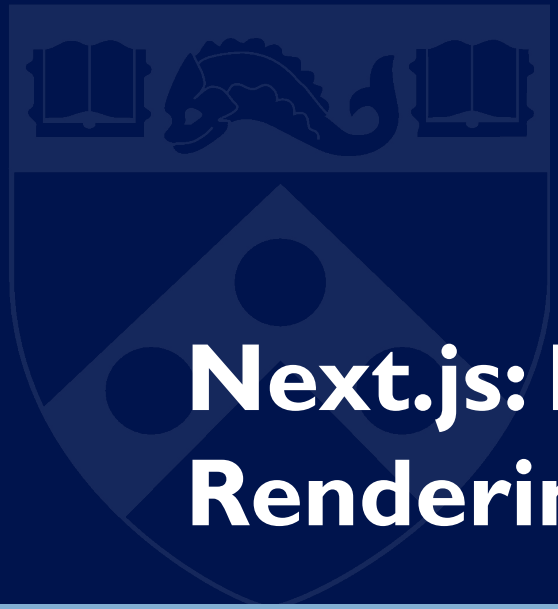


Next.js: Routing, Server-Side Rendering, Optimization

CIS 1962 (Fall 2025)
November 10th, 2025

Penn Engineering

Lesson Plan

4	React Styling
13	React Libraries
25	Next.js
30	Server-side Rendering
42	Routing
56	Optimization and SEO

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673

Styling in React

How do we write style code in React?

React Styles

We can write CSS the classic way with:

- **CSS Modules**
- **CSS-in-JS** with styled-components
- **Inline Styles and Utility Classes**

We can also make use of popular CSS and React UI Libraries:

- **CSS Libraries** like Bootstrap & Bulma
- **React UI** like MUI, React Bootstrap, & Chakra

CSS Modules

If you still want to write plain CSS, CSS Modules will allow you to classes scoped to components. This reduces collisions, and allows you to manage one CSS file per component easily.

```
import styles from './style.module.css';
// or import { myClass } from
"./style.module.css";

// function MyComponent(...) {
return <div
className={styles.myClass}>Hi!</div>;
```

```
/* style.css */

.myClass {
  font-size: 20rem;
}
```

CSS-In-JS

Alternatively, you can use a technique called CSS-In-JS where you write CSS directly in your JS code.

This keeps all your code in one place rather than in multiple files.

Additionally, since we're working with components, we extract repeated items into components for better readability!

CSS-In-JS: styled-components

`styled-components` is a popular library that allows you to insert styles right into JS.

We can define these style components with specific CSS styles and can insert them into the file immediately.

```
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`;

const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: ${(props) => props.color ??
'#bf4f74'};
`;

// function MyComponent(...) {
return (
  <Wrapper>
    <Title color="#ffccff">How Stylish</Title>
  </Wrapper>
);
```

Inline styles with Utility Classes

Writing styles inline is also a popular method, through libraries like Tailwind.

It's faster than writing full CSS properties, and many libraries will include utility classes to easily build UIs, including pseudo-classes and mobile-responsive support.

```
<div
     className={`h-full flex ${active ?"justify-center items-end lg:items-center
      lg:justify-end" : ""}`}
>
     ...
</div>
```

CSS Libraries

In contrast to writing traditional CSS or Tailwind utility classes, CSS libraries will include a lot more opinionated styles and components for different themes with little customization.

```
// Bulma
const BulmaNotification = ({ message,
type = "is-primary" }) => (
  <div className={`notification
${type}`}>
    {message}
  </div>
);
```

```
<BulmaNotification type="is-danger"
message="Something went wrong!" />
```

React UI Libraries

You can go further by installing a React UI library that provides ready-made components.

You provide props to these components that under the hood that turned into style classes.

```
// MUI
<Grid xs={12} sm={12} md={6} lg={3} xl={3}>
  <Box
    width="100%"
    backgroundColor={colors.primary[400]}
    display="flex"
    alignItems="center"
    justifyContent="center"
  >
    <StatBox
      title="32,441"
      subtitle="New Clients"
      progress="0.30"
      increase="+5%"
      icon={
        <PersonAddIcon
          sx={{ color: colors.greenAccent[600], fontSize: '26px' }}
        />
      }
    />
  </Box>
</Grid>
```

Popular UI Libraries

Utility-First CSS Libraries

- Tailwind ❤️
- Emotion
- UnoCSS
- WindiCSS

CSS-In-JS

- styled-components
- vanilla-extract
- Stiches

CSS Libraries

- Bootstrap
- Bulma
- Foundation
- Semantic UI
- Materialize
- Skeleton

React UI Libraries

- Material UI (MUI)
- React Bootstrap
- Chakra UI
- Ant Design
- … and many more!

React Libraries

What are some important React libraries to learn?

The React Ecosystem

React, while powerful for front-end, is small by itself, mostly handling UI and managing states.

It relies on libraries to better manage state, forms, animation, data fetching, style, and more.

We'll talk about 3 selected libraries:

- **Redux** (State Management)
- **React Hook Form** (Form States and Validation)
- **Framer Motion** ❤️ (Animations)

Review: Child ⇒ Parent States

React is considered unidirectional: it's easier to pass data from parent to child than from child to parent.

Function props may allow us to handle this, but this runs into issues with props (prop drilling). We can use `useContext`, but that gets messy…

```
const Parent = () => {
  const [flag, setFlag] = useState(true);
  return <Child setParentFlag={setFlag} />;
}

const Child = ({ setParentFlag }) => {
  return <button onClick={() => setParentFlag(false)}>Click Me</button>;
}
```

Enter… Redux!

Redux is a library for global state management.

We can hand off all state management to Redux rather than having React manage the states manually.

In contrast to **useContext**, which requires minimal setup but is useful for more simple, static values, Redux requires some setup and is useful for complex logic and frequently changed values across an app.

Redux Vocabulary

- **Store**
  - Holds all app states in one object/tree.
- **Actions**
  - Plain JS objects describing events (e.g., `{ type: "INCREMENT" }`).
  - You dispatch actions to update state.
- **Reducers**
  - Pure functions describing how state changes for each action.
  - Take **(state, action)** and return the next state.
- **Dispatch**
  - The method used to send actions to the reducer.

Redux with React

Using Redux with React can be streamlined with Redux Toolkit (`npm install @reduxjs/toolkit react-redux`).

This includes helpful functions for setting up stores, actions, reducers, and dispatches.

react-redux Example: Store

`configureStore` creates a store with reducers inside.

It will be provided at a top level using a Provider in `main.tsx`.

```
//  src/redux/store.ts
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from './counterSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer
  }
})

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

react-redux Example: Dispatch

`dispatch` sends your action to the store to run the specified reducer.

```
import { useSelector, useDispatch } from "react-redux";
import type { RootState, AppDispatch } from "./redux/store";
import { increment, decrement } from "./redux/counterSlice";

function App() {
  const count = useSelector((state: RootState) =>
state.counter.value);
  const dispatch = useDispatch<AppDispatch>();

  return (
    <div style={{ padding: 40 }}>
      <h1>Redux Toolkit + React</h1>
      <p>Count: {count}</p>
      <button onClick={() => dispatch(increment())}> + </button>
      <button onClick={() => dispatch(decrement())}> − </button>
    </div>
  );
}
```

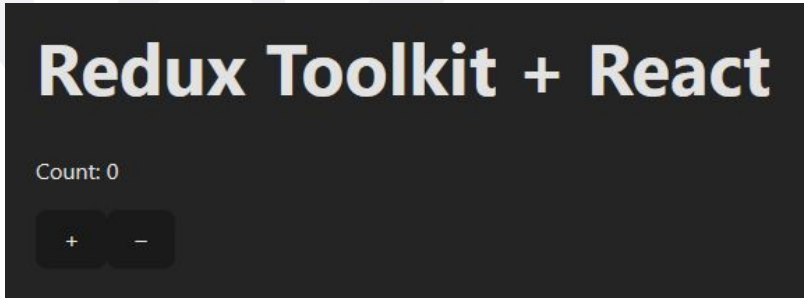
react-redux Example: Reducer

```
import { createSlice } from '@reduxjs/toolkit'

export interface CounterState { value: number }

const initialState: CounterState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 }
  }
})

export const { increment, decrement } =
counterSlice.actions

export default counterSlice.reducer
```

`createSlice` bundles together states, reducers, and action creators.

Essentially, defines **how** your states change and **what calls** those state changes.

react-redux Example



Redux Process:

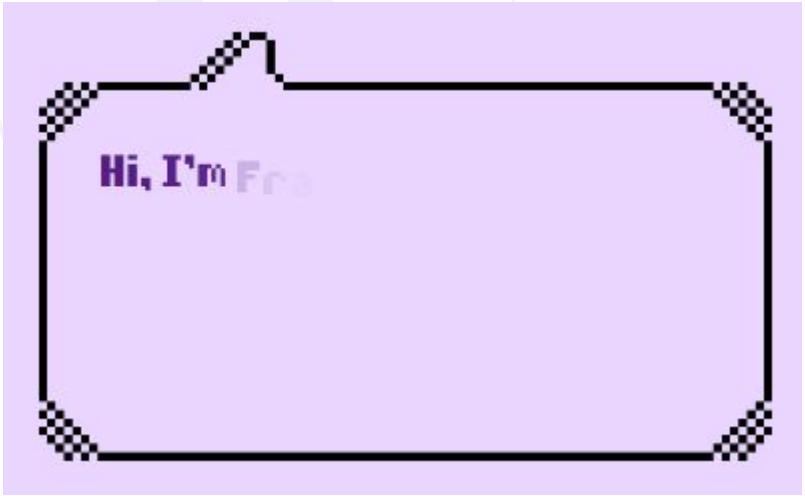
1. User clicks button
2. dispatch(increment())
3. Action { type: 'counter/increment' } sent to store
4. counterSlice.reducer runs and updates state
5. Subscribed components (count w/ useSelector) re-render with new state

react-hook-form

`react-hook-form` is a library that optimizes forms by removing re-renders and makes them easier to write and verify their inputs.

```tsx
import { useForm } from "react-hook-form";
import type { SubmitHandler } from "react-hook-form";

type FormData = {
  name: string;
  email: string;
};

const MyForm = () => {
  const { register, handleSubmit, formState: { errors } } =
useForm<FormData>();

  const onSubmit: SubmitHandler<FormData> = (data) => {
    console.log(data);
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name', { required: true, minLength: 8 })} />
      {errors.name && <span>Name required</span>}
      <input type="email" {...register('email')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Framer Motion



`framer-motion` helps you make beautiful page animations in React.

It includes support for things like mobile gestures (tapping, dragging), composite animations, orchestration/propagation animations, and much more!

Next.js

What does Next.js provide to turn React into a full framework?

Going Full Stack

So far our experience with React has mostly been dealing with the view layer of your app.

With **Next.js,** we can evolve to a full stack application with support for routing, modern server-side rendering, and a lot of optimization and SEO. Eventually we'll talk about how Next.js supports backend code with API routes, data fetching, and middleware.

Sidenote: pnpm

So far we've used npm for installations. It's often quite slow and disc-inefficient.

We recommend you try pnpm (https://pnpm.io/), which solves both of these problems!

```
npm install -g pnpm
```

Now you can run anything that used "npm" with "pnpm" instead!

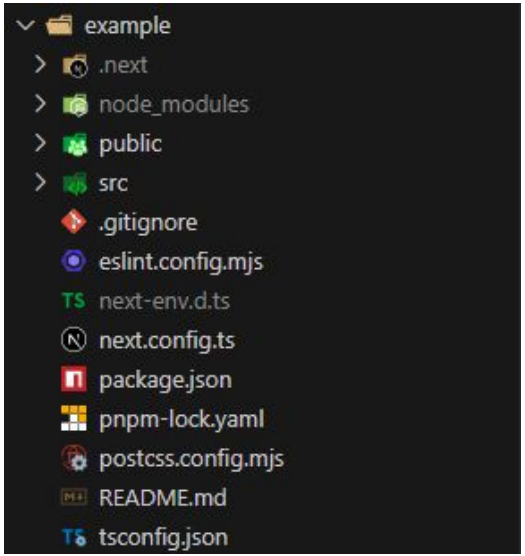
Next.js Installation

```
pnpm create next-app@latest
```

This command installs Next.js with a bunch of recommended default dependencies.

What do you get for defaults?

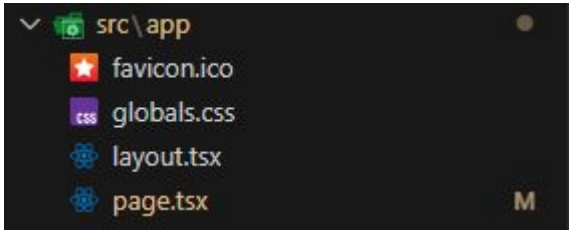
- TypeScript & zod
- Tailwind, PostCSS, and clsx for style
- bcrypt for authentication
- PostgreSQL for database connections
- A local git repository



But… where's the HTML?

Unlike normal React, you will notice that there is no index.html template file.

This is due to how Next.js uses the file system for routing with support for server-side rendering.



The HTML will be sent from the server, rather than populating an index.html template.

Server-Side Rendering

What is server-side rendering and how does it differ from the client-side rendering were used to?

Single Page Applications

Single page applications, or SPAs, are a web application architecture that uses client-side rendering to allow navigation on a page without reloading.

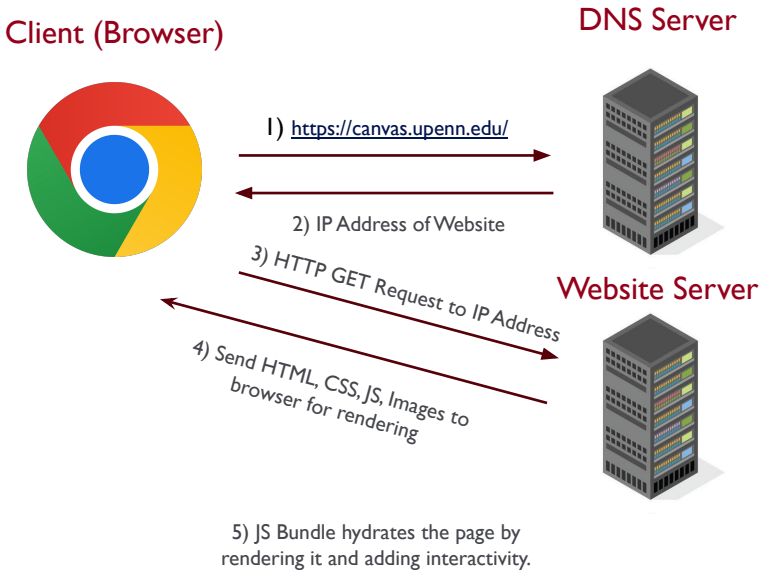
Often this delegates navigation and updates to the browser to run JS code for rendering, such as using React.

Client-Side Rendering

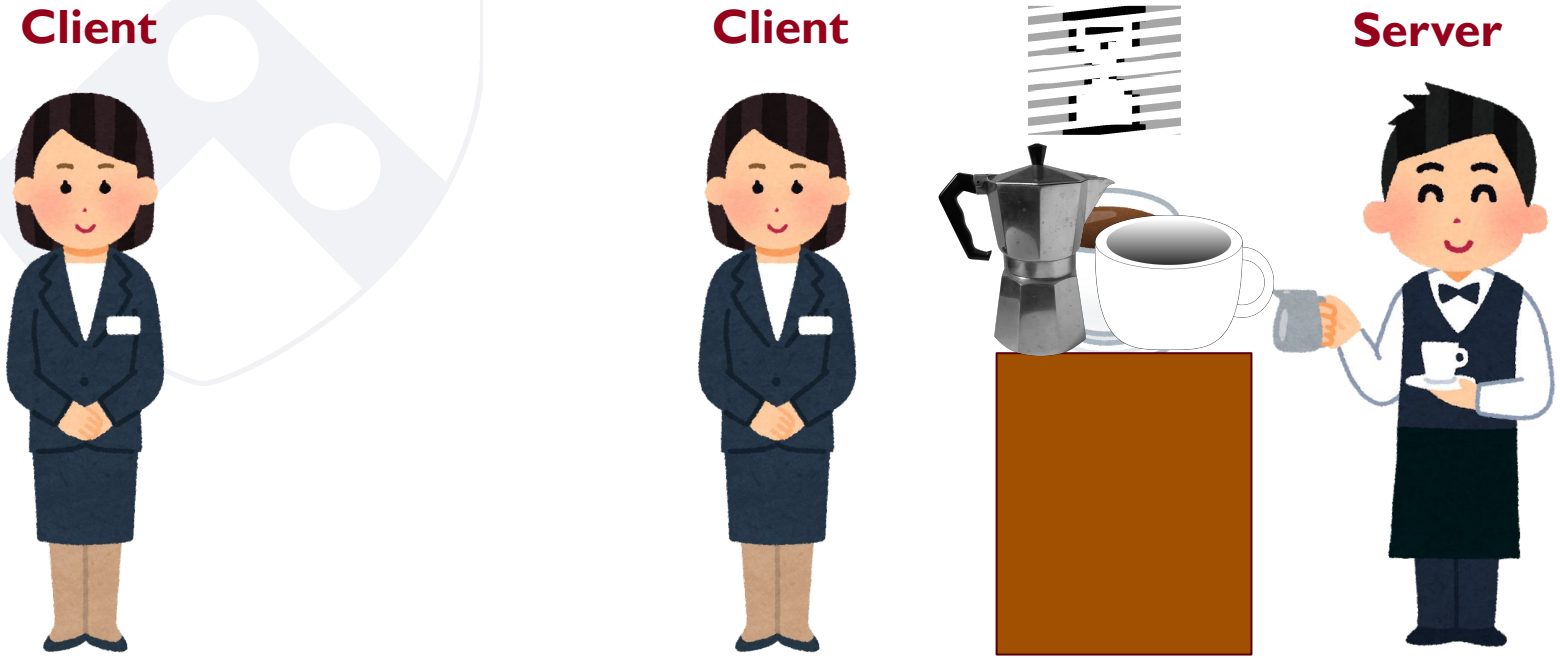
Client-side rendering is the standard for React apps.

When a user requests a page for a React app, the browser receives the template index.html (with just `<div id="root"></div>`).

Then React "hydrates" this page with UI after receiving bundled JS code.



Client-Side Rendering Analogy



"Imagine ordering a coffee and getting an empty coffee cup (index.html) and the materials to make your coffee yourself (JS). After taking a bit of time to make your coffee, you can make the coffee yourself a little faster.

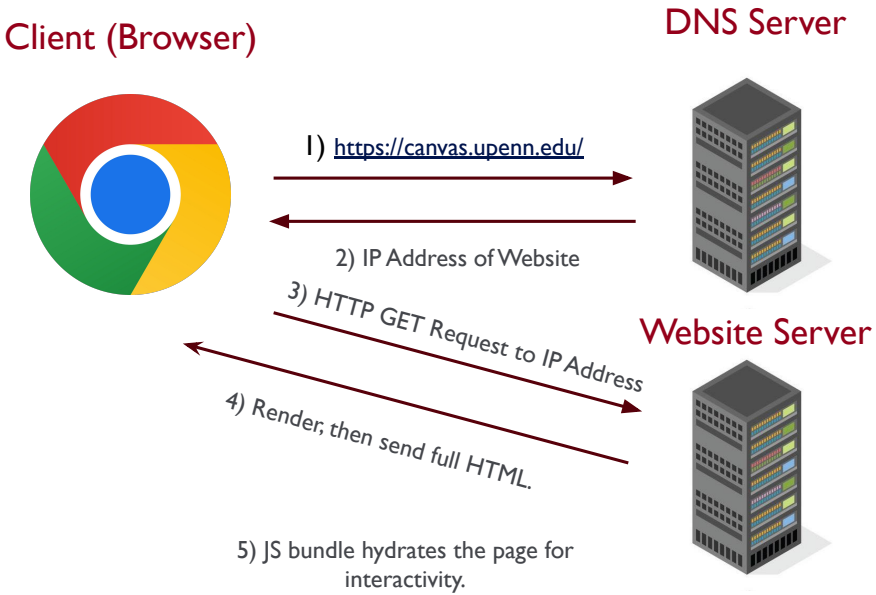
Cons of Client-Side Rendering

- Slow initial page load: while React is hydrating a page, only the empty template HTML is shown
- Bad SEO & Scraping: Search engines may be unable to see the actual content on the page, only your template HTML
- JS Dependency: CSR depends on the network to fetch JS, thus it may be slow to load.

Server-Side Rendering

In contrast to client-side rendering, server-side rendering (SSR) delegates the task of rendering a page to the website server, and then sends the full HTML back when requested.

JS will then hydrate the page with interactivity.



Server-Side Rendering Analogy

"Imagine ordering a coffee, waiting for the barista to finish everything, and then getting your ready-to-drink coffee (HTML and JS). Every time you want more coffee, you'll have to wait for the barista to make the coffee again.

React Server Components

React Server Components (RSC) are a relatively new option in React (stable in React v19) that generate a React tree as an output.

This reduces bundle sizes (bundles won't include libraries), and allow access to the backend using async functions.

However, cannot have interactive APIs (hooks, eventHandlers)

```
import marked from 'marked'; // no bundle size
import sanitizeHtml from 'sanitize-html'; // no bundle size

async function Page({page}) {
 const content = await file.readFile(`${page}.md`);
 return <div>{sanitizeHtml(marked(content))}</div>;
}
```

Next.js Hybrid Rendering

By default, using the **App Router** for Next.js gives you Server Components.

```
export default async function Home() {
  const data = await fetchData();
  return <div>{data.someValue}</div>;
}
```

Since Server Component cannot have interactivity, we can use the directive "`use client`" to turn it into a Client Component, so that the code can be hydrated with interactivity in the browser.

```
"use client";
import { useState } from 'react';
export default function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

RSC/Next Rendering Analogy

"Imagine ordering a coffee. The barista goes behind a curtain to prepare the secret ingredients to the coffee (server component). Then you are handed your coffee, which you then can add milk/sugar (client component).

Summary

Client-Side Rendering (CSR):

- Renders content in the browser using JavaScript.
- Fast client navigation, but slower initial load.

Server-Side Rendering (SSR):

- Renders HTML on the server and sends it to the browser.
- Faster initial content; better for SEO, but more stress on the server

React Server Components (RSC):

- Runs some components only on the server.
- Reduces bundle size; enables server-only data fetching.

5-Minute Break!

Routing with Next.js

What is routing and how does Next.js handle it?

Routing on the Web

Routing refers to mapping URLs to code that handle those URLs.

This can take the form of:

- **Front-end routing**: navigating to new pages without reloading the webpage
- **Back-end routing**: executing HTTP methods such as GET and POST to retrieve and submit data

Routing with React

While normal React doesn't include routing, there are some popular libraries that provide routing to React:

- React Router: Client-side routing through the browser, used with SPAs
- Express.js: Server-side routing through HTTP requests to a website server, used for SSR

React Router Example

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";
import Home from './Home';
import About from './About';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home/>} />
        <Route path="/about" element={<About/>} />
      </Routes>
    </BrowserRouter>
  );
}
```

Express Router Example

```
const express = require('express');
const app = express();

// Route for home page
app.get('/', (req, res) => {
  res.send('Homepage');
});

// Route for about page
app.get('/about', (req, res) => {
  res.send('About page');
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Routing with Next.js

Next.js uses the file system for routing.

It features 2 different routers:

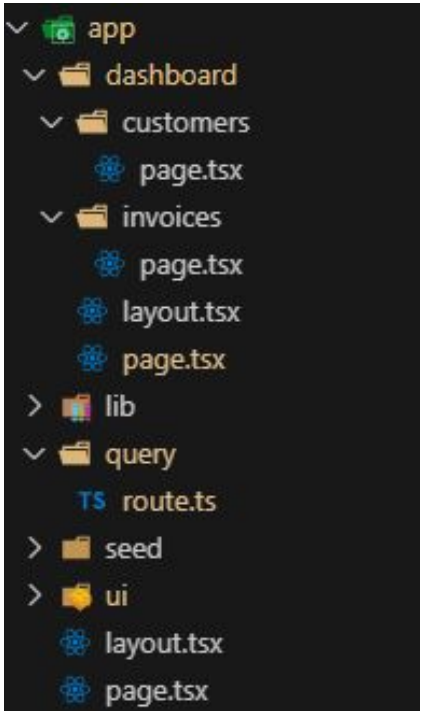
- **Pages Router**: the original router for Next.js, which is still supported. Uses the /pages directory.
- **App Router**: A new router that uses React Server Components. Uses the /app directory.

We'll be focusing on the App Router.

The App Router

The App Router uses specific files within the app folder to expose and provide layouts for routes:

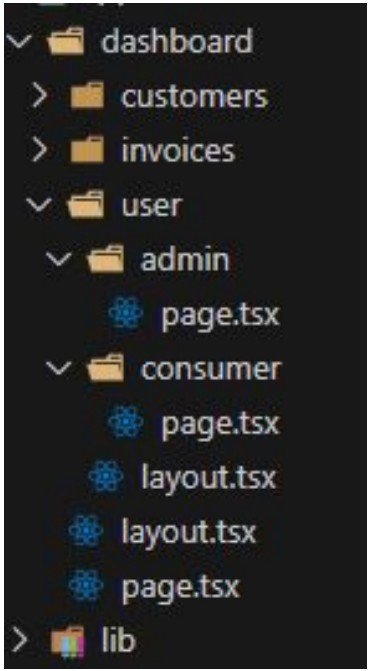
- `page`: exposes a route
- `layout`: shared UI to this route and it's children
- `route`: API endpoint
- `loading`: loading UI
- `not-found`: 404 Not Found UI



The App Router: Nesting Routes

You can use folders to nest routes. If a folder contains a page file, it will public as a URL that can be accessed.

For instance,
`/dashboard/user/consumer` and `/dashboard/user/admin` are public, but `/dashboard/user` only contains a layout file, thus it's not public.



page.[js,jsx,tsx]

The page file is a route segment, a special file that defines a UI for a specific route.

Any file path route that contains a page file gets exposed publicly, and can be used.

```
// Tailwind CSS classes excluded
export default function Page() {
  return (
    <div className="...">
      <h1 className="...">Hello World!</h1>
    </div>
  )
}
```

/example

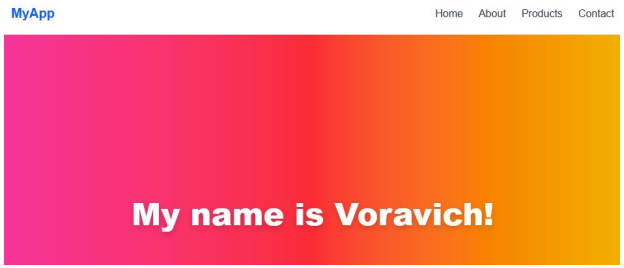
layout.[js,jsx,tsx]

The `layout` file is another route segment, a special file that defines shared UI for routes. Any route in the current folder and subfolders will have the layout defined in this file.

This file uses a similar structure to Wrapper Components (props.children) to render the layout in the children.

```
const navLinks = [...];

export default function Layout({ children }: {
children: React.ReactNode }) {
  return (
    <>
        <nav className="...">...</nav>
        <div className="...">{children}</div>
    </>
  );
}
```

/example/about

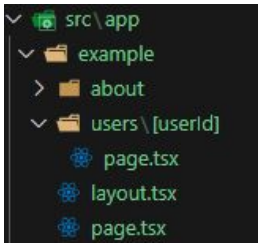


Dynamic Routes

Sometimes you want your routes to be dynamic, by taking in a parameter.

You can specify this by naming folders/segments with square brackets, treating those folders as normal routes with dynamic names. You can get these parameters back using params.[paramName] through async calls in the JSX.

```
export default async function Page({params}: PageProps) {
  const { userId } = await params
  const user = await getUser(userId);

  return (
    <div>
      <h1>{user.name}</h1>
      <p>User ID: {user.id}</p>
    </div>
  );
}
```

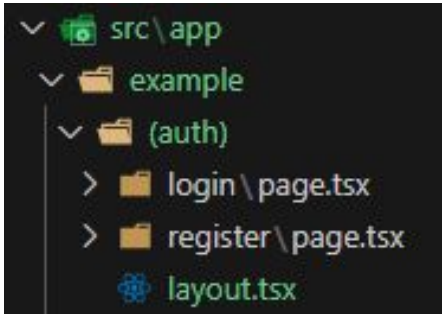


/example/users/12

Route Groups

You can wrap a folder name in parentheses () to define a **route group**.

This lets you organize your routes without affecting the folder structure, and is useful for sharing things like layouts or loading components.



Valid Routes:
/example/login
/example/register

Metadata files

Specific files can be used to embed metadata for SEO and add social media previews so pages can have better visibility when shared.

- `favicon.ico`
- `icon[.ico,.jpg,.jpeg,.png,.svg]`
- `apple-icon[.jpg,.jpeg,,.png]`
- `opengraph-image[.jpg,.jpeg,.png,.gif]` (preview image shown for social media links)
- `sitemap.xml`
- `robots.txt` (guidance for search engine crawlers)

App Router Overview

File system-based Routing starting from `/app` directory

Specific Routing files:

- `page` exposes a route
- `layout` for shared UI
- `loading` for loading states
- `not-found` for error handling

Metadata files like `favicon`, `icon`, `opengraph-image`, and `sitemap`

Dynamic Routes with [] for page parameters in routes

Route Groups with () for grouping in file system

Optimization and SEO

What features does Next.js provide for optimization and SEO?

Next.js Built-in Components

Next.js includes many built-in components you can import into your app, tailor-made for optimization and SEO.

These include Components like:

- `<Link>`
- `<Image>`
- `<Font>`
- `<Script>`

Code Splitting

Code splitting divides your app's JS and rendering into smaller chunks that load only when needed.

React can do this with React.lazy() and <React.Suspense> for lazy loading, while Next.js has this as default.

This results in faster page loads and better SEO search rankings.

Links & Route Pre-Fetching

`<Link>`, imported from `next/link`, allows for navigation between pages, similar to an `<a>` tag.

However, `<Link>` enables **client-side navigation**, not requiring a reload like `<a>`.

Additionally, when a `<Link>` is visible on a page, Next.js **pre-fetches** that page's bundle/data in the background, speeding up navigation.

```jsx
import Link from 'next/link';

export default function Navbar() {
  return (
    <nav>
      <Link href="/">Home</Link>
      <Link href="/about">About</Link>
      <Link href="/contact">Contact</Link>
    </nav>
  );
}
```

Fonts

`next/font` optimizes fonts by preventing layout shift.

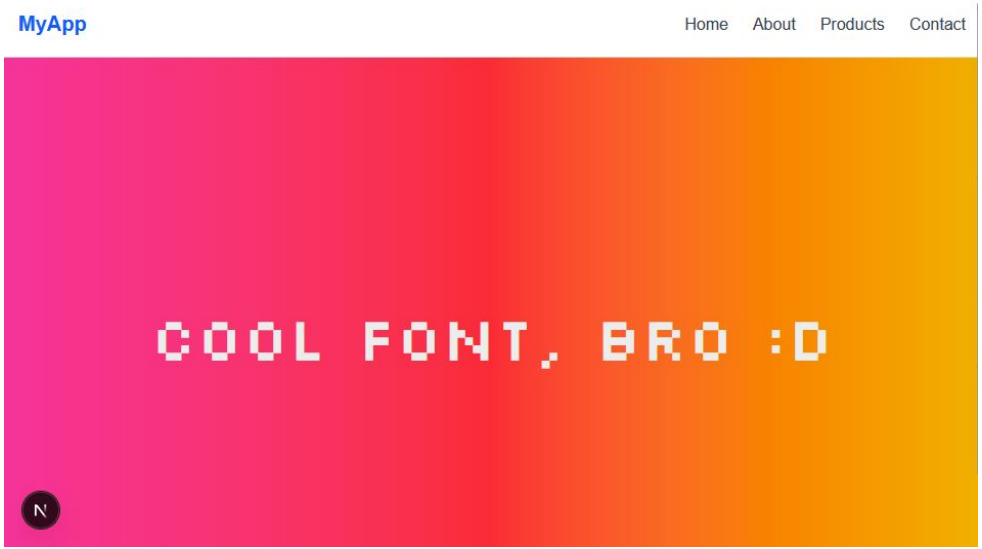
Instead of needing to fetch fonts during runtime (like from Google Fonts), it downloads them during **build time** and hosts them with your other static assets.

```
import { Silkscreen } from 'next/font/google';

const silkscreen = Silkscreen({
  subsets: ['latin'],
  weight: ['400', '700'],
});

export default function Page() {
  return (
    <div className={`${silkscreen.className} text-6xl`}>
        Cool font, bro :D
    </div>
  )
}
```

Fonts

`next/font` optimizes fonts by preventing layout shift.

Instead of needing to fetch fonts during runtime (like from Google Fonts), it downloads them during **build time** and hosts them with your other static assets.



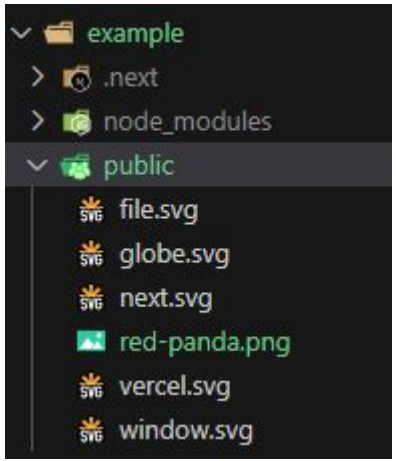
Images

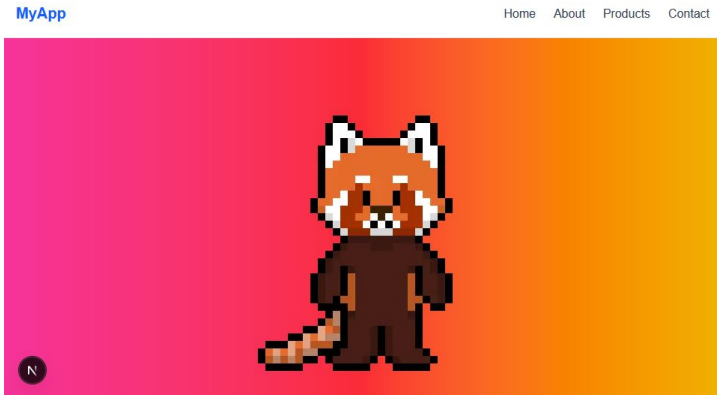
`<Image>, from next/image`, optimizes your images by providing:

- **Automatic Resizing/Mobile Responsive**: no need to manually resize for different screen sizes!
- **Lazy Loading**: Images only load when they enter the viewport
- **WebP/AVIF support**: Can use image formats for smaller file size
- **Reduce layout shift**: Improves SEO and UX metrics for websites

Images Example

```
import Image from 'next/image';

export default function Page() {
  return (
    <Image
      src="/red-panda.png"
      width={400}
      height={400}
      alt="pixel red panda"
    />
  )
}
```





400x400: exact size of image!

Streaming

Dynamic components that need to retrieve from a database can take a while to load.

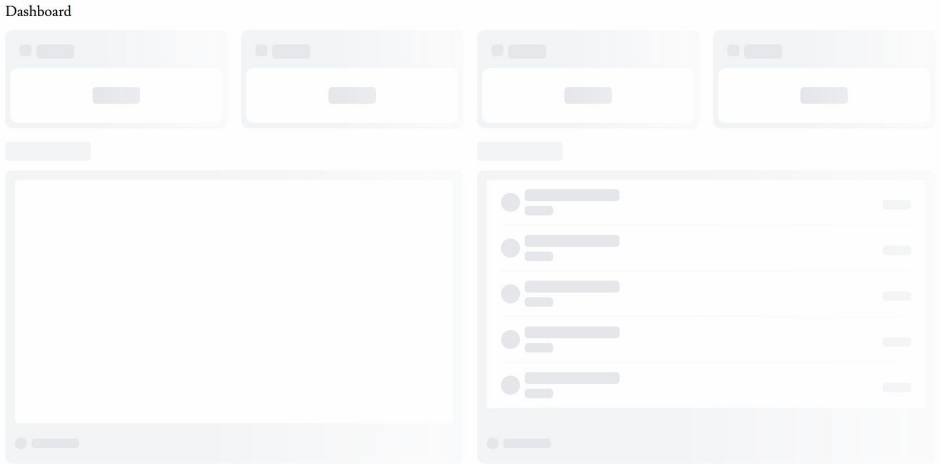
Streaming allows you to send parts of a webpage as soon as they are ready rather than waiting for the whole thing.

This improve the user experience and benefits user engagement metrics for SEO.

Streaming Example

```
export default async function Page() {
  return (
    <main>
      <h1 className="...">
        Dashboard
      </h1>
      <div className="...">
        <Suspense fallback={<CardsSkeleton />}>
          <CardWrapper />
        </Suspense>
      </div>
      <div className="...">
        <Suspense fallback={<RevenueChartSkeleton />}>
          <RevenueChart />
        </Suspense>
        <Suspense fallback={<LatestInvoicesSkeleton />}>
          <LatestInvoices />
        </Suspense>
      </div>
    </main>
  );
}
```

`<CardWrapper/>`, `<RevenueChart />` and `<LatestInvoices />` implement async database fetches



SSR and SSG

Server-Side Rendering (SSR): Each page request with Next.js returns the full HTML. This allows search engine crawlers to properly see the fully rendered page and index the content.

Static Site Generation (SSG): Pages are rendered at build time, served as static HTML files. `generateStaticParams` can be used to generate static data for these pages. This method allows for fast, cacheable pages.

Metadata

You can easily set metadata using `export const metadata` API in your files.

This allows for fine control over the data such as titles, descriptions, and social previews on each page of your app.

Root layout.tsx

```tsx
import type { Metadata } from "next";

export const metadata: Metadata = {
  title: {
    template: '%s | Example App',
    default: 'Example App',
  },
  description: 'Example code for lecture 9 of CIS 1962 Fall 2025.',
};
```

/example page.tsx

```tsx
export const metadata: Metadata = {
  title: 'Example',
};
```

Next.js Optimization Summary

Smart Loading

- Splits code, images, and fonts—delivers only what's needed, when it's needed.

Instant User Experience

- Prefetches and streams content for smooth, fast navigation.

Designed for SEO

- Renders optimized HTML and metadata for search engines and sharing.