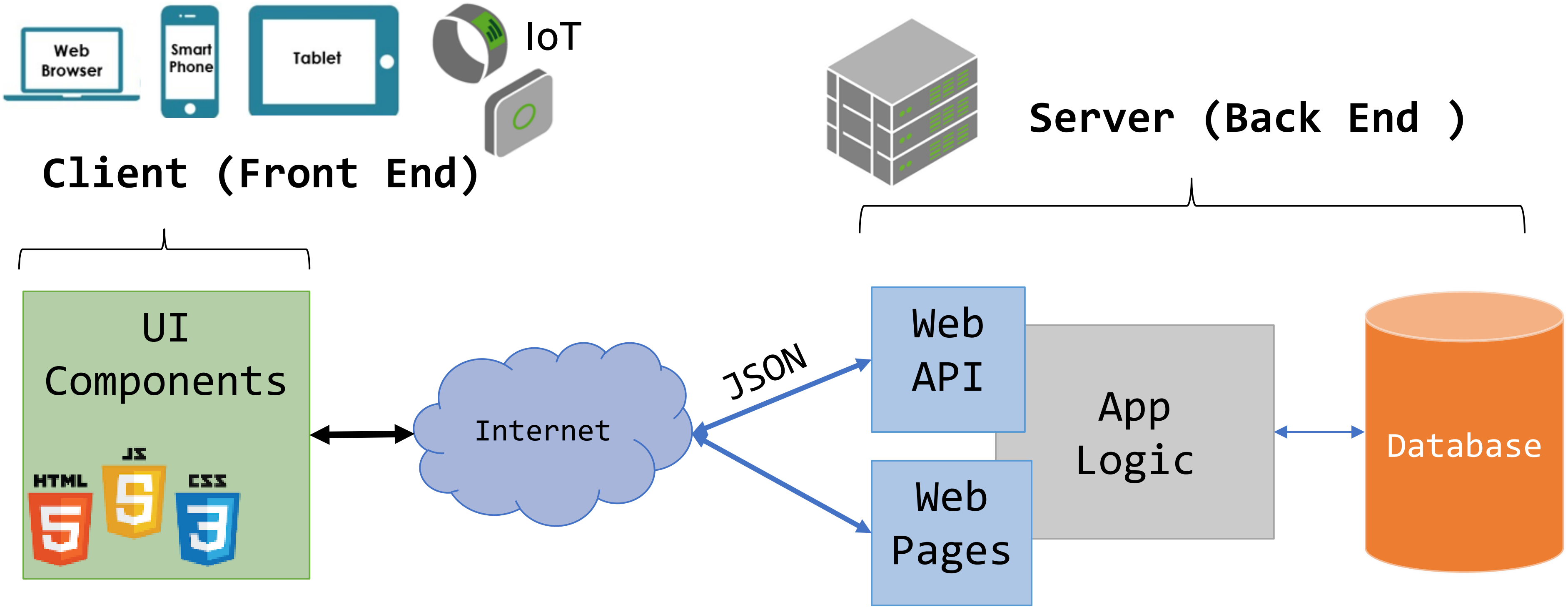
Web Pages using NEXT.JS

Outline

1. UI Components using React
2. Next.js routing
3. Server actions
4. Data fetching

Web App Architecture using Next.js

- Front-end made-up of **multiple UI components loaded** in response to user actions
- Back-end Web API and Web pages



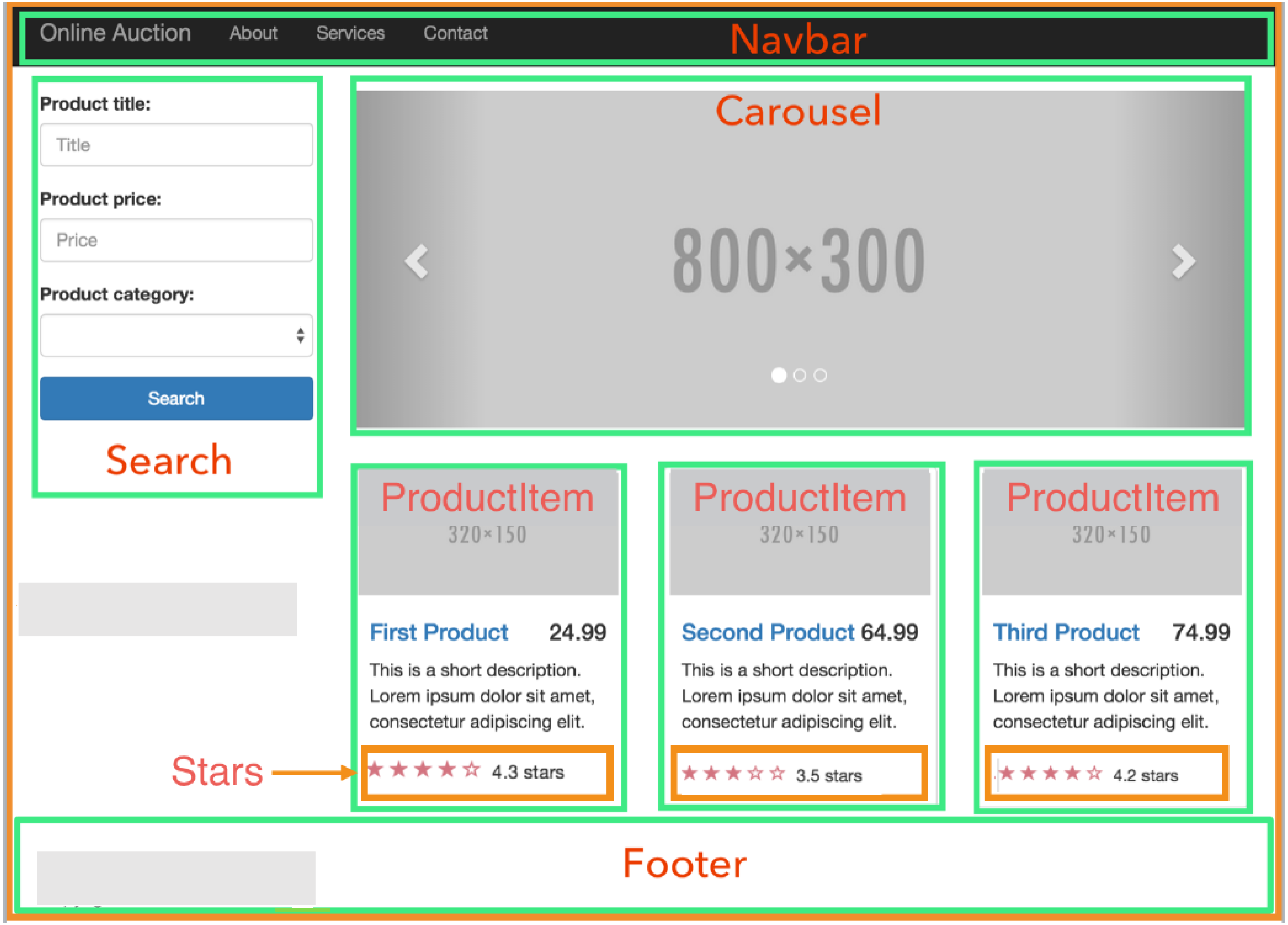
UI Components using React

Used by Facebook, Instagram, Netflix, Dropbox, Outlook, Yahoo, Khan Academy, ...

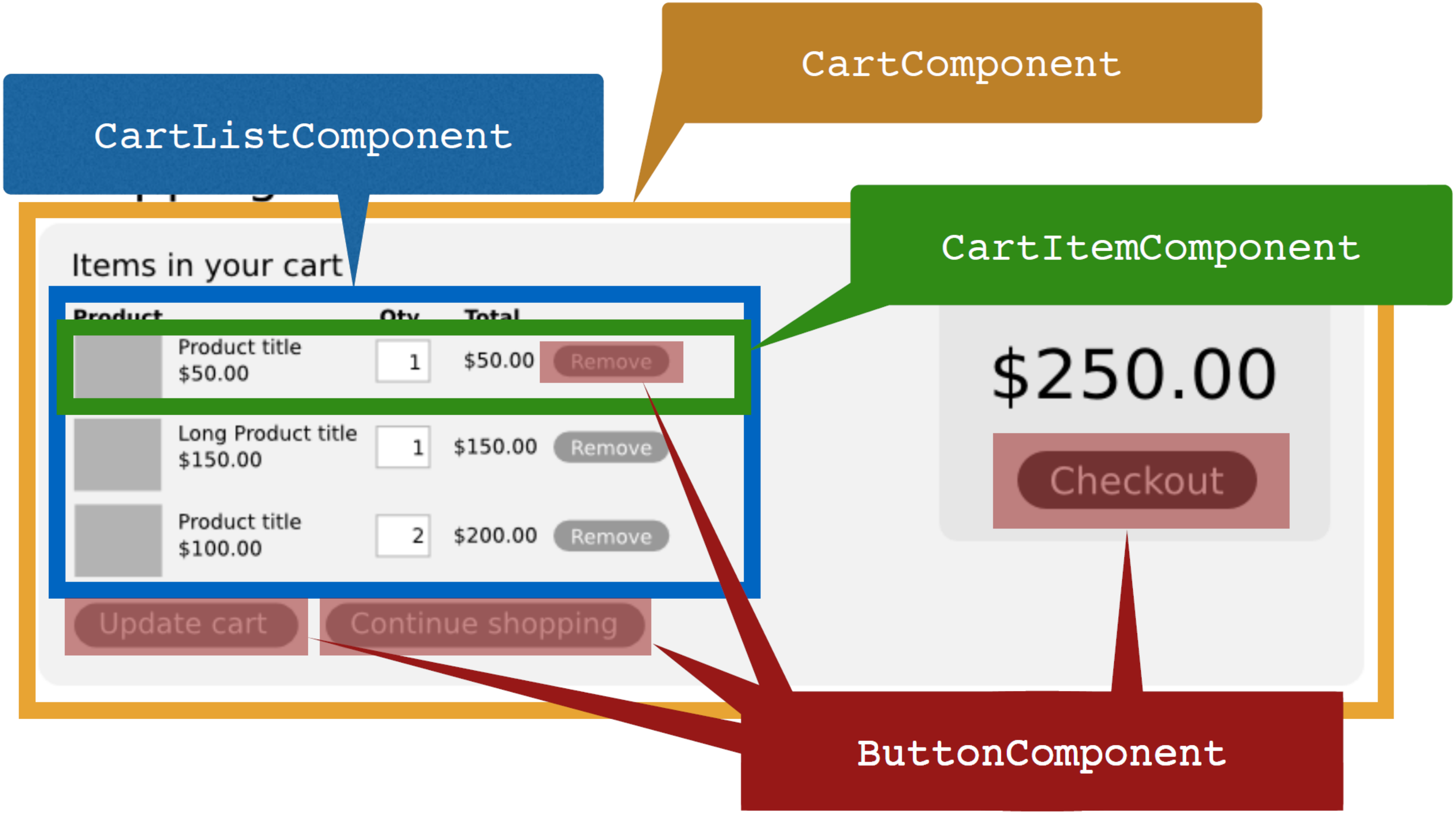
https://intellisoft.io/15-popular-sites-built-with-react-js/

< Back

A page = a composition of components



A component = a tree of components



UI Components using React

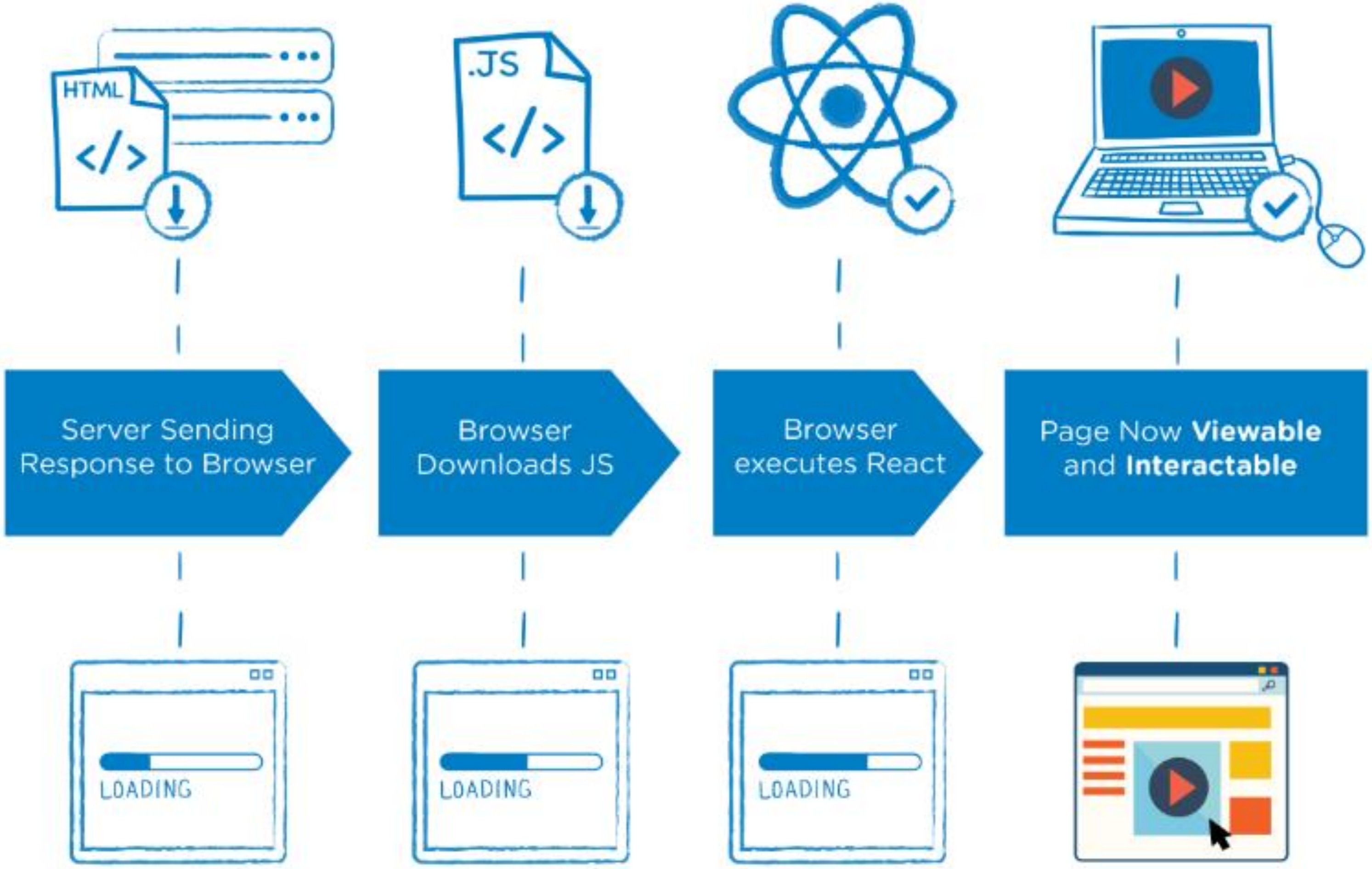
- React = an open-source JavaScript library for building **modular, components-based** user interfaces
  - It allows creating **reusable** UI components

  => thus, it enables reusability, and ease of maintenance
  - Open-sourced by Facebook mid-2013 - https://react.dev/
  - Competing with Angular https://angular.dev/ and Vue.js https://vuejs.org/
- **Components-based user interfaces (UI)**
  - UI is **composed** of small reusable **components**
  - A UI Component encapsulates **UI elements** and their associated **behavior** (i.e., UI logic)

React vs. Next.js

- Traditional Single-Page Applications (SPAs) built primarily with **client-side React** often face 2 challenges:
    - Search Engine Optimization (SEO) difficulty: Search engine crawlers receive minimal HTML shell before JavaScript execution, hindering the indexing of the full page content
    - Slower Initial Load & Interactivity: The browser must download, parse, and run large JavaScript bundles first before the user sees meaningful content, especially on slower networks or devices.
- Next.js builds on React, providing structure and features to overcome these limitations and enable building full-stack Web Apps including the **front-end** and **back-end**

Client-side rendering (CSR) in Traditional SPA



Server Sending Response to Browser

Browser Downloads JS

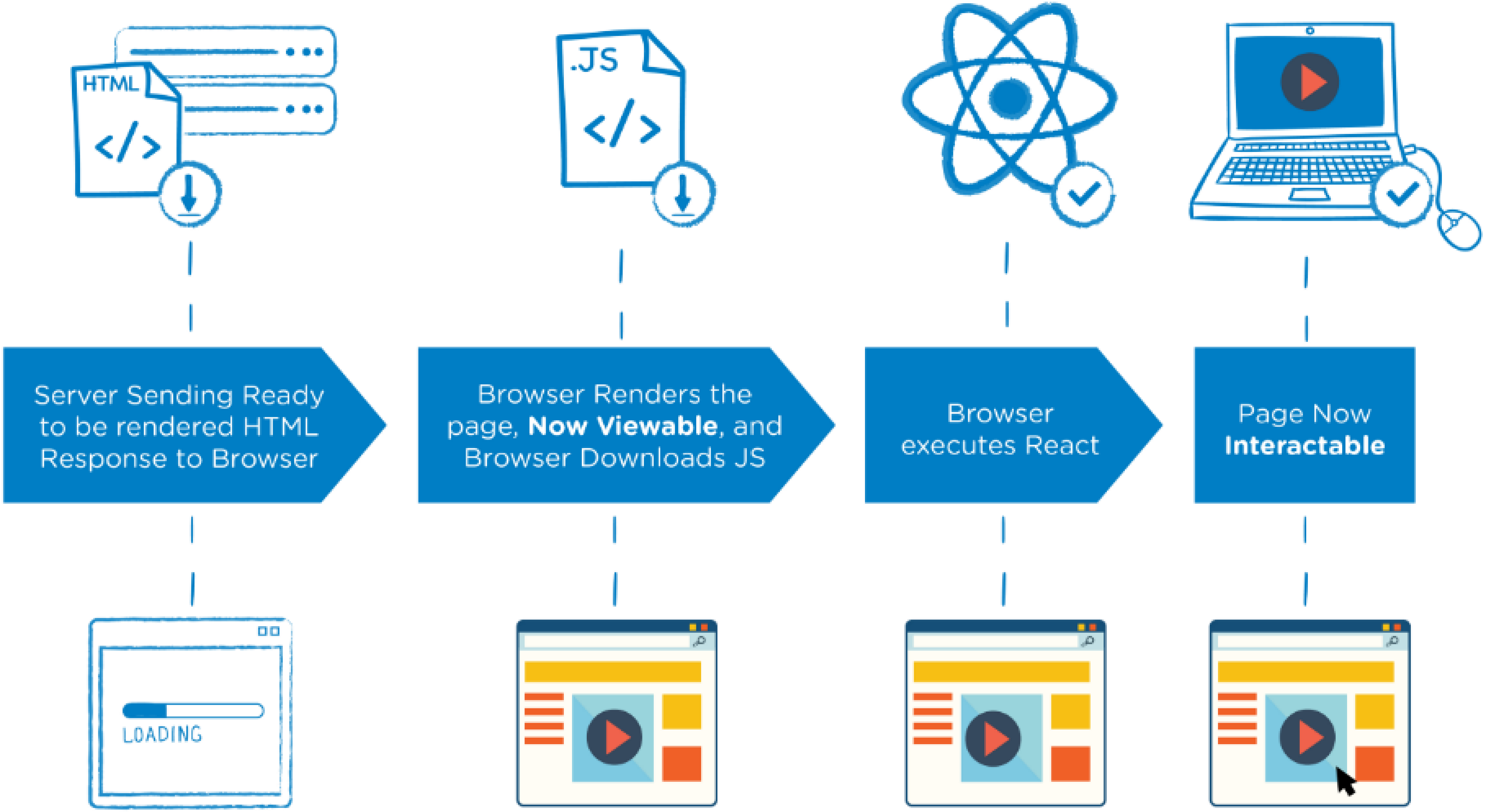
Browser executes React

Page Now **Viewable** and **Interactable**

Next.js Key Features

- Next.js = React-based full stack web framework that allows creating server-rendered pages, and Web API
- Key Enhancements over Client-Side React:
  - Enhanced SEO & Performance: Delivers server-side pre-rendered page content, resulting in faster initial loads and enhanced crawlability
  - Integrated Full-Stack: Unifies frontend and backend development through Server Components, Route Handlers (for APIs), and Server Actions
  - Built-in Production Optimizations: Including automatic code splitting, image optimization, route prefetching, and caching strategies.
  - Enhanced Developer Experience (DX): Fast Refresh provide instant feedback during development

Server-Side Rendering (SSR) using Next.js



Server Sending Ready to be rendered HTML Response to Browser

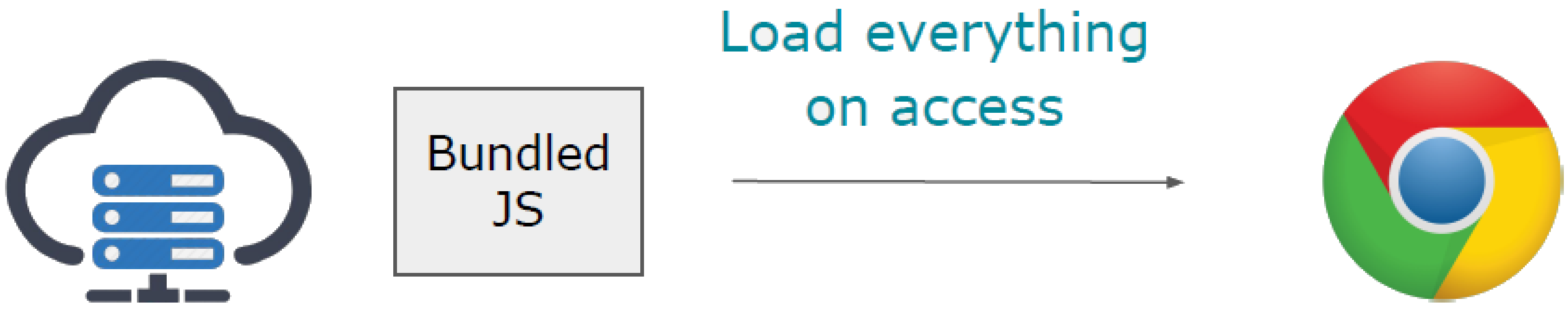
Browser Renders the page, **Now Viewable**, and Browser Downloads JS

Browser executes React

Page Now **Interactable**

Code splitting

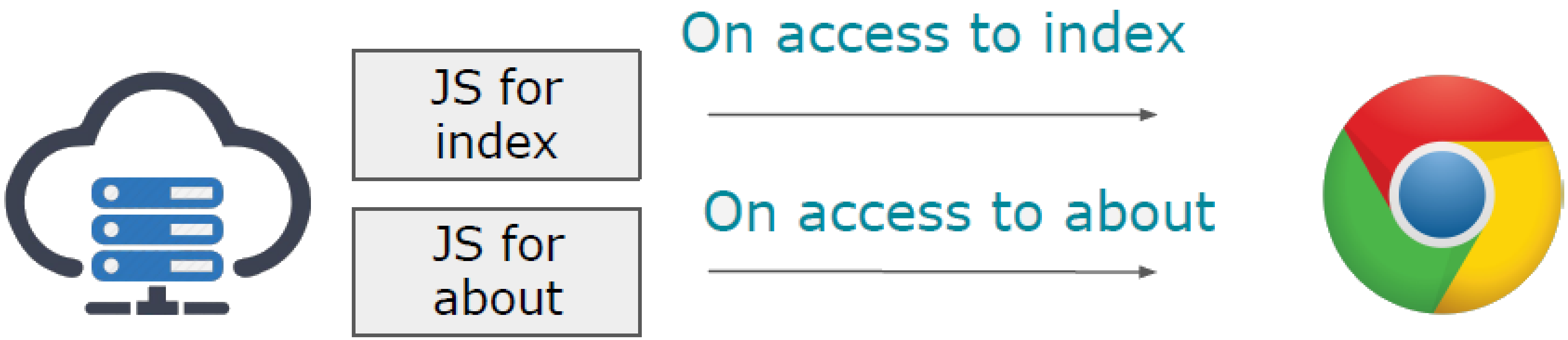
- In Single Page Architecture (SPA), a large bundled file will be loaded



Bundled JS

Load everything on access

- With Next.js , code will be split on per page base



JS for index

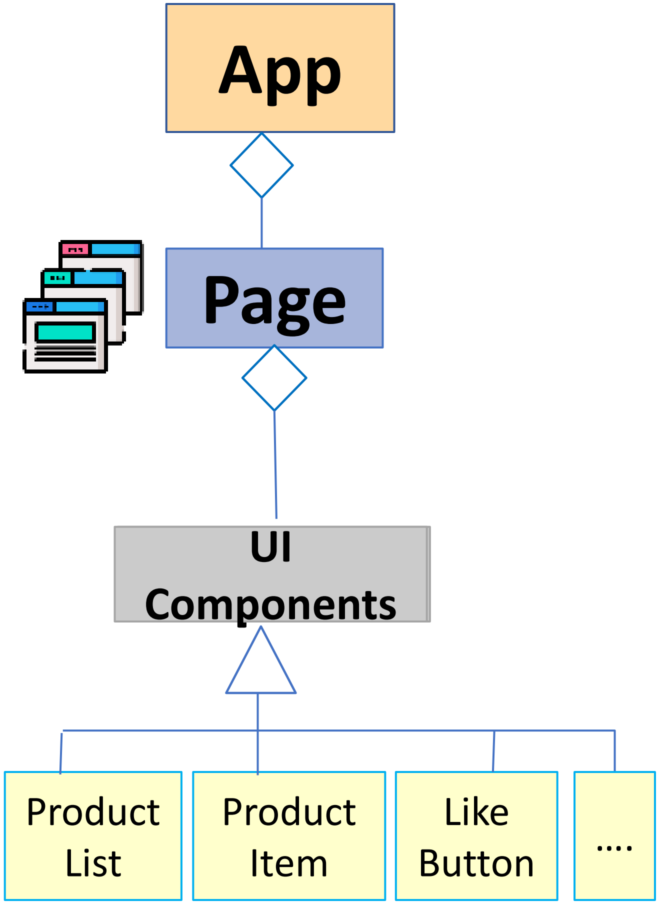
JS for about

On access to index

On access to about

UI Programming Model using Next.js

IMPORTANT



- An app consists of one or multiple **pages**, each representing a distinct route within the app

- A **page** is UI Component composed of multiple smaller UI Components, following a hierarchical structure that promotes modularity, reusability, and maintainability

- Pages can be wrapped in a **Layout component**, which acts as a shared container providing consistent UI elements across multiple pages, such as a header, footer, navigation bar, and sidebars

- A **UI Component** encapsulates UI elements and their associated behavior (i.e., UI logic)

- UI Components could be either **Server Components** (rendered on the server with optional caching) or **Client Components** (execute in the browser and handle client-side events)

- Client Components manage interactivity through:

(1) State variables, which store and update UI data dynamically, enabling reactive interfaces

(2) **Event Handlers**, which define responses to user interactions, such as button clicks or form submissions

Getting started

- Install latest **Node.js** https://nodejs.org/en/
- Download **VS Code** https://code.visualstudio.com/
- Create an empty folder (with no space in the name use **dash -** instead)
- Create a next.js app

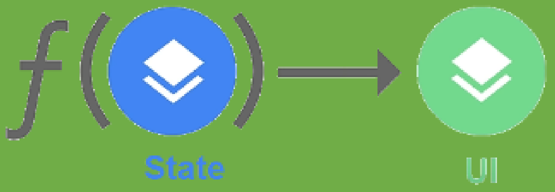
```
npx create-next-app .
```

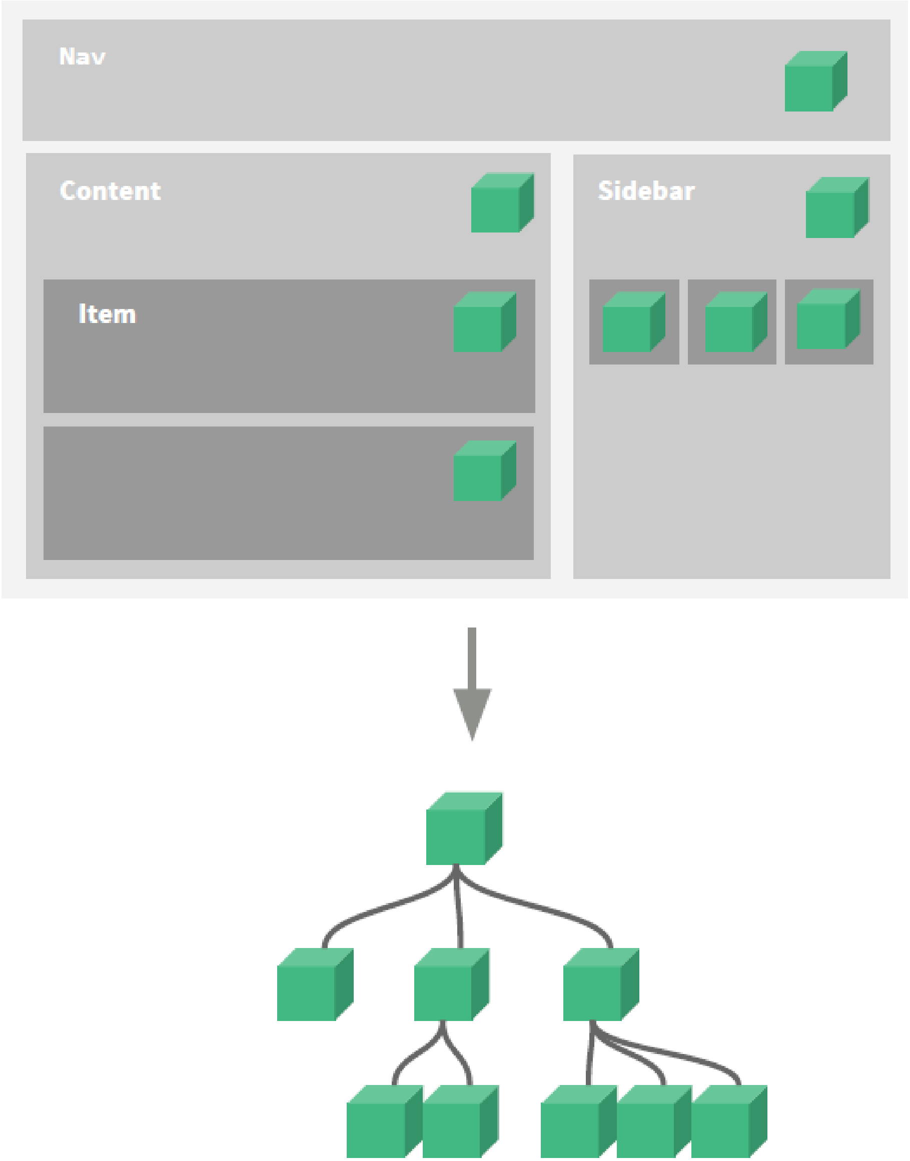
- Run the app in dev mode: `npm run dev`
- Build the app: `npm run build`
- Run the optimized build: `npm run start`

Project Folder Structure

- Next.js relies heavily on **convention** over configuration
  - Specific folder names (app/, public/) trigger core framework features
- Next.js uses **app/** folder for file-based routing
  - Folders = URL Segments (e.g., app/dashboard/ -> /dashboard)
  - **`page.jsx = Route UI`** defines the UI for that specific route segment
- **public/** serve static assets (e.g., images, font) from the app root (/)
  - E.g., `public/my-image.png -> /my-image.png`

UI Components





< Back

UI Component

- App UI = composition of small reusable **components**
- A UI ***component***:
    - Return **HTML *elements*** to provide the UI
    - Encapsulate **state** (internal component data) and **functions** to ***handle events*** raised from the UI elements
- Component = UI + display logic
- Components allows creating new '**HTML tags**'

Defining React Components: Functions of Data

IMPORTANT

- UI Component = a **function**:
  - Takes Inputs (Props & State) describing the data it needs
  - Returns Output (JSX) describing what the UI should be based on those inputs

- **UI = f(props, state)**: A component's rendered output is a function of its current props and state
  - **Props**: Data passed down to a component from its parent
    - Read-only within the component receiving them. Changes come from the parent
    - E.g., `<UserProfile name="Alice" />` - name is prop passed to UserProfile
  - **State**: Data managed internally by the component itself
    - It is mutable, changes trigger re-renders (see more details in the next lecture)
    - E.g., A counter state variable inside a `<Counter />` component

- **Reactivity:** React automatically re-renders the component (and updates the browser DOM efficiently) whenever its props or state change

React = Declarative UI Programming

- With React, you define components as functions that describe **what the UI should be** based on the input data (props) it receives
- When you use the component (e.g., `<Greeting name="Alice" />`), React uses the component's description and efficiently updates the browser's DOM to match that description whenever the data changes (You don't manually manipulate the DOM elements)

```
// Declarative:  Define a component that
takes props and returns a UI based on those
props
export default function Greeting({ name }) {
  // Declare what the UI should be based on
the 'name' prop
  if (name) {
    return <h1>Hello, {name}!</h1>;
  } else {
    return <h1>Hello, Guest!</h1>;
  }
}
```

Define the end result (WHAT), not the steps

```
// Imperative: Manually select element and update
it step-by-step
function displayGreeting(name) {
  // Step 1: Find the target DOM element
  const headingElement =
           document.querySelector("greeting");
  // Step 2: Manually set its content
  if (name) {
    headingElement.textContent = `Hello, ${name}!`;
  } else {
    headingElement.textContent = "Hello, Guest!";
  }
}
```

Define the steps (HOW), not the end result

Component Example

- Create a **Welcome** component
  - Returns **JSX** : an HTML-like syntax to define the component UI
  - Can accept a parameter, often called ***props***
    - to configure the component with different content / attributes - just like how HTML works (makes the component reusable)
    - **props** are read-only
  - Component name must start with a capital letter

You can embed JavaScript expressions in JSX

```
function Welcome(props) {
    return (<h1>Welcome to {props.appName}</h1>);
}
export default Welcome;
```

- Use the **Welcome** component

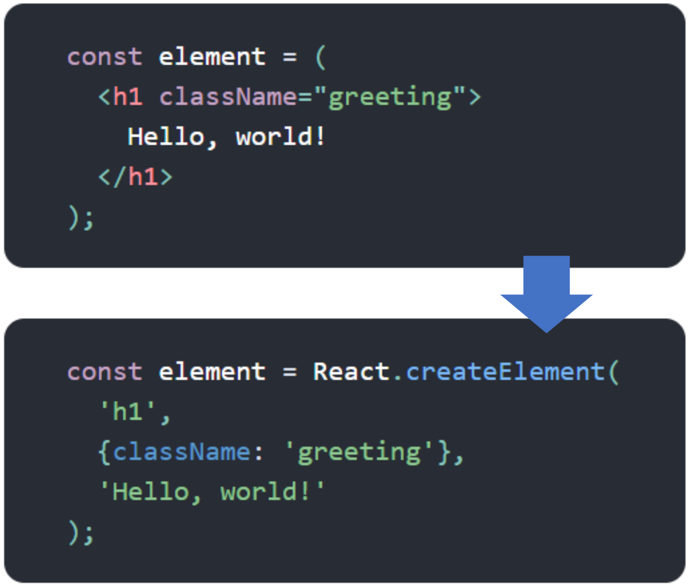
```
<Welcome appName='React Demo App' />
```

What is JSX?

- React uses JSX (JavaScript XML) HTML-like markup to describe the component's UI
- Embraces the fact that rendering logic is inherently coupled with other UI logic
- JSX allows us to write HTML like syntax which gets transformed to JavaScript objects

JSX

JavaScript



```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

It's just JavaScript!!

Props destructuring

- In a **react** component you can destructure props into variables

```
function UserInfo(props) {
    return (
        <div>
            First Name: {props.firstName}
            Last Name: {props.lastName}
        </div>
    );
}
```

Becomes

```
function UserInfo({ firstName, lastName }) {
    return (
        <div>
            First Name: {firstName}
            Last Name: {lastName}
        </div>
    );
}
```

Special "children" Prop

- The children property holds the content you might have provided between the component's opening and closing tags
    - A special children property auto-added by react

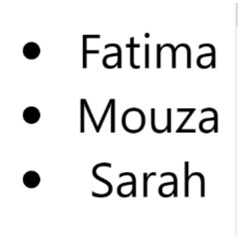
```
<Welcome name="Ali Faleh">
  <h2>Welcome to QU</h2>
</Welcome>
```

```
function Welcome({name, children}) {
    return (
        <>
            <h1>Welcome {name}</h1>
            {children}
        </>
    );
}
```

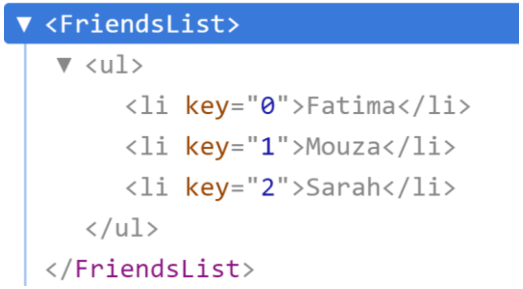
Rendering a List of items (with .map())

Lists are handled using **.map** array function

```
function FriendsList({friends}) {
  return <ul>
             {friends.map( (friend, i) =>
                 <li key={i}>{friend}</li>
             )}
         </ul>
}
```



- Fatima
- Mouza
- Sarah



```
▼ <FriendsList>
  ▼ <ul>
      <li key="0">Fatima</li>
      <li key="1">Mouza</li>
      <li key="2">Sarah</li>
    </ul>
  </FriendsList>
```

Key helps identify which items have changed, added or removed

- Use the **FriendsList** component

```
<FriendsList friends={['Fatima', 'Mouza', 'Sarah']}/>
```

List of item keys

Keys are very important in lists for the following reasons:

- A key is a unique identifier used to identify which list items have changed, are added, or are deleted from the list
- It also helps to determine which components need to be re-rendered instead of re-rendering all the components every time.
  - Therefore, it increases performance, as only the updated components are re-rendered

Routing

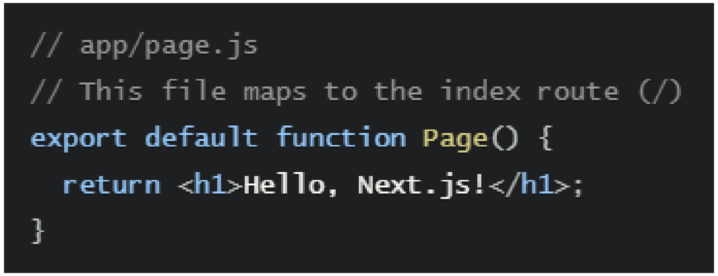
< Back

Routing

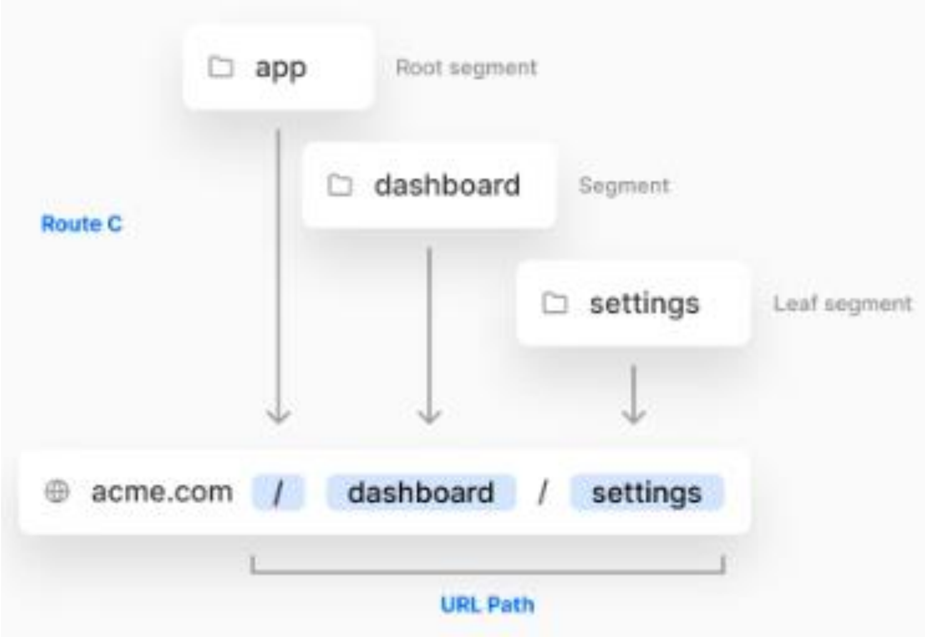
- Use folder hierarchy inside the **app** folder to define routes, and `page.jsx` files to define UI
  - A route is a single path of nested folders, from the root folder down to a leaf folder
  - Use a special **page.jsx** file to define the route UI

- 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



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

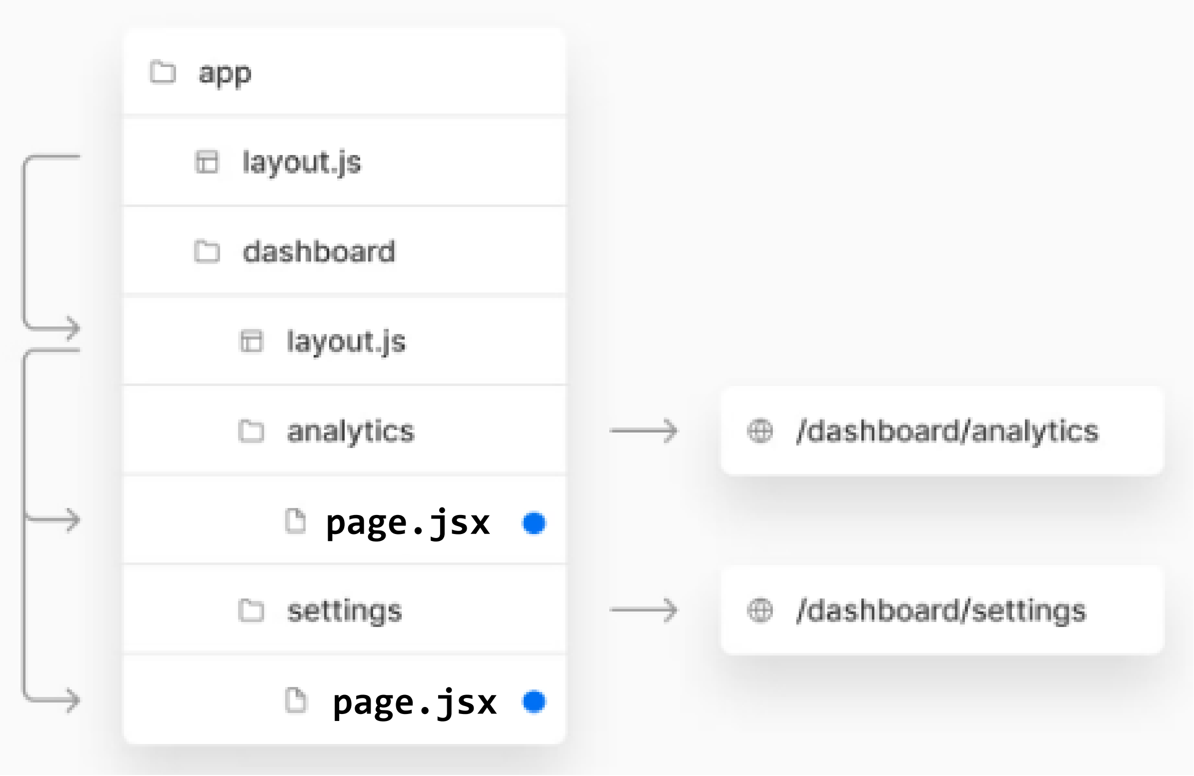


UI Pages

- You can create a page by adding a **`page.jsx`** file inside a subfolder (under the app folder)
  - You can colocate the page files (UI components, styles, images, test files, etc.) in the subfolder

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

- By default, page 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



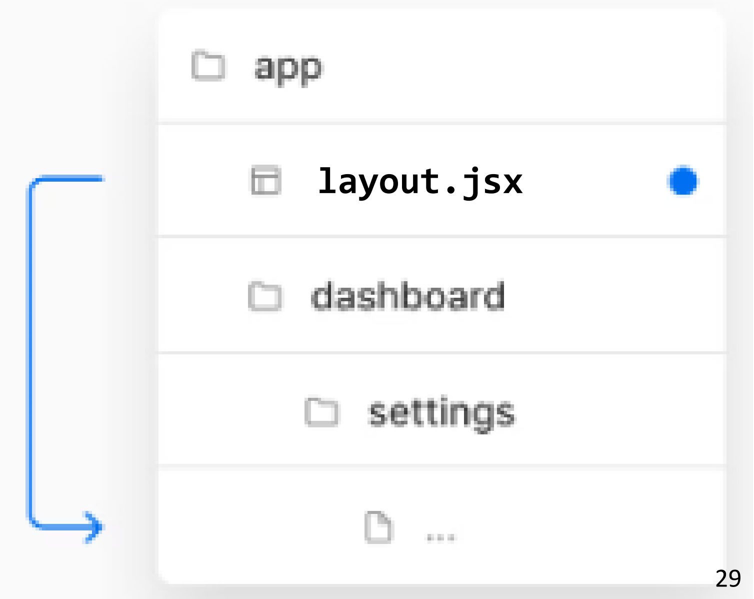
Layouts

- A layout is UI that is shared between route segments
  - Do not re-render (UI 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.jsx` 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:

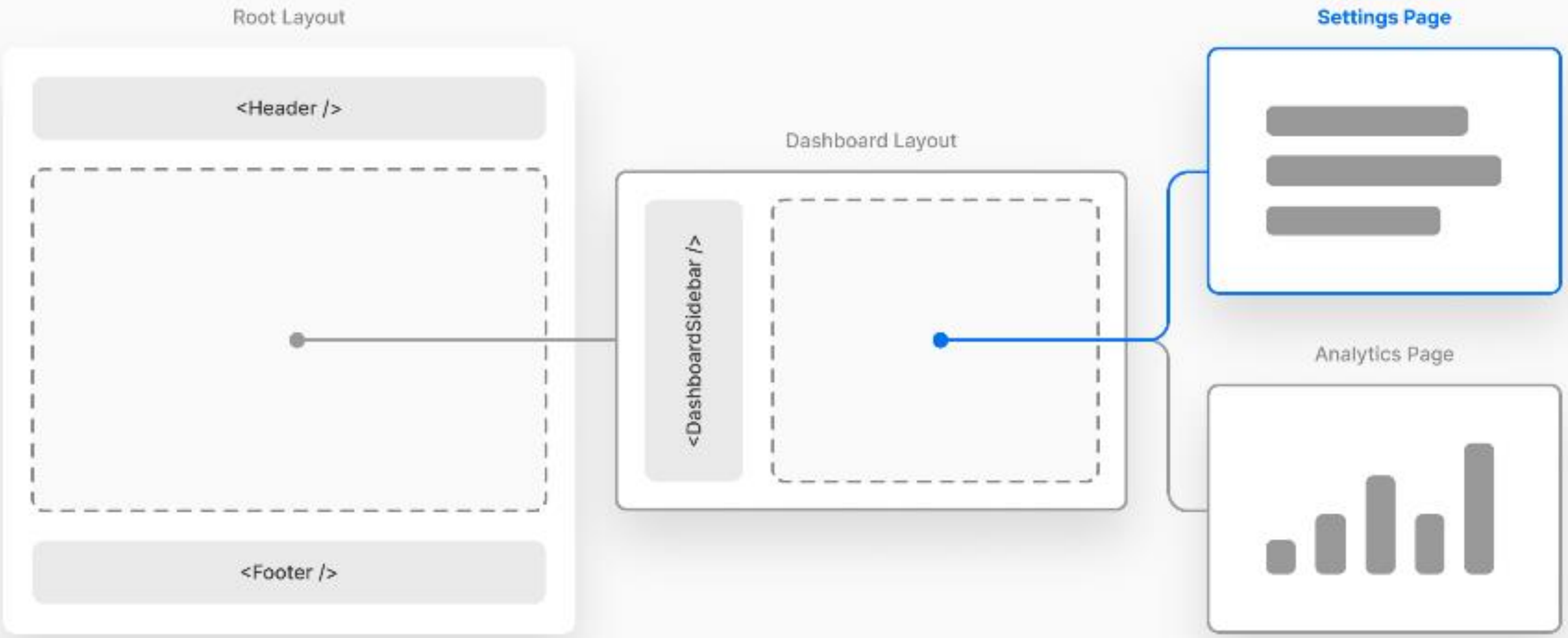
- **Root layout**: in **app** folder and applies to all routes

- **Regular layout**: inside a specific folder and applies to associated route segments



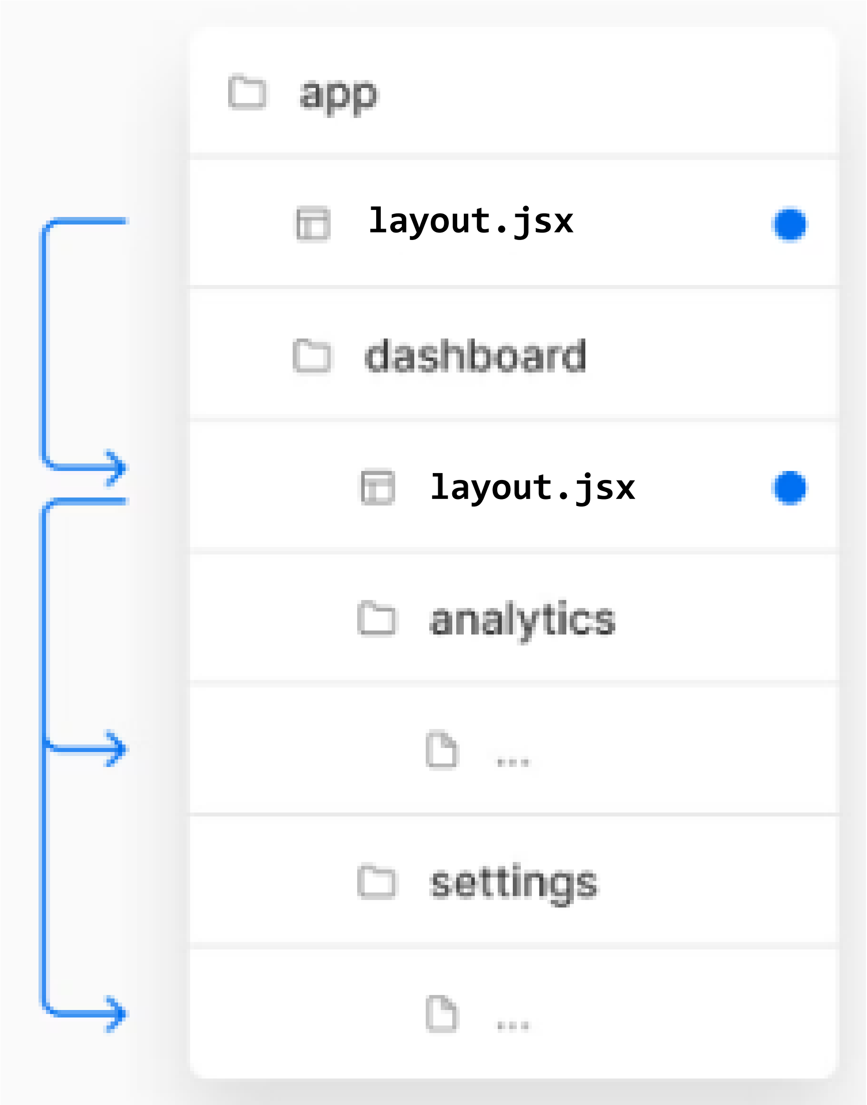
Pages are Wrapped in Layouts

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



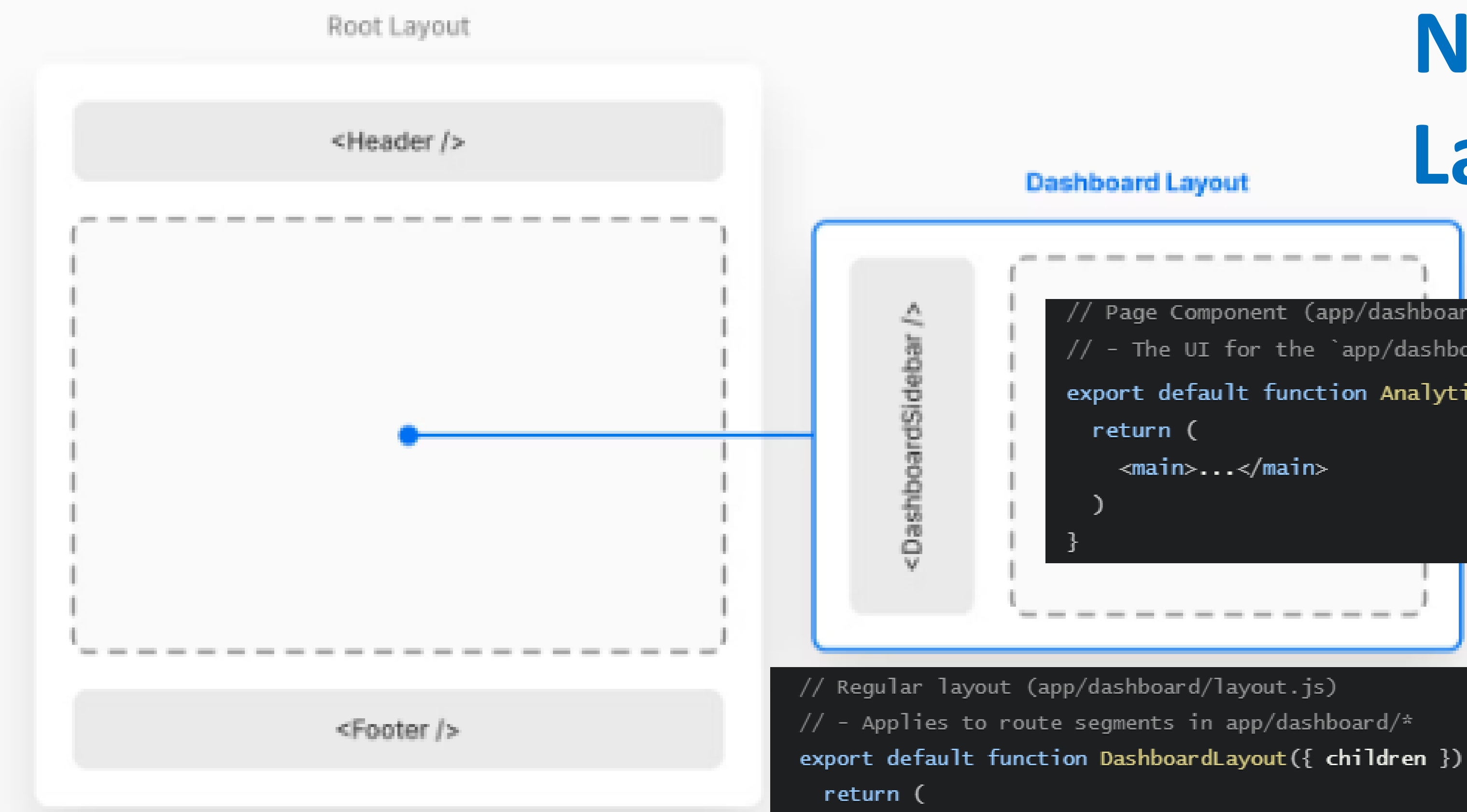
Nesting Layouts

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



```
app
  layout.jsx ●
  dashboard
    layout.jsx ●
    analytics
      ...
    settings
      ...
```

Nesting Layouts



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

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

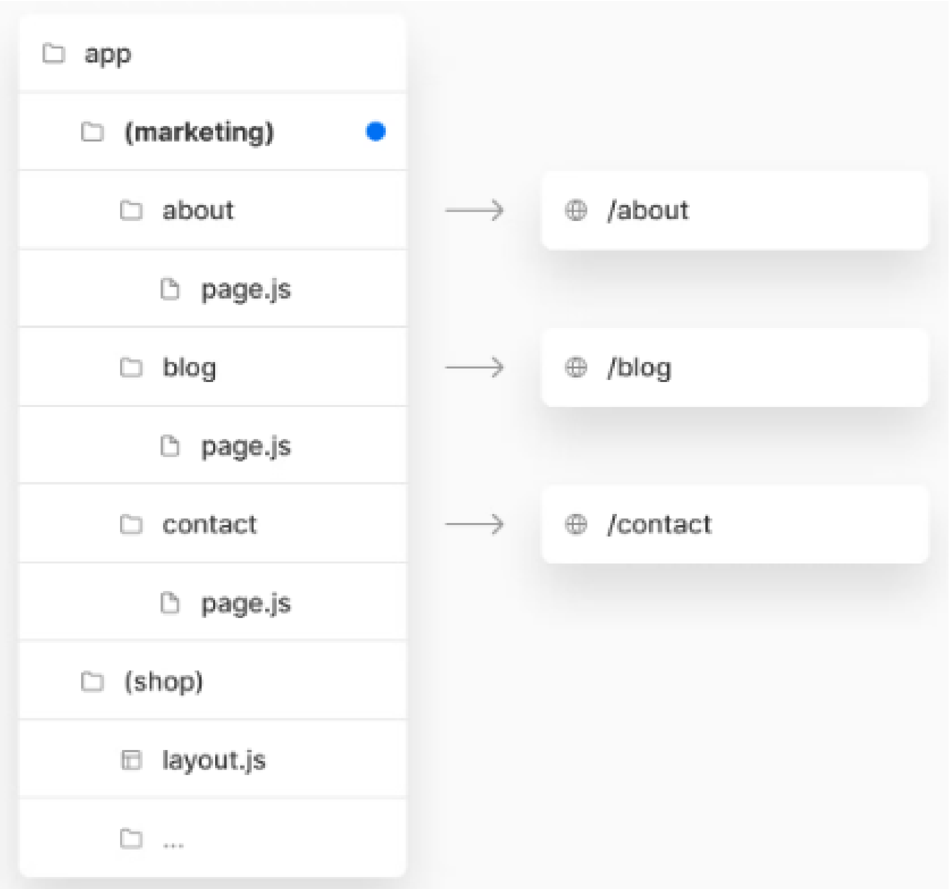
```
// Root layout (app/layout.js)
// - Applies to all routes
export default function RootLayout({ children })
  return (
    <html>
      <body>
        <Header />
        {children}
        <Footer />
      </body>
    </html>
  )
}
```

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

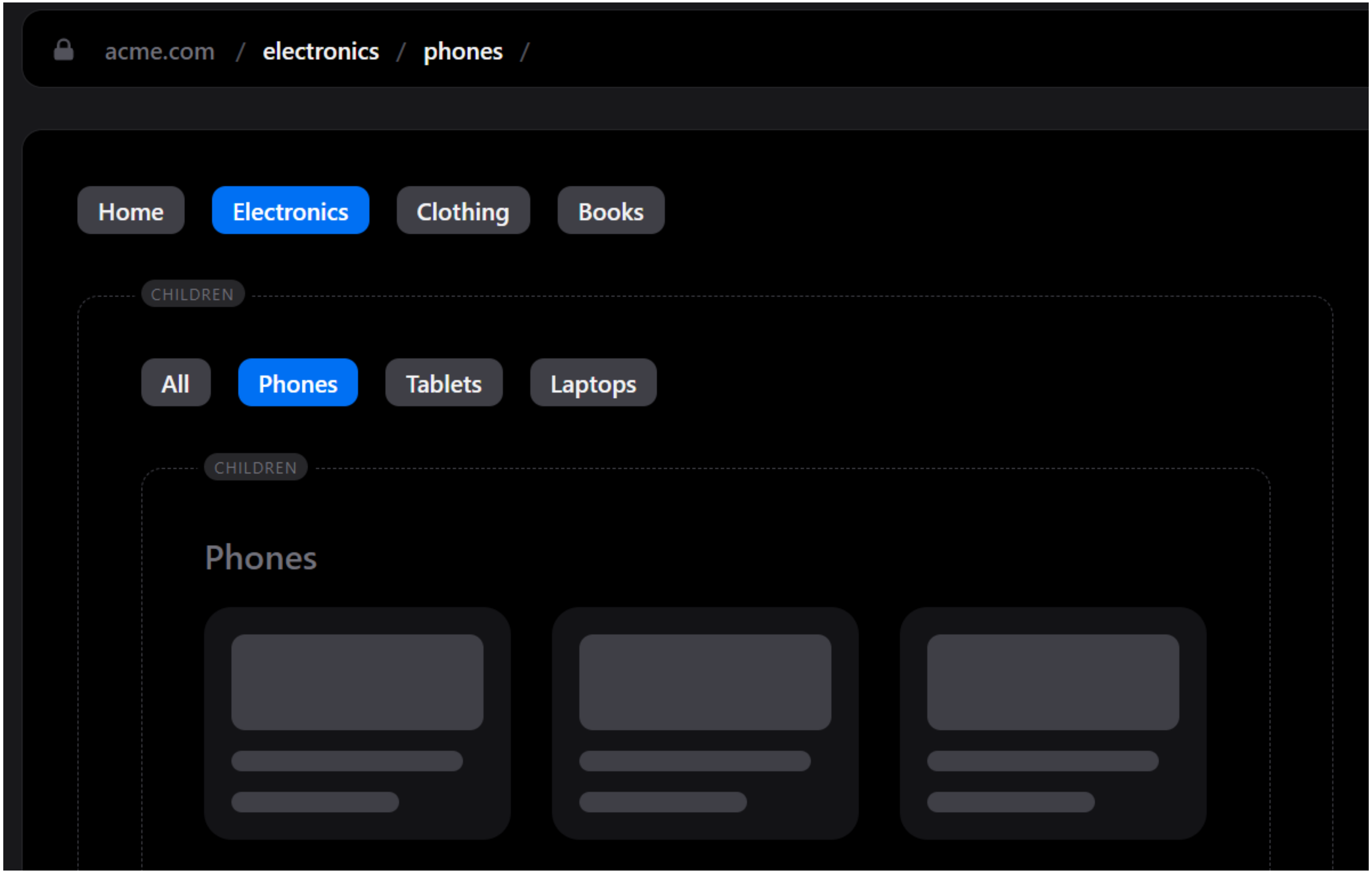
```
<RootLayout>
  <Header />
  <DashboardLayout>
    <DashboardSidebar />
    <AnalyticsPage>
      <main>...</main>
    </AnalyticsPage>
  </DashboardLayout>
  <Footer />
</RootLayout>
```

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))
  - This also allows having a different root layout per routes group



Nested Layout Example



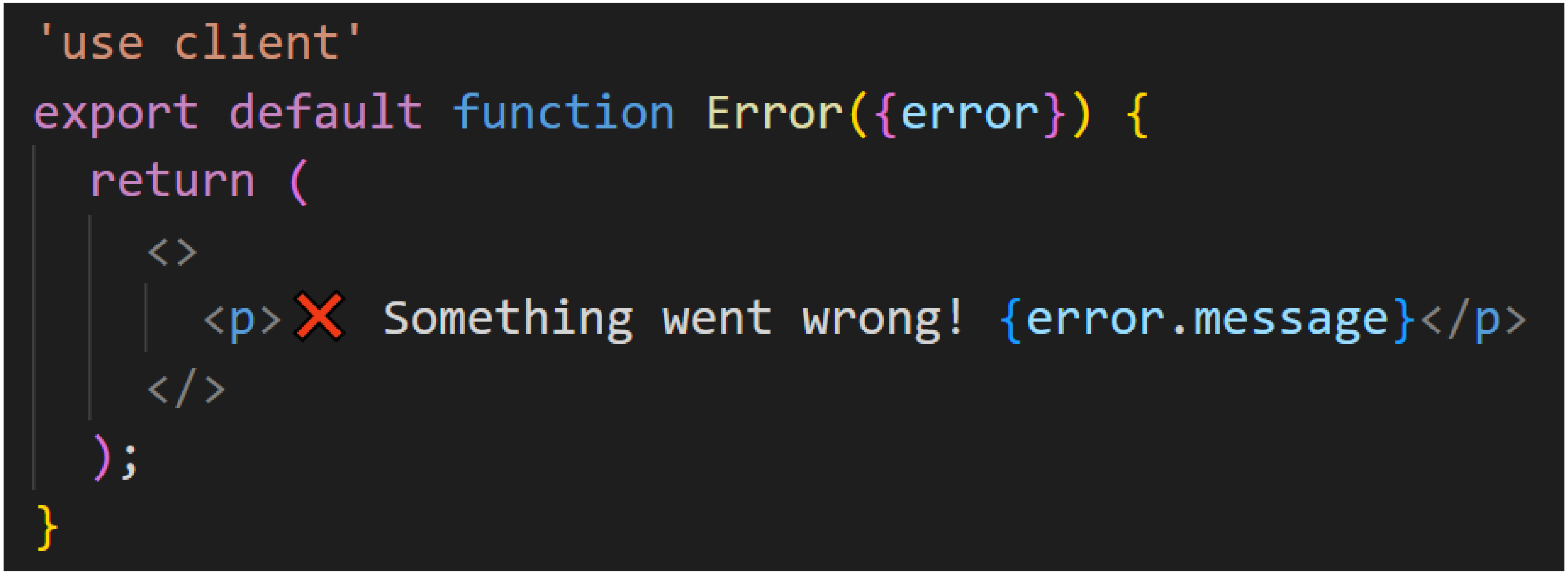
https://app-dir.vercel.app/layouts/electronics/phones

Special Files (Beyond page.jsx)

- app/ directory uses several other Special File Conventions to build complex UI:
  - **layout.jsx**: Shared **UI shell** that wraps child layouts or pages. Crucial for persisting state and avoiding re-renders during navigation
    - Every route segment can have a layout. The root layout (`app/layout.jsx`) is mandatory.
  - **error.jsx**: Defines error UI for a specific segment
  - **not-found.jsx**: Defines the UI shown when the notFound() function is thrown or a route doesn't match

error.jsx

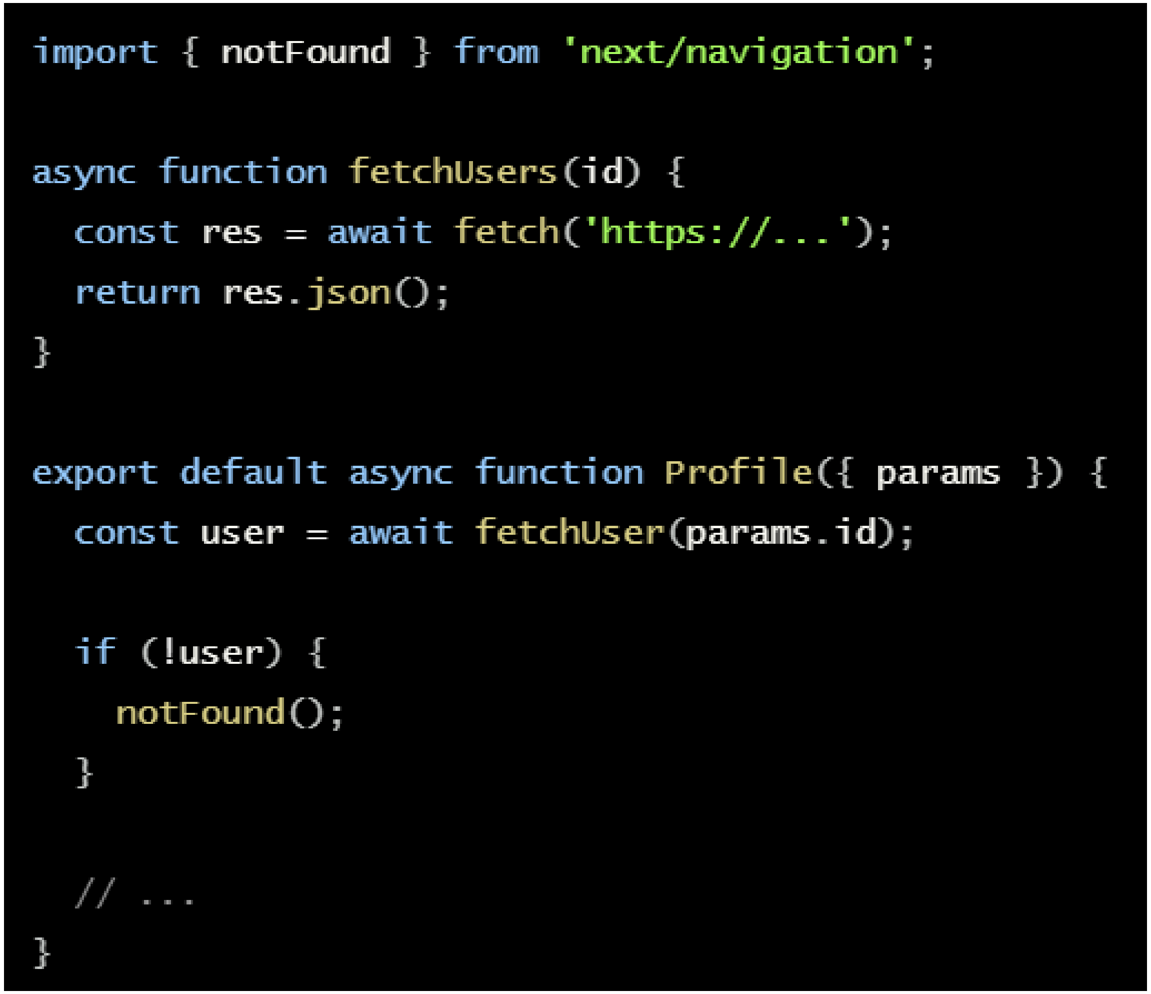
- `error.jsx` defines the error boundary for a route segment and the children below it. It can be used to show specific error information, and functionality to attempt to recover from the error
  - Should return a client-side component



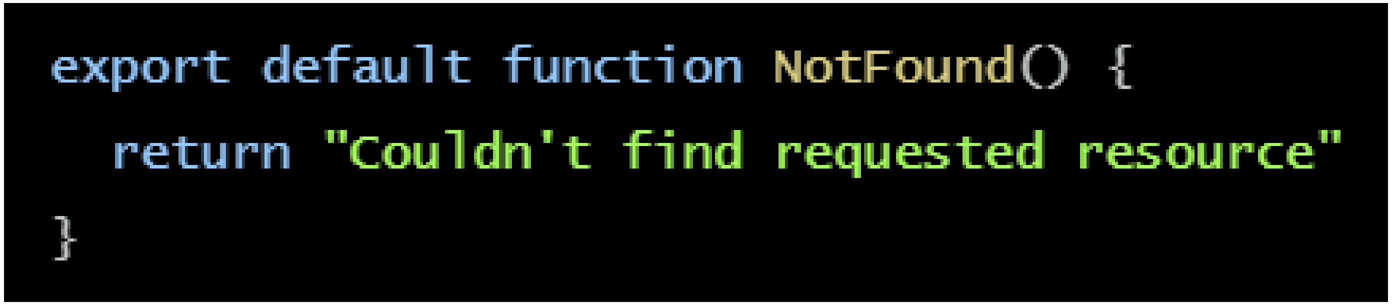
```
'use client'
export default function Error({error}) {
  return (
    <>
      <p>❌ Something went wrong! {error.message}</p>
    </>
  );
}
```

not-found.jsx

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

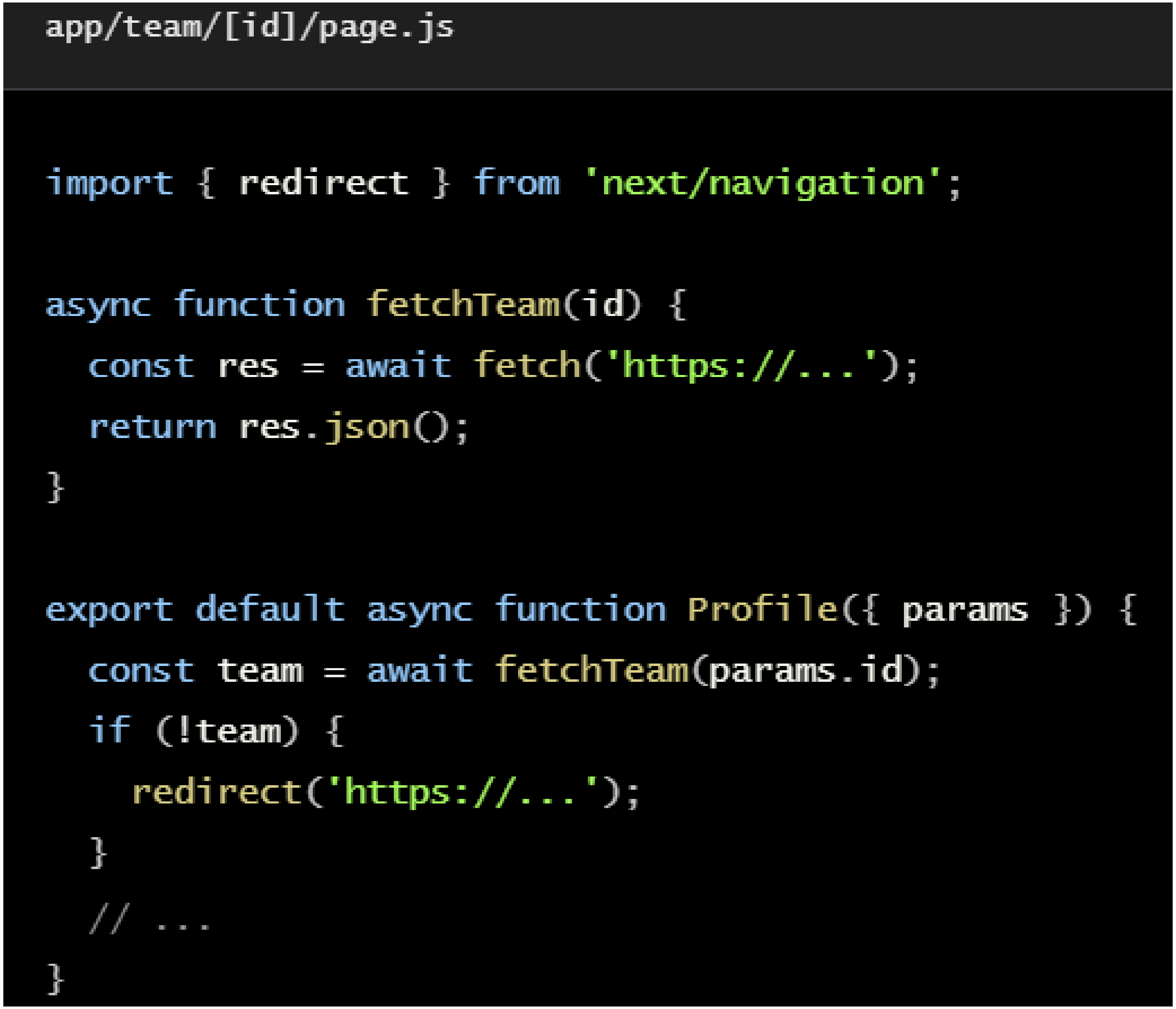


```
import { notFound } from 'next/navigation';

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()



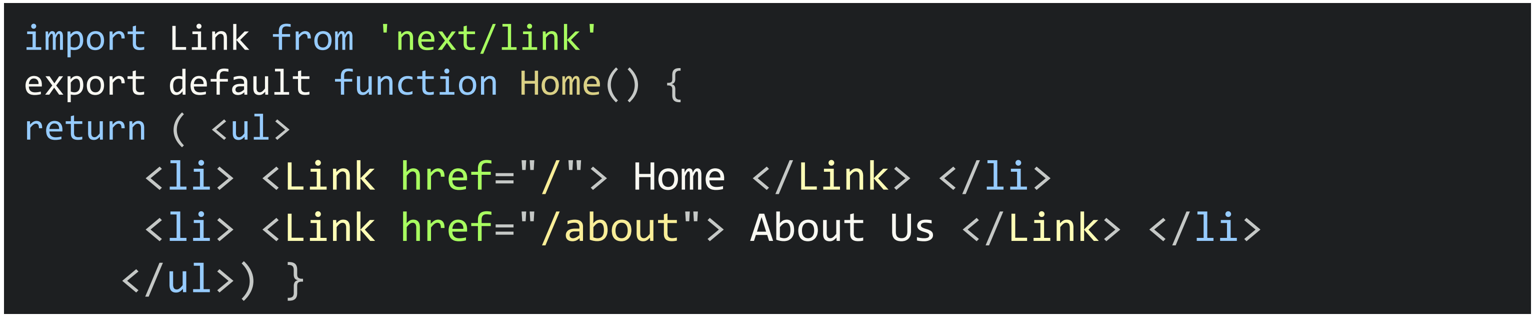
app/team/[id]/page.js

```
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

Linking between pages

- The Next.js router **Link** component to do **client-side** navigation between different routes
  - Prevents full page reloads for a faster, SPA-like experience
  - It does **partial page refresh** to display the UI of the target route in the href
  - Unlike a standard HTML <a> tag which causes a full page reload
- Prefetching (default): Pages for any <Link /> in the viewport (visible to the user) are prefetched (including static data), making subsequent navigation feel instantaneous
  - data for server-rendered routes is not prefetched.

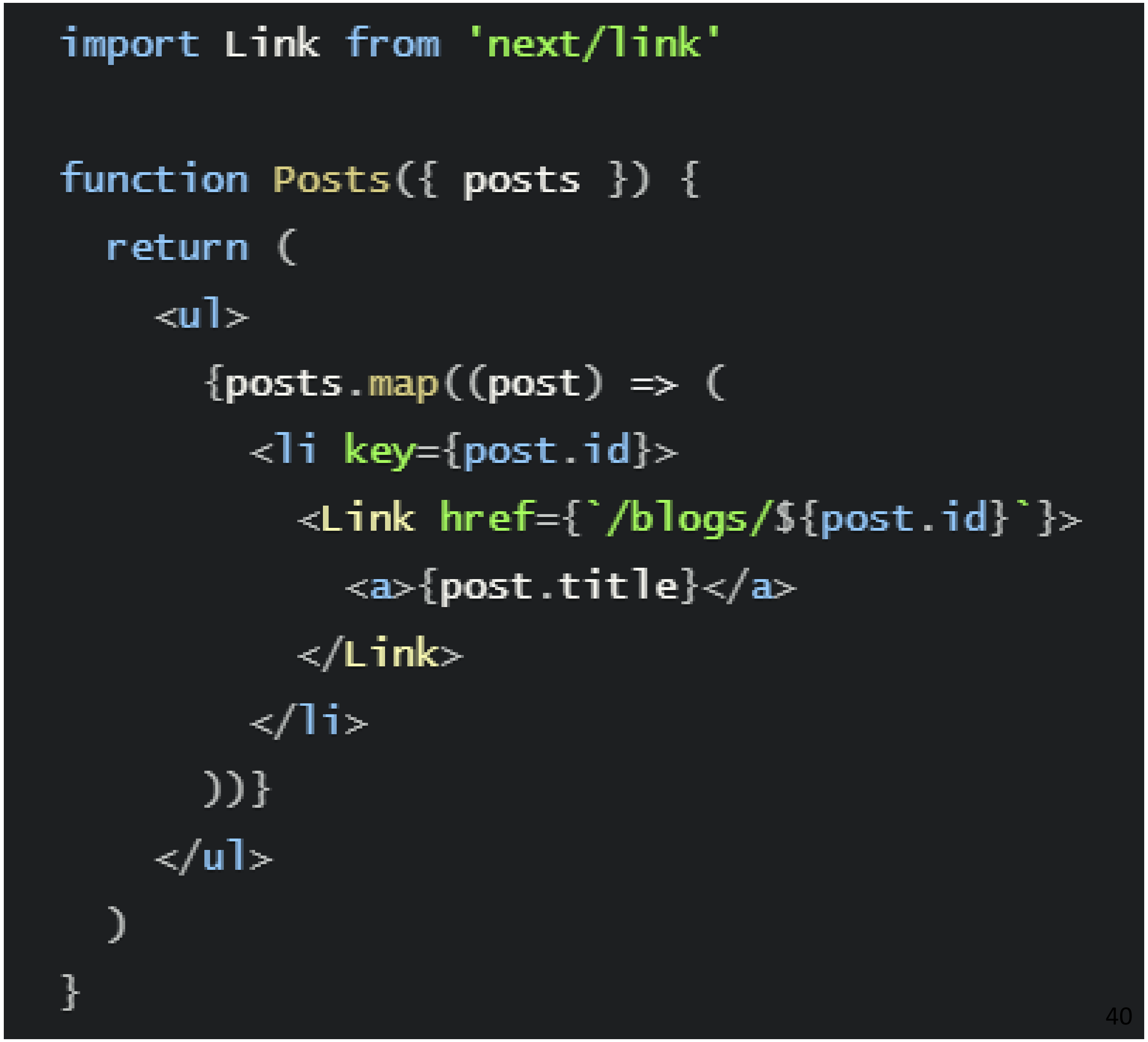


```
import Link from 'next/link'
export default function Home() {
return ( <ul>
     <li> <Link href="/"> Home </Link> </li>
     <li> <Link href="/about"> About Us </Link> </li>
    </ul>) }
```

Linking to dynamic paths

- Links can be created for dynamic paths

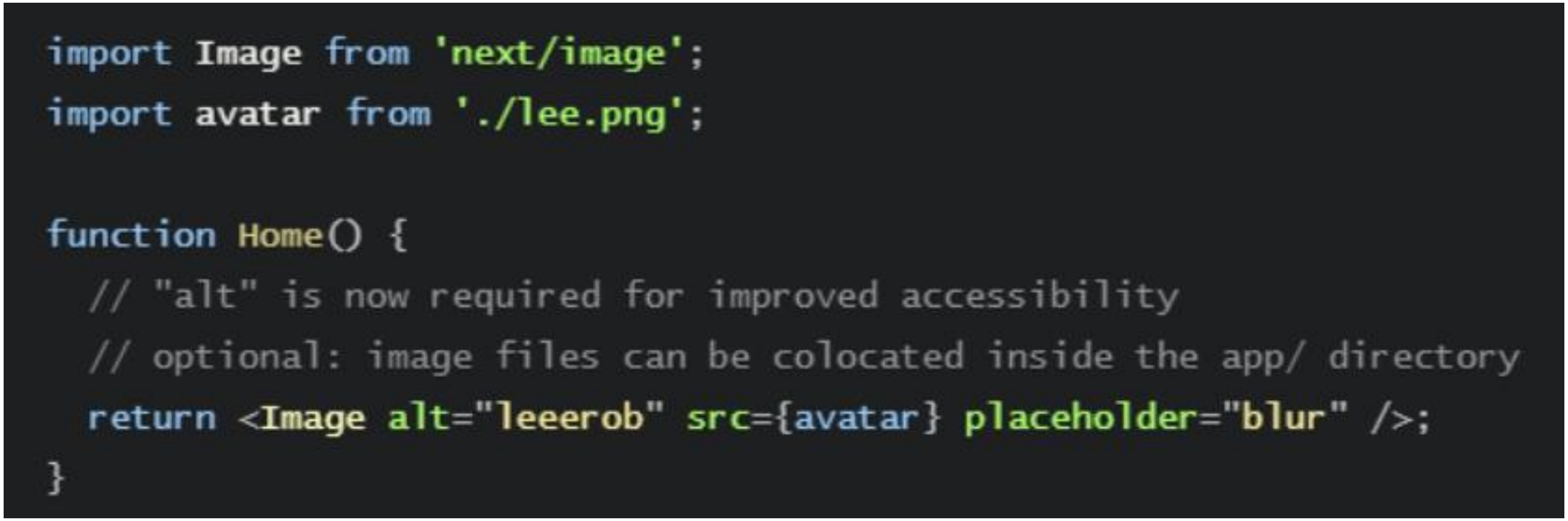
E.g., creating links to access posts for a list which have been passed to the component as a prop



```
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blogs/${post.id}`}>
            <a>{post.title}</a>
          </Link>
        </li>
      ))}
    </ul>
  )
}
```

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" />;
}
```

Server Actions

Server Actions

- Server Actions are asynchronous functions that run only **on the server** to perform server-side logic
  - E.g., Handling form submissions, data mutations (creating, updating, deleting)
    - E.g., User fills and submits a form, a server action could be used to create a new blog post, updates their profile, or adds an item to a wish list
  - They can be called directly from React components (both Server and Client Components) without manually creating separate Web API endpoints
  - **`'use server'`** Directive: to mark a function or an entire file as containing Server Actions
  - Security: Execute securely on the server, never exposing sensitive logic or credentials to the client

Server Action - Example

```
export default function Page() {
  async function createInvoice(formData) {
    'use server'
    const invoice = {
      customerId: formData.get('customerId'),
      amount: formData.get('amount'),
      status: formData.get('status'),
    }
     // Mutate data
    db.addInvoice(invoice);
    // revalidate cache to ensure the UI reflects the data changes
    revalidatePath('/invoices')
  }
   return <form action={createInvoice}>...</form>
}
```

Example Usage 1 - Handle Form Submission (CRUD Operations)

- Scenario: User fills out a contact form, creates a new blog post, updates their profile, or adds an item to a wish list
- Instead of creating a separate API route (`/api/contact`, `/api/posts`) to handle the POST request, you define a Server Action directly
    - It simplifies the code, keeps mutation logic closer to where it's triggered, and handles data submission securely on the server
    - Works seamlessly with html <form>

Example Usage 2 - Adding an Item to Card

- A list of products is displayed on a Server Component
  - Each product has an "Add to Cart" button that should add the item directly using `addToCart` Server Action
- The `addToCart` function is defined within or imported into the Server Component
  - It's marked with `'use server’`
  - The `<form>` uses the `action` prop to directly call this Server Action
  - When submitted, the form data is sent securely to the server, the action executes, interacts with the DB, and then revalidates the `/cart` path

Example Usage 3 - Quick Actions & Toggles (e.g., Likes, Bookmarks)

- While it's common to use Server Actions within <form> elements, they can also be invoked from a Client Components to handle events such as onClick
  - For example, to increment a like count:

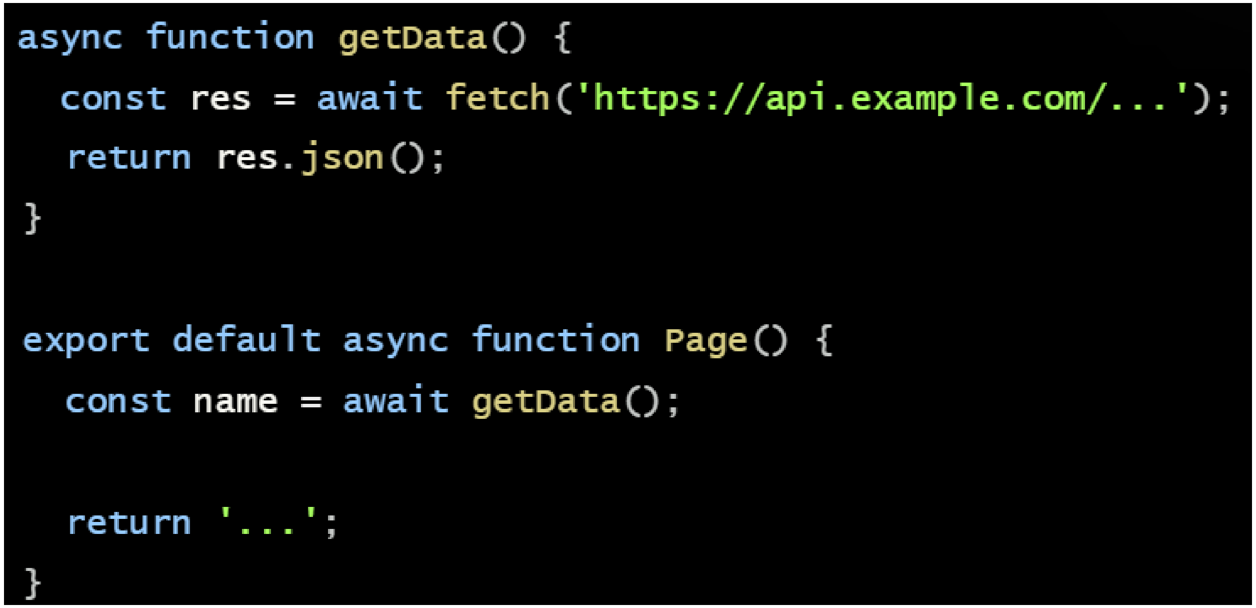
```
"use client";
import { incrementLike } from "@/app/actions/postActions.js";
import { useState } from "react";
export default function LikeButton({ postId, initialLikes }) {
  const [likes, setLikes] = useState(initialLikes);
  return (
    <> <span>Likes Count: {likes}</span>
      <button onClick={async () => {
          const updatedLikes = await incrementLike(postId);
          setLikes(updatedLikes);
        }}> Like
      </button>
    </>
  );
}
```

Data Fetching

< Back

Data Fetching

- fetch() is a Web API used to fetch remote resources and returns a promise
- 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 '...';
}
```

- Next.js extends the fetch options object to allow each request to set the desired caching and revalidating configuration

Data Fetching – Caching Config

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

- **auto no cache** (default): Next.js fetches the resource from the remote server on every request in development, but will fetch once during next build
  - If Dynamic APIs such as cookies, headers, or the searchParams are used on the route, Next.js will fetch the resource dynamically at request time to ensure the data is fresh
- **no-store**: Next.js fetches the resource from the remote server on every request, even if Dynamic APIs are not used on the route
- **force-cache**: Next.js looks for a matching request in its Data Cache
  - If there is a match and it is fresh, it will be returned from the cache
  - If there is no match or a stale match, Next.js will fetch the resource from the remote server and update the cache with the downloaded resource

Data Fetching Time-based revalidation

```
fetch(`https://...`, { next: { revalidate: number | false | 0 } })
```

Set the cache lifetime of a resource (in seconds)

- **number** - Specify the cache lifetime in seconds
  - The data is cached for the specified number of seconds (e.g., 60)
  - After the time expires, the next request gets the stale cached data immediately,
  - while Next.js triggers a background re-fetch to update cache updates for subsequent requests (i.e., revalidation)
  - Used for data that needs periodic refreshing without blocking the user
- **false** - Caches the data indefinitely (behaves like cache: 'force-cache')
  - The data is fetched once (at build time or first request) and stored indefinitely in the Data Cache until manually invalidated (e.g., using revalidatePath or revalidateTag)
- **0** - Prevents caching for this fetch request
  - Data is fetched fresh on every request
  - Behaves like `fetch(URL, { cache: 'no-store' })`

Data Fetching On-demand revalidation with revalidateTag

```
fetch(`https://...`, { next: { tags: ['posts'] } })
```

- Set the cache tags of a resource
- Data can then be revalidated on-demand using `revalidateTag`

```
'use server'
import { revalidateTag } from 'next/cache'
export default async function submit() {
  await addPost()
  revalidateTag('posts')
}
```

Summary

- Next.js = React-based full stack web framework that allows creating user interfaces, static pages, server-side rendered pages, and Web API

- Next.js has a **file-system based router**: Use folder hierarchy inside the **app** folder to define routes, and page.jsx files to define UI

- Server actions: Asynchronous functions that run only **on the server** to perform server-side logic can be used to handle form submissions or events

- Data fetching: Next.js implements the fetch() function to run the server and extends it to allow setting the desired caching and revalidating configuration

Resources

- Learn Next.js

http://nextjs.org/learn

- Next.js App Templates

https://vercel.com/templates

- Useful list of resources

https://github.com/unicodeveloper/awesome-nextjs