Icesi

Qué es navegar en el App Router



Sin recarga completa

El navegador no vuelve a descargar toda la página



Layouts y estado se conservan

Navbar, menú y estado de React siguen vivos



Solo cambia el segmento

Se pide y pinta únicamente la página nueva



Precarga en segundo plano

Los enlaces visibles se cargan antes del clic

Navegar en Next.js es cambiar de ruta en el cliente, sin recargar la página completa.

Formas de navegar en Next.js

Cliente ('use client')

declarativa

<Link>

Enlaces y menús

Precarga la ruta

También en servidor

```
<Link href="/productos">  
  Catálogo  
</Link>
```

<Link> : la forma declarativa. Lo que el usuario ve y pulsa.

Formas de navegar en Next.js

Cliente ('use client')

declarativa

<Link>

Enlaces y menús
Precarga la ruta
También en servidor

```
<Link href="/productos">  
  Catálogo  
</Link>
```

programática

useRouter()

push y replace
back y refresh
Tras un evento

```
router.push(  
  '/productos'  
)
```

useRouter : navegar desde código, cuando hay un evento o una validación.

Formas de navegar en Next.js

Cliente ('use client')

declarativa

<Link>

Enlaces y menús
Precarga la ruta
También en servidor

```
<Link href="/productos">  
  Catálogo  
</Link>
```

programática

useRouter()

push y replace
back y refresh
Tras un evento

```
router.push(  
  '/productos'  
)
```

nativa

pushState

Cambia la URL
sin navegar
Filtros y orden

```
history.pushState(  
  null, '', '?orden=asc'  
)
```

History API: cambiar la URL sin navegar, por ejemplo al ordenar.

Formas de navegar en Next.js

Cliente ('use client')

declarativa

<Link>

Enlaces y menús
Precarga la ruta
También en servidor

```
<Link href="/productos">  
  Catálogo  
</Link>
```

programática

useRouter()

push y replace
back y refresh
Tras un evento

```
router.push(  
  '/productos'  
)
```

nativa

pushState

Cambia la URL
sin navegar
Filtros y orden

```
history.pushState(  
  null, '', '?orden=asc'  
)
```

Servidor

antes de renderizar

redirect()

Sin sesión: va a /login

```
redirect('/login')
```

recurso inexistente

notFound()

Muestra el 404

```
notFound()
```

En el servidor no hay hooks: se usan `redirect` y `notFound`.

El componente Link

components/Menu.tsx

```
import Link from 'next/link';

export function Menu({ id }: { id: string }) {
  return (
    <nav className="flex gap-4">
      <Link href="/productos">Catálogo</Link>
      <Link href={` /productos/${id}`}>
        Detalle
      </Link>
      <Link href={{
        pathname: '/buscar',
        query: { q: 'tenis' },
      }}>Buscar</Link>
    </nav>
  );
}
```

`href="/productos"` ruta fija

`href={` /productos/${id}`}` ruta dinámica

`href={{ pathname, query }}` arma `/buscar?q=tenis`

Renderiza una etiqueta `<a>` accesible y precarga la ruta.

`<Link>` sirve en Server y Client Components: no necesita `use client`.

Props de Link y prefetch

PROP	POR DEFECTO	QUÉ HACE	EJEMPLO
<code>replace</code>	<code>false</code>	reemplaza la entrada del historial	<code><Link href="/login" replace></code>
<code>scroll</code>	<code>true</code>	con <code>false</code> conserva la posición vertical	<code><Link href="/p" scroll={false}></code>
<code>prefetch</code>	<code>"auto"</code>	cuándo precargar la ruta	<code><Link href="/p" prefetch={false}></code>
<code>onNavigate</code>	—	se ejecuta en la navegación de cliente	<code>onNavigate={(e) => e.preventDefault()}</code>

"auto" / null

Ruta estática: completa. Dinámica: hasta `loading.tsx`.

true

Precarga la ruta completa, también la dinámica.

false

Nunca precarga, ni al entrar al viewport ni al pasar el mouse.

El prefetch solo funciona en producción: en desarrollo no verás las precargas.

Enlace activo

components/NavLink.tsx

```
'use client';

import Link from 'next/link';
import { usePathname } from 'next/navigation';

export function NavLink({ href, children }) {
  const pathname = usePathname();
  const activo = pathname === href;

  return (
    <Link href={href}
      className={activo ? 'font-bold' : ''}>
      {children}
    </Link>
  );
}
```

useSelectedLayoutSegment

```
// Por segmento del Layout
const segmento = useSelectedLayoutSegment();
const activo = segmento === 'productos';
```

useLinkStatus (de next/link)

```
// Dentro de un <Link>
const { pending } = useLinkStatus();
return <span>{pending ? 'Cargando...' : 'Ir'}</span>;
```

`usePathname` compara la ruta; `useLinkStatus` muestra un estado de carga mientras navega.

useRouter: navegar desde código

```
components/BotonVer.tsx
'use client';

import { useRouter } from 'next/navigation';

export function BotonVer({ id }: { id: string }) {
  const router = useRouter();

  return (
    <button onClick={() => router.push(`/productos/${id}`)}>
      Ver producto
    </button>
  );
}
```



'use client'

Obligatorio: usa el historial del navegador



next/navigation

Importa useRouter de aquí, no de next/router



router.events ya no existe

Usa usePathname y useSearchParams

Next.js recomienda `<Link>` y reservar `useRouter` para cuando hay un evento.

Métodos del router



router.push(href)

Navega y agrega una entrada al historial.

```
router.push('/productos', { scroll: false })
```



router.replace(href)

Navega sin agregar entrada: reemplaza la actual.

```
router.replace('/dashboard')
```



router.refresh()

Vuelve a pedir los Server Components y conserva el estado de cliente.

```
router.refresh()
```



router.back() / forward()

Se mueve una posición por el historial.

```
router.back()
```



router.prefetch(href)

Precarga una ruta en segundo plano.

```
router.prefetch('/productos')
```



Seguridad

No pases a push o replace URLs sin validar.

```
router.push(urlValidada)
```

Seis herramientas del router: casi todo se resuelve con **push** y **replace**.

push agrega, replace reemplaza

1. Historial inicial

`router.push('/dashboard')`

agrega /dashboard encima



/login

/home

`router.replace('/dashboard')`

cambia /login por /dashboard



/login

/home

Estás en /login. Antes de /login estuviste en /home.

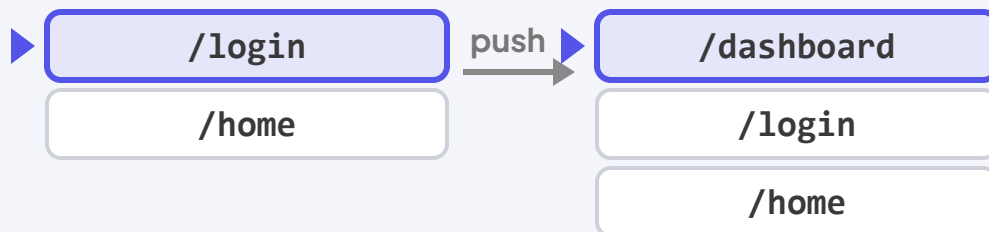
push agrega, replace reemplaza

1. Historial inicial

2. Navegas a /dashboard

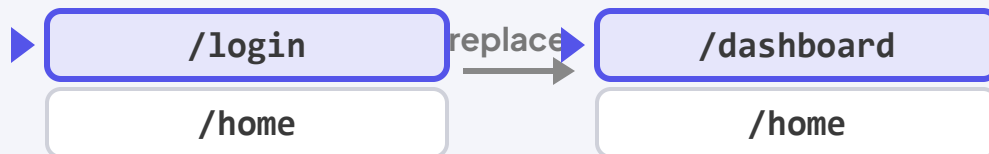
`router.push('/dashboard')`

agrega /dashboard encima



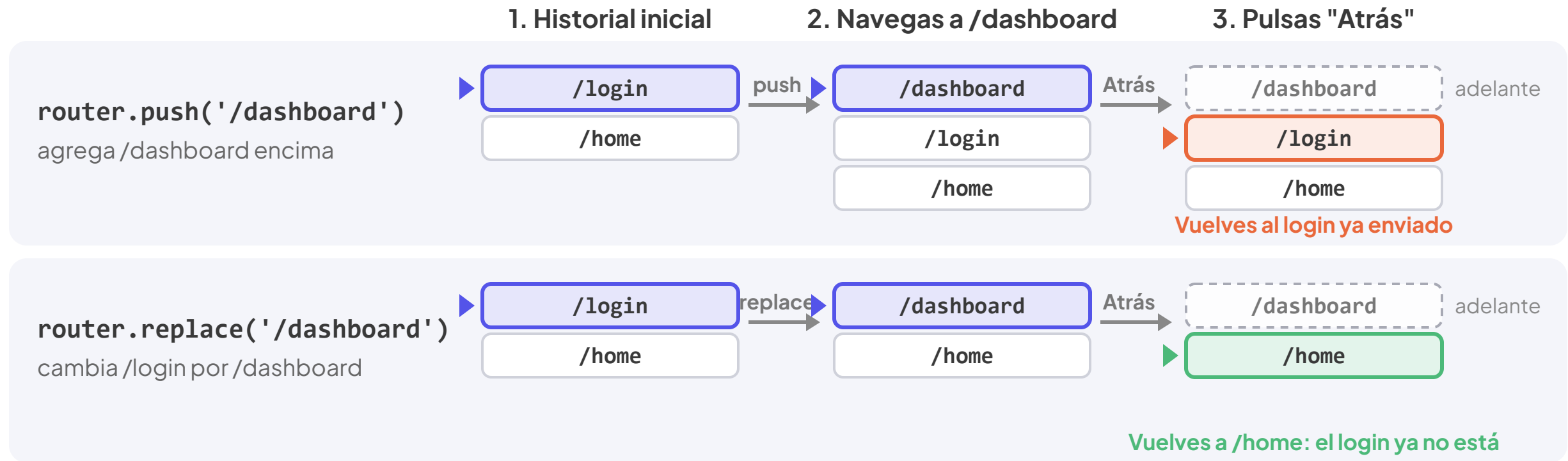
`router.replace('/dashboard')`

cambia /login por /dashboard



push agrega /dashboard encima; **replace** cambia /login por /dashboard.

push agrega, replace reemplaza



Con `push`, "Atrás" vuelve al login. Con `replace`, vuelve a `/home`.

Caso típico: login con replace

```
components/FormularioLogin.tsx
const handleLogin = async (e: React.FormEvent) => {
  e.preventDefault();
  const ok = await loginSimulado(email);

  if (ok) {
    router.replace('/dashboard');
  }
};
```



Con push

El usuario pulsa "Atrás" y regresa al formulario de login ya enviado.

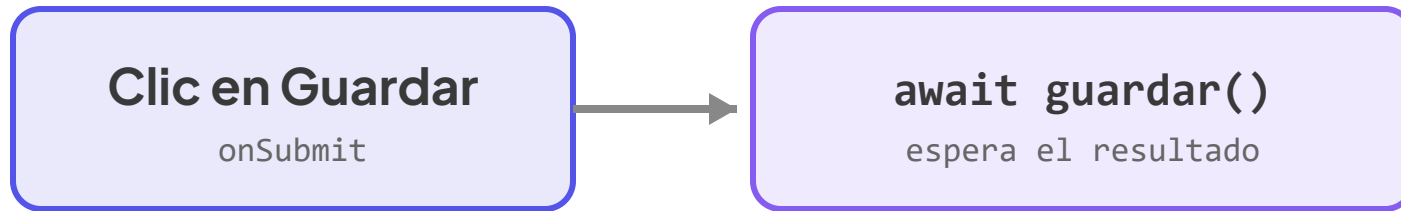


Con replace

La entrada del login desaparece del historial: "Atrás" sale de la zona de acceso.

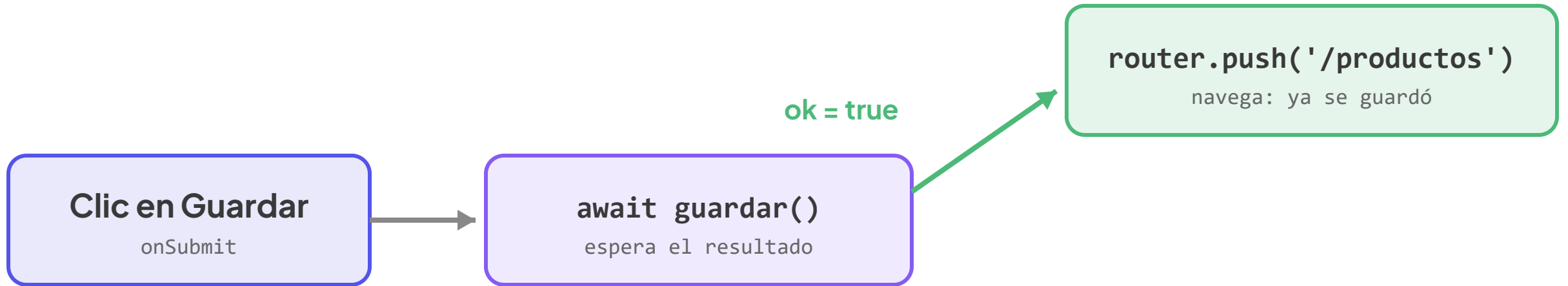
Usa **replace** en flujos que no deben reabrirse: login, pasarela de pago, confirmaciones.

Navegar después de un evento



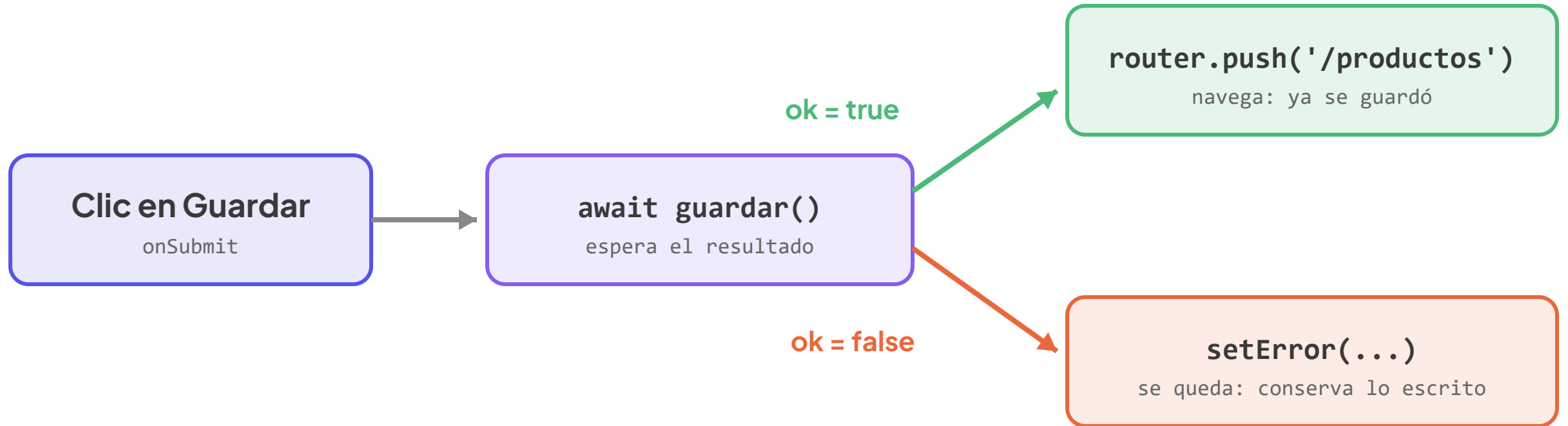
El usuario envía el formulario y el componente espera el resultado.

Navegar después de un evento



Si se guardó, recién ahí se navega con `router.push` .

Navegar después de un evento



Si falló, no se navega: se muestra el error y se conserva lo escrito.

Navegar después de un evento: el código

components/FormularioProducto.tsx

```
const handleSubmit = async (e: React.FormEvent) => {  
  e.preventDefault();  
  setError(null);  
  
  const ok = await guardar({ nombre });  
  
  if (ok) {  
    router.push('/productos');  
  } else {  
    setError('No se pudo guardar.');
```



1. onSubmit

El clic dispara la función asíncrona



2. await guardar()

Espera el resultado antes de decidir



3. ok → router.push

Navega solo si ya se guardó



4. error → setError

Se queda y conserva lo escrito

El orden importa: primero la operación, después la navegación.

Leer la URL



usePathname()

La ruta actual.

```
'/productos'
```



useParams()

Los segmentos dinámicos.

```
{id: '42'}
```



useSearchParams()

Los parámetros de consulta, dentro de Suspense.

```
?categoria=calzado
```



params

En una página de servidor, en Next 16 es una Promise.

```
const {id} = await params
```

Los tres hooks son de cliente; en servidor las páginas reciben **params** y **searchParams**.

Filtrar con parámetros de la URL

```
components/FiltroCategoria.tsx
'use client';

export function FiltroCategoria() {
  const router = useRouter();
  const pathname = usePathname();
  const searchParams = useSearchParams();

  const elegir = (categoria: string) => {
    const params = new URLSearchParams(searchParams.toString());
    params.set('categoria', categoria);
    router.push(`${pathname}?${params}`, { scroll: false });
  };
  // <select onChange={(e) => elegir(e.target.value)}>
}
```

Quien usa useSearchParams va en Suspense

```
<Suspense fallback={null}>
  <FiltroCategoria />
</Suspense>
```



Solo lectura

useSearchParams no tiene set : se arma un URLSearchParams nuevo y se navega.

Sin `<Suspense>`, el build de una ruta estática falla: "Missing Suspense boundary".

params y searchParams en el servidor

```
app/productos/[id]/page.tsx
interface Props {
  params: Promise<{ id: string }>;
  searchParams: Promise<{ categoria?: string }>;
}

export default async function Page({ params, searchParams }: Props) {
  const { id } = await params;
  const { categoria } = await searchParams;

  return <h1>{id} · {categoria ?? 'general'}</h1>;
}
```



Son Promise

En Next 16 se resuelven con `await`. Leerlas de forma síncrona ya no funciona.



Solo en páginas

Las páginas reciben `searchParams`; los layouts no.

Un Server Component no usa hooks: lee `params` directamente.

redirect(): navegar desde el servidor

app/dashboard/page.tsx

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

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

  if (!session) {
    redirect('/login');
  }

  return <Panel />;
}
```

redirect(path, 'replace' | 'push')



Dónde sirve

Servidor y render en cliente. No en event handlers: usa useRouter



Por defecto

push en Server Actions, replace en el resto. Responde 307

Si no hay sesión, el servidor redirige antes de renderizar la página.

permanentRedirect, notFound y try/catch

```
app/rutinas/[id]/page.tsx
import { notFound, permanentRedirect }
  from 'next/navigation';

export default async function Detalle({ params }) {
  const { id } = await params;
  const rutina = await buscar(id);

  if (!rutina) notFound();           // 404
  if (rutina.movida)
    permanentRedirect(rutina.nuevaUrl); // 308

  return <Detalle rutina={rutina} />;
}
```

```
redirect y notFound lanzan un error interno
// Mal: el catch atrapa el redirect
try {
  redirect('/login');
} catch {}

// Bien: fuera del try/catch
if (!sesion) redirect('/login');
```

No los pongas dentro de `try/catch` : lanzan un error interno a propósito para detener el render.

`notFound()` muestra `not-found.tsx` ; `permanentRedirect` responde 308.

Redirecciones fuera de los componentes

Rutas fijas

```
// next.config.js
module.exports = {
  async redirects() {
    return [
      {
        source: '/antiguo',
        destination: '/nuevo',
        permanent: true, // 308
      },
    ];
  },
};
```

Lógica en cada petición

```
// proxy.ts
import { NextResponse, type NextRequest }
  from 'next/server';

export function proxy(request: NextRequest) {
  return NextResponse.redirect(
    new URL('/home', request.url)
  );
}

export const config = { matcher: '/viejo/:path*' };
```

`next.config` para rutas fijas; `proxy.ts` (antes `middleware`) para decidir con la petición.

History API: cambiar la URL sin navegar

```
components/OrdenLista.tsx
'use client';

export function OrdenLista() {
  const searchParams = useSearchParams();

  const ordenar = (orden: string) => {
    const params = new URLSearchParams(searchParams.toString());
    params.set('orden', orden);
    window.history.pushState(null, '', `?${params}`);
  };
  // <button onClick={() => ordenar('asc')}>
}
```



No navega

Cambia la barra de direcciones sin pedir la página al servidor



Se sincroniza

usePathname y useSearchParams reflejan el cambio



pushState / replaceState

agregan o reemplazan la entrada del historial

Ideal para ordenar o abrir pestañas internas sin volver a cargar la vista.

Guía de decisión

 El usuario ve y pulsa un enlace o menú	→	<code><Link></code>
 La navegación depende de un evento o validación	→	<code>router.push</code>
 Terminó un flujo que no debe reabrirse (login)	→	<code>router.replace</code>
 Redirige desde el servidor: sin sesión o sin permisos	→	<code>redirect()</code>
 El recurso pedido no existe	→	<code>notFound()</code>
 Solo cambian parámetros de la URL	→	<code>history.pushState</code>

Pregúntate quién navega y desde dónde: eso elige el mecanismo.

Errores comunes

ERROR	CAUSA	SOLUCIÓN
Importar de <code>next/router</code>	Se confunde el Pages Router con el App Router	Importa <code>useRouter</code> de <code>next/navigation</code>
Llamar <code>router.push</code> durante el render	La navegación está en el cuerpo del componente	Muévela a un evento (<code>onClick</code> , <code>onSubmit</code>) o a un <code>useEffect</code>
Usar <code><a></code> para rutas internas	Se usa el ancla nativa	Usa <code><Link></code> : evita la recarga completa
Olvidar <code>'use client'</code> con <code>useRouter</code>	Un hook interactivo en un Server Component	Agrega <code>'use client'</code> en la primera línea
<code>redirect()</code> dentro de <code>try/catch</code>	El catch atrapa el error interno	Llámalo fuera del bloque

Casi todos los errores vienen de mezclar el entorno: cliente, servidor o Pages Router.

Trabajo en clase

Solo componentes visuales

Qué debes construir



Login

Correo, contraseña y
`router.replace` tras entrar.



Registro

Valida que las contraseñas
coincidan.



Header

Nombre de la app y avatar
de usuario.



Navbar

Enlace activo con
`usePathname` y menú móvil.

Con Tailwind v4 y daisyUI 5, mobile-first y sin backend.

Cuatro piezas: dos pantallas y dos componentes de layout.

Pistas con código

Enlazar pantallas

```
<Link href="/register" className="link link-primary">
  Regístrate
</Link>
```

Enlace activo en el navbar

```
const pathname = usePathname();
<Link href={href}
  className={pathname === href ? 'menu-active' : ''}>
```

Tras el login

```
if (correo && clave) {
  router.replace('/routines');
}
```

Menú móvil con drawer

```
<div className="drawer lg:drawer-open">
  <input id="menu" type="checkbox"
    className="drawer-toggle" />
  <div className="drawer-content">...</div>
  <div className="drawer-side">...</div>
</div>
```

Combina `<Link>`, `usePathname`, `router.replace` y un `drawer` o `dropdown`.

Entrega y evaluación



Repositorio privado

Dentro de la organización del curso, en GitHub. Entrega el enlace en la plataforma.

Login y registro completos



30%

Responsive: menú móvil y horizontal



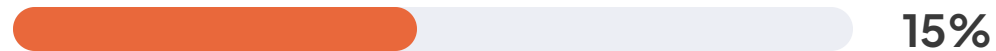
25%

Uso correcto de daisyUI y Tailwind



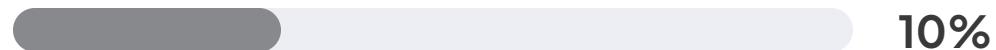
20%

Navegación: Link, usePathname, replace



15%

Organización del código



10%

El tiempo de trabajo lo indica el profesor durante la sesión.

Recursos para seguir



Tailwind CSS v4

Documentación oficial de utilidades, @theme y variantes.

tailwindcss.com/docs



daisyUI 5

Catálogo de componentes, temas y colores.

daisyui.com/components



Next.js App Router

Linking and Navigating, useRouter, redirect.

nextjs.org/docs/app

Las tres documentaciones oficiales cubren todo lo visto en esta sesión.