> Menu

App Router > ... > File Conventions > layout.js

layout.js

A **layout** is UI that is shared between routes.

app/dashboard/layout.tsx TypeScript

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

A **root layout** is the top-most layout in the root `app` directory. It is used to define the `<html>` and `<body>` tags and other globally shared UI.

app/layout.tsx TypeScript

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

Props

`children` (required)

Layout components should accept and use a `children` prop. During rendering, `children` will be populated with the route segments the layout is wrapping. These will primarily be the component of a child Layout (if it exists) or Page, but could also be other special files like Loading or Error when applicable.

`params` (optional)

The dynamic route parameters object from the root segment down to that layout.

Example	URL	`params`
`app/dashboard/[team]/layout.js`	`/dashboard/1`	`{ team: '1' }`
`app/shop/[tag]/[item]/layout.js`	`/shop/1/2`	`{ tag: '1', item: '2' }`
`app/blog/[...slug]/layout.js`	`/blog/1/2`	`{ slug: ['1', '2'] }`

For example:

app/shop/[tag]/[item]/layout.tsx TypeScript

```typescript
export default function ShopLayout({
  children,
  params,
}: {
  children: React.ReactNode
  params: {
    tag: string
    item: string
  }
}) {
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
```

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

Good to know

Root Layouts

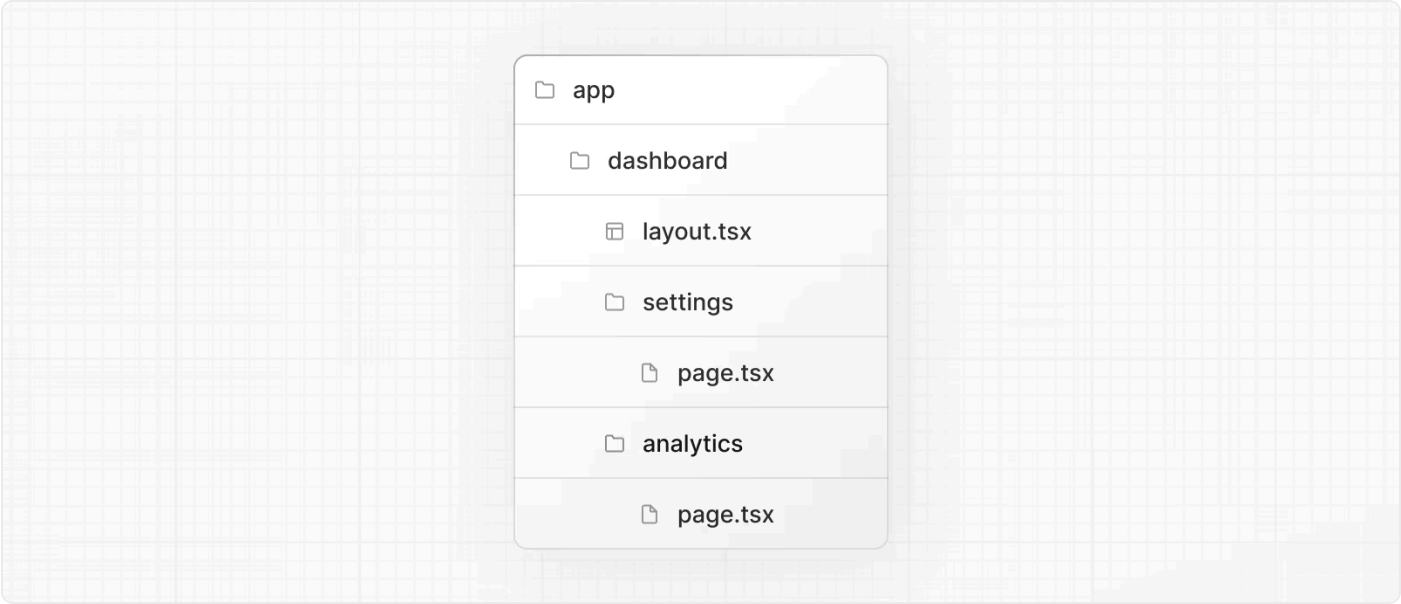
- The `app` directory **must** include a root `app/layout.js`.
- The root layout **must** define `<html>` and `<body>` tags.
  - You should **not** manually add `<head>` tags such as `<title>` and `<meta>` to root layouts. Instead, you should use the Metadata API which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.
- You can use route groups to create multiple root layouts.
  - Navigating **across multiple root layouts** will cause a **full page load** (as opposed to a client-side navigation). For example, navigating from `/cart` that uses `app/(shop)/layout.js` to `/blog` that uses `app/(marketing)/layout.js` will cause a full page load. This **only** applies to multiple root layouts.

Layouts do not receive `searchParams`

Unlike Pages, Layout components **do not** receive the `searchParams` prop. This is because a shared layout is not re-rendered during navigation which could lead to stale `searchParams` between navigations.

When using client-side navigation, Next.js automatically only renders the part of the page below the common layout between two routes.

For example, in the following directory structure, `dashboard/layout.tsx` is the common layout for both `/dashboard/settings` and `/dashboard/analytics`:



When navigating from `/dashboard/settings` to `/dashboard/analytics`, `page.tsx` in `/dashboard/analytics` will rerender on the server, while `dashboard/layout.tsx` will **not** rerender because it's a common UI shared between the two routes.

This performance optimization allows navigation between pages that share a layout to be quicker as only the data fetching and rendering for the page has to run, instead of the entire route that could include shared layouts that fetch their own data.

Because `dashboard/layout.tsx` doesn't re-render, the `searchParams` prop in the layout Server Component might become **stale** after navigation.

Instead, use the Page `searchParams` prop or the `useSearchParams` hook in a Client Component, which is re-rendered on the client with the latest `searchParams`.

Layouts cannot access `pathname`

Layouts cannot access `pathname`. This is because layouts are Server Components by default, and don't rerender during client-side navigation, which could lead to `pathname` becoming stale between navigations. To prevent staleness, Next.js would need to refetch all segments of a route, losing the benefits of caching and increasing the RSC payload size on navigation.

Instead, you can extract the logic that depends on pathname into a Client Component and import it into your layouts. Since Client Components rerender (but are not refetched) during

navigation, you can use Next.js hooks such as `usePathname` to access the current pathname and prevent staleness.

app/dashboard/layout.tsx TypeScript

```typescript
import { ClientComponent } from '@/app/ui/ClientComponent'

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <>
      <ClientComponent />
      {/* Other Layout UI */}
      <main>{children}</main>
    <>
  )
}
```

Common `pathname` patterns can also be implemented with `params` prop.

See the examples section for more information.

Version History

Version	Changes
`v13.0.0`	`layout` introduced.

Previous
instrumentation.js

Next
loading.js

Was this helpful?