Software Project 1

Tutorial 7

Course Staff

❑ Dr. Nada Sharaf	nada.hamed@giu-uni.de A1.211
❑ Eng. Donia Ali	donia.ali@giu-uni.de A1.110
❑ AL. Amany Hussein	amany.hussein@giu-uni.de A1.122
❑ Eng. Hania Ashraf	hania.ashraf@giu-uni.de A1.222
❑ Eng. Omar Ashraf	Berlin omar.ashraf@giu-uni.de 602

Next Js

- ➢ Introduction
- ➢ Setting up environment
- ➢ Pages and routing
    - routing files
    - nested routes
    - dynamic routes
    - Route Groups and private folders
    - Parallel and Intercepted Routes

Introduction

a React-based framework developed by Vercel that simplifies building high-performance, server-rendered full-stack web applications

It combines static and server-side rendering, optimizing for speed and SEO. Next.js offers file-based routing, automatic code splitting, built-in API routes, and image optimization, enhancing both developer experience and app efficiency.

Ideal for projects needing dynamic data and fast load times, it's popular for e-commerce, blogs, and SaaS apps.

With built-in TypeScript support and easy backend integration (e.g., with Nest.js), Next.js is versatile for both frontend and full-stack applications.

Setting up environment

System requirements

Node.js 18.18 or later.

macOS, Windows (including WSL), and Linux are supported.

Terminal

```
npx create-next-app@latest
```

then configure your app

To run your app

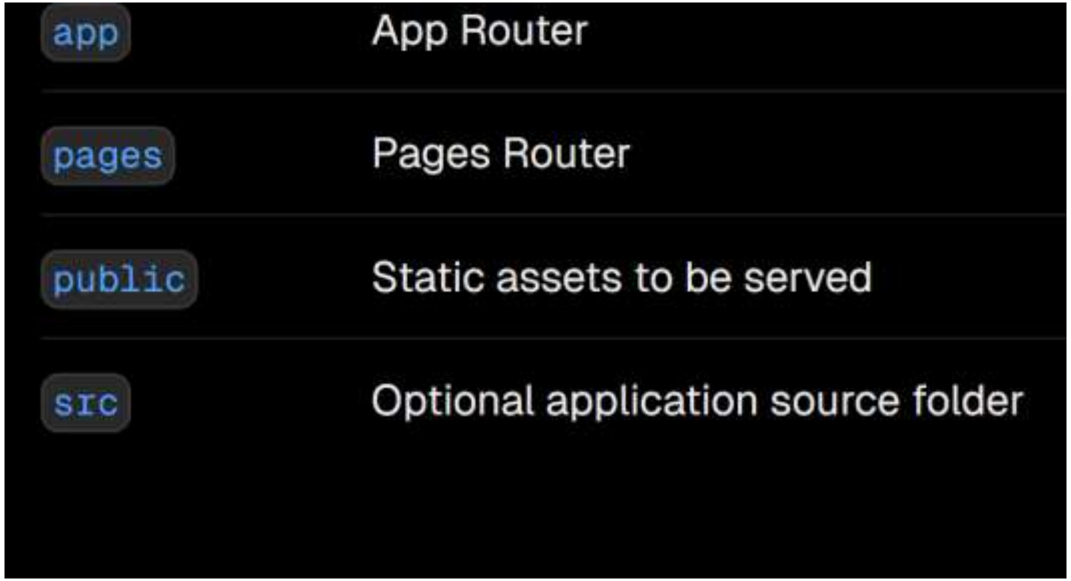
```
npm run dev
```

Pages and routing

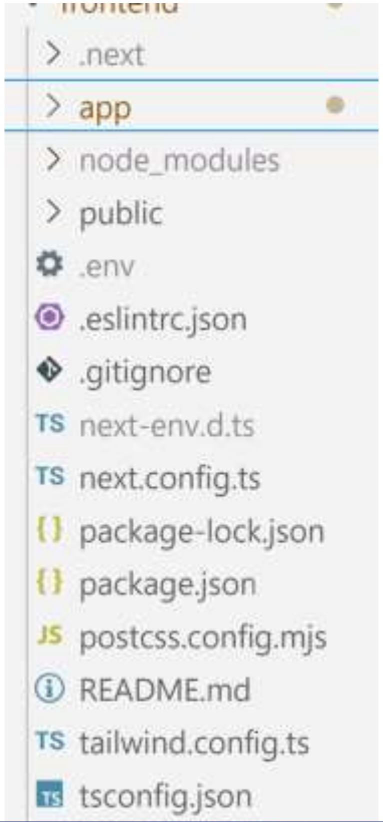
- Project structure
- Files Convention
- routing files
- nested routes
- dynamic routes
- Route Groups
- private folders
- Parallel
- Intercepted Routes

Pages and routing

❖ Project structure



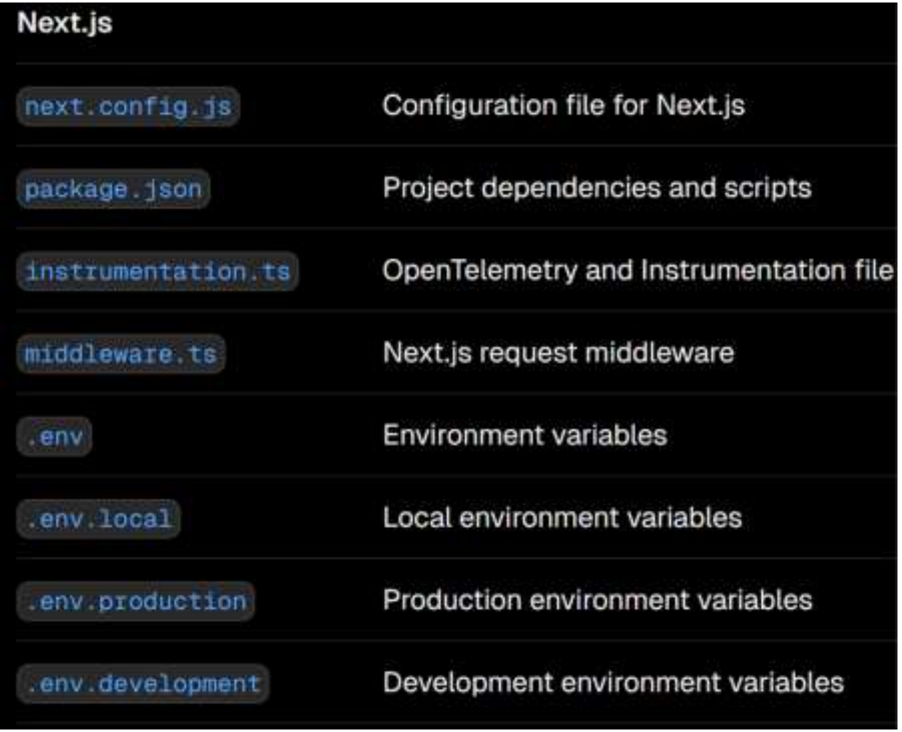
`app`	App Router
`pages`	Pages Router
`public`	Static assets to be served
`src`	Optional application source folder



```
frontend
> .next
> app
> node_modules
> public
.env
.eslintrc.json
.gitignore
next-env.d.ts
next.config.ts
package-lock.json
package.json
postcss.config.mjs
README.md
tailwind.config.ts
tsconfig.json
```

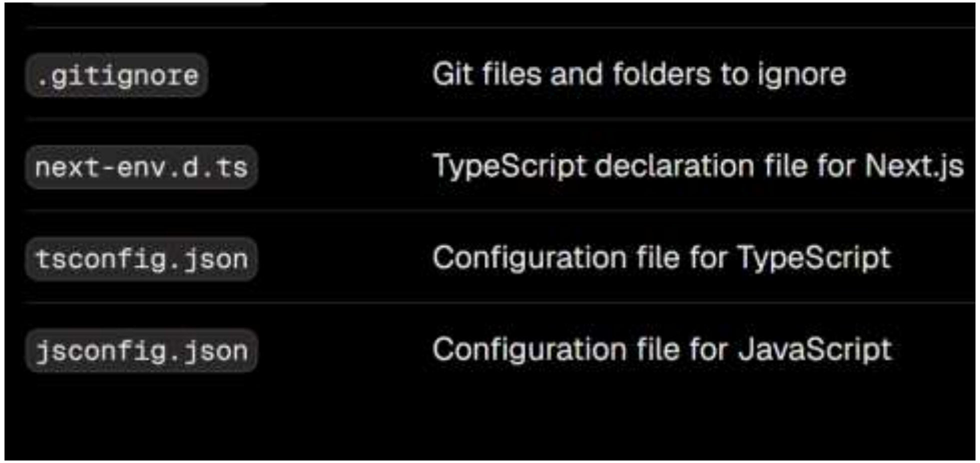
Pages and routing

❖ Project structure



Next.js

`next.config.js`	Configuration file for Next.js
`package.json`	Project dependencies and scripts
`instrumentation.ts`	OpenTelemetry and Instrumentation file
`middleware.ts`	Next.js request middleware
`.env`	Environment variables
`.env.local`	Local environment variables
`.env.production`	Production environment variables
`.env.development`	Development environment variables



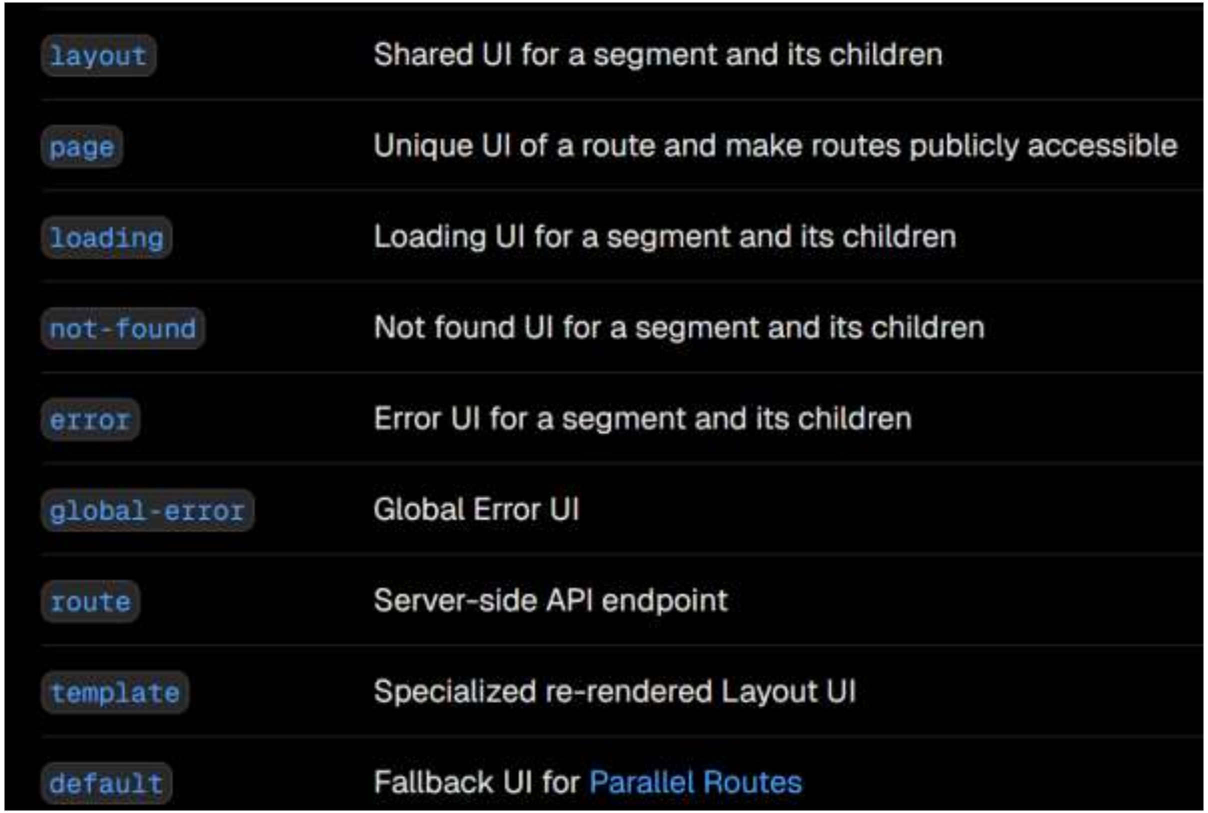
`.gitignore`	Git files and folders to ignore
`next-env.d.ts`	TypeScript declaration file for Next.js
`tsconfig.json`	Configuration file for TypeScript
`jsconfig.json`	Configuration file for JavaScript

Pages and routing

- ❖ Routing files

File Conventions

Next.js provides a set of special files to create UI with specific behavior in nested routes



`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

Pages and routing

Page (page.js/.jsx/.tsx)

A page file corresponds to a specific route in your app. These files define the content and structure for each page. Every file placed in the pages/ directory automatically becomes a route in the app based on its file name.

Layout (layout.js/.jsx/.tsx)

The layout file is used to define common structure and UI elements (like headers, footers, sidebars) that are shared across multiple pages. It helps in wrapping the main content of the page while maintaining consistent styling and structure.

Loading (loading.js/.jsx/.tsx)

The loading file displays a loading spinner or other UI components to indicate that content is still being fetched. It's helpful when dealing with asynchronous data or slow network requests, providing users with visual feedback during page load.

Not Found (not-found.js/.jsx/.tsx)

The not-found file is a custom 404 page shown when a user visits a non-existent route. It allows you to provide a friendly message or a redirection option when a user navigates to a route that doesn't exist in your app.

Pages and routing

Error (error.js/.jsx/.tsx)

This file defines an error UI that is triggered when something goes wrong during page rendering. For example, if there's an issue with data fetching or page rendering, the error component will catch it and display an appropriate error message.

Global Error (global-error.js/.jsx/.tsx)

A global error component is used to handle errors that occur anywhere in the app, not just on a specific page. It can capture issues such as server-side errors and show a unified error message or fallback UI to the user.

Template (template.js/.jsx/.tsx)

The template file defines layouts that re-render when navigating to different pages. This is useful when you need to update the layout or content dynamically, such as showing user-specific data that changes based on the route.

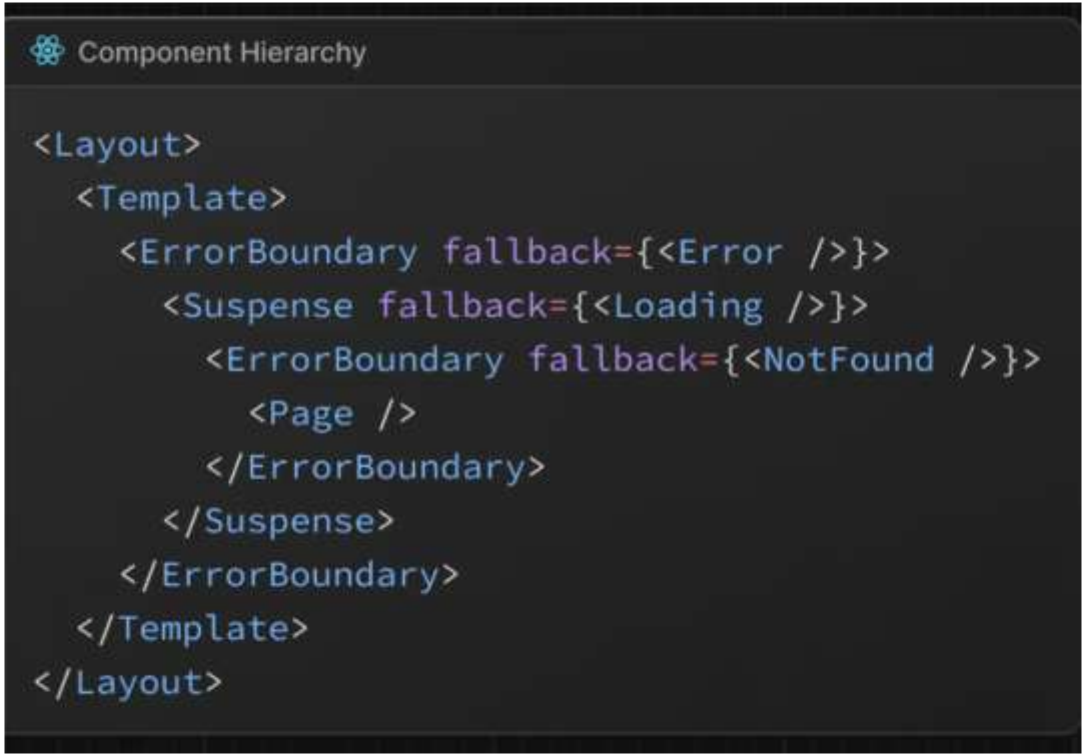
Default (default.js/.jsx/.tsx)

The default file is used in parallel routes to define a fallback UI when no specific parallel route is matched. It's typically shown as the default view when dealing with more complex routing structures, like nested or conditional routes.

Pages and routing

- Component Hierarchy

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

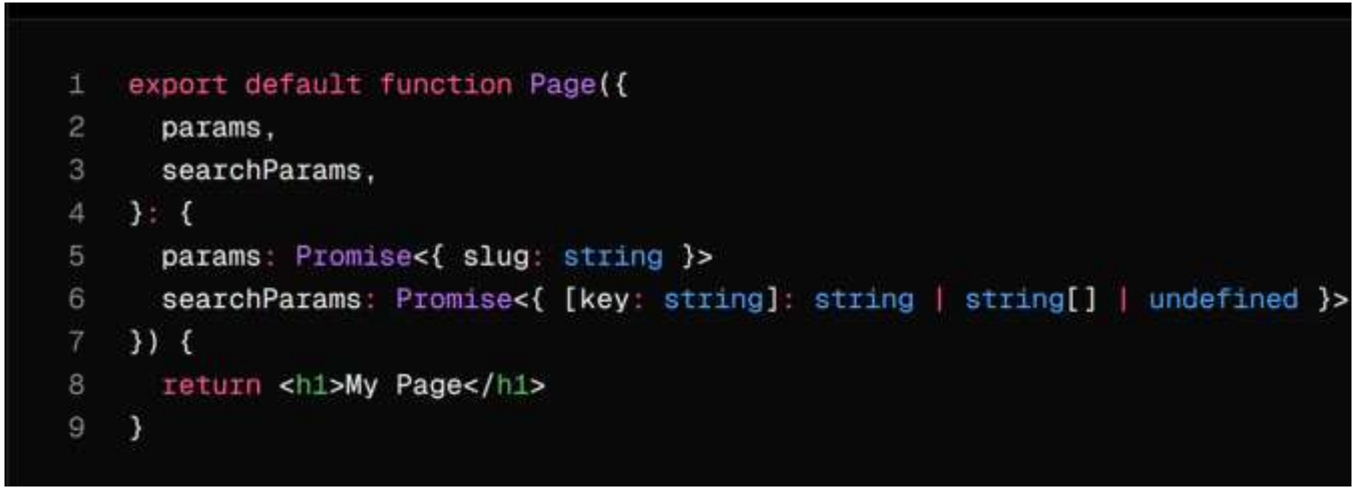


```
<Layout>
  <Template>
    <ErrorBoundary fallback={<Error />}>
      <Suspense fallback={<Loading />}>
        <ErrorBoundary fallback={<NotFound />}>
          <Page />
        </ErrorBoundary>
      </Suspense>
    </ErrorBoundary>
  </Template>
</Layout>
```

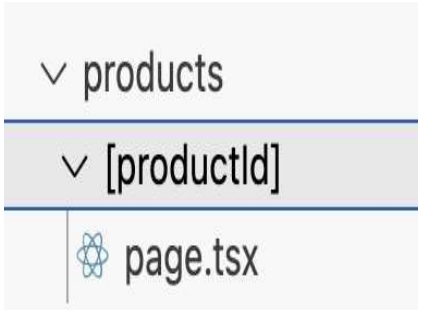
Page.tsx

Every route will have corresponding page.tsx to define UI (content and structure) corresponding to this route

Props: Any custom props you define for the page.



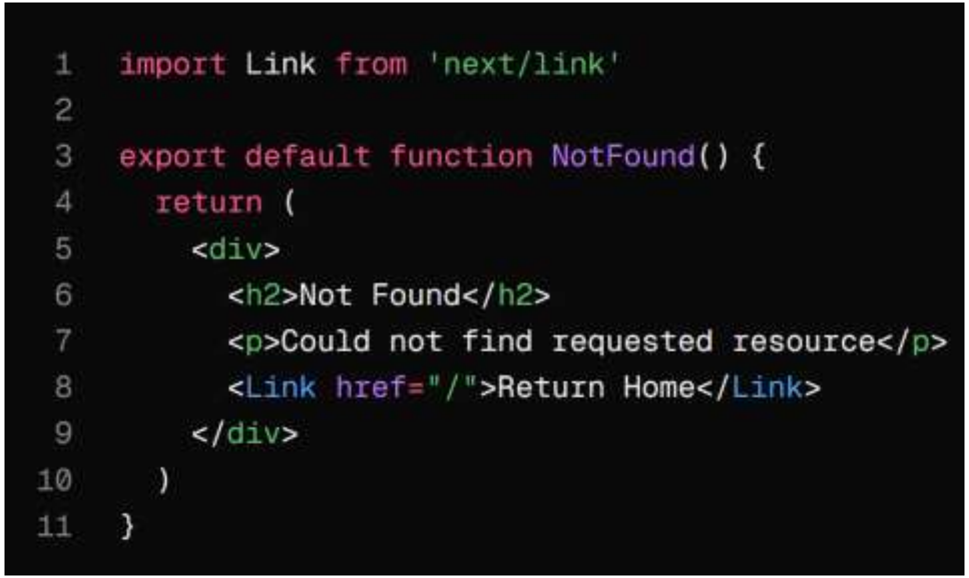
```tsx
export default function Page({
  params,
  searchParams,
}: {
  params: Promise<{ slug: string }>
  searchParams: Promise<{ [key: string]: string | string[] | undefined }>
}) {
  return <h1>My Page</h1>
}
```



not-found.tsx

The not-found file is used to render UI when the notFound function is thrown within a route segment. Along with serving a custom UI, Next.js will return a 200 HTTP status code for streamed responses, and 404 for non-streamed responses.

Props: None



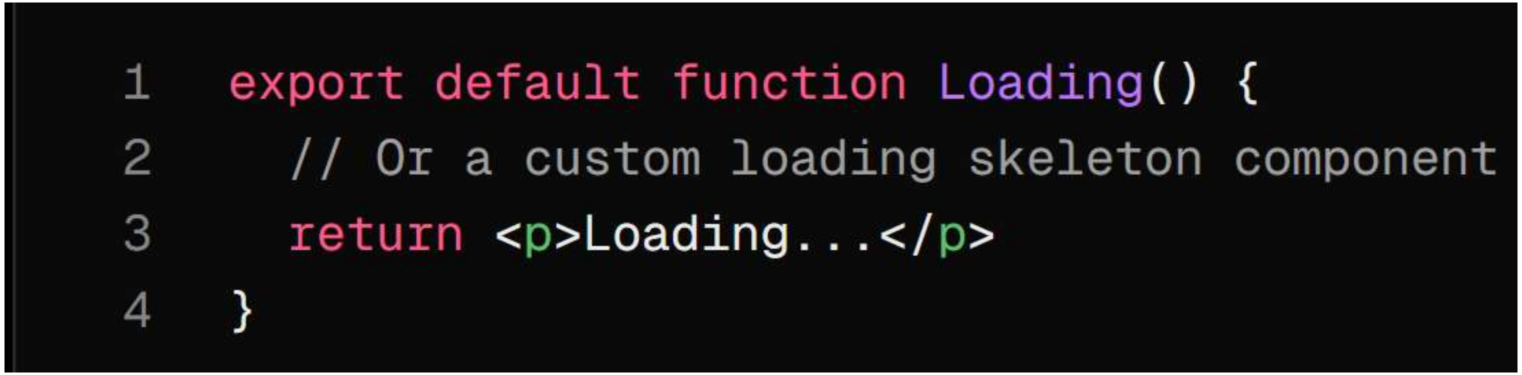
```tsx
import Link from 'next/link'

export default function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>Could not find requested resource</p>
      <Link href="/">Return Home</Link>
    </div>
  )
}
```

loading.tsx

A loading file can create instant loading states

it’s helpful when dealing with asynchronous data or slow network requests, providing users with visual feedback during page load.

Props: None



```tsx
export default function Loading() {
  // Or a custom loading skeleton component
  return <p>Loading...</p>
}
```

error.tsx

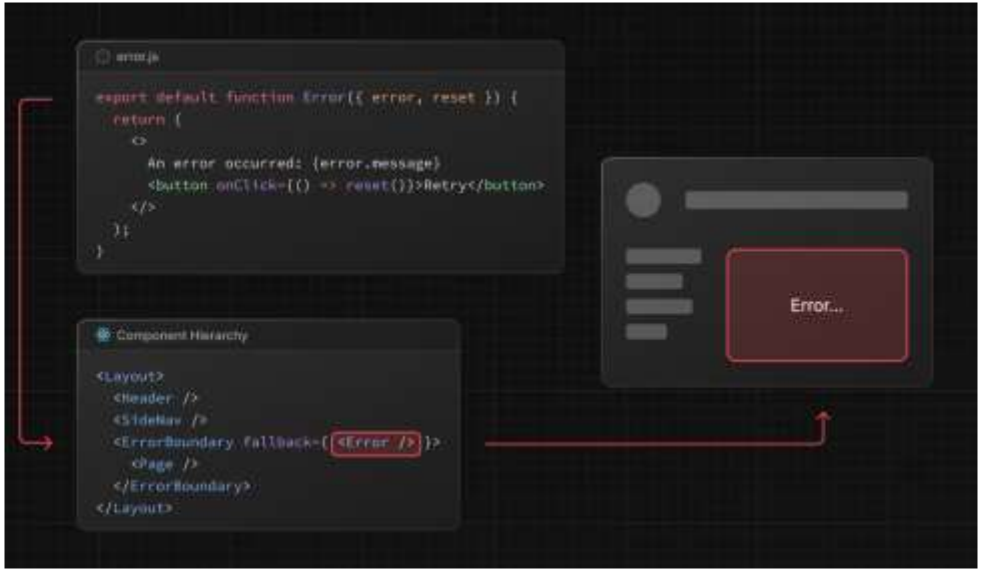
An error file allows you to handle unexpected runtime errors and display fallback UI.

error.tsx wraps a route segment and its nested children in a *React Error Boundary*. When an error throws within the boundary, the error component shows as the fallback UI.

Props:

error: The error object containing information about the error.

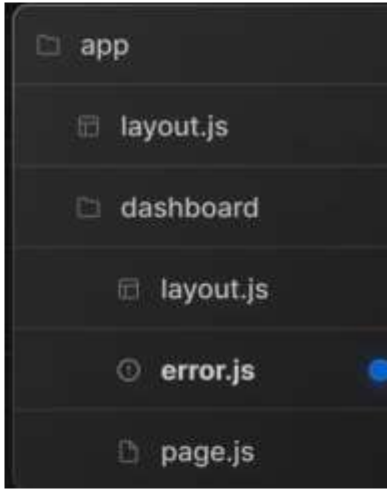
reset: function to prompt the user to attempt to recover from the error. When executed, the function will try to re-render the error boundary's contents. If successful, the fallback error component is replaced with the result of the re-render.


error.js

```
export default function Error({ error, reset }) {
  return (
    <>
      An error occurred: {error.message}
      <button onClick={() => reset()}>Retry</button>
    </>
  );
}
```

Component Hierarchy

```
<Layout>
  <Header />
  <SideNav />
  <ErrorBoundary fallback={<Error />}>
    <Page />
  </ErrorBoundary>
</Layout>
```

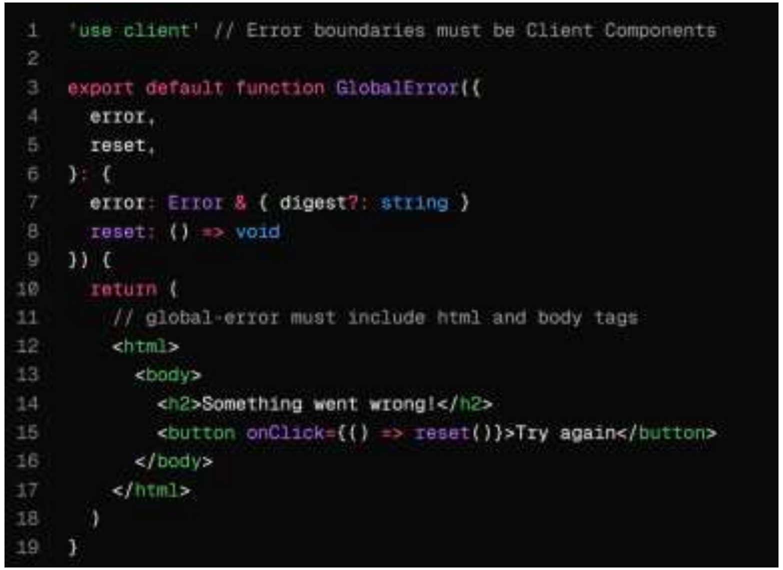


global-error.tsx

While less common, you can handle errors in the root layout or template using app/global-error.tsx,

Props:
error: The error object containing information about the error.

reset: function to prompt the user to attempt to recover from the error. When executed, the function will try to re-render the error boundary's contents. If successful, the fallback error component is replaced with the result of the re-render.



```tsx
'use client' // Error boundaries must be Client Components

export default function GlobalError({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  return (
    // global-error must include html and body tags
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
  )
}
```

layout.tsx

The **layout file** is used to define common structure and UI elements (like headers, footers, sidebars) that are shared across multiple pages. It helps in wrapping the main content of the page while maintaining consistent styling and structure.It **does not remount** shared components resulting in better performance. **The root layout** is defined at the top level of the app directory and applies to all routes. This layout is required and must contain html and body tags, allowing you to modify the initial HTML returned from the server.

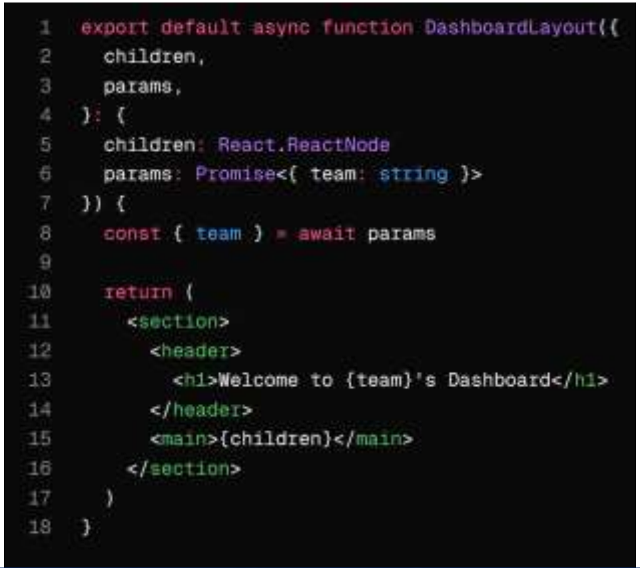
Props:

children (required)

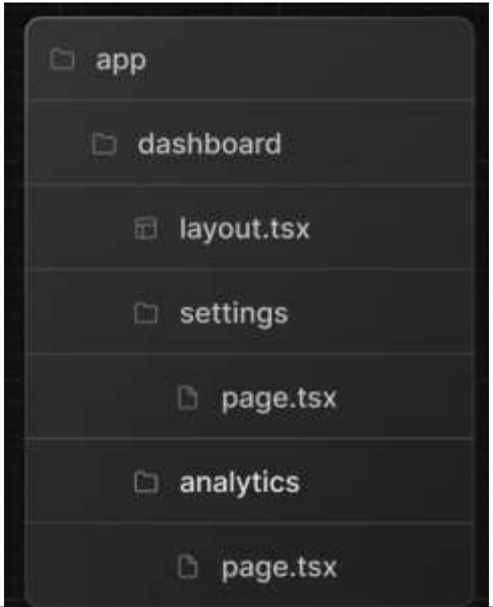
Layout components should accept and use a children prop.

During rendering, children will be populated with the route segments the layout is wrapping.

These will primarily be the component of a child Layout (if it exists) or Page, but could also be other special files like Loading or Error when applicable



```tsx
export default async function DashboardLayout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Promise<{ team: string }>
}) {
  const { team } = await params

  return (
    <section>
      <header>
        <h1>Welcome to {team}'s Dashboard</h1>
      </header>
      <main>{children}</main>
    </section>
  )
}
```



```
app
  dashboard
    layout.tsx
    settings
      page.tsx
    analytics
      page.tsx
```

template.tsx

Templates are similar to layouts in that they wrap a child page.

templates create a new instance for each of their components on navigation. This means that when a user navigates between routes that share a template, **a new instance** of the the component is mounted, DOM elements are recreated, state is not preserved in Client Components, and effects are re-synchronized.
use case:
enter exit animation or side effects using useEffect or reset the state

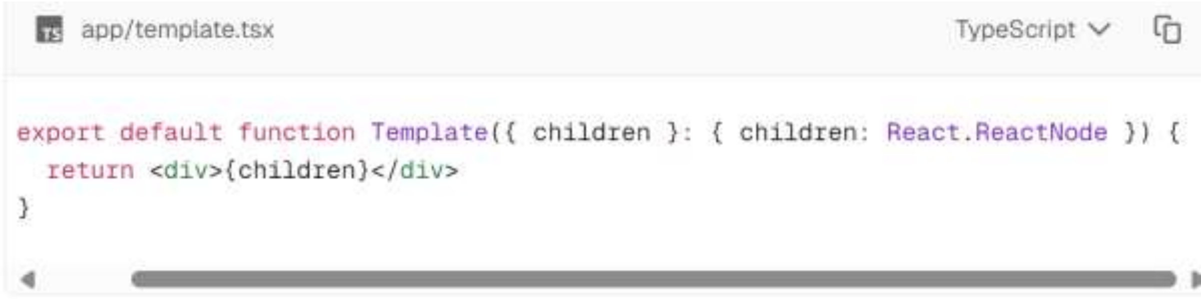
Props:

children (required)

Layout components should accept and use a children prop.

During rendering, children will be populated with the route segments the layout is wrapping.

These will primarily be the component of a child Layout (if it exists) or Page, but could also be other special files like Loading or Error when applicable



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

default.tsx

the default.tsx file is used to render a fallback within Parallel Routes when Next.js cannot recover a slot's active state after a full-page load.

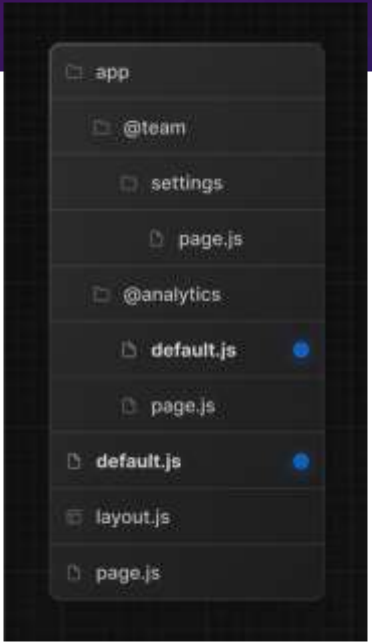
During soft navigation, Next.js keeps track of the active state (subpage) for each slot. However, for hard navigations (full-page load), Next.js cannot recover the active state.

In this case, a default.tsx file can be rendered for subpages that don't match the current URL.

Consider the following folder structure. The @team slot has a settings page, but @analytics does not.

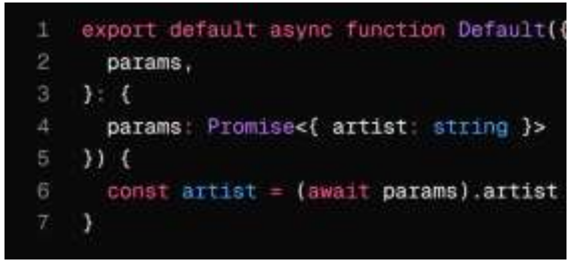
When navigating to /settings, the @team slot will render the settings page while maintaining the currently active page for the @analytics slot.

On refresh, Next.js will render a default.js for @analytics. If default.js doesn't exist, a 404 is rendered instead.



```
app
  @team
    settings
      page.js
  @analytics
    default.js
    page.js
  default.js
  layout.js
  page.js
```

Props: None



```
export default async function Default({
  params,
}: {
  params: Promise<{ artist: string }>
}) {
  const artist = (await params).artist
}
```

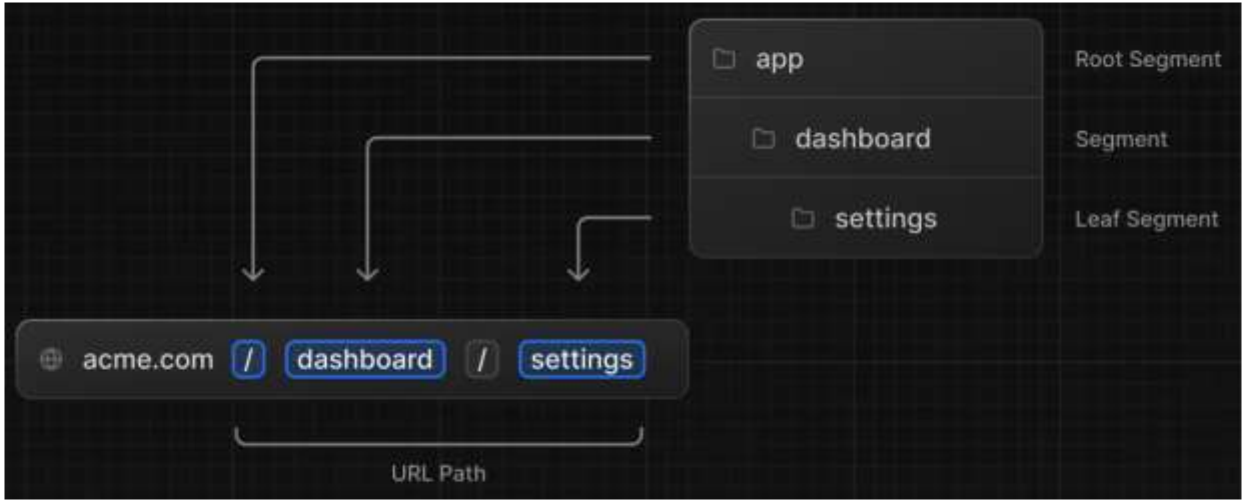
Roles of Folders and Files

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.

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

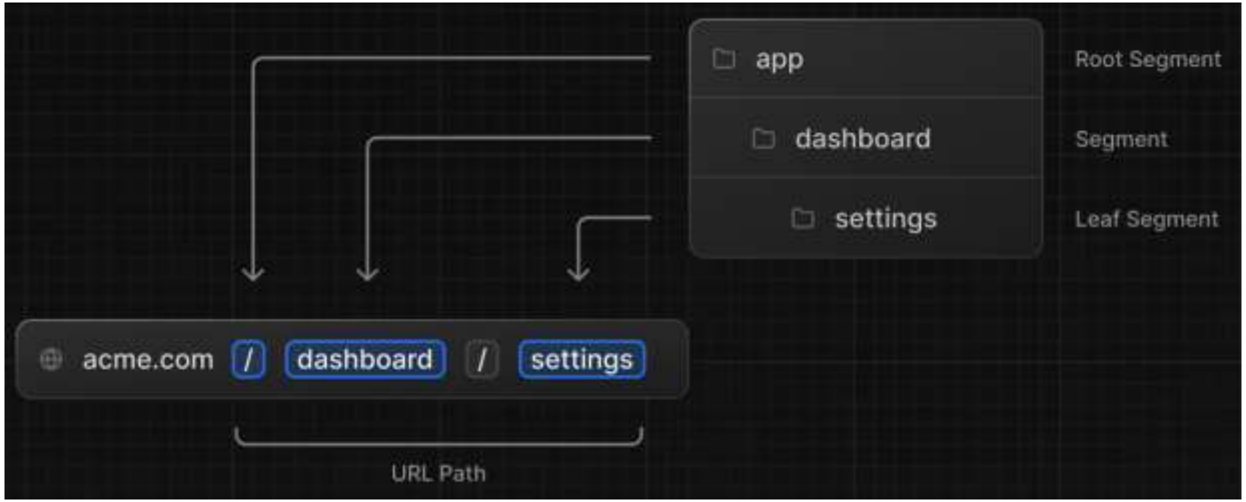


Nested routes

To create a nested route, you can **nest folders** inside each other. For example, you can add a new /dashboard/settings route by nesting two new folders in the app directory.

The /dashboard/settings route is composed of three segments:

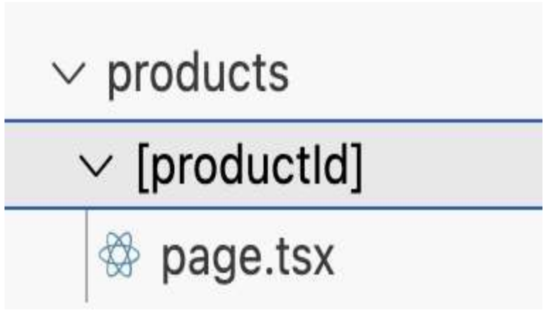
- / (Root segment)
- dashboard (Segment)
- settings (Leaf segment)



Dynamic routes

- A dynamic route is a flexible URL structure that adapts to different values. Instead of hardcoding every URL, you create a template that changes based on the data provided.
- Use square brackets to define a dynamic segment.
- Example: [id].ts creates a route where /products/[id] renders content based on the id parameter.

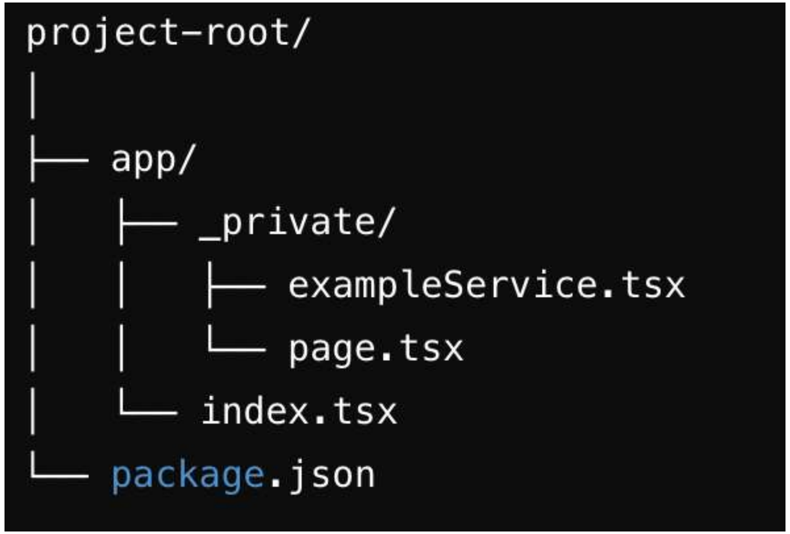
Note
Next.js 15 onwards, Params and SearchParams are now Promise.



Private Folders

- Private folders indicate that it is a private implementation detail and shouldn’t be considered by the routing system.
- The folder and all its subfolders are excluded from the routing system.
- To create a private folder : Prefix the folder name with an underscore

- **Advantages**
  - For separating UI logic from routing logic.
  - For consistently organizing internal files across a project.
  - For avoiding potential naming conflicts in future Next.js file convention.



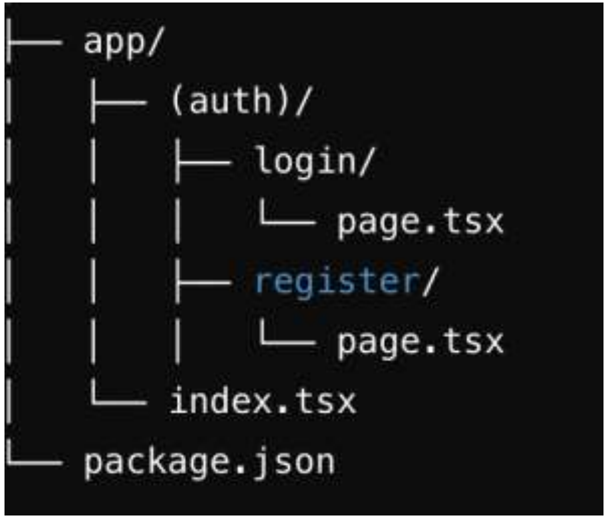
```
project-root/
│
├── app/
│   ├── _private/
│   │   ├── exampleService.tsx
│   │   └── page.tsx
│   └── index.tsx
└── package.json
```

Route Groups

- Route groups allow you to organize related routes into a single group, making the routing structure clearer.
- They are defined by enclosing folder names in parentheses, such as (auth), (dashboard), etc.

Why Use Route Groups?

- **Organization**: Keeps your file structure clean by grouping related routes together.
- **Scalability**: Easily scale your app by adding more grouped routes as needed.



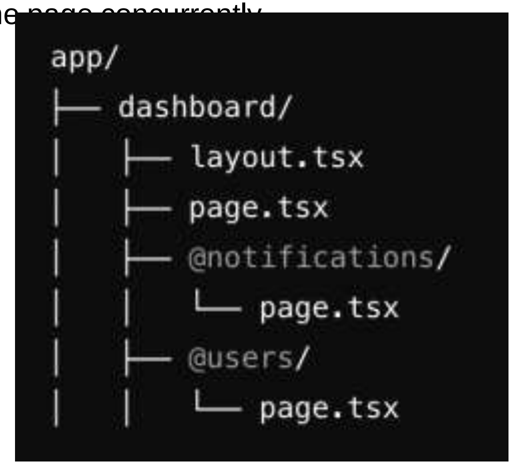
```
├── app/
│   ├── (auth)/
│   │   ├── login/
│   │   │   └── page.tsx
│   │   ├── register/
│   │   │   └── page.tsx
│   └── index.tsx
└── package.json
```

Parallel Routes

- **Parallel routes allow you to render multiple routes at the same time in different sections of the layout, providing an efficient way to organize components and optimize performance.**

Why Use Parallel Routes?

- **Independent Rendering**: Components like notifications and user data can load independently without waiting for the other and have their own loading and error states
- **Optimized User Experience**: Reduces waiting time by loading different parts of the page concurrently.



```
app/
├── dashboard/
│   ├── layout.tsx
│   ├── page.tsx
│   ├── @notifications/
│   │   └── page.tsx
│   ├── @users/
│   │   └── page.tsx
```

How Parallel Routes Work

1. `layout.tsx`: The main layout for the dashboard, which will contain the overall structure.
2. `page.tsx`: The main dashboard page content.
3. **Parallel Routes**:
   - **@notifications**: Rendered in a section of the dashboard layout.
   - **@users**: Another section rendered alongside notifications.

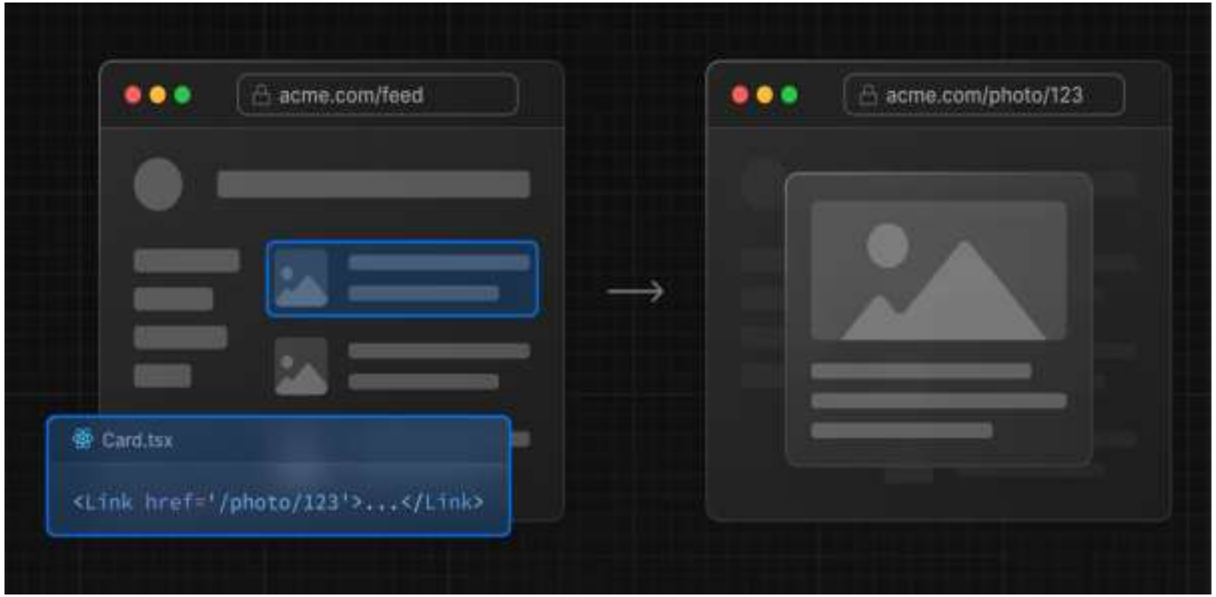
Parallel routes are created using named **slots**. Slots are defined with the **@folder** However, slots are **not route segments**

Intercepting Routes

- **Intercepting routes allows you to load a route from another part of your application within the current layout. This routing paradigm can be useful when you want to display the content of a route without the user switching to a different context.**

For example, when clicking on a photo in a feed, you can display the photo in a modal, overlaying the feed. In this case, Next.js intercepts the /photo/123 route, masks the URL, and overlays it over /feed.

However, when navigating to the photo by clicking a shareable URL or by refreshing the page, the entire photo page should render instead of the modal. No route interception should occur.



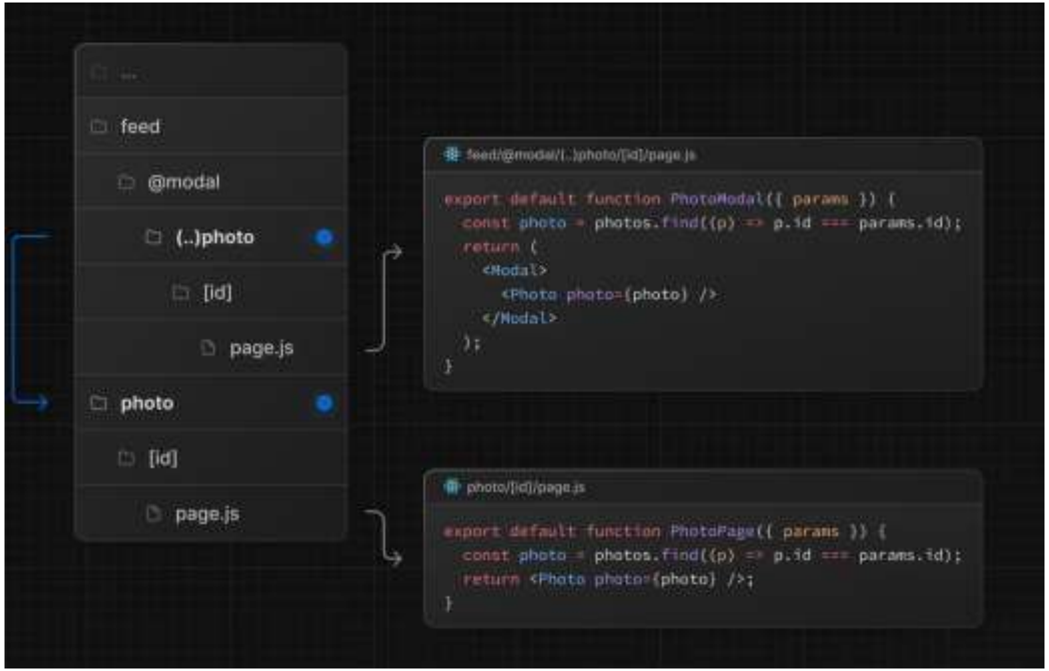
```
<Link href='/photo/123'>...</Link>
```

Intercepting Routes

Intercepting routes can be defined with the (..) convention, which is similar to relative path convention ../ but for segments.

You can use:

- (.) to match segments on the same level
- (..) to match segments one level above
- (..)(..) to match segments two levels above
- (...) to match segments from the root app directory



feed/@modal/(..)photo/[id]/page.js

```
export default function PhotoModal({ params }) {
  const photo = photos.find((p) => p.id === params.id);
  return (
    <Modal>
      <Photo photo={photo} />
    </Modal>
  );
}
```

photo/[id]/page.js

```
export default function PhotoPage({ params }) {
  const photo = photos.find((p) => p.id === params.id);
  return <Photo photo={photo} />;
}
```