Layouts and Templates

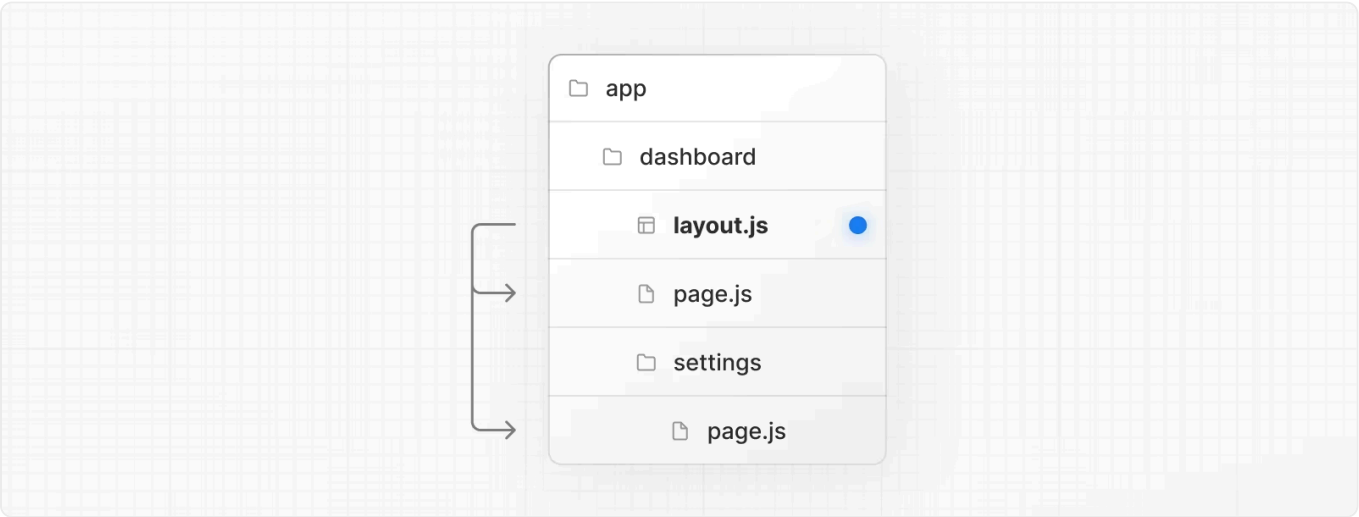
The special files layout.js and template.js allow you to create UI that is shared between routes. This page will guide you through how and when to use these special files.

Layouts

A layout is UI that is **shared** between multiple routes. On navigation, layouts preserve state, remain interactive, and do not re-render. Layouts can also be nested.

You can define a layout by default exporting a React component from a `layout.js` file. The component should accept a `children` prop that will be populated with a child layout (if it exists) or a page during rendering.

For example, the layout will be shared with the `/dashboard` and `/dashboard/settings` pages:



app/dashboard/layout.tsx TypeScript

```typescript
export default function DashboardLayout({
  children, // will be a page or nested layout
}: {
  children: React.ReactNode
}) {
  return (
    <section>
      {/* Include shared UI here e.g. a header or sidebar */}
      <nav></nav>

      {children}
    </section>
  )
}
```

Root Layout (Required)

The root layout is defined at the top level of the `app` directory and applies to all routes. This layout is **required** and must contain `html` and `body` tags, allowing you to modify the initial HTML returned from the server.

app/layout.tsx TypeScript

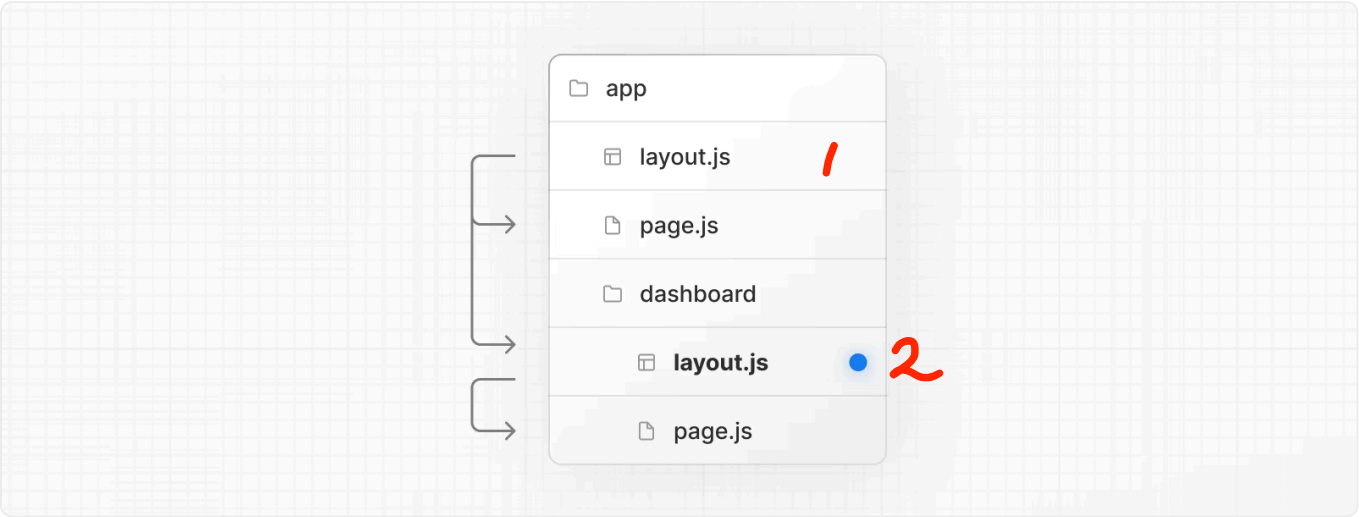
```typescript
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>
        {/* Layout UI */}
        <main>{children}</main>
      </body>
    </html>
  )
}
```

Nesting Layouts

By default, layouts in the folder hierarchy are **nested**, which means they wrap child layouts via their `children` prop. You can nest layouts by adding `layout.js` inside specific route

segments (folders).

For example, to create a layout for the `/dashboard` route, add a new `layout.js` file inside the `dashboard` folder:

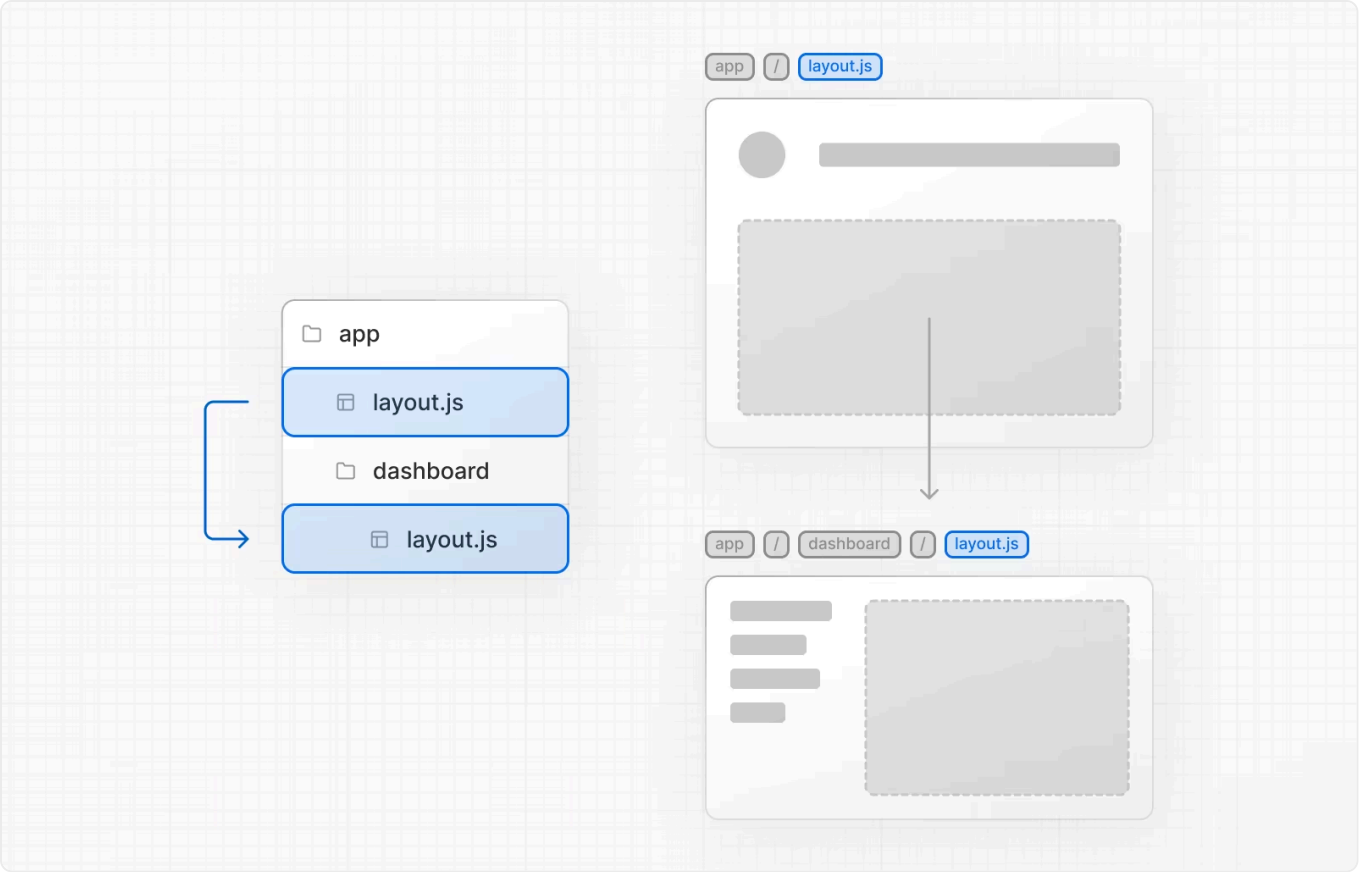


app/dashboard/layout.tsx TypeScript

```typescript
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

If you were to combine the two layouts above, the root layout (`app/layout.js`) would wrap the dashboard layout (`app/dashboard/layout.js`), which would wrap route segments inside `app/dashboard/*`.

The two layouts would be nested as such:



Good to know:

- `.js`, `.jsx`, or `.tsx` file extensions can be used for Layouts.
- Only the root layout can contain `<html>` and `<body>` tags.
- When a `layout.js` and `page.js` file are defined in the same folder, the layout will wrap the page.
- Layouts are Server Components by default but can be set to a Client Component.
- Layouts can fetch data. View the Data Fetching section for more information.
- Passing data between a parent layout and its children is not possible. However, you can fetch the same data in a route more than once, and React will automatically dedupe the requests without affecting performance.
- Layouts do not have access to `pathname` (learn more). But imported Client Components can access the pathname using `usePathname` hook.
- Layouts do not have access to the route segments below itself. To access all route segments, you can use `useSelectedLayoutSegment` or `useSelectedLayoutSegments` in a Client Component.
- You can use Route Groups to opt specific route segments in and out of shared layouts.
- You can use Route Groups to create multiple root layouts. See an example here.
- **Migrating from the `pages` directory:** The root layout replaces the `_app.js` and `_document.js` files. View the migration guide.

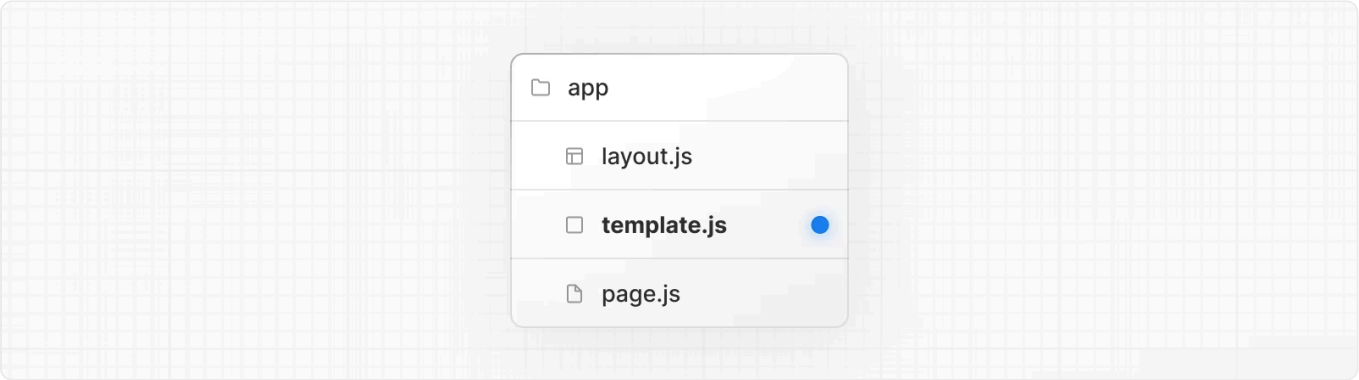
Templates

Templates are similar to layouts in that they wrap a child layout or page. Unlike layouts that persist across routes and maintain state, templates create a new instance for each of their children on navigation. This means that when a user navigates between routes that share a template, a new instance of the child is mounted, DOM elements are recreated, state is **not** preserved in Client Components, and effects are re-synchronized.

There may be cases where you need those specific ~~behaviors~~, and templates would be a more suitable option than layouts. For example:

- To resynchronize `useEffect` on navigation.
- To reset the state of a child Client Components on navigation.

A template can be defined by exporting a default React component from a `template.js` file. The component should accept a `children` prop.



app/template.tsx TypeScript

```typescript
export default function Template({ children }: { children: React.ReactNode }) {
  return <div>{children}</div>
}
```

In terms of nesting, `template.js` is rendered between a layout and its children. Here's a simplified output:

Output

```
<Layout>
  {/* Note that the template is given a unique key. */}
  <Template key={routeParam}>{children}</Template>
</Layout>
```

Examples

Metadata

You can modify the `<head>` HTML elements such as `title` and `meta` using the Metadata APIs.

Metadata can be defined by exporting a `metadata` object or `generateMetadata` function in a `layout.js` or `page.js` file.

app/page.tsx TypeScript

```
import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'Next.js',
}

export default function Page() {
  return '...'
}
```

Good to know: You should **not** manually add `<head>` tags such as `<title>` and `<meta>` to root layouts. Instead, use the Metadata API which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.

Learn more about available metadata options in the API reference.

Active Nav Links

You can use the usePathname() hook to determine if a nav link is active.

Since `usePathname()` is a client hook, you need to extract the nav links into a Client Component, which can be imported into your layout or template:

app/ui/nav-links.tsx TypeScript

```typescript
'use client'

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

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

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

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

app/layout.tsx TypeScript

```typescript
import { NavLinks } from '@/app/ui/nav-links'

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <html lang="en">
      <body>
        <NavLinks />
        <main>{children}</main>
      </body>
    </html>
  )
}
```

Previous
Pages

Next
Linking and Navigating

Was this helpful?