useRouter

If you want to access the `router` object inside any function component in your app, you can use the `useRouter` hook, take a look at the following example:

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

function ActiveLink({ children, href }) {
  const router = useRouter();
  const style = {
    marginRight: 10,
    color: router.asPath === href ? 'red' : 'black',
  };

  const handleClick = (e) => {
    e.preventDefault();
    router.push(href);
  };

  return (
    <a href={href} onClick={handleClick} style={style}>
      {children}
    </a>
  );
}

export default ActiveLink;
```

`useRouter` is a React Hook ↗, meaning it cannot be used with classes. You can either use withRouter or wrap your class in a function component.

`router` object

The following is the definition of the `router` object returned by both `useRouter` and `withRouter`:

- `pathname`: `String` - The path for current route file that comes after `/pages`. Therefore, `basePath`, `locale` and trailing slash (`trailingSlash: true`) are not included.
- `query`: `Object` - The query string parsed to an object, including dynamic route parameters. It will be an empty object during prerendering if the page doesn't use Server-side Rendering. Defaults to `{}`
- `asPath`: `String` - The path as shown in the browser including the search params and respecting the `trailingSlash` configuration. `basePath` and `locale` are not included.
- `isFallback`: `boolean` - Whether the current page is in fallback mode.
- `basePath`: `String` - The active basePath (if enabled).
- `locale`: `String` - The active locale (if enabled).
- `locales`: `String[]` - All supported locales (if enabled).
- `defaultLocale`: `String` - The current default locale (if enabled).
- `domainLocales`: `Array<{domain, defaultLocale, locales}>` - Any configured domain locales.
- `isReady`: `boolean` - Whether the router fields are updated client-side and ready for use. Should only be used inside of `useEffect` methods and not for conditionally rendering on the server. See related docs for use case with automatically statically optimized pages
- `isPreview`: `boolean` - Whether the application is currently in preview mode.

Using the `asPath` field may lead to a mismatch between client and server if the page is rendered using server-side rendering or automatic static optimization. Avoid using `asPath` until the `isReady` field is `true`.

The following methods are included inside `router`:

router.push

▶ **Examples**

Handles client-side transitions, this method is useful for cases where `next/link` is not enough.

```
router.push(url, as, options);
```

- `url`: `UrlObject | String` - The URL to navigate to (see Node.JS URL module documentation ↗ for `UrlObject` properties).
- `as`: `UrlObject | String` - Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes.
- `options` - Optional object with the following configuration options:

- `scroll` - Optional boolean, controls scrolling to the top of the page after navigation. Defaults to `true`
- `shallow`: Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`
- `locale` - Optional string, indicates locale of the new page

You don't need to use `router.push` for external URLs. window.location ↗ is better suited for those cases.

Navigating to `pages/about.js`, which is a predefined route:

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

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

  return (
    <button type="button" onClick={() => router.push('/about')}>
      Click me
    </button>
  );
}
```

Navigating `pages/post/[pid].js`, which is a dynamic route:

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

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

  return (
    <button type="button" onClick={() => router.push('/post/abc')}>
      Click me
    </button>
  );
}
```

Redirecting the user to `pages/login.js`, useful for pages behind authentication:

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

// Here you would fetch and return the user
const useUser = () => ({ user: null, loading: false });

export default function Page() {
  const { user, loading } = useUser();
  const router = useRouter();

  useEffect(() => {
    if (!(user || loading)) {
      router.push('/login');
    }
  }, [user, loading]);

  return <p>Redirecting...</p>;
}
```

Resetting state after navigation

When navigating to the same page in Next.js, the page's state **will not** be reset by default as React does not unmount unless the parent component has changed.

pages/[slug].js

```
import Link from 'next/link';
import { useState } from 'react';
import { useRouter } from 'next/router';

export default function Page(props) {
  const router = useRouter();
  const [count, setCount] = useState(0);
  return (
    <div>
      <h1>Page: {router.query.slug}</h1>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase count</button>
      <Link href="/one">one</Link> <Link href="/two">two</Link>
    </div>
  );
}
```

In the above example, navigating between `/one` and `/two` **will not** reset the count . The `useState` is maintained between renders because the top-level React component, `Page`, is the same.

If you do not want this behavior, you have a couple of options:

- Manually ensure each state is updated using `useEffect`. In the above example, that could look like:

  ```
  useEffect(() => {
    setCount(0);
  }, [router.query.slug]);
  ```

- Use a React `key` to tell React to remount the component ↗. To do this for all pages, you can use a custom app:

pages/_app.js

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

export default function MyApp({ Component, pageProps }) {
  const router = useRouter();
  return <Component key={router.asPath} {...pageProps} />;
}
```

With URL object

You can use a URL object in the same way you can use it for `next/link`. Works for both the `url` and `as` parameters:

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

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

  return (
    <button
      type="button"
      onClick={() => {
        router.push({
          pathname: '/post/[pid]',
          query: { pid: post.id },
        });
      }}
    >
      Click here to read more
    </button>
  );
}
```

router.replace

Similar to the `replace` prop in `next/link`, `router.replace` will prevent adding a new URL entry into the `history` stack.

```js
router.replace(url, as, options);
```

- The API for `router.replace` is exactly the same as the API for `router.push`.

Take a look at the following example:

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

```jsx

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

  return (
    <button type="button" onClick={() => router.replace('/home')}>
      Click me
    </button>
  );
}
```

router.prefetch

Prefetch pages for faster client-side transitions. This method is only useful for navigations without `next/link`, as `next/link` takes care of prefetching pages automatically.

> This is a production only feature. Next.js doesn't prefetch pages in development.

```js
router.prefetch(url, as, options);
```

- `url` - The URL to prefetch, including explicit routes (e.g. `/dashboard`) and dynamic routes (e.g. `/product/[id]`)
- `as` - Optional decorator for `url`. Before Next.js 9.5.3 this was used to prefetch dynamic routes.
- `options` - Optional object with the following allowed fields:
  - `locale` - allows providing a different locale from the active one. If `false`, `url` has to include the locale as the active locale won't be used.

Let's say you have a login page, and after a login, you redirect the user to the dashboard. For that case, we can prefetch the dashboard to make a faster transition, like in the following example:

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

export default function Login() {
  const router = useRouter();
  const handleSubmit = useCallback((e) => {
    e.preventDefault();

    fetch('/api/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        /* Form data */
      }),
    }).then((res) => {
      // Do a fast client-side transition to the already prefetched dashboard page
      if (res.ok) router.push('/dashboard');
    });
  }, []);

  useEffect(() => {
    // Prefetch the dashboard page
    router.prefetch('/dashboard');
  }, [router]);

  return (
    <form onSubmit={handleSubmit}>
      {/* Form fields */}
      <button type="submit">Login</button>
    </form>
  );
}
```

router.beforePopState

In some cases (for example, if using a Custom Server), you may wish to listen to popstate ↗ and do something before the router acts on it.

```
router.beforePopState(cb);
```

- `cb` - The function to run on incoming `popstate` events. The function receives the state of the event as an object with the following props:
  - `url`: `String` - the route for the new state. This is usually the name of a `page`
  - `as`: `String` - the url that will be shown in the browser
  - `options`: `Object` - Additional options sent by router.push

If `cb` returns `false`, the Next.js router will not handle `popstate`, and you'll be responsible for handling it in that case. See Disabling file-system routing.

You could use `beforePopState` to manipulate the request, or force a SSR refresh, as in the following example:

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

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

  useEffect(() => {
    router.beforePopState(({ url, as, options }) => {
      // I only want to allow these two routes!
      if (as !== '/' && as !== '/other') {
        // Have SSR render bad routes as a 404.
        window.location.href = as;
        return false;
      }

      return true;
    });
  }, [router]);

  return <p>Welcome to the page</p>;
}
```

router.back

Navigate back in history. Equivalent to clicking the browser's back button. It executes `window.history.back()`.

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

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

  return (
    <button type="button" onClick={() => router.back()}>
      Click here to go back
    </button>
  );
}
```

router.reload

Reload the current URL. Equivalent to clicking the browser's refresh button. It executes `window.location.reload()`.

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

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

  return (
    <button type="button" onClick={() => router.reload()}>
      Click here to reload
    </button>
  );
}
```

router.events

▶ **Examples**

You can listen to different events happening inside the Next.js Router. Here's a list of supported events:

- `routeChangeStart(url, { shallow })` - Fires when a route starts to change
- `routeChangeComplete(url, { shallow })` - Fires when a route changed completely
- `routeChangeError(err, url, { shallow })` - Fires when there's an error when changing routes, or a route load is cancelled
  - `err.cancelled` - Indicates if the navigation was cancelled
- `beforeHistoryChange(url, { shallow })` - Fires before changing the browser's history
- `hashChangeStart(url, { shallow })` - Fires when the hash will change but not the page
- `hashChangeComplete(url, { shallow })` - Fires when the hash has changed but not the page

> **Note:** Here `url` is the URL shown in the browser, including the `basePath`.

For example, to listen to the router event `routeChangeStart`, open or create `pages/_app.js` and subscribe to the event, like so:

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

export default function MyApp({ Component, pageProps }) {
  const router = useRouter();

  useEffect(() => {
    const handleRouteChange = (url, { shallow }) => {
      console.log(
        `App is changing to ${url} ${
          shallow ? 'with' : 'without'
        } shallow routing`,
      );
    };

    router.events.on('routeChangeStart', handleRouteChange);

    // If the component is unmounted, unsubscribe
    // from the event with the `off` method:
    return () => {
      router.events.off('routeChangeStart', handleRouteChange);
    };
  }, [router]);

  return <Component {...pageProps} />;
}
```

We use a Custom App (`pages/_app.js`) for this example to subscribe to the event because it's not unmounted on page navigations, but you can subscribe to router events on any component in your application.

Router events should be registered when a component mounts (useEffect ↗ or componentDidMount ↗ / componentWillUnmount ↗) or imperatively when an event happens.

If a route load is cancelled (for example, by clicking two links rapidly in succession), `routeChangeError` will fire. And the passed `err` will contain a `cancelled` property set to `true`, as in the following example:

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

export default function MyApp({ Component, pageProps }) {
  const router = useRouter();

  useEffect(() => {
    const handleRouteChangeError = (err, url) => {
      if (err.cancelled) {
        console.log(`Route to ${url} was cancelled!`);
      }
    };

    router.events.on('routeChangeError', handleRouteChangeError);

    // If the component is unmounted, unsubscribe
    // from the event with the `off` method:
    return () => {
      router.events.off('routeChangeError', handleRouteChangeError);
    };
  }, [router]);

  return <Component {...pageProps} />;
}
```

Potential ESLint errors

Certain methods accessible on the `router` object return a Promise. If you have the ESLint rule, no-floating-promises ↗ enabled, consider disabling it either globally, or for the affected line.

If your application needs this rule, you should either `void` the promise – or use an `async` function, `await` the Promise, then void the function call. **This is not applicable when the method is called from inside an** `onClick` **handler.**

The affected methods are:

- `router.push`
- `router.replace`
- `router.prefetch`

Potential solutions

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

// Here you would fetch and return the user
const useUser = () => ({ user: null, loading: false });

export default function Page() {
  const { user, loading } = useUser();
  const router = useRouter();

  useEffect(() => {
    // disable the linting on the next line - This is the cleanest solution
    // eslint-disable-next-line no-floating-promises
    router.push('/login');

    // void the Promise returned by router.push
    if (!(user || loading)) {
      void router.push('/login');
    }
    // or use an async function, await the Promise, then void the function call
    async function handleRouteChange() {
      if (!(user || loading)) {
        await router.push('/login');
      }
    }
    void handleRouteChange();
  }, [user, loading]);

  return <p>Redirecting...</p>;
}
```

---

withRouter

If `useRouter` is not the best fit for you, `withRouter` can also add the same `router` object to any component.

Usage

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

function Page({ router }) {
  return <p>{router.pathname}</p>;
}

export default withRouter(Page);
```

TypeScript

To use class components with `withRouter`, the component needs to accept a router prop:

```
import React from 'react';
import { withRouter, NextRouter } from 'next/router';

interface WithRouterProps {
  router: NextRouter;
}

interface MyComponentProps extends WithRouterProps {}

class MyComponent extends React.Component<MyComponentProps> {
  render() {
    return <p>{this.props.router.pathname}</p>;
  }
}

export default withRouter(MyComponent);
```