Purwadhika
Digital Technology School

Full Stack Web Development

Server side rendering & static site generation - Part 1

Job Connector Program

Outline

- Introduction to Next.js
- Rendering
- Routing
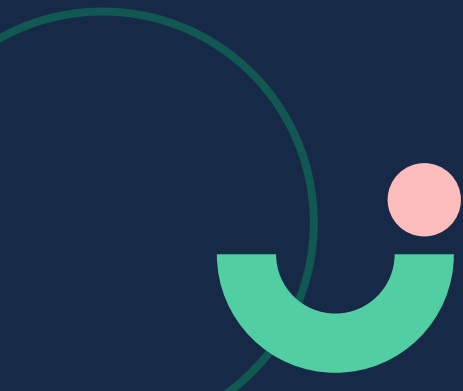
What is Next.js ?

Next.js is a popular open-source JavaScript framework that simplifies the development of React-based web applications. Developed and maintained by Vercel, Next.js extends the capabilities of React by adding server-side rendering (SSR), static site generation (SSG), and other powerful features.

Documentation ➜ https://nextjs.org/docs
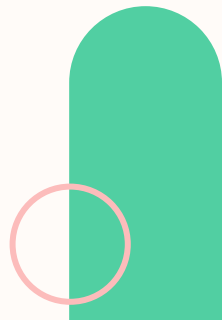
NEXT.JS

Why use Next.js ?

Using Next.js offers several advantages in the development of React-based web applications:

- **Server-Side Rendering (SSR):** Next.js provides built-in support for SSR, which enhances performance by rendering pages on the server and sending pre-rendered HTML to the client. This can lead to faster initial page loads and improved SEO.
- **Static Site Generation (SSG):** For content-heavy websites where the data doesn't change frequently, Next.js allows you to generate static sites at build time. This further improves performance by serving pre-rendered pages without the need for server-side processing.
- **Automatic Code Splitting:** Next.js automatically splits the JavaScript code into smaller, manageable chunks. This helps in optimizing the loading time of your application by only sending the necessary code to the client, reducing the initial page load time.
- **File-Based Routing:** Next.js simplifies routing by adopting a file-based approach. You can create new pages by adding React components to the "pages" directory, making it easy to organize and understand the structure of your application.

- **CSS-in-JS Support:** Next.js has built-in support for popular CSS-in-JS libraries like Styled Components and Emotion, making it convenient to manage styles alongside components.
- **Developer-Friendly:** Next.js comes with features like hot module replacement (HMR) for a more efficient development experience. It also provides a straightforward setup process, allowing developers to focus on building features rather than configuring complex setups.
- **Vercel Integration:** While Next.js is not limited to any specific hosting solution, it integrates seamlessly with Vercel, a platform for deploying serverless functions and static websites. This integration simplifies the deployment process and provides additional features like automatic SSL and global CDN.
- **Adoption in the Industry:** Next.js is widely adopted by both small and large companies for building production-ready applications. Its popularity and the backing of Vercel contribute to its credibility and support in the industry.

Installation

To create a project, run :

- *npx create-next-app@latest*

On installation, you'll see the following prompts:

- What is your project named? **my-app**
- Would you like to use TypeScript? No / **Yes**
- Would you like to use ESLint? No / **Yes**
- Would you like to use Tailwind CSS? **No** / Yes
- Would you like to use `src/` directory? No / **Yes**
- Would you like to use App Router? (recommended) No / **Yes**
- Would you like to customize the default import alias (@/*)? **No** / Yes

Reference ➜ https://nextjs.org/docs/getting-started/installation
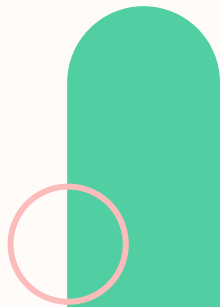
Project Structure

```
my-app/
    ├── .next/                  # Next.js build output (automatically generated)
    ├── node_modules/           # Node.js modules (automatically generated)
    ├── public/                 # Static assets (e.g., images, fonts)
    ├── src/                    # Application source code
    │   ├── app/                # App router
    │   │   ├── page.js         # Home page component
    │   │   ├── about           # /about path
    │   │   │   ├── page.js     # About page component
    │   │   ├── contact.js      # Contact page component
    │   ├── pages/              # Pages router
    │   ├── components/         # Reusable React components
    │   │   ├── Header.js       # Header component
    │   │   ├── Footer.js       # Footer component
    ├── middleware.ts           # Next.js request middleware
    ├── next.config.js          # Next.js configuration file
    ├── package.json            # Node.js package configuration file
    ├── .env                    # Environment variables
    ├── .env.local              # Local environment variables
    ├── .env.production         # Production environment variables
    ├── .env.development        # Development environment variables
    ├── .eslintrc.json          # Configuration file for ESLint
    ├── .gitignore              # Git ignore file
    ├── README.md               # Project documentation
```

For typescript there will be other files which is next-env.d.ts for typescript declaration for Next.js and tsconfig.json for typescript configuration file
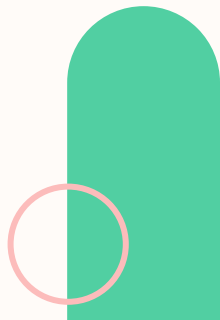
Run the App

How to run :

- Run **_npm run dev_** to start the development server.
- Visit _http://localhost:3000_ to view your application.
- Edit app/layout.js (or pages/index.js) file and save it to see the updated result in your browser.

Other scripts :

- **dev**: runs next dev to start Next.js in development mode.
- **build**: runs next build to build the application for production usage.
- **start**: runs next start to start a Next.js production server.
- **lint**: runs next lint to set up Next.js' built-in ESLint configuration.
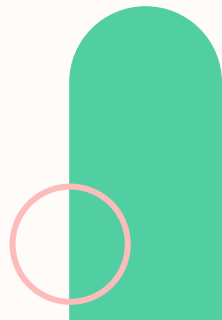
App Router vs Pages Router

Next.js has two different routers: the **App Router** and the **Pages Router**. The **App Router** is a newer router that allows you to use React's latest features, such as **Server Components** and **Streaming**. The **Pages Router** is the original Next.js router, which allowed you to build server-rendered React applications and continues to be supported for older Next.js applications.
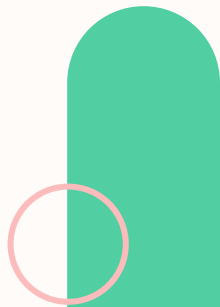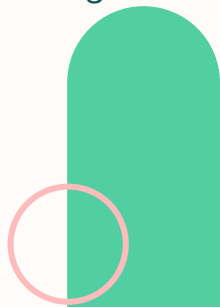
Rendering

Rendering converts the code you write into user interfaces. React and Next.js allow you to create hybrid web applications where parts of your code can be rendered on the server or the client. This section will help you understand the differences between these rendering environments, strategies, and runtimes.

There are two environments where web applications can be rendered: the **client** and the **server**.

- The client refers to the browser on a user's device that sends a request to a server for your application code. It then turns the response from the server into a user interface.
- The server refers to the computer in a data center that stores your application code, receives requests from a client, and sends back an appropriate response.
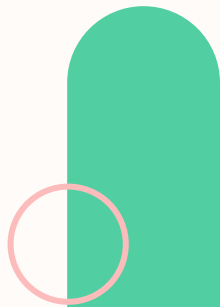
By default, Next.js uses **Server Components**. This allows you to automatically implement server rendering with no additional configuration, and you can opt into using **Client Components** when needed.

Server Component

React **Server Components** allow you to write UI that can be rendered and optionally cached on the server. In Next.js, the rendering work is further split by route segments to enable streaming and partial rendering, and there are three different server rendering strategies:
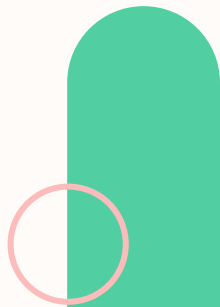
- Static Rendering
- Dynamic Rendering
- Streaming

Server Component - Static Rendering

By default Next.js use **Static Rendering**. With Static Rendering, routes are rendered at build time, or in the background after data revalidation. The result is cached and can be pushed to a Content Delivery Network (CDN). This optimization allows you to share the result of the rendering work between users and server requests.

Static rendering is useful when a route has data that is not personalized to the user and can be known at build time, such as a static blog post or a product page.

Server Component - Dynamic Rendering

With **Dynamic Rendering**, routes are rendered for each user at request time. Dynamic rendering is useful when a route has data that is personalized to the user or has information that can only be known at request time, such as cookies or the URL's search params.

As a developer, you **do not need to choose between static and dynamic rendering** as Next.js will automatically choose the best rendering strategy for each route based on the features and APIs used. Instead, **you choose when to cache or revalidate specific data**, and you may choose to stream parts of your UI.
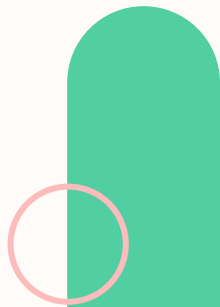
Server Component - Streaming

Streaming enables you to progressively render UI from the server. Work is split into chunks and streamed to the client as it becomes ready. This allows the user to see parts of the page immediately, before the entire content has finished rendering.

Streaming is built into the Next.js App Router by default. This helps improve both the initial page loading performance, as well as UI that depends on slower data fetches that would block rendering the whole route. For example, reviews on a product page. You can start streaming route segments using loading.js and UI components with React Suspense.

We will discuss this in the next slide.
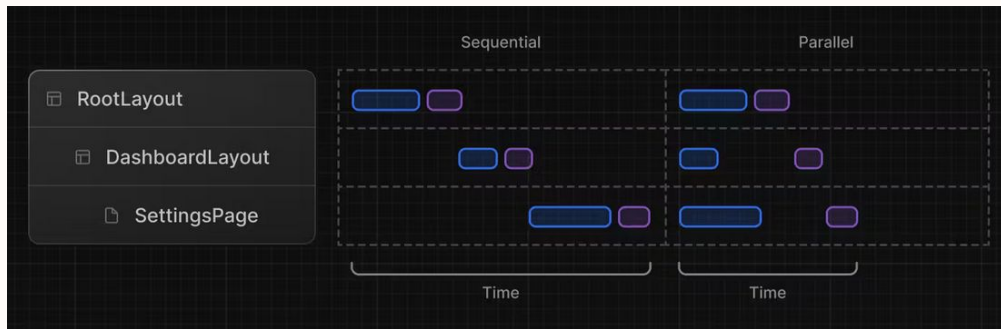
Client Component

Client Components allows you to write interactive UI that can be rendered on the client at request time. In Next.js, client rendering is opt-in, meaning you have to explicitly decide what components React should render on the client.

There are a couple of benefits to doing the rendering work on the client, including:

- Interactivity: Client Components can use state, effects, and event listeners, meaning they can provide immediate feedback to the user and update the UI.
- Browser APIs: Client Components have access to browser APIs, like geolocation or localStorage, allowing you to build UI for specific use cases.

To use Client Components, you can add the React **"use client"** directive at the top of a file, above your imports. **"use client"** is used to declare a boundary between a Server and Client Component modules. This means that by defining a "use client" in a file, all other modules imported into it, including child components, are considered part of the client bundle.

Client Component

```
"use client";

import { useState } from "react";

export default function Counter() {
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  )
}
```

Routing

The skeleton of every application is **routing**. Next.js uses a file-system based router where folders are used to define routes.

- **Tree:** A convention for visualizing a hierarchical structure. For example, a component tree with parent and children components, a folder structure, etc.
- **Subtree:** Part of a tree, starting at a new root (first) and ending at the leaves (last).
- **Root:** The first node in a tree or subtree, such as a root layout.
- **Leaf:** Nodes in a subtree that have no children, such as the last segment in a URL path.
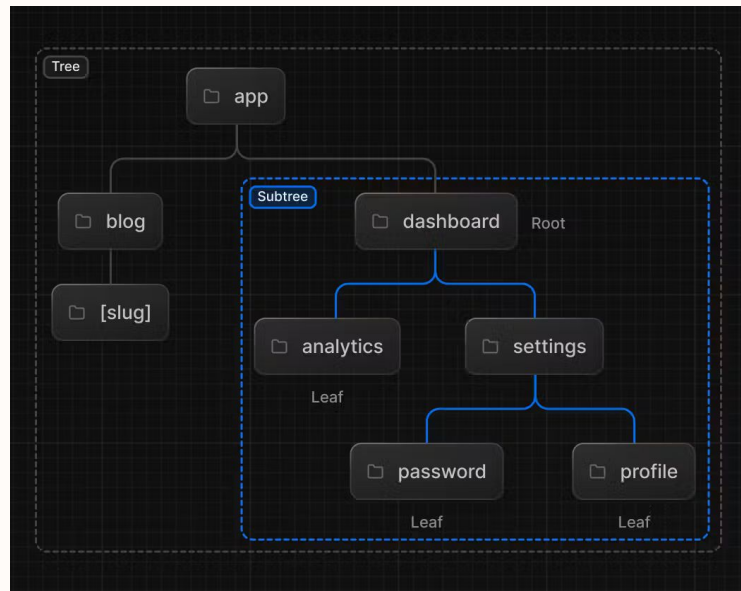
Roles of Folders and Files

Next.js uses a file-system based router where:

- **Folders** are used to define routes. A route is a single path of nested folders, following the file-system hierarchy from the root folder down to a final leaf folder that includes a page.js file
- **Files** are used to create UI that is shown for a route segment
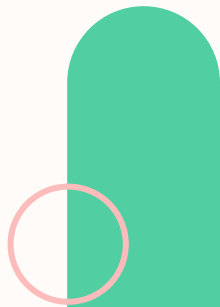
Creating Routes - Route Segment

Each folder in a route represents a route segment.
Each route segment is mapped to a corresponding
segment in a **URL path**.

- **URL Segment:** Part of the URL path delimited
  by slashes.
- **URL Path:** Part of the URL that comes after the
  domain (composed of segments).
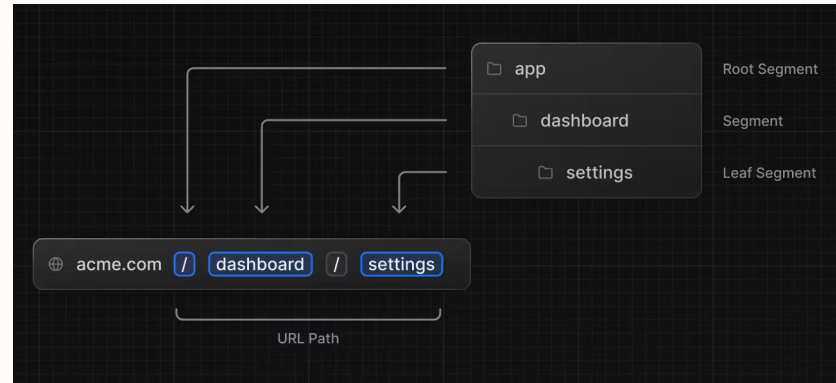
Creating Routes - Show the Page

A special ***page.tsx*** file is used to make route segments publicly accessible.
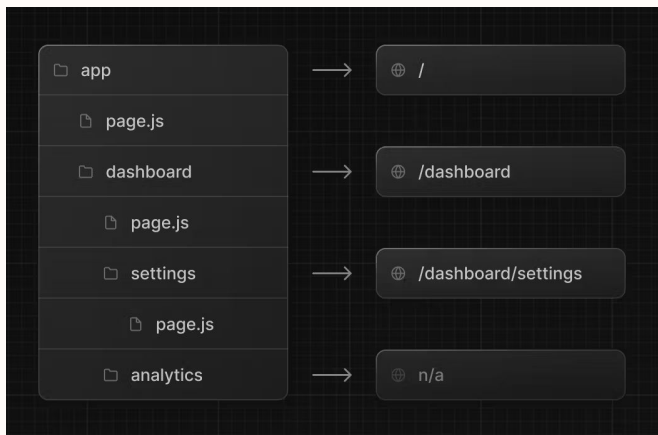
In this example, the /dashboard/analytics URL path is not publicly accessible because it does not have a corresponding page.js file. This folder could be used to store components, stylesheets, images, or other colocated files.

File Conventions

layout	Shared UI for a segment and its children
page	Unique UI of a route and make routes publicly accessible
loading	Loading UI for a segment and its children
not-found	Not found UI for a segment and its children
error	Error UI for a segment and its children
global-error	Global Error UI
route	Server-side API endpoint
template	Specialized re-rendered Layout UI
default	Fallback UI for Parallel Routes
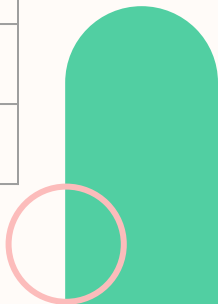
Component Hierarchy

The React components defined in special files of a route segment are rendered in a specific hierarchy:
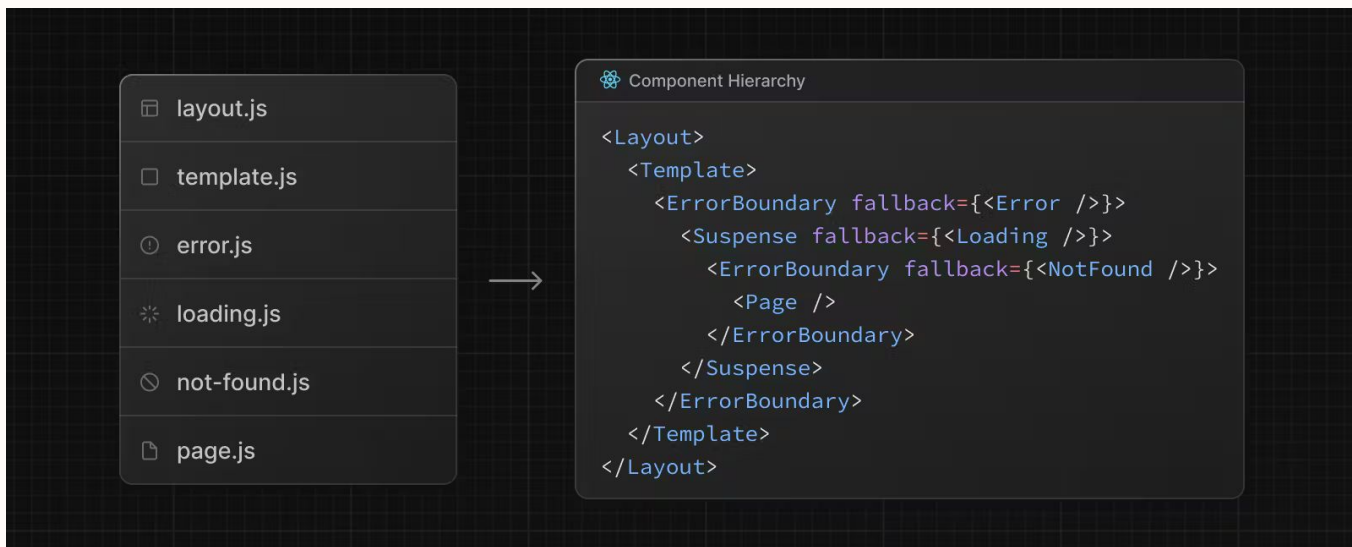
Pages

A **page** is UI that is unique to a route. You can define pages by exporting a component from a page.js file. Use nested folders to define a route and a page.js file to make the route publicly accessible.
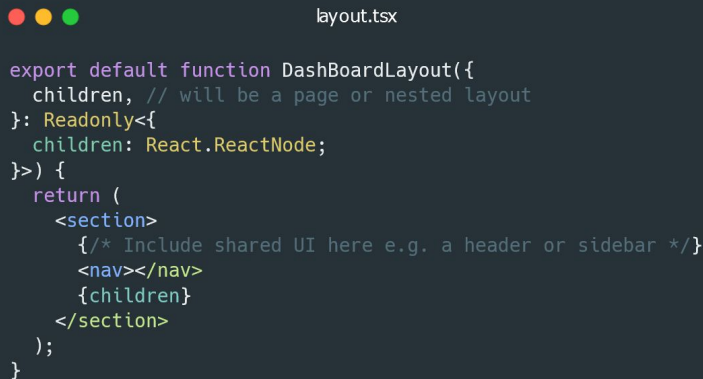
```
// `app/page.js` is the UI for the `/` URL
export default function Page() {
  return <h1>Hello, Home page!</h1>
}
```
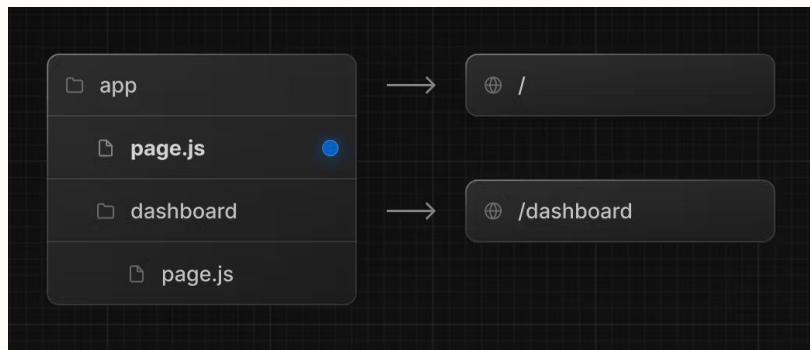
Layout

A **layout** is UI that is shared between multiple pages. On navigation, layouts preserve state, remain interactive, and do not re-render. Layouts can also be nested.

You can define a layout by default exporting a React component from a layout.tsx file. The component should accept a children prop that will be populated with a child layout (if it exists) or a child page during rendering.
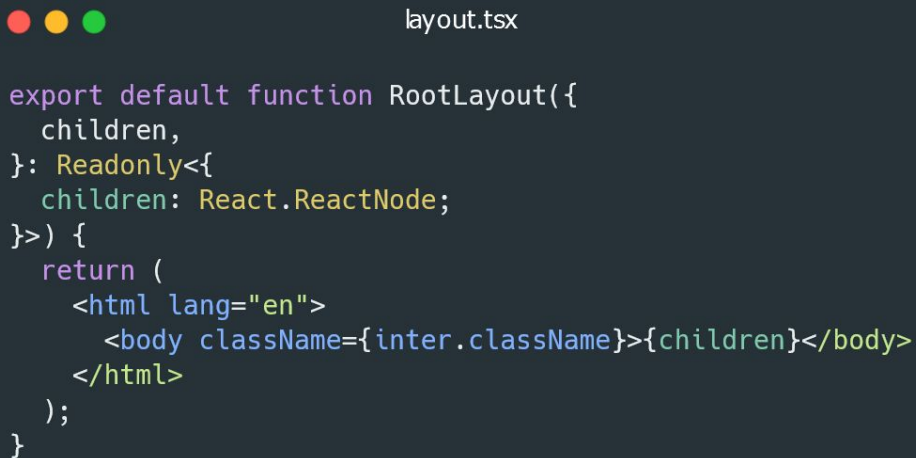
```tsx
// layout.tsx
export default function DashBoardLayout({
  children, // will be a page or nested layout
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <section>
      {/* Include shared UI here e.g. a header or sidebar */}
      <nav></nav>
      {children}
    </section>
  );
}
```

Root Layout (Required)

The root layout is defined at the top level of the app directory and applies to all routes. This layout enables you to modify the initial HTML returned from the server.

```tsx
                              layout.tsx

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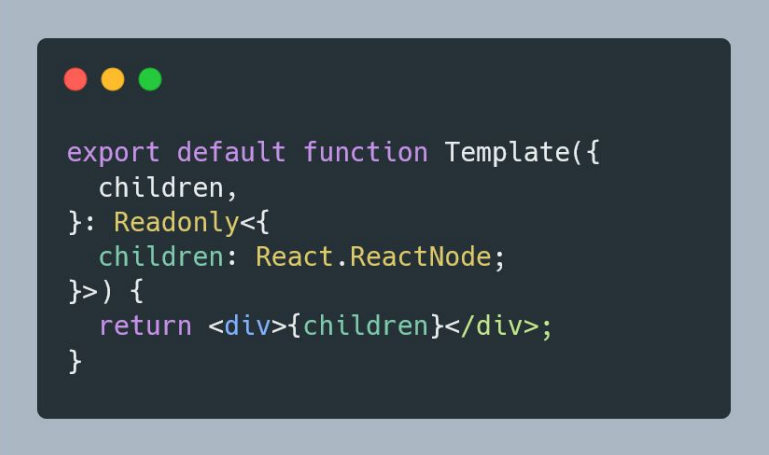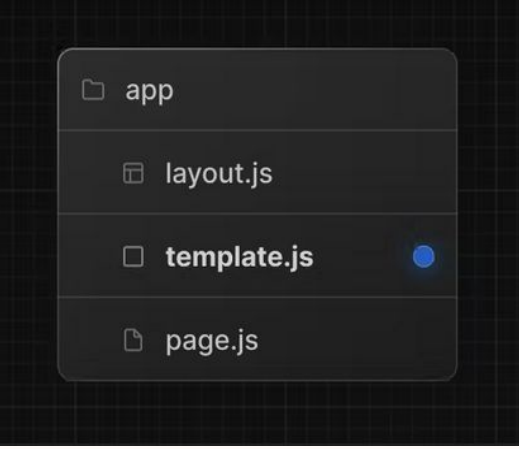
Templates

Templates are similar to layouts in that they wrap each child layout or page. Unlike layouts that persist across routes and maintain state, templates create a new instance for each of their children on navigation. This means that when a user navigates between routes that share a template, a new instance of the component is mounted, DOM elements are recreated, state is not preserved, and effects are re-synchronized.

```
export default function Template({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return <div>{children}</div>;
}
```

app
layout.js
template.js
page.js

Dynamic Routes

When you don't know the exact segment names ahead of time and want to create routes from dynamic data, you can use **Dynamic Segments** that are filled in at request time or prerendered at build time.

A Dynamic Segment can be created by wrapping a folder's name in square brackets: **[folderName]**. For example, **[id]** or **[slug]**. Dynamic Segments are passed as the params prop to layout, page, route, and generateMetadata functions.
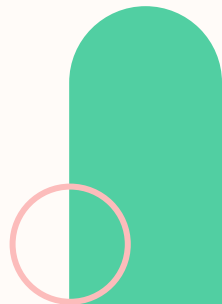
Dynamic Routes

For example, a blog could include the following route **app/blog/[slug]/page.js** where **[slug]** is the Dynamic Segment for blog posts.

app/blog/[slug]/page.tsx

```tsx
export default function Page({ params }: { params: { slug: string } }) {
  return <div>My Post: {params.slug}</div>;
}
```

Linking and Navigating

The **App Router** uses a hybrid approach for routing and navigation. On the server, your application code is automatically code-split by route segments. And on the client, Next.js prefetches and caches the route segments. This means, when a user navigates to a new route, the browser doesn't reload the page, and only the route segments that change re-render - improving the navigation experience and performance.

There are two ways to navigate between routes in Next.js:

- Using the ***<Link>*** Component
- Using the ***useRouter*** Hook
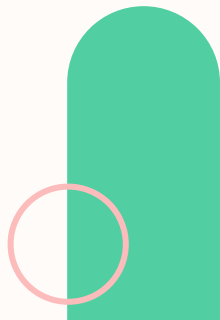
\<Link\> Component

\<Link\> is a built-in component that extends the HTML \<a\> tag to provide prefetching and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.

You can use it by importing it from **next/link**, and passing a href prop to the component:

```
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

useRouter() Hook

The **useRouter** hook allows you to programmatically change routes. This hook can only be used inside Client Components and is imported from *next/navigation*.

```jsx
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```
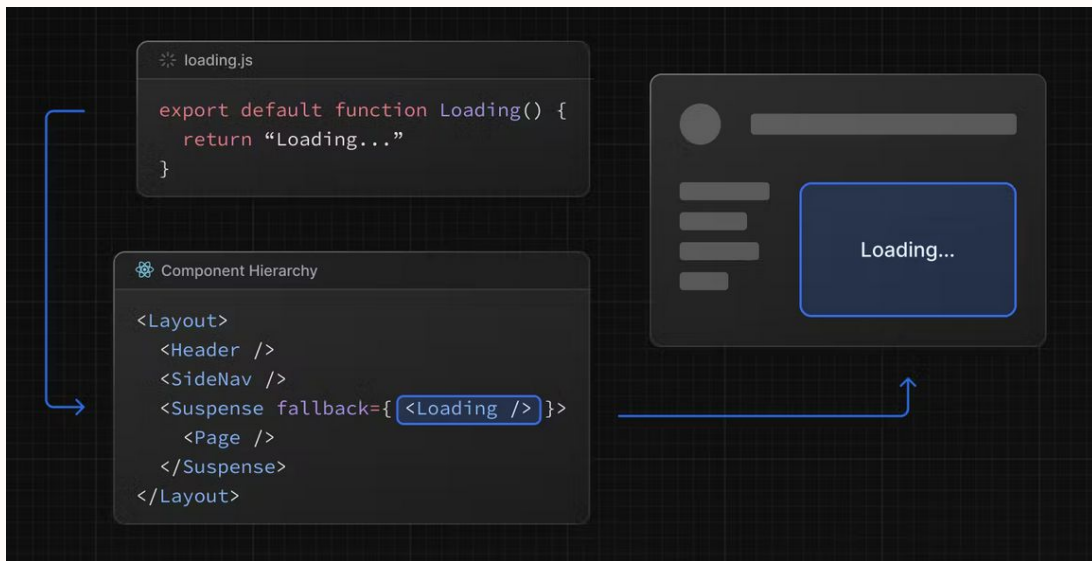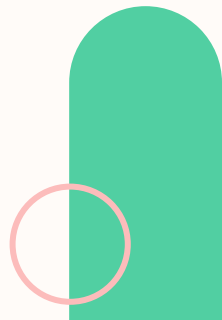
Loading

The special file loading.tsx helps you create meaningful **Loading UI** with [React Suspense](#). With this convention, you can show an instant loading state from the server while the content of a route segment loads. The new content is automatically swapped in once rendering is complete.
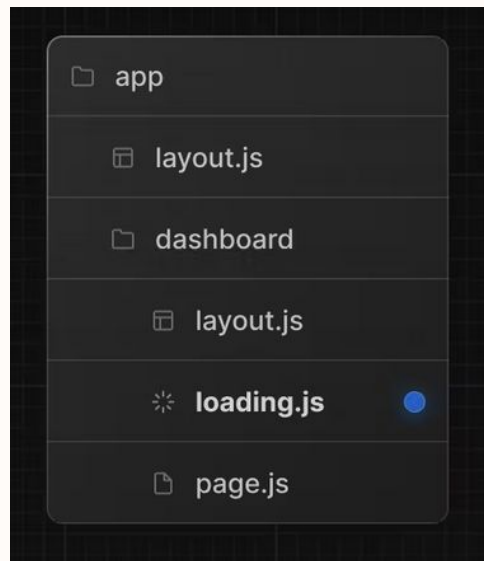
Error Handling

The **error.js** file convention allows you to gracefully handle unexpected runtime errors in nested routes.

- Automatically wrap a route segment and its nested children in a React Error Boundary.
- Create error UI tailored to specific segments using the file-system hierarchy to adjust granularity.
- Isolate errors to affected segments while keeping the rest of the application functional.
- Add functionality to attempt to recover from an error without a full page reload.

Error Handling

Route Segment Config

The **Route Segment Options** allows you configure the behavior of a **Page**, **Layout**, or **Route Handler** by directly exporting the following variables:

| Option	Type	Default
dynamic	'auto' | 'force-dynamic' | 'error' | 'force-static'	'auto'
dynamicParams	boolean	true
revalidate	false | 'force-cache' | 0 | number	false
fetchCache	'auto' | 'default-cache' | 'only-cache' | 'force-cache' | 'force-no-store' | 'default-no-store' | 'only-no-store'	'auto'
runtime	'nodejs' | 'edge'	'nodejs'
preferredRegion	'auto' | 'global' | 'home' | string | string[]	'auto'
maxDuration	number	Set by deployment platform

Route Segment Config

```
export const dynamic: string = 'auto';
export const dynamicParams: boolean = true;
export const revalidate: boolean = false;
export const fetchCache: string = 'auto';
export const runtime: string = "nodejs";
export const preferredRegion: string = "auto";
export const maxDuration: number = 5;

export default function MyComponent() {}
```

Route Handler

Route Handlers allow you to create custom request handlers for a given route using the **Web Request and Response APIs**.
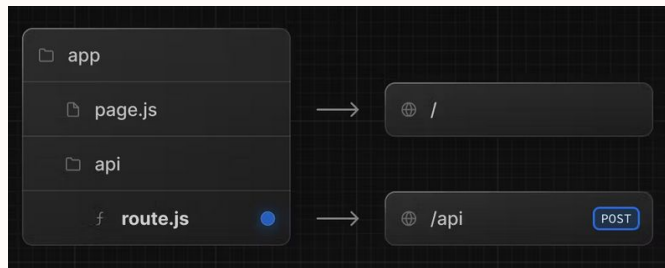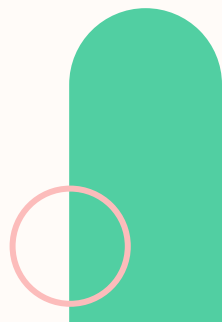


```
export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY,
    },
  })
  const data = await res.json()

  return Response.json({ data })
}

export async function HEAD(request) {}

export async function POST(request) {}

export async function PUT(request) {}

export async function DELETE(request) {}

export async function PATCH(request) {}

export async function OPTIONS(request) {}
```

Exercise

Convert your portfolio website to Next.js !

Thank You!