Training on Next.js 13^

COSDD SysDev

July 11, 2024

Training Coverage

Day 1

- Introduction and understanding the new project structure in Next.js 13
- Project Organization and File Colocation
- Pages and routing in Next.js
- Creating and navigating between pages using the `Link` component
- Adding child component
- Adding images and other assets
- Basic use of tailwind

What's in Next.js?

- React framework for building **full-stack** web applications
- Next.js allow you to create hybrid web applications where parts of your code can be **rendered on the server or the client**.
- Next.js allows you to create **backend endpoints** within the same project, simplifying full-stack development.
- File-based routing
- Built-in CSS and Tailwind Support

Next.js vs Create React App (CRA)

Features	Next.js	CRA
Rendering Option	SSR, SSG, ISR, CSR	CSR only
Routing	File-based routing	Manual routing with react-router
API Routes	Built-in	Requires separate backend
Performance	Automatic optimizations	Requires manual optimizations
CSS and Styling	CSS modules, global styles, Sass	CSS modules, Sass
Internationalization	Built-in	Requires third-party libraries
SEO	Excellent with SSR and SSG	Limited due to CSR
Development Setup	Requires some initial learning	Simple and quick to start
Use Cases	Complex, high-performance, SEO-focused	Small to medium projects, SPAs, prototypes

Requirements and other sources

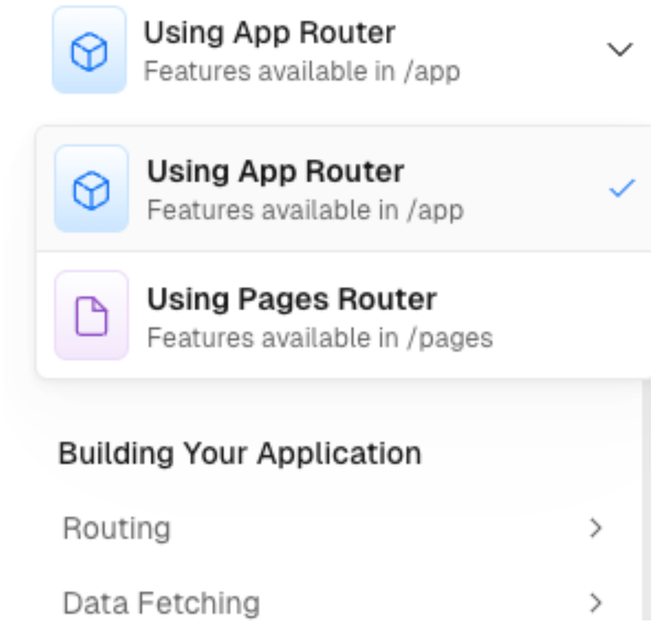
System Requirements:

- Node.js 18.17 or later
- npm

Sources:

- Next.js Documentation
  https://nextjs.org/docs

- Tailwind Documentation
  https://tailwindcss.com/



Installation

* npx create-next-app@latest or npx create-next-app@latest **app_name**

```
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
What import alias would you like configured? @/* (if yes)
```

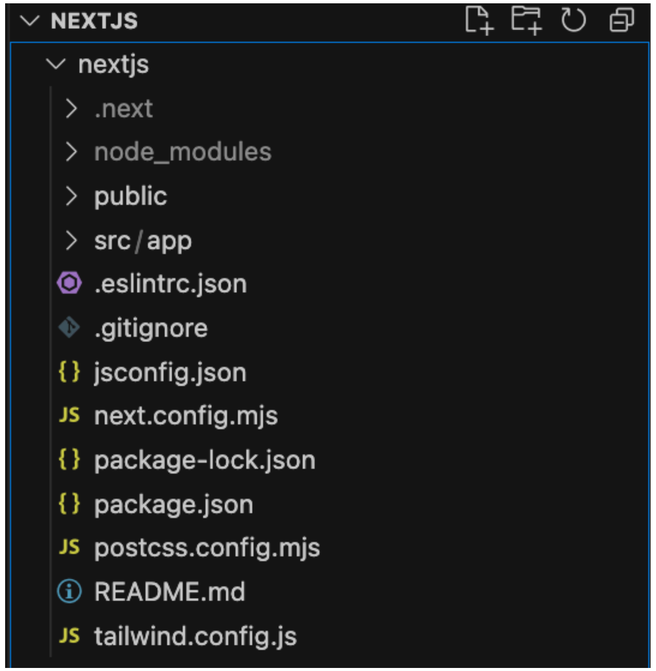
```
Installing dependencies:
- react
- react-dom
- next

Installing devDependencies:
- postcss
- tailwindcss
- eslint
- eslint-config-next
```

Installation

```
npm run dev: runs next dev to start Next.js in development mode.
npm run build: runs next build to build the application for production usage.
npm run start: runs next start to start a Next.js production server.
npm run lint: runs next lint to set up Next.js' built-in ESLint configuration.
```

Project Structure

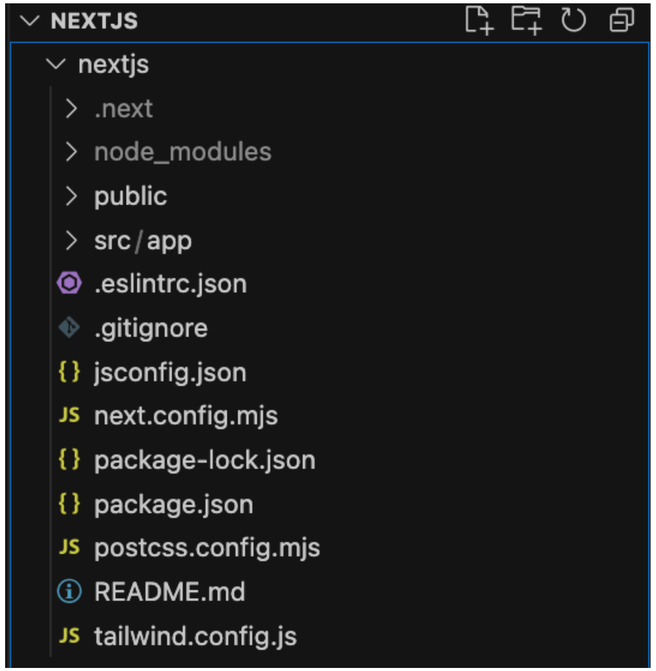
The `.next` directory is automatically created by Next.js. It contains all the build outputs and temporary files needed for your application to run. This directory should generally be excluded from version control (like Git) because it is automatically generated and can be rebuilt at any time.



Project Structure

The node_modules directory is where all the dependencies (packages) for your Node.js project are stored.

- The node_modules directory is typically very large and is not included in version control systems like Git.
- Instead, you only include the package.json and package-lock.json (or yarn.lock) files.



Installation: npm install

Adding dependency: npm install <package-name> –save

Rebuilding: rm -rf node_modules
npm install

Project Structure

The public folder can serve static files, like images in the root directory.
Files inside public can then be referenced by your code starting from the base URL (/).

A .gitignore file is used in Git to specify which files and directories should be ignored by Git when you add files to your repository.

The .eslintrc.json file is a configuration file used by ESLint, which is a popular linting tool for JavaScript and TypeScript code.

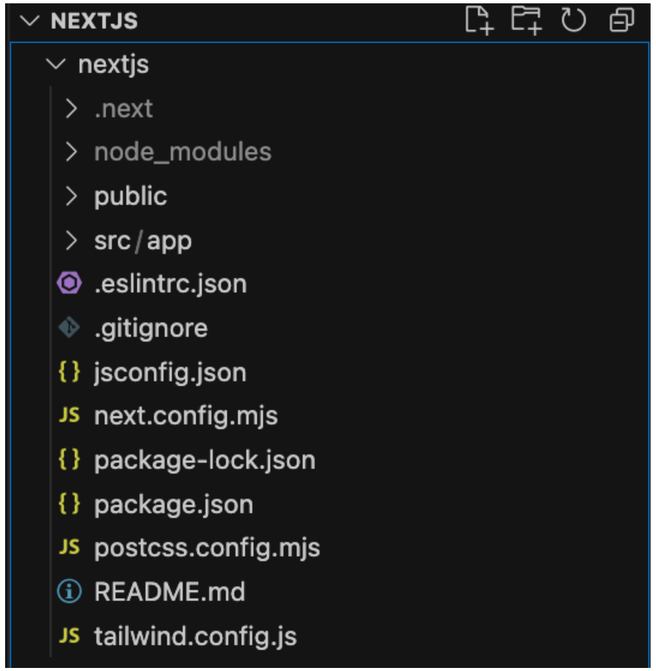
next.config.mjs you to override and customize various settings of the Next.js framework, such as webpack configurations, environment variables, and more.

package-lock.json locks down the exact version of each dependency and its transitive dependencies (node_modules)

package.json defines the project's requirements and scripts

postcss.config.mjs is a configuration file used by PostCSS

tailwind.config.js is a configuration file used by Tailwind CSS



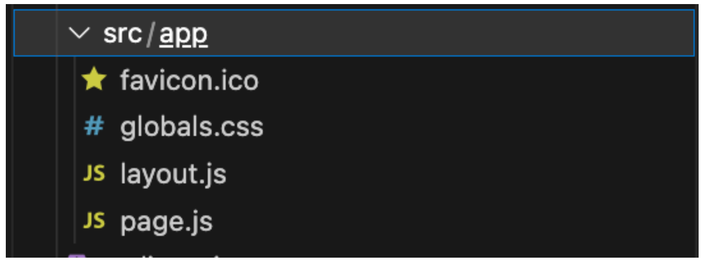
Project Structure

`global.css` can be imported into any layout, page, or component inside the app directory.

`styles.module.css` locally scope CSS by automatically creating a unique class name.

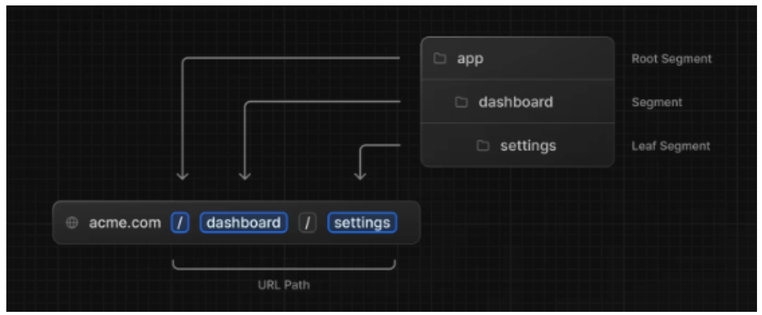
`layout.js` is UI that is shared between routes.

`page.js` is UI that is unique to a route.

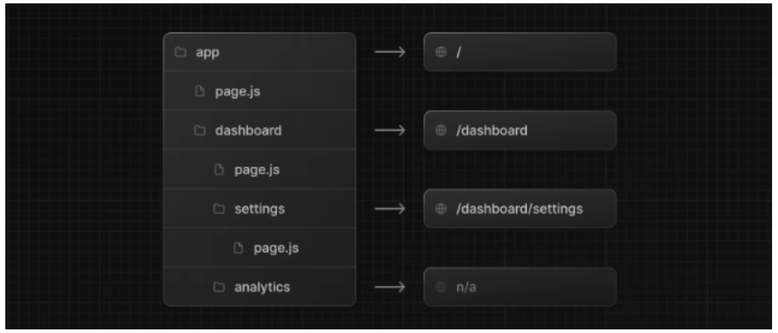


Routing

Next.js uses a file-system based router where folders are used to define routes.



A special page.js file is used to make route segments publicly accessible.



.js, .jsx, or .tsx file extensions can be used for Layouts, Pages and other child components

Routing

1. Static Route

app/dashboard/page.js

```
    <Link href='/dashboard'>Dashboard</Link>
```

```
export default function Home() {
  return (
    <div className='w-full h-screen justify-center items-center flex'>
      <div className='w-6/12 items-center justify-center flex-column'>
        <div className='w-full flex items-center justify-center'>
          <h5>Dashboard</h5>
        </div>
      </div>
    </div>
  );
}
```

Routing

2. Dynamic route can be created by wrapping a folder's name in square brackets: [folderName]

app/profile/[user_id]/page.js

```
  <Link href='/profile/2'>User Profile</Link>
```

```
export default function Profile({ params }) {
  const user_id = params.user_id;
  return (
    <main className='flex min-h-screen flex-col items-center gap-4 p-24'>
      <h5>User ID: {user_id}</h5>
    </main>
  );
}
```

Routing

3. Query String is a part of a URL that contains key-value pairs of parameters and their values.

app/account/page.js

```
      <Link href='/account?user_id=2'>Account</Link>
```

```
import { useSearchParams } from "next/navigation";
export default function Account() {
  const query = useSearchParams();
  const sub = query.get("user_id");
  return (
    <main className='flex min-h-screen flex-col items-center gap-4 p-24'>
      <h5>User ID: {sub}</h5>
    </main>
  );
}
```

Routing

4. Dynamic Segments can be extended to catch-all subsequent segments by adding an ellipsis inside the brackets […folderName].

app/settings/[…user]/page.js

```
<Link href='/settings/Hello/new/world'>Setting</Link>
```

```
export default function Setting({ params }) {
  const myUser = params.user;
  return (
    <main className='flex min-h-screen flex-col items-center gap-4 p-24'>
      <h5>
        What I say: {myUser[0]} {myUser[1]} {myUser[2]}
      </h5>
    </main>
  );
}
```

<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 and recommended way to navigate between routes in Next.js.

```
import Link from "next/link";

export default function Home() {
  return (
    <main className='flex min-h-screen flex-col items-center gap-4 p-24'>
      <Link href='/dashboard'>Dashboard</Link>
      <Link href='/profile/2'>User Profile</Link>
      <Link href='/account?user_id=2'>Account</Link>
      <Link href='/settings/Hello/new/world'>Setting</Link>
    </main>
  );
}
```

useRouter hook

The useRouter hook allows you to programmatically change routes inside Client Components.

```tsx
import { useRouter } from "next/navigation";

export default function Home() {
  const router = useRouter();
  return (
    <main className='flex min-h-screen flex-col items-center gap-4 p-24'>
      <button type='button' onClick={() => router.push("/dashboard")}>
        Dashboard
      </button>
      <button type='button' onClick={() => router.push("/profile/2")}>
        Profile
      </button>
      <button type='button' onClick={() => router.push("/account?user_id=2")}>
        Account
      </button>
      <button
        type='button'
        onClick={() => router.push("/settings/Hello/new/world")}>
        Setting
      </button>
    </main>
  );
}
```

```
const router: AppRouterInstance
```

```
const router = useRouter();
```

Child Component

Child components are modular chunks of the user interface using Static import

Static Import : Use this approach for components that are critical for the initial render of the page or that are small and don't significantly impact the bundle size.

Child components:
app/button/buttonA.jsx
app/button/buttonB.jsx

Parent Component:
App/dashboard/pages.js

```jsx
import ButtonA from "@/app/(components)/button/buttonA";
import ButtonB from "@/app/(components)/button/buttonB";

export default function Home() {
  return (
    <div className='w-full h-screen justify-center items-center flex'>
      <div className='w-6/12 items-center justify-center flex-column'>
        <div className='w-full flex items-center justify-center'>
          <h5>Dashboard</h5>
        </div>
        <div className='w-full flex'>
          <div className='w-6/12'>
            <ButtonA />
          </div>
          <div className='w-6/12'>
            <ButtonB />
          </div>
        </div>
      </div>
    </div>
  );
}
```

Child Component

Child components are modular chunks of the user interface using Dynamic import.

Dynamic Import : Use this approach for large components, components that are conditionally rendered, or components that are not essential for the initial render of the page. This helps in improving the performance and load time of your application.

Child components:
app/button/buttonA.jsx
app/button/buttonB.jsx

Parent Component:
App/dashboard/pages.js

```jsx
import ButtonA from "@/app/(components)/button/buttonA";
import ButtonB from "@/app/(components)/button/buttonB";

export default function Home() {
  return (
    <div className='w-full h-screen justify-center items-center flex'>
      <div className='w-6/12 items-center justify-center flex-column'>
        <div className='w-full flex items-center justify-center'>
          <h5>Dashboard</h5>
        </div>
        <div className='w-full flex'>
          <div className='w-6/12'>
            <ButtonA />
          </div>
          <div className='w-6/12'>
            <ButtonB />
          </div>
        </div>
      </div>
    </div>
  );
}
```

Child Component

Child components are modular chunks of the user interface using Static and Dynamic import.

Child components:
app/button/buttonA.jsx
app/button/buttonB.jsx

Parent Component:
App/dashboard/pages.js

```jsx
import ButtonA from "@/app/(components)/button/buttonA";
import dynamic from "next/dynamic";

export default function Home() {
  const ButtonB = dynamic(() => import("@/app/(components)/button/buttonB"), {
    ssr: false,
  });
  return (
    <div className='w-full h-screen justify-center items-center flex'>
      <div className='w-6/12 items-center justify-center flex-column'>
        <div className='w-full flex items-center justify-center'>
          <h5>Dashboard</h5>
        </div>
        <div className='w-full flex'>
          <div className='w-6/12'>
            <ButtonA />
          </div>
          <div className='w-6/12'>
            <ButtonB />
          </div>
        </div>
      </div>
    </div>
  );
}
```

Adding Image and other assets

The Next.js Image component extends the HTML <img> element with features for automatic image optimization:

Size Optimization: Automatically serve correctly sized images for each device, using modern image formats like WebP and AVIF

Visual Stability: Prevent layout shift automatically when images are loading.

Faster Page Loads: Images are only loaded when they enter the viewport using native browser lazy loading, with optional blur-up placeholders.

Asset Flexibility: On-demand image resizing, even for images stored on remote servers

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

export default function Home() {
  return (
    <main className='flex min-h-screen flex-col items-center gap-4 p-24'>
      <Image
        src='/nha-logo.png'
        alt='NHA Logo'
        className='dark:invert'
        width={100}
        height={24}
        priority
      />
    </main>
  );
}
```