Continue

You can't perform that action at this time. A common critique of design patterns is that they needlessly add complexity. Our perspective is that patterns are valuable for solving specific problems, often helping to communicate comminalities in code problems for humans. If a project doesn't have those problems, there isn't a need to apply them. Patterns can also be very language or framework-specific (e.g. React), which can often mean thinking beyond the scope of just the original GoF design patterns. Learn more about web performance patterns for loading your code more efficiently. Unsure how to think about modern approaches to loading or rendering user-experiences? We've got you covered. The useRouter hook allows you to programmatically change routes inside Client Components. Recommendation: Use the component for navigation unless you have a specific requirement for using useRouter. 'use client' import { useRouter } from 'next/navigation' export default function Page() { const router = useRouter() return (router.push('/dashboard')}> Dashboard) } useRouter() router.push(href: string, { scroll: boolean }): Perform a client-side navigation to the provided route. Adds a new entry into the browser's history stack. router.replace(href: string, { scroll: boolean }): Perform a client-side navigation to the provided route without adding a new entry into the browser's history stack. router.refresh(): Refresh the current route. Making a new request to the server, re-fetching data requests, and re-rendering Server Components. The client will merge the updated React Server Component payload without losing unaffected client-side React (e.g. useState) or browser state (e.g. scroll position). router.prefetch(href: string): Prefetch the provided route for faster client-side transitions. router.back(): Navigate back to the previous route in the browser's history stack. router.forward(): Navigate forwards to the next page in the browser's history stack. Good to know: You must not send untrusted or unsanitized URLs to router.push or router.replace, as this can open your site to cross-site scripting (XSS) vulnerabilities. For example, javascript: URLs sent to router.push or router.replace will be executed in the context of your page. The component automatically prefetch routes as they become visible in the viewport. refresh() could re-produce the same result if fetch requests are cached. Other Dynamic APIs like cookies and headers could also change the response. Migrating from next/router The useRouter hook should be imported from next/navigation and not next/router when using the App Router The pathname string has been removed and is replaced by usePathname() The query object has been removed and is replaced by useSearchParams() router.events has been replaced. See below. View the full migration guide. Examples Router events You can listen for page changes by composing other Client Component hooks like usePathname and useSearchParams. 'use client' import { useEffect } from 'react' import { usePathname, useSearchParams } from 'next/navigation' export function NavigationEvents() { const pathname = usePathname() const searchParams = useSearchParams() useEffect(() => { const url = `${pathname}?${searchParams}` console.log(url) // You can now use the current URL // ... }, [pathname, searchParams]) return '...' } Which can be imported into a layout. import { Suspense } from 'react' import { NavigationEvents } from './components/navigation-events' export default function Layout({ children }) { return ({children}) } Good to know: is wrapped in a Suspense boundary becauseuseSearchParams() causes client-side rendering up to the closest Suspense boundary during static rendering. Learn more. By default, Next.js will scroll to the top of the page when navigating to a new route. You can disable this behavior by passing scroll: false to router.push() or router.replace(). 'use client' import { useRouter } from 'next/navigation' export default function Page() { const router = useRouter() return (router.push('/dashboard', { scroll: false })} > Dashboard) } Version History VersionChangesv13.0.0useRouter from next/navigation introduced. Next.js BoilerplateA Next.js starter from create-next-app.Image Gallery StarterAn image gallery built on Next.js and Cloudinary.Next.js CommerceAn all-in-one starter kit for high-performance ecommerce sites. 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 ({children}) } 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 - 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 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 (router.push('/about')}> Click me) } Navigating pages/post/[pid].js, which is a dynamic route: import { useRouter } from 'next/router' export default function Page() { const router = useRouter() return (router.push('/post/abc')}> Click me) } 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 Redirecting... } 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. 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 (Page: {router.query.slug} Count: {count} setCount(count + 1)}>Increase count one two) } 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: import { useRouter } from 'next/router' export default function MyApp({ Component, pageProps }) { const router = useRouter() return } 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: import { useRouter } from 'next/router' export default function ReadMore({ post }) { const router = useRouter() return ({ router.push({ pathname: '/post/[pid]', query: { pid: post.id }, }) }} > Click here to read more) } router.replace Similar to the replace prop in next/link, router.replace will prevent adding a new URL entry into the history stack. 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: import { useRouter } from 'next/router' export default function Page() { const router = useRouter() return (router.replace('/home')}> Click me) } 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. 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: 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 fields */} Login) } 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 Welcome to the page } 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 (router.back()}> Click here to go back) } 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 (router.reload()}> Click here to reload) } router.events 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 Good to know: 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 } 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 } The next/compat/router export This is the same useRouter hook, but can be used in both app and pages directories. It differs from next/router in that it does not throw an error when the pages router is not mounted, and instead has a return type of NextRouter | null. This allows developers to convert components to support running in both app and pages as they transition to the app router. A component that previously looked like this: import { useRouter } from 'next/router' const MyComponent = () => { const { isReady, query } = useRouter() // ... } Will error when converted over to next/compat/router, as null can not be destructured. Instead, developers will be able to take advantage of new hooks: import { useEffect } from 'react' import { useRouter } from 'next/compat/router' import { useSearchParams } from 'next/navigation' const MyComponent = () => { const router = useRouter() // may be null or a NextRouter instance const searchParams = useSearchParams() useEffect(() => { if (router && !router.isReady) { return } // In `app/`, searchParams will be ready immediately with the values, in // `pages/` it will be available after the router is ready. const search = searchParams.get('search') // ... }, [router, searchParams]) // ... } This component will now work in both pages and app directories. When the component is no longer used in pages, you can remove the references to the compat router: import { useSearchParams } from 'next/navigation' const MyComponent = () => { const searchParams = useSearchParams() // As this component is only used in `app/`, the compat router can be removed. const search = searchParams.get('search') // ... } Using useRouter outside of Next.js context in pages Another specific use case is when rendering components outside of a Next.js application context, such as inside getServerSideProps on the pages directory. In this case, the compat router can be used to avoid errors: import { renderToString } from 'react-dom/server' import { useRouter } from 'next/compat/router' const MyComponent = () => { const router = useRouter() // may be null or a NextRouter instance // ... } export async function getServerSideProps() { const renderedComponent = renderToString() return { props: { renderedComponent, }, } } 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 Redirecting... } 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 {router.pathname} } 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 { render() { return {this.props.router.pathname} } } export default withRouter(MyComponent) Used by some of the world's largest companies, Next.js enables you to create full-stack web applications by extending the latest React features, and integrating powerful Rust-based JavaScript tooling for the fastest builds. Visit to view the full documentation. The Next.js community can be found on GitHub Discussions where you can ask questions, voice ideas, and share your projects with other people. To chat with other community members you can join the Next.js Discord server. Do note that our Code of Conduct applies to all Next.js community channels. Users are highly encouraged to read and adhere to them to avoid repercussions. Contributions to Next.js are welcome and highly appreciated. However, before you jump right into it, we would like you to review our Contribution Guidelines to make sure you have a smooth experience contributing to Next.js. We have a list of good first issues that contain bugs that have a relatively limited scope. This is a great place for newcomers and beginners alike to get started, gain experience, and get familiar with our contribution process. If you believe you have found a security vulnerability in Next.js, we encourage you to responsibly disclose this and NOT open a public issue. We will investigate all legitimate reports. Our preference is that you make use of GitHub's private vulnerability reporting feature to disclose potential security vulnerabilities in our Open Source Software. To do this, please visit and click the "Report a vulnerability" button. Operating System: Platform: linux Arch: x64 Version: #22 SMP Tue Jan 10 18:39:00 UTC 2023 Binaries: Node: 18.14.2 npm: 9.5.0 Yarn: 1.22.19 pnpm: N/A Relevant packages: next: 13.3.1 eslint-config-next: 13.3.1 react: 18.2.0 react-dom: 18.2.0 App directory (appDir: true), Routing (next/router, next/navigation, next/link) Link to the code that reproduces this issue To Reproduce use this codesandbox link. write a letter to the input you will see a refresh and the query string will be removed Describe the Bug useRouter's replace method is not working as expected. in Nextjs 13.2.4 it was working correctly. when you type something in the search it starts the transition using useTransition hook. and replaces the url with the provided search value. but in Nextjs version 13.3.1 it is not working. it is also not working in latest canary relase (13.3.2-canary.7) Expected Behavior Not remove the search params and replace the url. Which browser are you using? (if relevant) Google Chrome Version 112.0.5615.137 (Official Build) (x86_64) How are you deploying your application? (if relevant) No response Authentication can be a bit sketchy sometimes as we have to keep so much in mind, like session management, protecting several routes/pages, hashing passwords, validating user's credentials during sign-up and sign-in. Also, creating an authentication from scratch can be a lot of work. If you're working with Next.JS, then you should try using Next-Auth as it provides many authentication schemes like JWT, cookie, etc. And also using third-party authentication providers like Google, Facebook, and (yes!) even with Discord. Also, next-auth helps in session management so that the server can't be tricked easily. Providers aside, we will be looking into setting up authentication based on users' credentials like email and password. Things to consider during authentication Client form validation Server form value validation Hashing users' passwords during sign-up for the obvious reason Storing into a database Checking of the hashed password during sign-in Protecting routes for the non-authenticated user Proper error handling for both frontend and backend Packages we need I am using Next.js as the framework for the demonstration. Along with that next-auth for authentication bycryptjs for hashing the passwords mongodb for MongoDB functions NOTE This is not a frontend tutorial so I'll not be covering any notifications on successful events and/or CSS stuff. Website scaffolding The website is very simple consisting of 4 pages and obviously a navbar for better demonstration: Install packages and setting up database npm i next-auth mongodb bcryptjs During install, we will sign up for a free MongoDB account on their website. Now, we can connect to that database using the connect code from their dashboard. We should use the MongoURL from inside of a .env.local file for more refined and secure code. Sign Up Route Before sign-in, users need to signup for that particular website. NextJS provides us to write API codes in the pages/api folder using the NodeJS environment. It will also follow the same folder-structured route. For the sign-up route, we will create a route pages/api/auth/signup.js. We also need to make sure that only the POST method is accepted and nothing else. Things to do in the signup route Get users credentials Validate Send error code if any Connect to database Check if any existing user is present with the same email address Hash password using bycrypt js bycrypt js returns a Promise during hashing of password so we need to await for the response. password: await hash(password, 12) //hash(plain text, no. of salting rounds) If all goes well, send a response and close connection with the database import { MongoClient } from 'mongodb'; import { hash } from 'bcryptjs'; async function handler(req, res) { //Only POST mothod is accepted if (req.method === 'POST') { //Getting email and password from body const { email, password } = req.body; //Validate if (!email || !email.includes('@') || !password) { res.status(422).json({ message: 'Invalid Data' }); return; } //Connect with database const client = await MongoClient.connect(`mongodb+srv://${process.env.MONGO_USER}:${process.env.MONGO_PASS}@${process.env.MONGO_CLUSTER}.n4tnm.mongodb.net/${process.env.MONGO_DB}?retryWrites=true&w=majority`, { useNewUrlParser: true, useUnifiedTopology: true }); const db = client.db(); //Check existing const checkExisting = await db .collection('users') .findOne({ email: email }); //Send error response if duplicate user is found if (checkExisting) { res.status(422).json({ message: 'User already exists' }); client.close(); return; } //Hash password const status = await db.collection('users').insertOne({ email, password: await hash(password, 12), }); //Send success response res.status(201).json({ message: 'User created', ...status }); //Close DB connection client.close(); } else { //Response for other than POST method res.status(500).json({ message: 'Route not valid' }); } } export default handler; Now that our signup route is in place, it's time to connect the frontend to the backend. Posting Sign Up form import { signIn } from 'next-auth/client'; //... const onFormSubmit = async (e) => { e.preventDefault(); //Getting value from useRef() const email = emailRef.current.value; const password = passwordRef.current.value; //Validation if (!email || !email.includes('@') || !password) { alert('Invalid details'); return; } //POST form values const res = await fetch('/api/auth/signup', { method: 'POST', headers: { 'Content-Type': 'application/json', }, body: JSON.stringify({ email: email, password: password, }), }); //Await for data for any desirable next steps const data = await res.json(); console.log(data); }; //... With the Sign Up login in place, let's work with the Sign In logic. Sign In using Next-Auth Next-Auth provides us with Client API as well as REST API The NextAuth.js client library makes it easy to interact with sessions from React applications. NextAuth.js exposes a REST API which is used by the NextAuth.js client. We will use both for signing in the users. To add NextAuth.js to a project create a file called [...nextauth].js in pages/api/auth. All requests to /api/auth/* (signin, callback, signout, etc) will automatically be handed by NextAuth.js. With this help from next-auth, we need to implement our own sign-in logic for checking users stored on the database. Things to do in the sign-in route: Configure to use JWT Specify provider from next-auth (Credentials) For more providers, check Connect to database Check if the user is present Send error response if any Compare hashed password with the password stored on DB Send response Close DB connection In [...nextauth].js: import NextAuth from 'next-auth'; import Providers from 'next-auth/providers'; import { MongoClient } from 'mongodb'; import { compare } from 'bcryptjs'; export default NextAuth({ //Configure JWT session: { jwt: true, }, //Specify Provider providers: [Providers.Credentials({ async authorize(credentials) { //Connect to DB const client = await MongoClient.connect(`mongodb+srv://${process.env.MONGO_USER}:${process.env.MONGO_PASS}@${process.env.MONGO_CLUSTER}.n4tnm.mongodb.net/${process.env.MONGO_DB}?retryWrites=true&w=majority`, { useNewUrlParser: true, useUnifiedTopology: true }); //Get all the users const users = await client.db().collection('users'); //Find user with the email const result = await users.findOne({ email: credentials.email, }); //Not found - send error res if (!result) { client.close(); throw new Error('No user found with the email'); } //Check hased password with DB password const checkPassword = await compare(credentials.passowrd, result.passowrd); //Incorrect password - send response if (!checkPassword) { client.close(); throw new Error('Password doesnt match'); } //Else send success response client.close(); return { email: result.email }; }, }),], }); This guide will help you: Upgrading Node.js Version The minimum Node.js version is now v18.17. See the Node.js documentation for more information. Next.js Version To update to Next.js version 13, run the following command using your preferred package manager: npm install next@latest react@latest react-dom@latest ESLint Version If you're using ESLint, you need to upgrade your ESLint version: npm install -D eslint-config-next@latest Good to know: You may need to restart the ESLint server in VS Code for the ESLint changes to take effect. Open the Command Palette (cmd+shift+p on Mac; ctrl+shift+p on Windows) and search for ESLint: Restart ESLint Server. Next Steps After you've updated, see the following sections for next steps: Upgrading New Features Next.js 13 introduced the new App Router with new features and conventions. The new Router is available in the app directory and co-exists with the pages directory. Upgrading to Next.js 13 does not require using the App Router. You can continue using pages with new features that work in both directories, such as the updated Image component, Link component, Script component, and Font optimization. Component Next.js 12 introduced new improvements to the Image Component with a temporary import: next/future/image. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading. In version 13, this new behavior is now the default for next/image. There are two codemods to help you migrate to the new Image Component: next-image-to-legacy-image codemod: Safely and automatically renames next/image imports to next/legacy/image. Existing components will maintain the same behavior. next-image-experimental codemod: Dangerously adds inline styles and removes unused props. This will change the behavior of existing components to match the new defaults. To use this codemod, you need to run the next-image-to-legacy-image codemod first. The Component no longer requires manually adding an tag as a child. This behavior was added as an experimental option in version 12.2 and is now the default. In Next.js 13, always renders and allows you to forward props to the underlying tag. For example: import Link from 'next/link' // Next.js 12: `` has to be nested otherwise it's excluded About // Next.js 13: `` always renders `` under the hood About To upgrade your links to Next.js 13, you can use the new-link codemod. Component The behavior of next/script has been updated to support both pages and app, but some changes need to be made to ensure a smooth migration: Move any beforeInteractive scripts you previously included in _document.js to the root layout file (app/layout.tsx). The experimental worker strategy does not yet work in app and scripts denoted with this strategy will either have to be removed or modified to use a different strategy (e.g. lazyOnload). onLoad, onReady, and onError handlers will not work in Server Components so make sure to move them to a Client Component or remove them altogether. Font Optimization Previously, Next.js helped you optimize fonts by inlining font CSS. Version 13 introduces the new next/font module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy. next/font is supported in both the pages and app directories. While inlining CSS still works in pages, it does not work in app. You should use next/font instead. See the Font Optimization page to learn how to use next/font. Migrating from pages to app Watch: Learn how to incrementally adopt the App Router → YouTube (16 minutes). Moving to the App Router may be the first time using React features that Next.js builds on top of such as Server Components, Suspense, and more. When combined with new Next.js features such as special files and layouts, migration means new concepts, mental models, and behavioral changes to learn. We recommend reducing the combined complexity of these updates by breaking down your migration into smaller steps. The app directory is intentionally designed to work simultaneously with the pages directory to allow for incremental page-by-page migration. The app directory supports nested routes and layouts. Learn more. Use nested folders to define routes and a special page.js file to make a route segment publicly accessible. Learn more. Special file conventions are used to create UI for each route segment. The most common special files are page.js and layout.js. Use page.js to define UI unique to a route. Use layout.js to define UI that is shared across multiple routes. .js, .jsx, or .tsx file extensions can be used for special files. You can colocate other files inside the app directory such as components, styles, tests, and more. Learn more. Data fetching functions like getServerSideProps and getStaticProps have been replaced with a new API inside app. getStaticPaths has been replaced with generateStaticParams. pages/_app.js and pages/_document.js have been replaced with a single app/layout.js root layout. Learn more. pages/_error.js has been replaced with more granular error.js special files. Learn more. pages/404.js has been replaced with the not-found.js file. pages/api/* API Routes have been replaced with the route.js (Route Handler) special file. Step 1: Creating the app directory Update to the latest Next.js version (requires 13.4 or greater): Then, create a new app directory at the root of your project (or src/ directory). Step 2: Creating a Root Layout Create a new app/layout.tsx file inside the app directory. This is a root layout that will apply to all routes inside app. export default function RootLayout({ // Layouts must accept a children prop. // This will be populated with nested layouts or pages children, }: { children: React.ReactNode }) { return ({children}) } The app directory must include a root layout. The root layout must define , and tags since Next.js does not automatically create them The root layout replaces the pages/_app.tsx and pages/_document.tsx files. .js, .jsx, or .tsx extensions can be used for layout files. To manage HTML elements, you can use the built-in SEO support: import type { Metadata } from 'next' export const metadata: Metadata = { title: 'Home', description: 'Welcome to Next.js', } Migrating _document.js and _app.js If you have an existing _app or _document file, you can copy the contents (e.g. global styles) to the root layout (app/layout.tsx). Styles in app/layout.tsx will not apply to pages/*. You should keep _app/_document while migrating to prevent your pages/* routes from breaking. Once fully migrated, you can then safely delete them. If you are using any React Context providers, they will need to be moved to a Client Component. Migrating the getLayout() pattern to Layouts (Optional) Next.js recommended adding a property to Page components to achieve per-page layouts in the pages directory. This pattern can be replaced with native support for nested layouts in the app directory. See before and after exampleBeforeexport default function DashboardLayout({ children }) { return (

My Dashboard

{children}) }import DashboardLayout from '../components/DashboardLayout' export default function Page() { return My Page } Page.getLayout = function getLayout(page) { return {page} }After Remove the Page.getLayout property from pages/dashboard/index.js and follow the steps for migrating pages to the app directory. export default function Page() { return My Page } Move the contents of DashboardLayout into a new Client Component to retain pages directory behavior. 'use client' // this directive should be at top of the file, before any imports. // This is a Client Component export default function DashboardLayout({ children }) { return (

My Dashboard

{children}) } Import the DashboardLayout into a new layout.js file inside the app directory. import DashboardLayout from './DashboardLayout' // This is a Server Component export default function Layout({ children }) { return {children} } You can incrementally move non-interactive parts of DashboardLayout.js (Client Component) into layout.js (Server Component) to reduce the amount of component JavaScript you send to the client. In the pages directory, the next/head React component is used to manage HTML elements such as title and meta . In the app directory, next/head is replaced with the new built-in SEO support. Before: import Head from 'next/head' export default function Page() { return (My page title) } After: import type { Metadata } from 'next' export const metadata: Metadata = { title: 'My Page Title', } export default function Page() { return '...' } See all metadata options. Step 4: Migrating Pages Pages in the app directory are Server Components by default. This is different from the pages directory where pages are Client Components. Data fetching has changed in app. getServerSideProps, getStaticProps and getInitialProps have been replaced with a simpler API. The app directory uses nested folders to define routes and a special page.js file to make a route segment publicly accessible. pages Directoryapp DirectoryRouteindex.jspage.js/about.jsabout/page.js/aboutblog/[slug].jsblog/[slug]/page.js/blog/post-1 We recommend breaking down the migration of a page into two main steps: Step 1: Move the default exported Page Component into a new Client Component. Step 2: Import the new Client Component into a new page.js file inside the app directory. Good to know: This is the easiest migration path because it has the most comparable behavior to the pages directory. Step 1: Create a new Client Component Create a new separate file inside the app directory (i.e. app/home-page.tsx or similar) that exports a Client Component. To define Client Components, add the 'use client' directive to the top of the file (before any imports). Similar to the Pages Router, there is an optimization step to prerender Client Components to static HTML on the initial page load. Move the default exported page component from pages/index.js to app/home-page.tsx. 'use client' // This is a Client Component (same as components in the `pages` directory) // It receives data as props, has access to state and effects, and is // prerendered on the server during the initial page load. export default function HomePage({ recentPosts }) { return ({recentPosts.map((post) => ({post.title}))}) } Step 2: Create a new page Create a new app/page.tsx file inside the app directory. This is a Server Component by default. Import the home-page.tsx Client Component into the page. If you were fetching data in pages/index.js, move the data fetching logic directly into the Server Component using the new data fetching APIs. See the data fetching upgrade guide for more details. // Import your Client Component import HomePage from './home-page' async function getPosts() { const res = await fetch('https://...') const posts = await res.json() return posts } export default async function Page() { // Fetch data directly in a Server Component const recentPosts = await getPosts() // Forward fetched data to your Client Component return } If your previous page used useRouter, you'll need to update to the new routing hooks. Learn more. Start your development server and visit . You should see your existing index route, now served through the app directory. Step 5: Migrating Routing Hooks A new router has been added to support the new behavior in the app directory. In app, you should use the three new hooks imported from next/navigation: useRouter(), usePathname(), and useSearchParams(). The new useRouter hook is imported from next/navigation and has different behavior to the useRouter hook in pages which is imported from next/router. The new useRouter does not return the pathname string. Use the separate usePathname hook instead. The new useRouter does not return the query object. Search parameters and dynamic route parameters are now separate. Use the useSearchParams and useParams hooks instead. You can use useSearchParams and usePathname together to listen to page changes. See the Router Events section for more details. These new hooks are only supported in Client Components. They cannot be used in Server Components. 'use client' import { useRouter, usePathname, useSearchParams } from 'next/navigation' export default function ExampleClientComponent() { const router = useRouter() const pathname = usePathname() const searchParams = useSearchParams() // ... } In addition, the new useRouter hook has the following changes: isFallback has been removed because fallback has been replaced. The locale, locales, defaultLocales, domainLocales values have been removed because built-in i18n Next.js features are no longer necessary in the app directory. Learn more about i18n. basePath has been removed. The alternative will not be part of useRouter. It has not yet been implemented. asPath has been removed because the concept of as has been removed from the new router. isReady has been removed because it is no longer necessary. During static rendering, any component that uses the useSearchParams() hook will skip the prerendering step and instead be rendered on the client at runtime. route has been removed. usePathname or useSelectedLayoutSegments() provide an alternative. View the useRouter() API reference. Sharing components between pages and app To keep components compatible between the pages and app routers, refer to the useRouter hook from next/compat/router. This is the useRouter hook from the pages directory, but intended to be used while sharing components between routers. Once you are ready to use it only on the app router, update to the new useRouter from next/navigation. Step 6: Migrating Data Fetching Methods The pages directory uses getServerSideProps and getStaticProps to fetch data for pages. Inside the app directory, these previous data fetching functions are replaced with a simpler API built on top of fetch() and async React Server Components. export default async function Page() { // This request should be cached until manually invalidated. // Similar to `getStaticProps`. // `force-cache` is the default and can be omitted. const staticData = await fetch(`https://...`, { cache: 'force-cache' }) // This request should be refetched on every request. // Similar to `getServerSideProps`. const dynamicData = await fetch(`https://...`, { cache: 'no-store' }) // This request should be cached with a lifetime of 10 seconds. // Similar to `getStaticProps` with the `revalidate` option. const revalidatedData = await fetch(`https://...`, { next: { revalidate: 10 }, }) return ... } In the pages directory, getServerSideProps is used to fetch data on the server and forward props to the default exported React component in the file. The initial HTML for the page is prerendered from the server, followed by "hydrating" the page in the browser (making it interactive). // `pages` directory export async function getServerSideProps() { const res = await fetch(`https://...`) const projects = await res.json() return { props: { projects } } } export default function Dashboard({ projects }) { return ({projects.map((project) => ({project.name}))}) } In the App Router, we can colocate our data fetching inside our React components using Server Components. This allows us to send less JavaScript to the client, while maintaining the rendered HTML from the server. By setting the cache option to no-store, we can indicate that the fetched data should never be cached. This is similar to getServerSideProps in the pages directory. // `app` directory // This function can be named anything async function getProjects() { const res = await fetch(`https://...`, { cache: 'no-store' }) const projects = await res.json() return projects } export default async function Dashboard() { const projects = await getProjects() return ({projects.map((project) => ({project.name}))}) } Accessing Request Object In the pages directory, you can retrieve request-based data based on the Node.js HTTP API. For example, you can retrieve the req object from getServerSideProps and use it to retrieve the request's cookies and headers. // `pages` directory export async function getServerSideProps({ req, query }) { const authHeader = req.getHeaders()['authorization']; const theme = req.cookies['theme']; return { props: { ... }} } export default function Page(props) { return ... } The app directory exposes new read-only functions to retrieve request data: // `app` directory import { cookies, headers } from 'next/headers' async function getData() { const authHeader = (await headers()).get('authorization') return '...' } export default async function Page() { // You can use `cookies` or `headers` inside Server Components // directly or in your data fetching function const theme = (await cookies()).get('theme') const data = await getData() return '...' } Static Site Generation (getStaticProps) In the pages directory, the getStaticProps function is used to pre-render a page at build time. This function can be used to fetch data from an external API or directly from a database, and pass this data down to the entire page as it's being generated during the build. // `pages` directory export async function getStaticProps() { const res = await fetch(`https://...`) const projects = await res.json() return { props: { projects } } } export default function Index({ projects }) { return projects.map((project) => {project.name}) } In the app directory, data fetching with fetch() will default to cache: 'force-cache', which will cache the request data until manually invalidated. This is similar to getStaticProps in the pages directory. // `app` directory // This function can be named anything async function getProjects() { const res = await fetch(`https://...`) const projects = await res.json() return projects } export default async function Index() { const projects = await getProjects() return projects.map((project) => {project.name}) } Dynamic paths (getStaticPaths) In the pages directory, the getStaticPaths function is used to define the dynamic paths that should be pre-rendered at build time. // `pages` directory import PostLayout from '@/components/post-layout' export async function getStaticPaths() { return { paths: [{ params: { id: '1' } }, { params: { id: '2' } }], } } export async function getStaticProps({ params }) { const res = await fetch(`https://.../posts/${params.id}`) const post = await res.json() return { props: { post } } } export default function Post({ post }) { return } In the app directory, getStaticPaths is replaced with generateStaticParams. generateStaticParams behaves similarly to getStaticPaths, but has a simplified API for returning route parameters and can be used inside layouts. The return shape of generateStaticParams is an array of segments instead of an array of nested param objects or a string of resolved paths. // `app` directory import PostLayout from '@/components/post-layout' export async function generateStaticParams() { return [{ id: '1' }, { id: '2' }] } async function getPost(params) { const res = await fetch(`https://.../posts/${(await params).id}`) const post = await res.json() return post } export default async function Post({ params }) { const post = await getPost(params) return } Using the name generateStaticParams is more appropriate than getStaticPaths for the new model in the app directory. The get prefix is replaced with a more descriptive generate, which sits better alone now that getStaticProps and getServerSideProps are no longer necessary. The Paths suffix is replaced by Params, which is more appropriate for nested routing with multiple dynamic segments. Replacing fallback In the pages directory, the fallback property returned from getStaticPaths is used to define the behavior of a page that isn't pre-rendered at build time. This property can be set to true to show a fallback page while the page is being generated, false to show a 404 page, or blocking to generate the page at request time. // `pages` directory export async function getStaticPaths() { return { paths: [], fallback: 'blocking' }; } export async function getStaticProps({ params }) { ... } export default function Post({ post }) { return ... } In the app directory the config.dynamicParams property controls how params outside of generateStaticParams are handled: true: (default) Dynamic segments not included in generateStaticParams are generated on demand. false: Dynamic segments not included in generateStaticParams will return a 404. This replaces the fallback: true | false | 'blocking' option of getStaticPaths in the pages directory. The fallback: 'blocking' option is not included in dynamicParams because the difference between 'blocking' and true is negligible with streaming. // `app` directory export const dynamicParams = true; export async function generateStaticParams() { return [...] } async function getPost(params) { ... } export default async function Post({ params }) { const post = await getPost(params); return ... } With dynamicParams set to true (the default), when a route segment is requested that hasn't been generated, it will be server-rendered and cached. In the pages directory, the getStaticProps function allows you to add a revalidate field to automatically regenerate a page after a certain amount of time. // `pages` directory export async function getStaticProps() { const res = await fetch(`https://.../posts`) const posts = await res.json() return { props: { posts }, revalidate: 60, } } export default function Index({ posts }) { return ( ) } In the app directory, data fetching with fetch() can use revalidate, which will cache the request for the specified amount of seconds. // `app` directory async function getPosts() { const res = await fetch(`https://.../posts`, { next: { revalidate: 60 } }) const data = await res.json() return data.posts } export default async function PostList() { const posts = await getPosts() return posts.map((post) => {post.name}) } API Routes API Routes continue to work in the pages/api directory without any changes. However, they have been replaced by Route Handlers in the app directory. Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs. export async function GET(request: Request) {} Good to know: If you previously used API routes to call an external API from the client, you can now use Server Components instead to securely fetch data. Learn more about data fetching. Single-Page Applications If you are also migrating to Next.js from a Single-Page Application (SPA) at the same time, see our documentation to learn more. Step 7: Styling In the pages directory, global stylesheets are restricted to only pages/_app.js. With the app directory, this restriction has been lifted. Global styles can be added to any layout, page, or component. CSS Modules Tailwind CSS Global Styles CSS-in-JS External Stylesheets Sass Tailwind CSS If you're using Tailwind CSS, you'll need to add the app directory to your tailwind.config.js file: module.exports = { content: ['./app/**/*.{js,ts,jsx,tsx,mdx}', //