Linking and Navigating

The Next.js router allows you to do client-side route transitions between pages, similar to a single-page application.

A React component called `Link` is provided to do this client-side route transition.

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

function Home() {
  return (
    <ul>
      <li>
        <Link href="/">Home</Link>
      </li>
      <li>
        <Link href="/about">About Us</Link>
      </li>
      <li>
        <Link href="/blog/hello-world">Blog Post</Link>
      </li>
    </ul>
  );
}

export default Home;
```

The example above uses multiple links. Each one maps a path (`href`) to a known page:

- `/` → `pages/index.js`
- `/about` → `pages/about.js`
- `/blog/hello-world` → `pages/blog/[slug].js`

Any `<Link />` in the viewport (initially or through scroll) will be prefetched by default (including the corresponding data) for pages using Static Generation. The corresponding data for server-rendered routes is fetched *only when* the `<Link />` is clicked.

Linking to dynamic paths

You can also use interpolation to create the path, which comes in handy for dynamic route segments. For example, to show a list of posts which have been passed to the component as a prop:

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

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

export default Posts;
```

`encodeURIComponent` ↗ is used in the example to keep the path utf-8 compatible.

Alternatively, using a URL Object:

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

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link
            href={{
              pathname: '/blog/[slug]',
              query: { slug: post.slug },
            }}
          >
            {post.title}
          </Link>
        </li>
      ))}
    </ul>
  );
}
```

```
export default Posts;
```

Now, instead of using interpolation to create the path, we use a URL object in `href` where:

- `pathname` is the name of the page in the `pages` directory. `/blog/[slug]` in this case.
- `query` is an object with the dynamic segment. `slug` in this case.

Injecting the router

▶ **Examples**

To access the `router` object in a React component you can use `useRouter` or `withRouter`.

In general we recommend using `useRouter`.

Imperative Routing

▶ **Examples**

`next/link` should be able to cover most of your routing needs, but you can also do client-side navigations without it, take a look at the documentation for `next/router`.

The following example shows how to do basic page navigations with `useRouter`:

```
import { useRouter } from 'next/router';

export default function ReadMore() {
  const router = useRouter();

  return (
    <button onClick={() => router.push('/about')}>
      Click here to read more
    </button>
  );
}
```

Shallow Routing

▶ **Examples**

Shallow routing allows you to change the URL without running data fetching methods again, that includes `getServerSideProps`, `getStaticProps`, and `getInitialProps`.

You'll receive the updated `pathname` and the `query` via the `router` object (added by `useRouter` or `withRouter`), without losing state.

To enable shallow routing, set the `shallow` option to `true`. Consider the following example:

```
import { useEffect } from 'react';
import { useRouter } from 'next/router';

// Current URL is '/'
function Page() {
  const router = useRouter();

  useEffect(() => {
    // Always do navigations after the first render
    router.push('/?counter=10', undefined, { shallow: true });
  }, []);

  useEffect(() => {
    // The counter changed!
  }, [router.query.counter]);
}

export default Page;
```

The URL will get updated to `/?counter=10`. and the page won't get replaced, only the state of the route is changed.

You can also watch for URL changes via `componentDidUpdate` ↗ as shown below:

```
componentDidUpdate(prevProps) {
  const { pathname, query } = this.props.router
  // verify props have changed to avoid an infinite loop
  if (query.counter !== prevProps.router.query.counter) {
    // fetch data based on the new query
  }
}
```

Caveats

Shallow routing **only** works for URL changes in the current page. For example, let's assume we have another page called `pages/about.js`, and you run this:

```
router.push('/?counter=10', '/about?counter=10', { shallow: true });
```

Since that's a new page, it'll unload the current page, load the new one and wait for data fetching even though we asked to do shallow routing.

When shallow routing is used with middleware it will not ensure the new page matches the current page like previously done without middleware. This is due to middleware being able to rewrite dynamically and can't be verified client-side without a data fetch which is skipped with shallow, so a shallow route change must always be treated as shallow.