Linking and Navigating

There are four ways to navigate between routes in Next.js:

- Using the `<Link>` Component
- Using the `useRouter` hook (Client Components)
- Using the `redirect` function (Server Components)
- Using the native History API

This page will go through how to use each of these options, and dive deeper into how navigation works.

`<Link>` Component

`<Link>` is a built-in component that extends the HTML `<a>` tag to provide prefetching and client-side navigation between routes. It is the primary and recommended way to navigate between routes in Next.js.

You can use it by importing it from `next/link`, and passing a `href` prop to the component:

app/page.tsx TypeScript

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

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

There are other optional props you can pass to `<Link>`. See the API reference for more.

Examples

Linking to Dynamic Segments

When linking to dynamic segments, you can use template literals and interpolation ↗ to generate a list of links. For example, to generate a list of blog posts:

app/blog/PostList.js

```
import Link from 'next/link'

export default function PostList({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${post.slug}`}>{post.title}</Link>
        </li>
      ))}
    </ul>
  )
}
```

Checking Active Links

You can use `usePathname()` to determine if a link is active. For example, to add a class to the active link, you can check if the current `pathname` matches the `href` of the link:

@/app/ui/nav-links.tsx TypeScript

```
'use client'

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

export function Links() {
  const pathname = usePathname()

  return (
    <nav>
      <Link className={`link ${pathname === '/' ? 'active' : ''}`} href="/">
        Home
      </Link>

```

```
        <Link
          className={`link ${pathname === '/about' ? 'active' : ''}`}
          href="/about"
        >
          About
        </Link>
      </nav>
    )
  }
```

Scrolling to an `id`

The default behavior of the Next.js App Router is to **scroll to the top of a new route or to maintain the scroll position for backwards and forwards navigation.**

If you'd like to scroll to a specific `id` on navigation, you can append your URL with a `#` hash link or just pass a hash link to the `href` prop. This is possible since `<Link>` renders to an `<a>` element.

```
<Link href="/dashboard#settings">Settings</Link>

// Output
<a href="/dashboard#settings">Settings</a>
```

Good to know:

- Next.js will scroll to the Page if it is not visible in the viewport upon navigation.

Disabling scroll restoration

The default behavior of the Next.js App Router is to **scroll to the top of a new route or to maintain the scroll position for backwards and forwards navigation.** If you'd like to disable this behavior, you can pass `scroll={false}` to the `<Link>` component, or `scroll: false` to `router.push()` or `router.replace()`.

```
// next/link
<Link href="/dashboard" scroll={false}>
  Dashboard
</Link>
```

```
// useRouter
```

```
import { useRouter } from 'next/navigation'

const router = useRouter()

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

`useRouter()` hook

The `useRouter` hook allows you to programmatically change routes from Client Components.

app/page.js

```
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

For a full list of `useRouter` methods, see the API reference.

Recommendation: Use the `<Link>` component to navigate between routes unless you have a specific requirement for using `useRouter`.

`redirect` function

For Server Components, use the `redirect` function instead.

app/team/[id]/page.tsx TypeScript

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

async function fetchTeam(id: string) {
  const res = await fetch('https://...')
  if (!res.ok) return undefined
  return res.json()
}

export default async function Profile({ params }: { params: { id: string } }) {
  const team = await fetchTeam(params.id)
  if (!team) {
    redirect('/login')
  }

  // ...
}
```

Good to know:

- `redirect` returns a 307 (Temporary Redirect) status code by default. When used in a Server Action, it returns a 303 (See Other), which is commonly used for redirecting to a success page as a result of a POST request.
- `redirect` internally throws an error so it should be called outside of `try/catch` blocks.
- `redirect` can be called in Client Components during the rendering process but not in event handlers. You can use the `useRouter` hook instead.
- `redirect` also accepts absolute URLs and can be used to redirect to external links.
- If you'd like to redirect before the render process, use `next.config.js` or Middleware.

See the `redirect` API reference for more information.

Using the native History API

Next.js allows you to use the native `window.history.pushState` ↗ and `window.history.replaceState` ↗ methods to update the browser's history stack without reloading the page.

`pushState` and `replaceState` calls integrate into the Next.js Router, allowing you to sync with `usePathname` and `useSearchParams`.

`window.history.pushState`

Use it to add a new entry to the browser's history stack. The user can navigate back to the previous state. For example, to sort a list of products:

```
'use client'

import { useSearchParams } from 'next/navigation'

export default function SortProducts() {
  const searchParams = useSearchParams()

  function updateSorting(sortOrder: string) {
    const params = new URLSearchParams(searchParams.toString())
    params.set('sort', sortOrder)
    window.history.pushState(null, '', `?${params.toString()}`)
  }

  return (
    <>
      <button onClick={() => updateSorting('asc')}>Sort Ascending</button>
      <button onClick={() => updateSorting('desc')}>Sort Descending</button>
    </>
  )
}
```

`window.history.replaceState`

Use it to replace the current entry on the browser's history stack. The user is not able to navigate back to the previous state. For example, to switch the application's locale:

```
'use client'

import { usePathname } from 'next/navigation'

export function LocaleSwitcher() {
  const pathname = usePathname()

  function switchLocale(locale: string) {
    // e.g. '/en/about' or '/fr/contact'
    const newPath = `/${locale}${pathname}`
    window.history.replaceState(null, '', newPath)
```

```
  }

  return (
    <>
      <button onClick={() => switchLocale('en')}>English</button>
      <button onClick={() => switchLocale('fr')}>French</button>
    </>
  )
}
```

How Routing and Navigation Works

The App Router uses a hybrid approach for routing and navigation. On the server, your application code is automatically code-split by route segments. And on the client, Next.js prefetches and caches the route segments. This means, when a user navigates to a new route, the browser doesn't reload the page, and only the route segments that change re-render - improving the navigation experience and performance.

1. Code Splitting

Code splitting allows you to split your application code into smaller bundles to be downloaded and executed by the browser. This reduces the amount of data transferred and execution time for each request, leading to improved performance.

Server Components allow your application code to be automatically code-split by route segments. This means only the code needed for the current route is loaded on navigation.

2. Prefetching

Prefetching is a way to preload a route in the background before the user visits it.

There are two ways routes are prefetched in Next.js:

- `<Link>` **component**: Routes are automatically prefetched as they become visible in the user's viewport. Prefetching happens when the page first loads or when it comes into view through scrolling.
- `router.prefetch()`: The `useRouter` hook can be used to prefetch routes programmatically.

The `<Link>`'s default prefetching behavior (i.e. when the `prefetch` prop is left unspecified or set to `null`) is different depending on your usage of `loading.js`. Only the shared layout, down the rendered "tree" of components until the first `loading.js` file, is prefetched and cached for `30s`. This reduces the cost of fetching an entire dynamic route, and it means you can show an instant loading state for better visual feedback to users.

You can disable prefetching by setting the `prefetch` prop to `false`. Alternatively, you can prefetch the full page data beyond the loading boundaries by setting the `prefetch` prop to `true`.

See the `<Link>` API reference for more information.

Good to know:

- Prefetching is not enabled in development, only in production.

3. Caching

Next.js has an **in-memory client-side cache** called the Router Cache. As users navigate around the app, the React Server Component Payload of prefetched route segments and visited routes are stored in the cache.

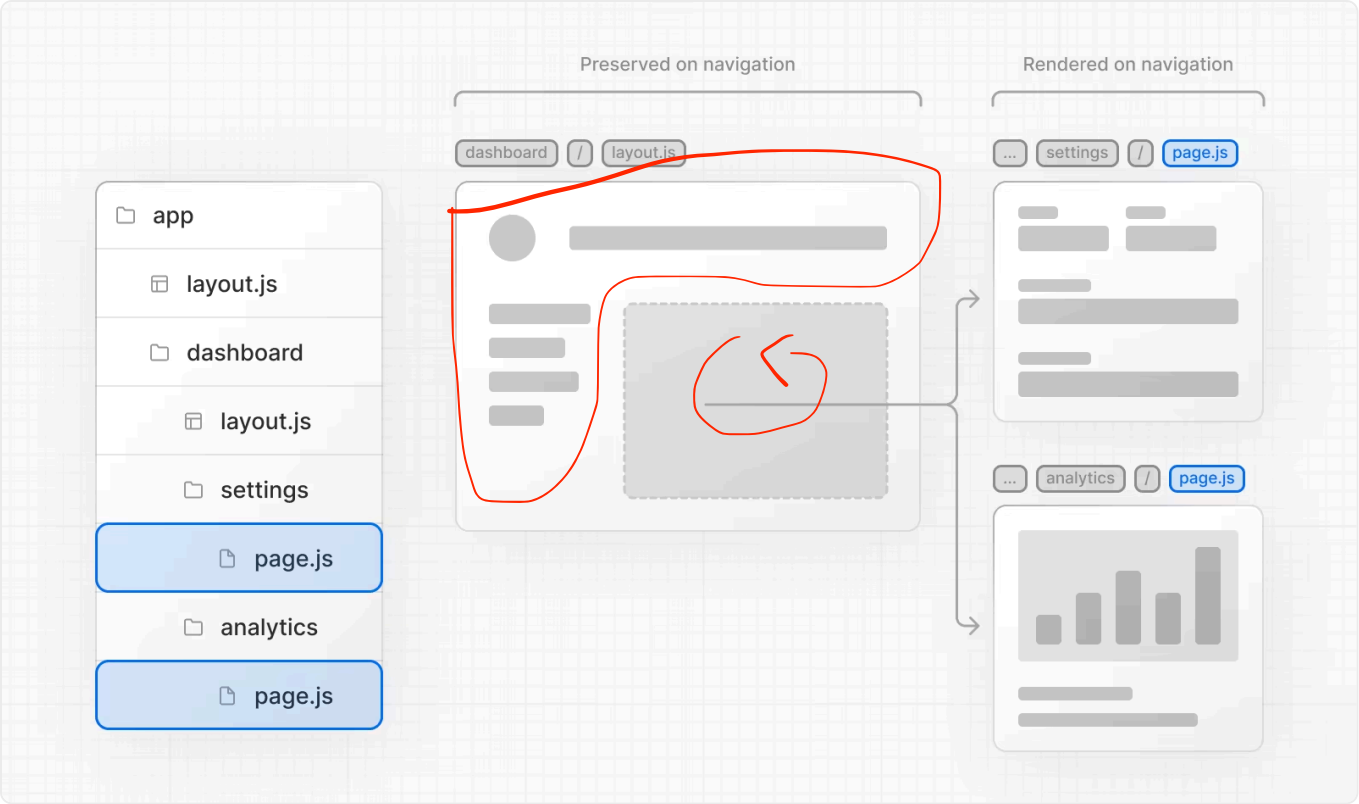
This means on navigation, the cache is reused as much as possible, instead of making a new request to the server - improving performance by reducing the number of requests and data transferred.

Learn more about how the Router Cache works and how to configure it.

4. Partial Rendering

Partial rendering means only the route segments that change on navigation re-render on the client, and any shared segments are preserved.

For example, when navigating between two sibling routes, `/dashboard/settings` and `/dashboard/analytics`, the `settings` and `analytics` pages will be rendered, and the shared `dashboard` layout will be preserved.



Without partial rendering, each navigation would cause the full page to re-render on the client. Rendering only the segment that changes reduces the amount of data transferred and execution time, leading to improved performance.

5. Soft Navigation

Browsers perform a "hard navigation" when navigating between pages. The Next.js App Router enables "soft navigation" between pages, ensuring only the route segments that have changed are re-rendered (partial rendering). This enables client React state to be preserved during navigation.

6. Back and Forward Navigation

By default, Next.js will maintain the scroll position for backwards and forwards navigation, and re-use route segments in the Router Cache.

7. Routing between `pages/` and `app/`

When incrementally migrating from `pages/` to `app/`, the Next.js router will automatically handle hard navigation between the two. To detect transitions from `pages/` to `app/`, there is a client router filter that leverages probabilistic checking of app routes, which can occasionally result in false positives. By default, such occurrences should be very rare, as

we configure the false positive likelihood to be 0.01%. This likelihood can be customized via the `experimental.clientRouterFilterAllowedRate` option in `next.config.js`. It's important to note that lowering the false positive rate will increase the size of the generated filter in the client bundle. 오탐률 0.01%

Alternatively, if you prefer to disable this handling completely and manage the routing between `pages/` and `app/` manually, you can set `experimental.clientRouterFilter` to false in `next.config.js`. When this feature is disabled, any dynamic routes in pages that overlap with app routes won't be navigated to properly by default.

Next Steps

App Router > Building Your Application

Caching

An overview of caching mechanisms in Next.js.

App Router > ... > Configuring

TypeScript

Next.js provides a TypeScript-first development experience for building your React application.

Previous
Layouts and Templates

Next
Error Handling

Next.js Conf Governance

Previews

Subscribe to our newsletter

Stay updated on new releases and features, guides, and case studies.

you@domain.com Subscribe