From Pages to App

This guide will help you:

- Update your Next.js application from version 12 to version 13
- Upgrade features that work in both the `pages` and the `app` directories
- Incrementally migrate your existing application from `pages` to `app`

Upgrading

Node.js Version

The minimum Node.js version is now **v16.8**. 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:

Terminal

```
npm install next@latest react@latest react-dom@latest
```

ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

Terminal

```
npm install -D eslint-config-next@latest
```

Note: 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:

- Upgrade new features: A guide to help you upgrade to new features such as the improved Image and Link Components.
- Migrate from the `pages` to `app` directory: A step-by-step guide to help you incrementally migrate from the `pages` to the `app` directory.

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 new 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.

`<Image/>` 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.

`<Link>` Component

The `<Link>` Component no longer requires manually adding an `<a>` 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, `<Link>` always renders `<a>` and allows you to forward props to the underlying tag.

For example:

```
import Link from 'next/link'

// Next.js 12: `<a>` has to be nested otherwise it's excluded
<Link href="/about">
  <a>About</a>
</Link>

// Next.js 13: `<Link>` always renders `<a>` under the hood
<Link href="/about">
  About
</Link>
```

To upgrade your links to Next.js 13, you can use the `new-link` codemod.

`<Script>` 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`

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.
- You can colocate other files inside the `app` directory such as components, styles, tests, and more. Learn more.
- `pages/api/*` currently remain inside the `pages` directory.

Step 1: Creating the `app` directory

Update to the latest Next.js version (requires 13.4 or greater):

```
npm install next@latest
```

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`.

app/layout.tsx

```tsx
export default function RootLayout({
  // Layouts must accept a children prop.
  // This will be populated with nested layouts or pages
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

- The `app` directory **must** include a root layout.
- The root layout must define `<html>`, and `<body>` 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 `<head>` HTML elements, you can use the built-in SEO support:

app/layout.tsx

```tsx
import { 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 example**

Step 3: Migrating `next/head`

In the `pages` directory, the `next/head` React component is used to manage `<head>` HTML elements such as `title` and `meta`. In the `app` directory, `next/head` is replaced with the new built-in SEO support.

Before:

pages/index.tsx

```tsx
import Head from 'next/head';

export default function Page() {
  return (
    <>
      <Head>
        <title>My page title</title>
      </Head>
    </>
  );
}
```

After:

app/page.tsx

```tsx
import { 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 for 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` Directory	`app` Directory	Route
`index.js`	`page.js`	`/`
`about.js`	`about/page.js`	`/about`
`blog/[slug].js`	`blog/[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.

Note: 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).
- Move the default exported page component from `pages/index.js` to `app/home-page.tsx`.

app/home-page.tsx

```tsx
'use client';

// This is a Client Component. It receives data as props and
// has access to state and effects just like Page components
// in the `pages` directory.
export default function HomePage({ recentPosts }) {
  return (
    <div>
      {recentPosts.map((post) => (
        <div key={post.id}>{post.title}</div>
      ))}
```

```
    </div>
  );
}
```

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.

app/page.tsx

```tsx
// 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 <HomePage recentPosts={recentPosts} />;
}
```

- If your previous page used `useRouter`, you'll need to update to the new routing hooks. Learn more.
- Start your development server and visit `http://localhost:3000`. 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 `useRouter` hook imported from `next/router` is not supported in the `app` directory but can continue to be used in the `pages` directory.
- The new `useRouter` does not return the `pathname` string. Use the separate `usePathname` hook instead.
- The new `useRouter` does not return the `query` object. Use the separate `useSearchParams` hook 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.

app/example-client-component.tsx

```tsx
'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.

View the `useRouter()` API reference.

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.

app/page.tsx

```tsx
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 <div>...</div>;
}
```

Server-side Rendering (`getServerSideProps`)

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/dashboard.js

```jsx
// `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 (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  );
```

```
}
```

In the `app` directory, 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/dashboard/page.tsx

```tsx
// `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 (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  );
}
```

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/index.js

```jsx
// `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:

- `headers()`: Based on the Web Headers API, and can be used inside Server Components to retrieve request headers.
- `cookies()`: Based on the Web Cookies API, and can be used inside Server Components to retrieve cookies.

app/page.tsx

```tsx
// `app` directory
import { cookies, headers } from 'next/headers';

async function getData() {
  const authHeader = 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 = 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/index.js

```jsx
// `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) => <div>{project.name}</div>);
}
```

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/page.js

```
// `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) => <div>{project.name}</div>);
}
```

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/posts/[id].js

```
// `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 <PostLayout post={post} />;
}
```

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/posts/[id]/page.js

```
// `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/${params.id}`);
  const post = await res.json();

  return post;
}

export default async function Post({ params }) {
  const post = await getPost(params);

  return <PostLayout post={post} />;
}
```

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/posts/[id].js

```
// `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/posts/[id]/page.js

```
// `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 as static data on success.

Incremental Static Regeneration (`getStaticProps` with `revalidate`)

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. This is called Incremental Static Regeneration (ISR) and helps you update static content without redeploying.

pages/index.js

```
// `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 (
    <Layout>
      <PostList posts={posts} />
    </Layout>
  );
}
```

In the `app` directory, data fetching with `fetch()` can use `revalidate`, which will cache the request for the specified amount of seconds.

app/page.js

```
// `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) => <div>{post.name}</div>);
```

```
14  }
```

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.

app/api/route.ts

```ts
export async function GET(request: Request) {}
```

Note: 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.

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:

tailwind.config.js

```
module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}', // <-- Add this line
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',
  ],
};
```

You'll also need to import your global styles in your `app/layout.js` file:

app/layout.js

```
import '../styles/globals.css';

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

Learn more about styling with Tailwind CSS

Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See Codemods for more information.