Next.js vs. React: A Comparison for Developers

As a React developer, moving to Next.js is less about learning a completely new language or paradigm and more about understanding a powerful framework built *on top* of React. The core concepts of building UIs with components, managing state, and using JSX remain the same. However, Next.js introduces structure and built-in features that significantly change how you handle routing, data fetching, rendering, and API creation, especially when compared to a standard Create React App (CRA) or a similar client-side React setup.

Let's break down the similarities and differences:

Similarities

1. **Core UI Library:** Both use React as the foundation for building user interfaces. You'll still write components, use hooks, manage state (with useState, useReducer, Context API, etc.), and work with props.
2. **JSX:** You'll continue writing your UI using JSX syntax.
3. **Component-Based Architecture:** Both encourage breaking down your UI into reusable components.
4. **Ecosystem:** You can use most React libraries and npm packages in a Next.js project.
5. **Development Experience:** Features like Fast Refresh (hot reloading) are present in both environments, providing a smooth development workflow.

Differences

This is where Next.js shines and introduces new concepts:

Feature	Typical React (e.g., CRA)	Next.js (App Router)	Explanation
Routing	Uses client-side libraries (e.g., React Router) to define routes programmatically.	File-system based routing (app directory structure). Folders define routes, page.jsx makes a route active.	Next.js simplifies routing setup by using your folder structure.
Rendering	Primarily Client-Side Rendering (CSR).	Multiple strategies: SSR, SSG, ISR, CSR.	Pre-rendering (SSR/SSG) improves

	HTML is minimal, JavaScript renders content in the browser.	Next.js automatically chooses or allows you to specify.	performance, SEO, and initial load time.
Data Fetching	Typically done client-side using fetch, Axios, SWR, React Query within useEffect or component lifecycle methods.	Optimized server-side data fetching in Server Components. fetch is automatically extended with caching/revalidation. Client-side fetching is still possible.	Moving data fetching to the server can reduce client-side JavaScript and improve performance.
API Creation	Requires a separate backend project (e.g., Node.js with Express, Python with Django/Flask).	Built-in API Routes (Route Handlers in App Router) within the same project (app/api directory).	Allows building full-stack applications within a single codebase for simpler projects or microservices.
Project Structure	More flexible, often centered around a src or components folder.	More opinionated, centered around the app directory for routing and special files (page.jsx, layout.jsx).	Provides a clear structure for defining routes and shared UI.
Code Splitting	Often requires manual configuration or is handled by bundlers (like Webpack in CRA).	Automatic code splitting by route segment. Only loads the JavaScript needed for the current page.	Improves performance by reducing the initial download size.
Server Components	Not a core concept. All components render on the client.	Introduced with the App Router. Components can render on the server (default) or client ('use client').	Enables fetching data directly on the server and reduces client-side JavaScript bundle size.

Your Next Project: A Guide to Getting Started with Next.js

Since you're coming from React, the transition should feel quite natural once you grasp the core Next.js concepts, especially the App Router structure. Here's a step-by-step guide:

Step 1: Install Node.js and npm/yarn/pnpm

Ensure you have a recent version of Node.js installed. This comes with npm. You might prefer yarn or pnpm as well.

Step 2: Create a New Next.js Project

Open your terminal and run the create-next-app command:

npx create-next-app@latest my-next-project

Follow the prompts. For a modern project, it's recommended to:

- Use TypeScript (even if you write JSX, it helps with tooling and catching errors).
- Enable ESLint.
- Use the src directory.
- Use the **App Router** (this is the newer, recommended routing system).
- (Optional) Configure Tailwind CSS if you like utility-first styling.

Step 3: Understand the Project Structure (App Router)

Navigate into your new project folder:

cd my-next-project

Familiarize yourself with the key directories and files:

- app/: This is the core of the App Router. Folders here define routes.
  - app/layout.jsx (or .tsx): The root layout. Wraps all pages. Ideal for global elements like <html>, <body>, Navbar, and Footer.
  - app/page.jsx (or .tsx): The root page for the / route (your homepage).
  - app/globals.css: Global styles, often used for Tailwind CSS directives or base styles.
  - app/[folder-name]/page.jsx: Creates a route at /[folder-name].
  - app/[folder-name]/layout.jsx: Creates a layout for /[folder-name] and its nested routes.

    - app/api/[route]/route.jsx (or .tsx): Creates an API endpoint at /api/[route]. These are Route Handlers.
- components/: Create this folder (or src/components) to store your reusable React components (like the Hero, Navbar, Footer components you saw). These are imported into your page.jsx, layout.jsx, or other components.
- public/: For static assets like images, fonts, etc. Files here can be referenced directly from the root URL (e.g., /my-image.png).
- package.json: Project dependencies and scripts (npm run dev, npm run build, npm run start).

Step 4: Create Your Pages and Layouts

- **Root Layout (app/layout.jsx):** This is where you'll place elements that appear on every page, like the Navbar and Footer. You'll import your Navbar and Footer components here and render them around the children prop, which represents the current page content.
- **Home Page (app/page.jsx):** This file will contain the component for your homepage. You'll import and use your Hero component here.
- **Other Pages (app/[route]/page.jsx):** Create folders inside app for each route (e.g., app/about/ for the /about route, app/contact/ for the /contact route). Inside each folder, create a page.jsx file and export a React component for that page's content.

Step 5: Build Your Components

Create your reusable components (Navbar.jsx, Hero.jsx, Footer.jsx, etc.) inside the components folder. These are standard React components.

- **Client Components:** If a component needs client-side interactivity (like state, event listeners, browser APIs), add 'use client'; at the very top of the file. Your Navbar with a mobile menu toggle is a good example.
- **Server Components:** By default, components are Server Components. They are great for static content, fetching data on the server, and reducing client-side JavaScript. Your Hero and Footer can likely be Server Components.

Step 6: Styling

Choose your preferred styling method (Tailwind CSS, CSS Modules, global CSS, CSS-in-JS) and apply styles to your components. Next.js has excellent built-in support for CSS Modules and works seamlessly with Tailwind CSS.

Step 7: Data Fetching (Optional but Common)

If your pages need data:

- **Server Components:** Use async/await directly in your Server Component to fetch data, often using the native fetch API. Next.js automatically handles caching and revalidation.
- **Client Components:** Use useEffect with a data fetching library or fetch if the data needs to be fetched client-side (e.g., based on user interaction).

Step 8: Run and Build

- npm run dev: Starts the development server with hot reloading.
- npm run build: Builds your application for production. Next.js will automatically optimize your code, determine rendering strategies, and prepare files for deployment.
- npm run start: Starts the production server after building.

Step 9: Deployment

Deploy your built Next.js application. Vercel and Netlify are popular choices that offer easy deployment by connecting to your Git repository.

In Summary

Think of Next.js as providing the "structure" and "performance optimizations" around your React components. You still write React, but you leverage Next.js's conventions for routing and rendering to build more robust, performant, and SEO-friendly applications with less manual configuration than a purely client-side React app. The file structure in the app directory is the main difference you'll notice immediately, as it dictates your application's navigation.

Good luck with your next project!