Next.js

CIS 1962 (Winter 2026)
April 2nd, 2026

Penn Engineering

Lesson Plan

5	Next.js
10	Server-side Rendering
21	Next.js Routing
34	Next.js Backend
41	Optimization and SEO

Logistics

Homework 7 and Project proposal due 4/5 (Sunday!)

Please submit the proposal in a timely manner to get feedback!

Review Activity

https://edstem.org/us/courses/91614/lessons/163507/slides/961370

Next.js

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

Next.js: React for 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.

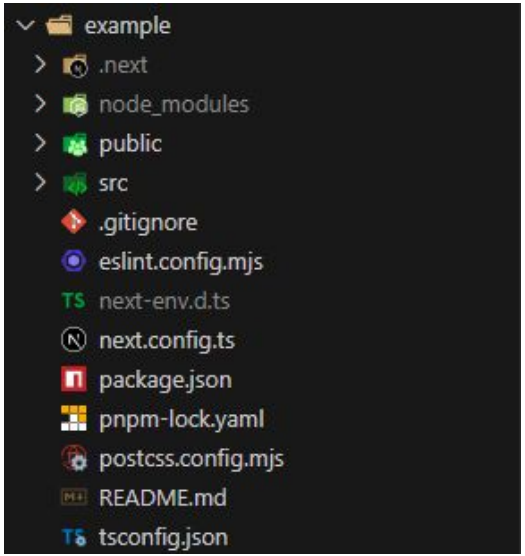
Next.js Installation

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

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

What do you get for defaults?

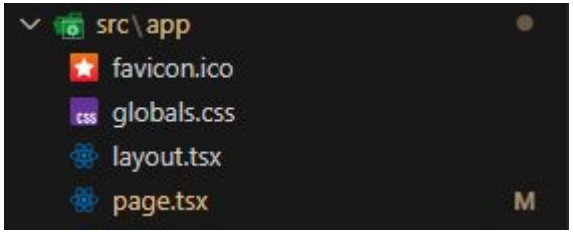
- TypeScript
- Tailwind for style
- eslint
- A local git repository



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.

Class Demo: Next.js Tour

https://edstem.org/us/courses/91614/lessons/163507/slides/961419

pnpm create next-app@latest

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.

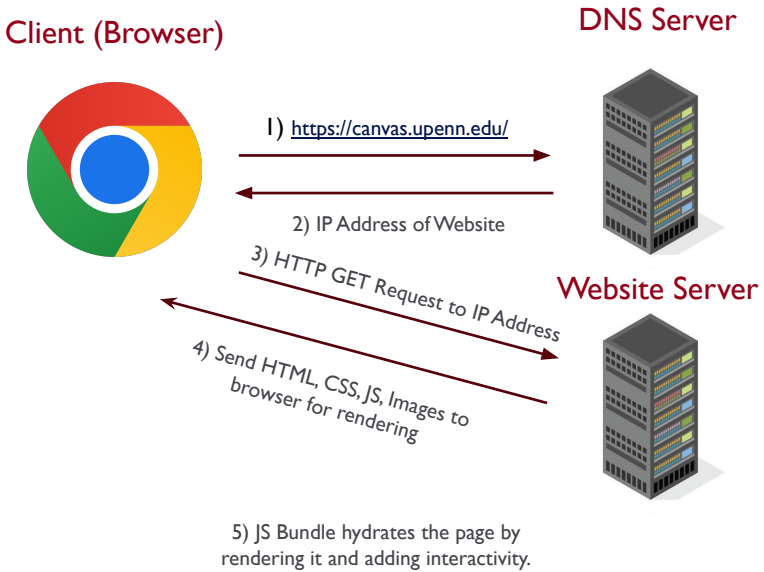
Often this delegates navigation and updates to the browser to run JS code for rendering, such as using React.

Client-Side Rendering

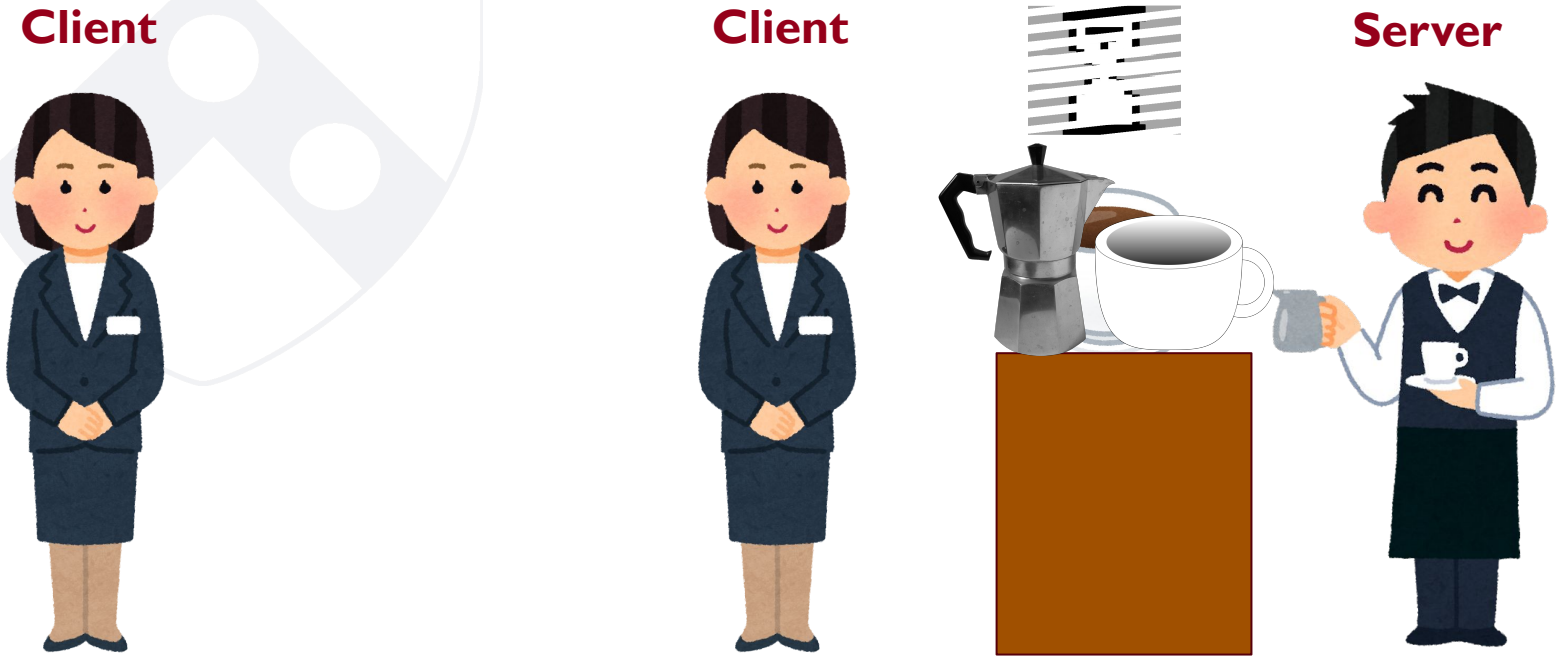
Client-side rendering is the standard for React apps.

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.

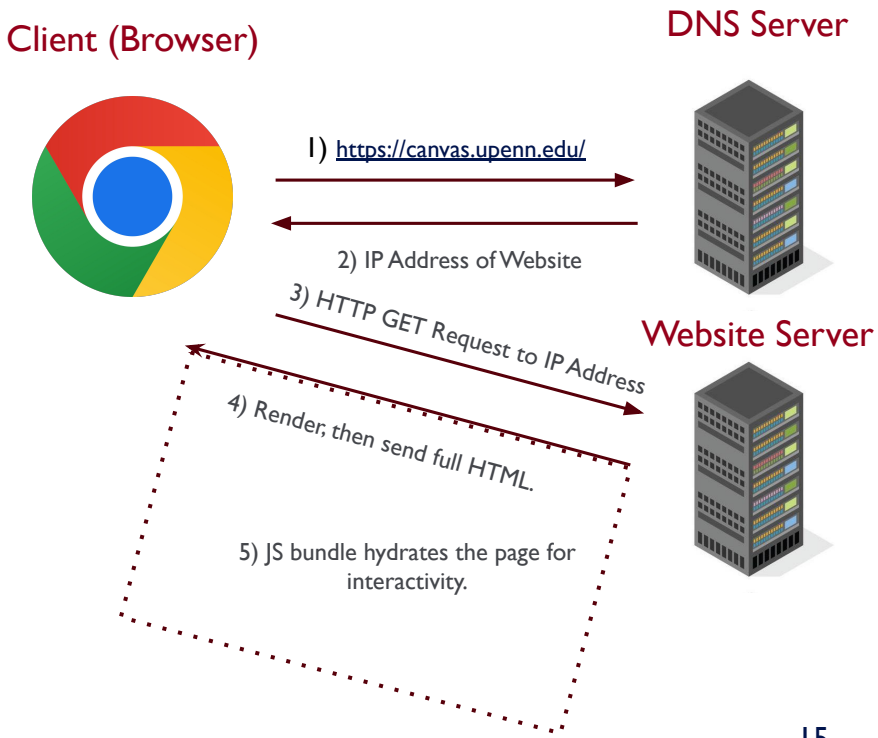
Cons of Client-Side Rendering

- 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 Components 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) & Next:

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

Routing with Next.js

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

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:

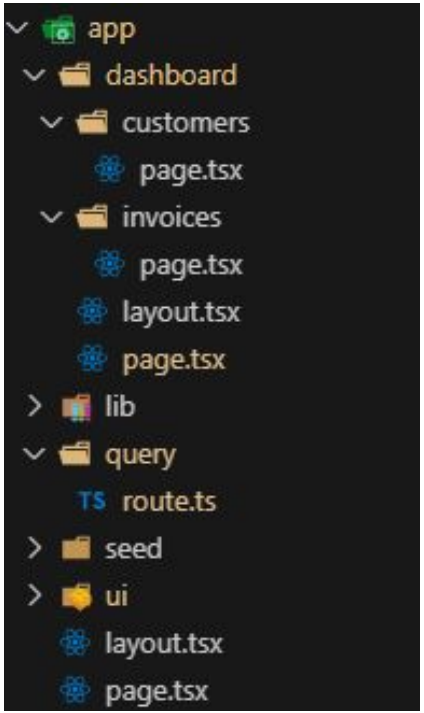
- **Pages Router**: the original router for Next.js, which is still supported. Uses the /pages directory.
- **App Router**: A new router that uses React Server Components. Uses the /app directory.

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:

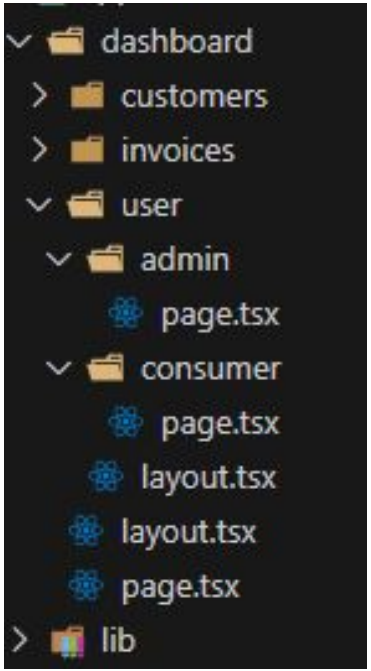
- `page`: exposes a route
- `layout`: shared UI to this route and it's children
- `route`: API endpoint
- `loading`: loading UI
- `not-found`: 404 Not Found UI



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.

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

/example

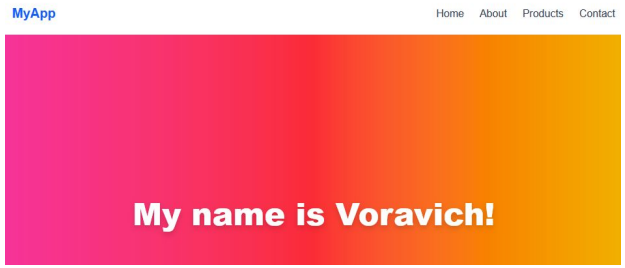
layout.[js,jsx,tsx]

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

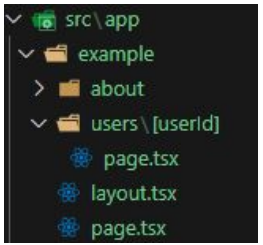


Dynamic Routes

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.

```tsx
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>
  );
}
```



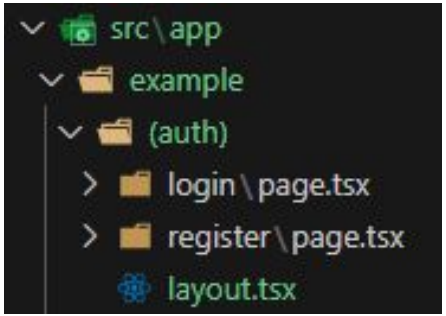
/example/users/12

User #12
User ID: 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

Next.js Backend

How do we build a backend with Next.js?

App Router & route.ts

Next.js's App Router provides us ways to define components, layouts, and API endpoints through the file system in the `/app` folder.

One of these files is the `route` file, which defines an API endpoint complete with support for HTTP method handlers that allow server functions to run.

HTTP Methods in route.ts

In `route.ts`, you can export specific functions for each HTTP method. For instance:

```
// api/helloworld/route.ts
export async function GET() {
  return Response.json({ message: 'Hello World!' });
}

export async function POST(request) {
  const data = await request.json();
  // ...do something with data
  return Response.json({ received: data });
}
```

Route Parameters

We can access route parameters with Next.js's dynamic routes (ones with []) similar to how we did with normal routes:

```
// app/api/users/[id]/route.ts
export async function GET(request: Request, { params }: { params: { id: string } }) {
  const userId = params.id;
  return Response.json({ userId });
}
```

Query Strings

We can also access query strings by accessing the URL object associated with the current URL:

```
export async function GET(request: Request) {
  const { searchParams } = new URL(request.url);

  const role = searchParams.get('role');
  const page = searchParams.get('page');

  return Response.json({ role, page });
}
```

Middleware with proxy.ts

Within the Next.js backend, you can use `proxy.ts` to handle middleware, any code that runs before incoming requests.

```
export function proxy(request: Request) {
  if (!isAuthenticated(request)) {
    return Response.json(
      { success: false, message: 'authentication failed' },
      { status: 401 }
    )
  }
}
```

Fetching

The benefit of Next.js's Server Components is that components **run on the server** rather than the client, and thus can handle async operations.

```
export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts = await data.json()
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

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>`

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>
  );
}
```

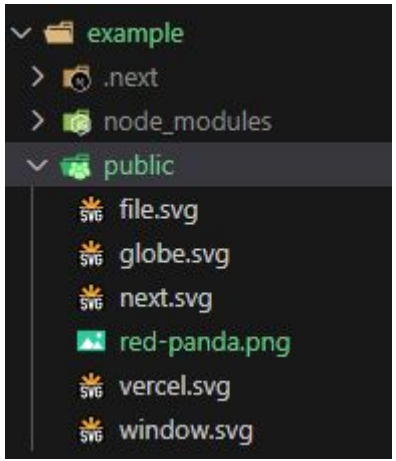
Images

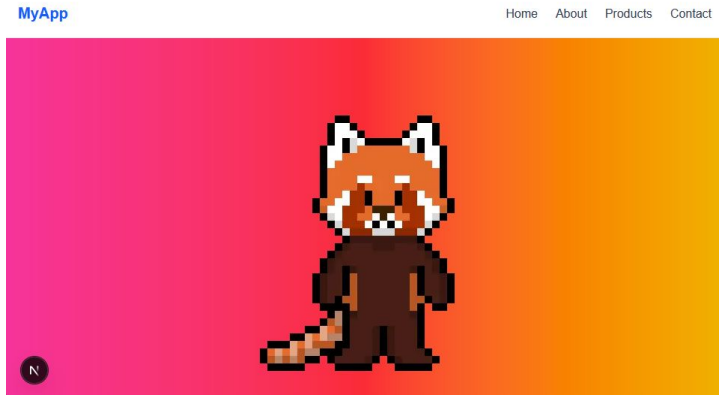
`<Image>, from next/image`, optimizes your images by providing:

- **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!

Fonts

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

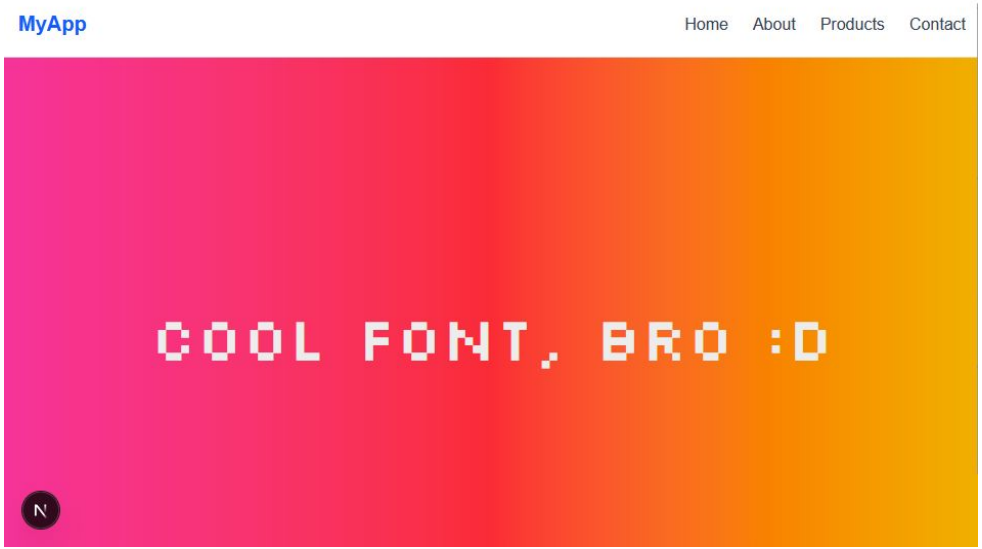
Instead of needing to fetch fonts during runtime (like from Google Fonts), it downloads them during **build time** and hosts them with your other static assets.

```
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.



Streaming

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

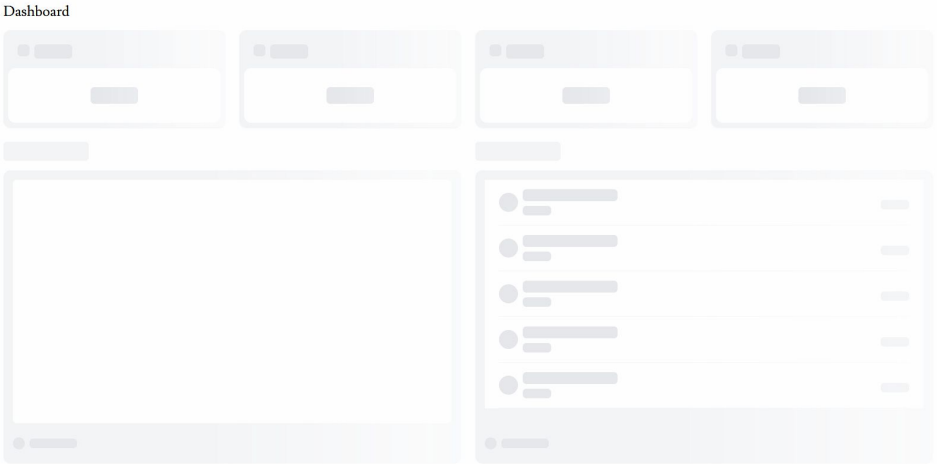
Streaming allows you to send parts of a webpage as soon as they are ready rather than waiting for the whole thing.

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



SSG and ISR

Static Site Generation (SSG): Pages are rendered at build time, served as static HTML files. This method allows for fast, cacheable pages.

- If a page has static content determined at build time, it'll use SSG!

Incremental Static Rendering (ISR): Static pages update after deployment

- Next caches versions of the page, then update them when specified

ISR Example

```tsx
// app/products/page.tsx
async function getProducts() {
  const res = await fetch('https://api.example.com/products', {
    next: { revalidate: 30 },
  })

  return res.json()
}

export default async function ProductsPage() {
  const products = await getProducts()

  return (
    <div>
      <h1>Products</h1>
      {products.map(p => (
        <p key={p.id}>{p.name}</p>
      ))}
    </div>
  )
}
```

Cache Components

‘use cache’ allow certain return values from async functions to be stored in a cache, to be served faster without needing to re-fetch.

```
// next.config.js
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  cacheComponents: true,
}

export default nextConfig
```

Cache Components

```
import { cacheLife } from 'next/cache'

export async function getUsers() {
  'use cache'
  cacheLife('hours')
  return db.query('SELECT * FROM users')
}
```

Caching function output

```
'use cache'
import { cacheLife } from 'next/cache'

export default async function BlogPage() {
  cacheLife('days') // Blog content updated daily

  const posts = await getBlogPosts()
  return <div>{/* render posts */}</div>
}
```

Caching components

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 11
of CIS 1962 Fall 2025.',
};
```

/example page.tsx

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

Class Activity

https://edstem.org/us/courses/91614/lessons/163507/slides/961482