CMPS 356

Next.js 13
New Features

Dr. Abdelkarim Erradi

CSE@QU

Outline

- New app folder and Layout
- Data Fetching

New app folder and Layout

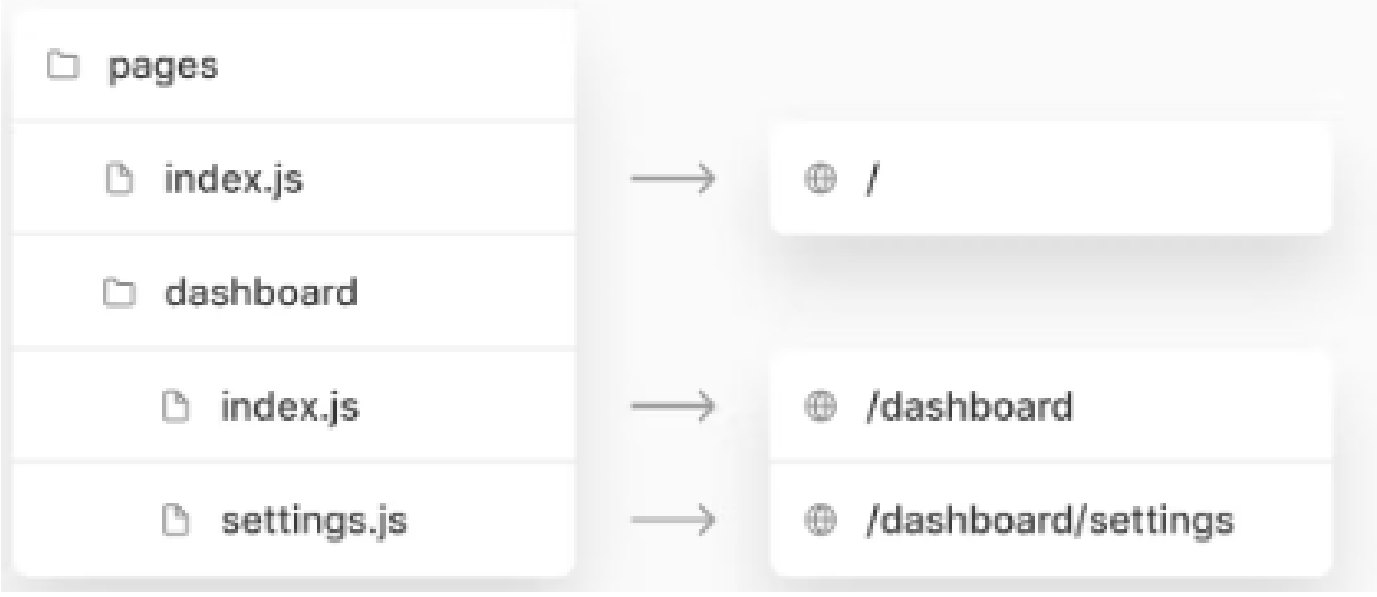
New app Directory

Next.js introduced the **app/** directory offering:

- **Layouts**: Easily share UI while preserving state and avoiding re-renders
- Server Components: Making server-first the default to reduce client-side JS
- Streaming: Display instant loading states and stream in updates
- Suspense for Data Fetching: async/await support and the **use** hook for component-level fetching
- The app/ directory can coexist with the existing pages directory for incremental adoption

Routing prior to Next.js 13

- Next.js uses the file system to map individual folders and files in the **pages** directory to routes accessible through URLs
  - Each page file exports a React Component and has an associated route based on its file name
  - Supports Dynamic Routes (including catch all variations) with the `[param].js`, `[...param].js` and `[[...param]].js` conventions

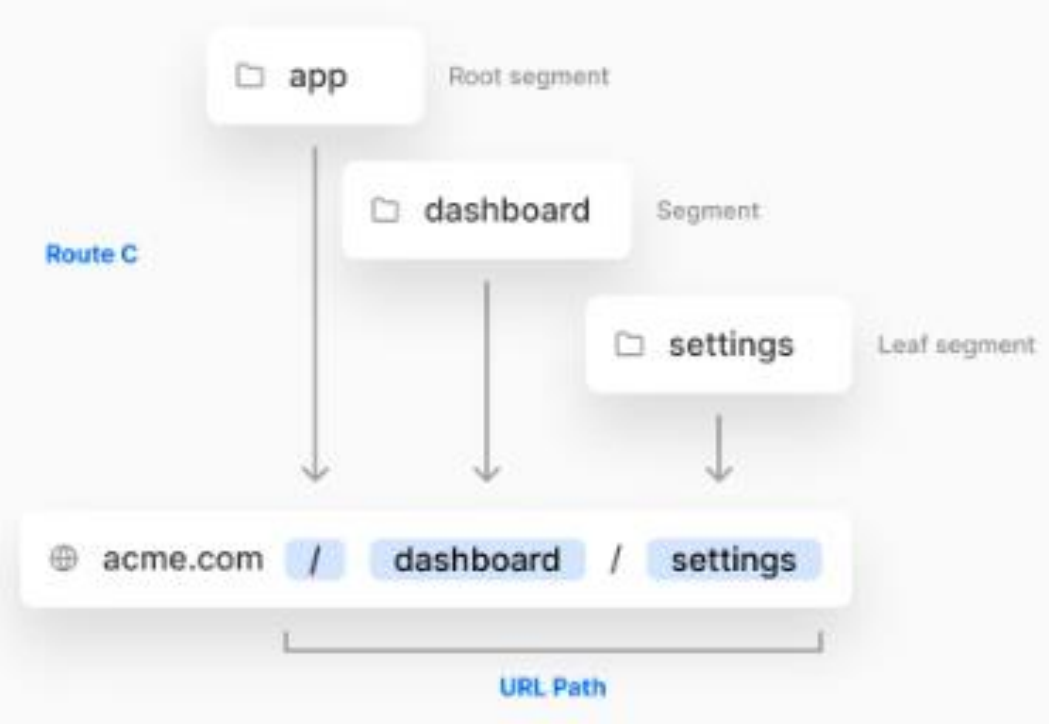


Next.js 13 Routing

- Use folder hierarchy inside the **app** folder to define routes, and files to define UI
  - A route is a single path of nested folders, from the root folder down to a leaf folder
  - Use a special **page.js** file to make a route segment publicly accessible

- Each folder in the subtree represents a route segment in a URL path

- E.g., create `/dashboard/settings` route by nesting two subfolders in the app directory

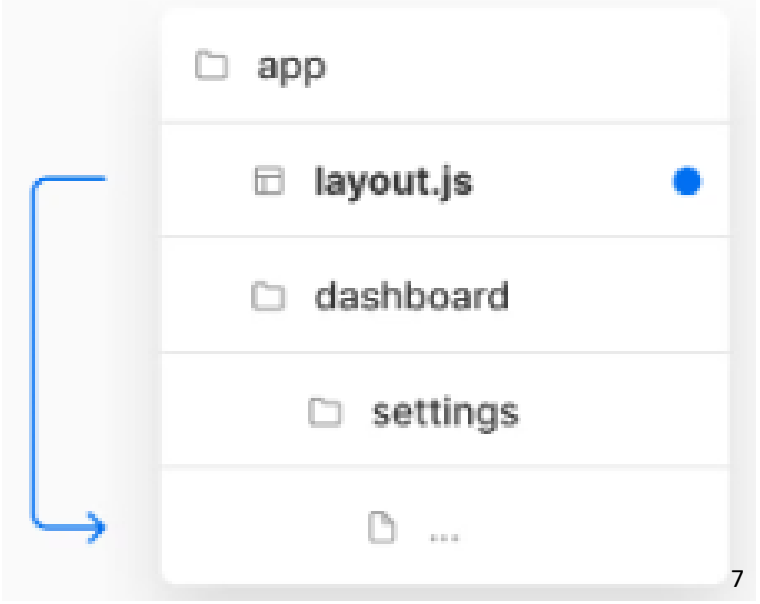


Layouts

- A layout is UI that is shared between route segments
  - Do not re-render (React state is preserved) when a user navigates between sibling segments
  - Navigating between routes only fetches and renders the segments that change
- A layout can be defined by exporting a React component from a `layout.js` file
  - The component should accept a `children` prop which will be populated with the segments the layout is wrapping

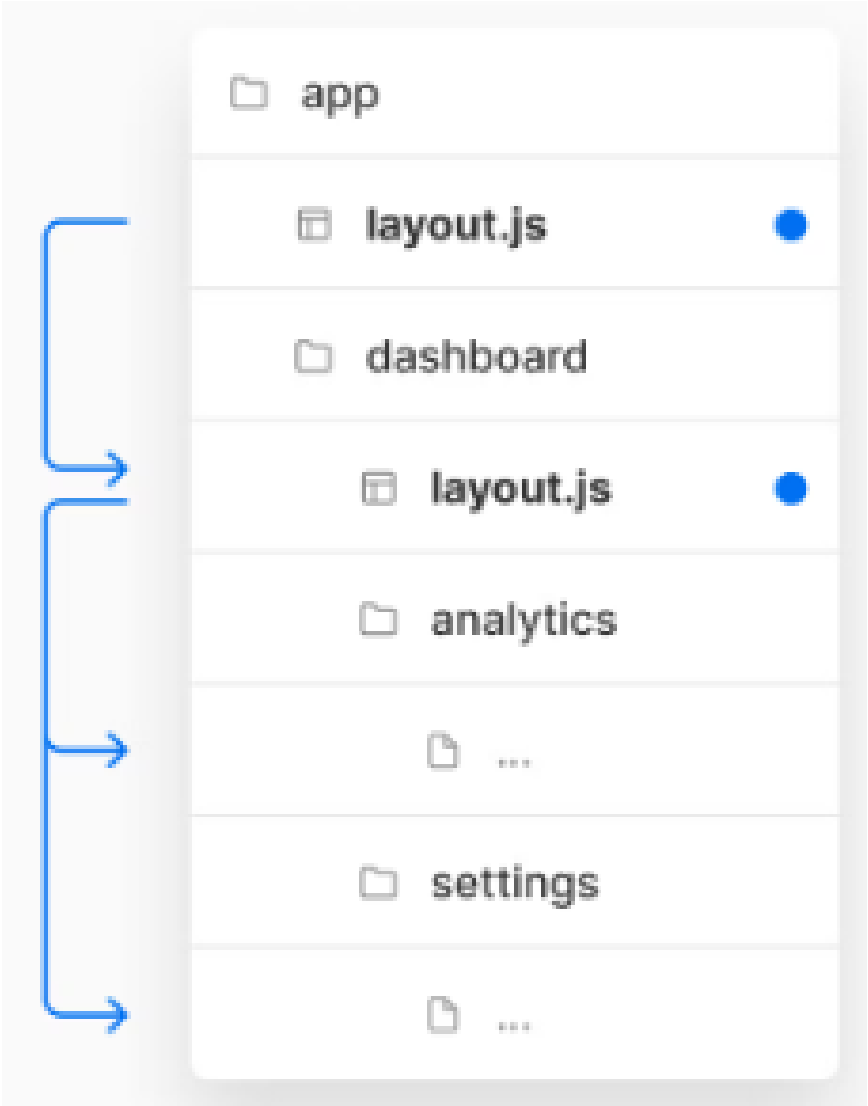
There are 2 types of layouts:

- **Root layout**: in **app** folder and applies to all routes
- **Regular layout**: inside a specific folder and applies to associated route segments

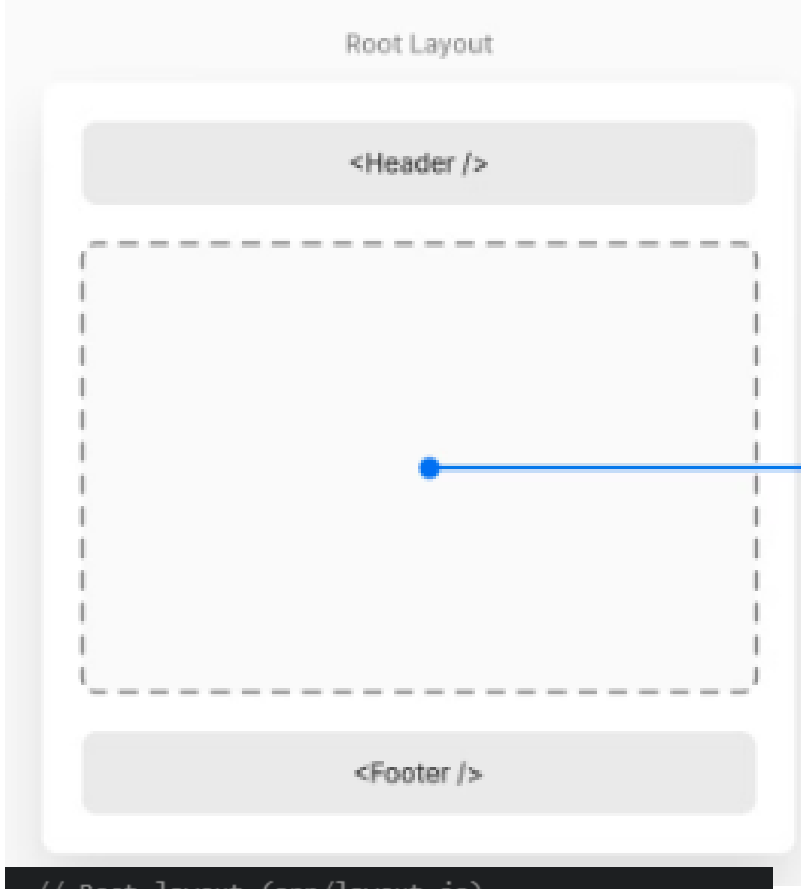


Nesting Layouts

- Layouts that can be nested and shared across routes
- E.g., the root layout (`app/layout.js`) would be applied to the dashboard layout, which would also apply to all route segments inside `dashboard/*`



Nesting Layouts




```tsx
// Page Component (app/dashboard/analytics/page.js)
// - The UI for the `app/dashboard/analytics` segment
export default function AnalyticsPage() {
  return (
    <main>...</main>
  )
}
```

```tsx
// Root layout (app/layout.js)
// - Applies to all routes
export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <Header />
        {children}
        <Footer />
      </body>
    </html>
  )
}
```

```tsx
// Regular layout (app/dashboard/layout.js)
// - Applies to route segments in app/dashboard/*
export default function DashboardLayout({ children }) {
  return (
    <>
      <DashboardSidebar />
      {children}
    </>
  )
}
```

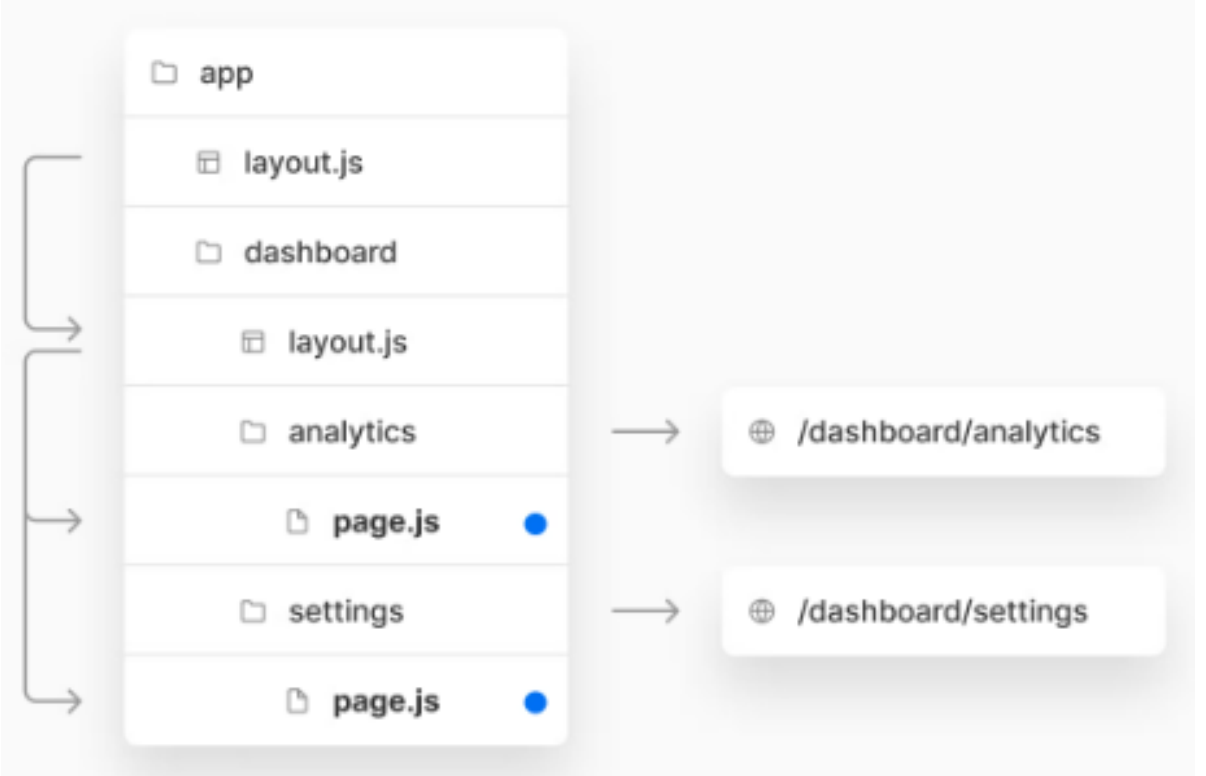
The above combination of layouts and pages would render the following component hierarchy:

```tsx
<RootLayout>
  <Header />
  <DashboardLayout>
    <DashboardSidebar />
    <AnalyticsPage>
      <main>...</main>
    </AnalyticsPage>
  </DashboardLayout>
  <Footer />
</RootLayout>
```

UI Pages

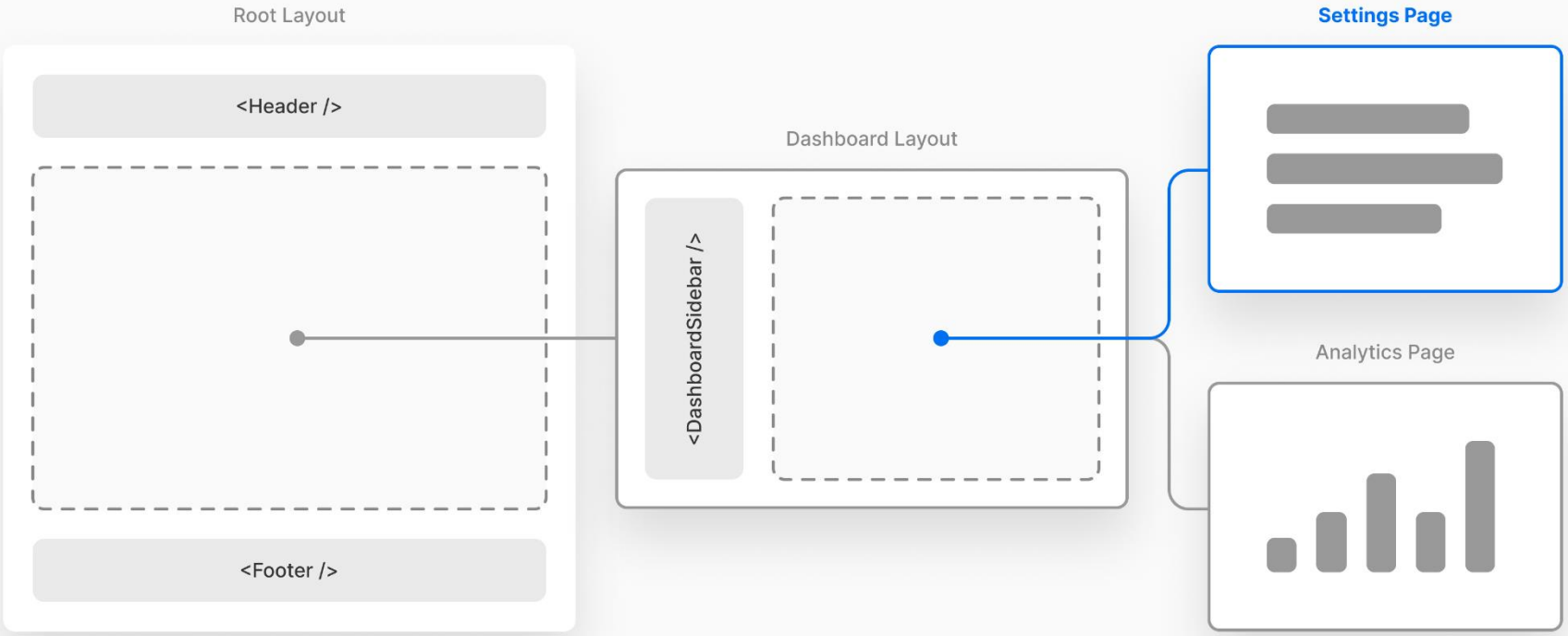
- You can create a page by adding a `page.js` file inside a folder
  - Can colocate your own project files (UI components, styles, images, test files, etc.) inside the app folder & subfolders

When a user visits `/dashboard/settings` Next.js will render the `page.js` file inside the settings folder



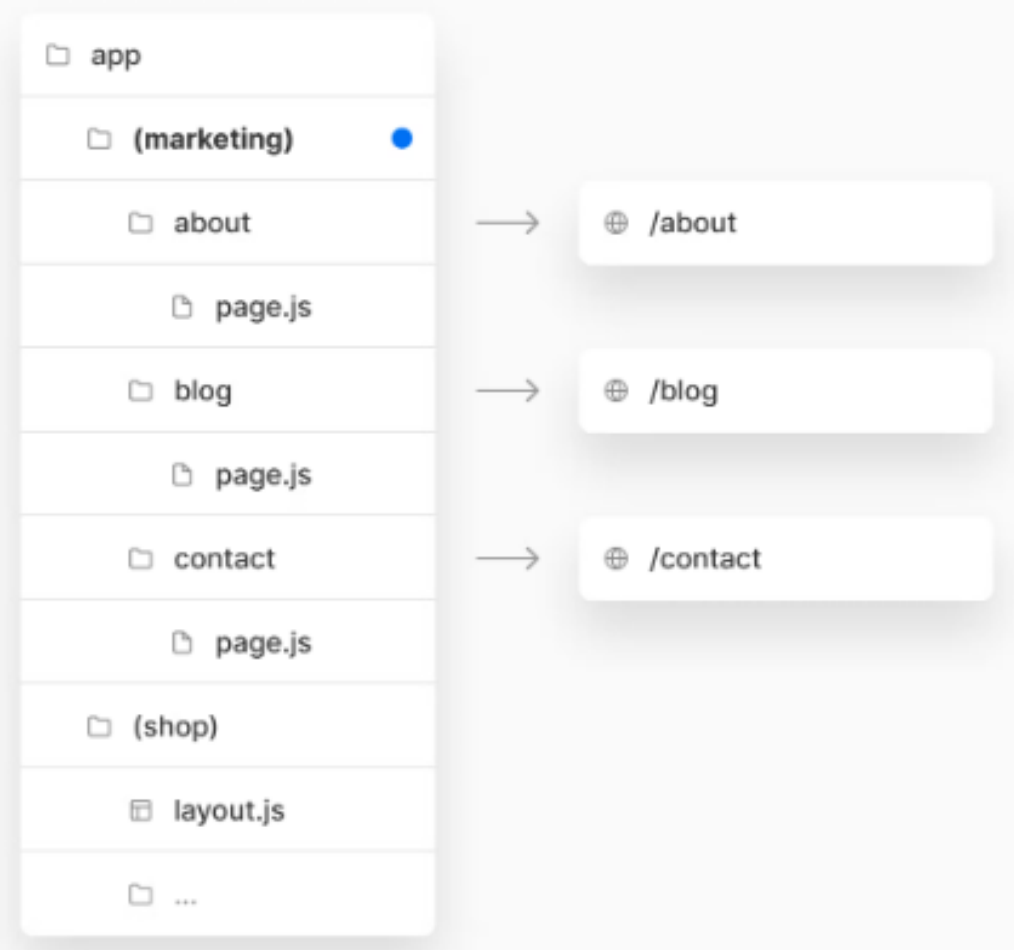
Pages are Wrapped in Layouts

- When a user visits `/dashboard/settings` Next.js will render the `page.js` file inside the settings folder wrapped in any layouts that exist further up the subtree

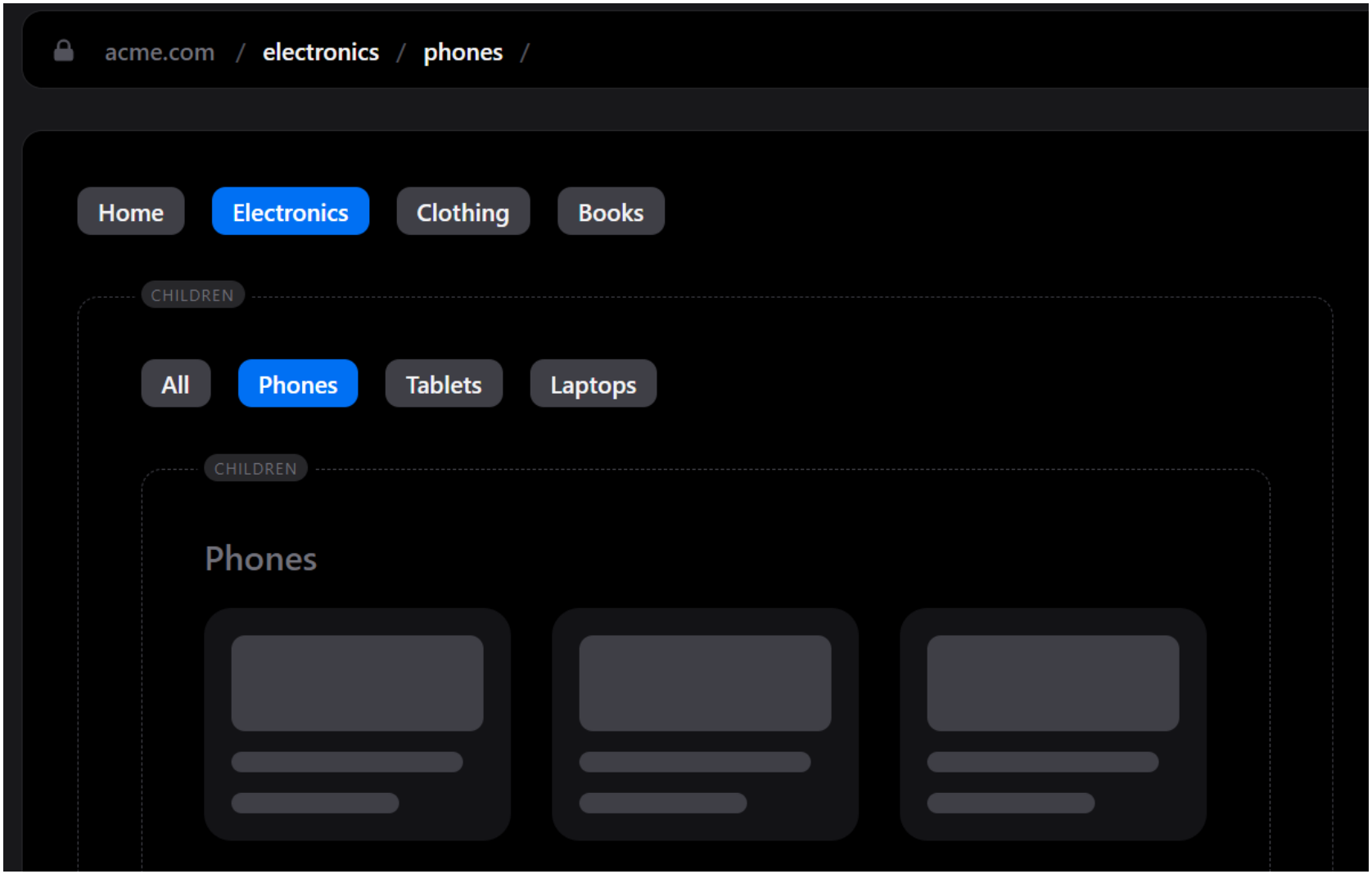


Organizing routes without affecting the URL path

- To organize routes, create a group to keep related routes together. The folders in parenthesis will be omitted from the URL (e.g. (marketing) or (shop))



Nested Layout Example



https://app-dir.vercel.app/layouts/electronics/phones

React Server Components

- By default, files inside **app** folder and its subfolders will be rendered on the server as **React Server Components**
  - resulting in less client-side JavaScript and better performance
- Making the route accessible requires adding **page.js** file

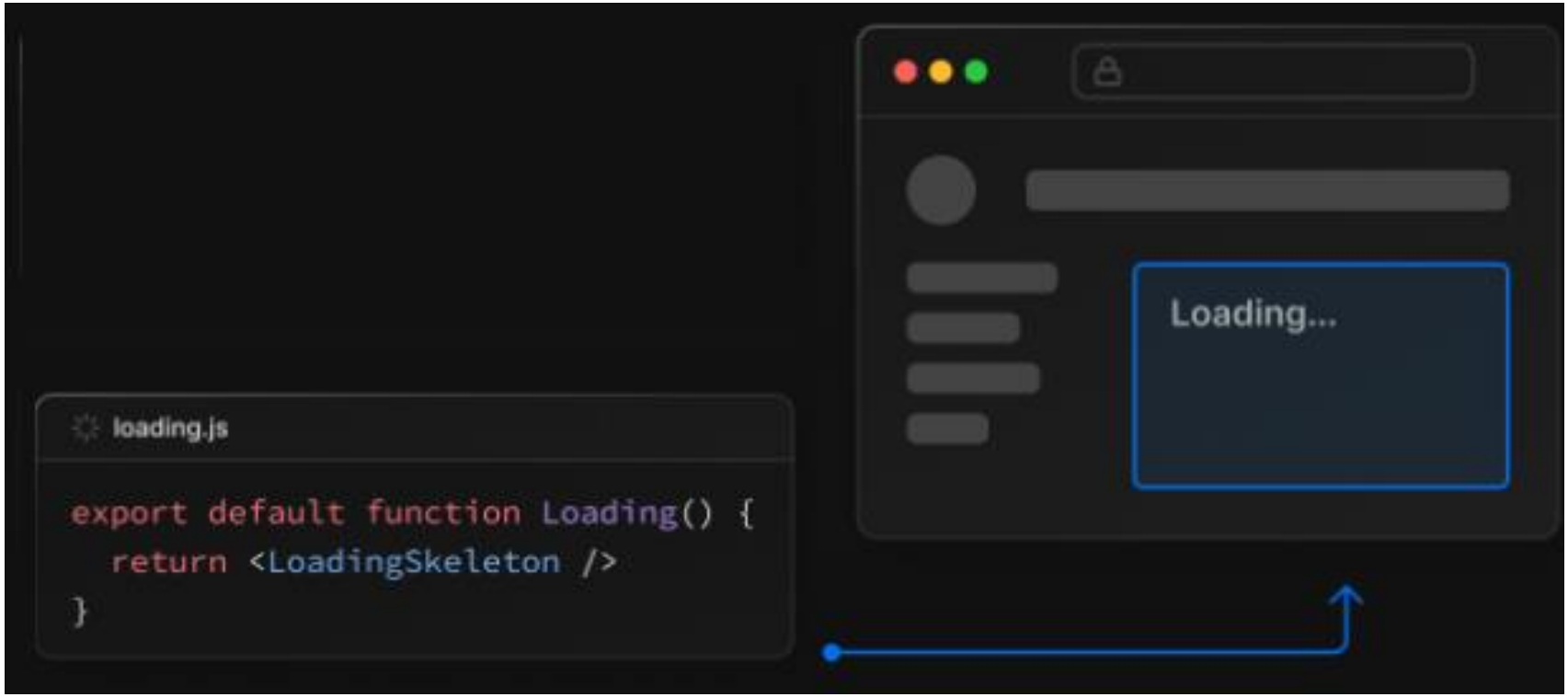
```
// app/page.js
// This file maps to the index route (/)
export default function Page() {
  return <h1>Hello, Next.js!</h1>;
}
```

UI Pages

- You can create a page by adding a **`page.js`** file inside a folder
- Files are used to define UI with new file conventions such as:
  - **layout.js**: define UI that is shared across multiple routes
  - **page.js**: define UI unique to a route
  - **loading.js**: show a loading indicator such as a spinner
  - **error.js**: show specific error information
  - **not-found.js**: render UI when the notFound function is thrown within a route segment

Loading UI

- **loading.js** return a loading indicator such as a spinner while the content of the route segment loads. The new content is automatically swapped in once rendering on the server is complete
  - This provides a better user experience by indicating that the app is responding



error.js

- `error.js` defines the error boundary for a route segment and the children below it. It can be used to show specific error information, and functionality to attempt to recover from the error
  - Should return a client-side component

```
'use client'
export default function Error({error}) {
  return (
    <>
      <p>❌ Something went wrong! {error.message}</p>
    </>
  );
}
```

not-found.js

- `not-found.js`: is used to render UI when the notFound function is thrown within a route segment

```
import { notFound } from 'next/navigation';

async function fetchUsers(id) {
  const res = await fetch('https://...');
  return res.json();
}

export default async function Profile({ params }) {
  const user = await fetchUser(params.id);

  if (!user) {
    notFound();
  }

  // ...
}
```

```
export default function NotFound() {
  return "Couldn't find requested resource"
}
```

redirect()

app/team/[id]/page.js

```
import { redirect } from 'next/navigation';

async function fetchTeam(id) {
  const res = await fetch('https://...');
  return res.json();
}

export default async function Profile({ params }) {
  const team = await fetchTeam(params.id);
  if (!team) {
    redirect('https://...');
  }
  // ...
}
```

The redirect function allows you to redirect the user to another URL

next/link

- next/link component no longer requires manually adding <a> tag as a child

```
import Link from 'next/link'

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

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

next/image

- Lazy loading and optimized files for increased performance with less client-side JavaScript

```
import Image from 'next/image';
import avatar from './lee.png';

function Home() {
  // "alt" is now required for improved accessibility
  // optional: image files can be colocated inside the app/ directory
  return <Image alt="leeerob" src={avatar} placeholder="blur" />;
}
```

Data Fetching

Data Fetching prior to Next.js 13

- Next.js provides data fetching methods which can be used at the page (route) level
  - Statically Generated (getStaticProps)
  - Server-Side Rendered (getServerSideProps)
  - Incremental Static Regeneration (ISR) to create or update static pages after a site is built

Data Fetching Next.js 13

- You can call fetch with async/await directly within Server Components

```
// This request should be cached until manually invalidated.
// Similar to `getStaticProps`.
// `force-cache` is the default and can be omitted.
fetch(URL, { cache: 'force-cache' });

// This request should be refetched on every request.
// Similar to `getServerSideProps`.
fetch(URL, { cache: 'no-store' });

// This request should be cached with a lifetime of 10 seconds.
// Similar to `getStaticProps` with the `revalidate` option.
fetch(URL, { next: { revalidate: 10 } });
```

Data Fetching Next.js 13

- fetch() is a Web API used to fetch remote resources and returns a promise
- Next.js extends the fetch options object to allow each request to set its own caching and revalidating
- You can fetch data in a component, a page or a layout
  - e.g., a blog layout could fetch categories which can be used to populate a sidebar component

```
async function getData() {
  const res = await fetch('https://api.example.com/...');
  return res.json();
}

export default async function Page() {
  const name = await getData();

  return '...';
}
```

Server-Side Rendering (SSR)

- To refetch data on every fetch() request, use the `cache: 'no-store'` option
  - This is equivalent to getServerSideProps()

```
fetch('https://...', { cache: 'no-store' });
```

Static Site Generation (SSG)

- By default, fetch will automatically fetch static data (cached data)
  - This is equivalent to getStaticProps() in the pages directory

```
fetch('https://...'); // cache: 'force-cache' is the default
```

```
async function getNavItems() {
  const navItems = await fetch('https://api.example.com/...');
  return navItems.json();
}

export default async function Layout({ children }) {
  const navItems = await getNavItems();

  return (
    <>
      <nav>
        <ul>
          {navItems.map((item) => (
            <li key={item.id}>
              <Link href={item.href}>{item.name}</Link>
            </li>
          ))}
        </ul>
      </nav>
      {children}
    </>
  );
}
```

Static Site Generation Example

Revalidating Data

- To revalidate cached data, you can use the **`next.revalidate`** option in fetch()
  - This equivalent to Incremental Static Regeneration (ISR)

```
fetch('https://...', { next: { revalidate: 10 } });
```

Generate Static Params

- The `generateStaticParams` function can be used in combination with dynamic route segments to define the list of route segment parameters that will be statically generated at build time
  - This is equivalent to **`getStaticPaths`**

```
export default function Page({ params }) {
  const { slug } = params;

  return ...
}

export async function generateStaticParams() {
  const posts = await getPosts();

  return posts.map((post) => ({
    slug: post.slug,
  }));
}
```

Resources

- Next.js 13 Documentation

https://beta.nextjs.org/

- Next.js 13 Fetch API

https://beta.nextjs.org/docs/api-reference/fetch