Next JS

Table of Content

- Introduction
- Setting up development environment

Introduction

- What is Next.js?
- Prerequisites
- Why Learn Next.js?
- Why Next.js over Traditional React
- SSR + SSG + ISR + RSC Quick Reference
- App Router vs Pages Router

What is Next.js?

Next.js is a powerful React framework used to build **full-stack, production-ready web applications**. While React itself is a library focused solely on building user interfaces, it lacks the structure and features required for creating complete applications.

Next.js builds on top of React, providing a robust framework that includes:

- File-based routing
- Optimized rendering (server-side and client-side)
- Integrated API routes
- Bundling and compiling
- Performance optimizations

These features come built-in, so there's no need to manually install or configure third-party libraries for most use cases. Next.js follows certain conventions and opinions, but they are based on real-world experience building production applications and make development more efficient and scalable.

Prerequisites

Before starting with Next.js, you should be familiar with:

- **HTML and CSS**
- **Modern JavaScript** (ES6+)
- **React fundamentals**, including:
    - Functional components
    - Props and state
    - JSX
    - React Hooks

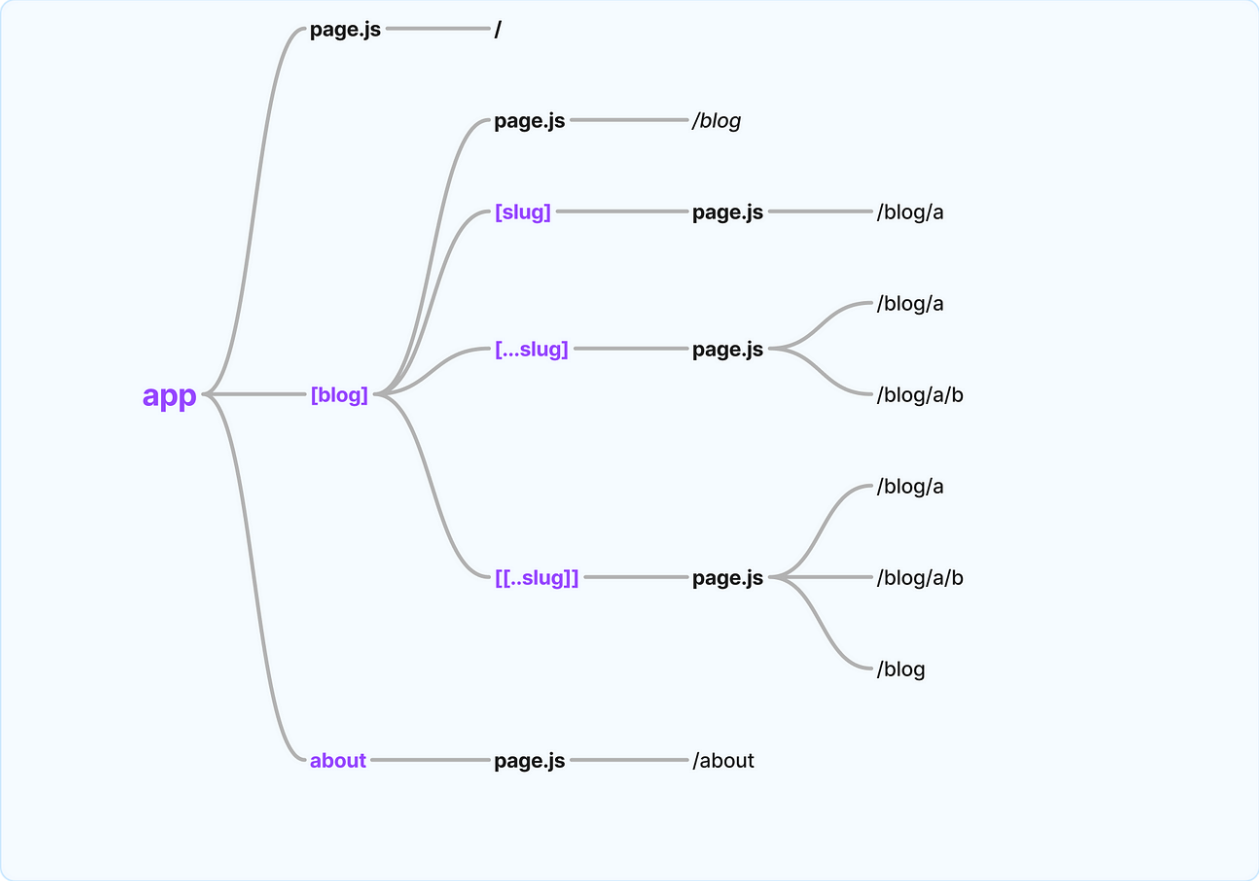
Why Learn Next.js?

Next.js greatly simplifies the process of building production-ready web applications. Some of the standout features include:

- **File-Based Routing**
- **Built-In API Routes**
- **Rendering Flexibility**
- **Streamlined Data Fetching**
- **Performance Optimizations**
- **Optimized Build System**

File-Based Routing

Instead of configuring a routing library manually (as in traditional React apps), you simply create files in the `app` or `pages` directory, and routes are generated automatically.



Built-In API Routes

Next.js allows you to build both frontend React components and backend APIs in the same codebase. This tight integration streamlines development and simplifies full-stack applications.

Rendering Flexibility

Next.js supports multiple rendering methods:

- Server-Side Rendering (SSR)
- Static Site Generation (SSG)
- Client-Side Rendering (CSR)

This flexibility allows for better performance and SEO (Search Engine Optimization).

Streamlined Data Fetching

With built-in support for asynchronous data fetching in React components, fetching data becomes more intuitive and efficient.

Styling Options

Next.js supports various styling methods:

- CSS Modules
- Tailwind CSS
- CSS-in-JS solutions

Performance Optimizations

Next.js includes automatic optimizations for:

- Images
- Fonts
- Scripts

These help improve **Core Web Vitals** and enhance the overall user experience.

Optimized Build System

Next.js offers a seamless development experience and an efficient production build pipeline—allowing you to focus on building features rather than dealing with configurations.

Why Next.js over Traditional React

When comparing Next.js to traditional React applications (created with Create React App or Vite), there are several key differences that make Next.js a more powerful choice for building modern web applications.

Feature	React (CRA/Vite)	Next.js 15 (App Router)
Routing	Manual (React Router)	Filesystem-based, automatic

Feature	React (CRA/Vite)	Next.js 15 (App Router)
Server-side rendering	❌	☑ Built-in with `fetch()`
Metadata / SEO	Manual	Automatic via `metadata` export
Static + Dynamic Pages	Custom logic	Built-in (`revalidate`, `cache`)
Server components	❌	☑ `app/` supports RSC out-of-box

SSR + SSG + ISR + RSC Quick Reference

SSR (Server-Side Rendering): Fetches data on each request, generating HTML on the server. This is useful for dynamic content that changes frequently.

SSG (Static Site Generation): Pre-renders pages at build time, generating static HTML. This is ideal for content that doesn't change often.

ISR (Incremental Static Regeneration): Combines the benefits of SSR and SSG. It allows you to update static pages after the build process, making it suitable for content that changes occasionally.

RSC (React Server Components): A new way to build components that run on the server. This allows for better performance and reduced client-side JavaScript.

Key features of each rendering method:

Feature	How you write it	Result
SSR	`await fetch(...)` with `cache: 'no-store'`	Runs on every request
SSG	`await fetch(...)` with `cache: 'force-cache'`	Static HTML
ISR	`await fetch(...)` with `next: { revalidate: 60 }`	Rebuilds after 60s
RSC	Default when no `use client`	Server-rendered with streaming

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.

Setting up development environment

- **Installing Node.js and npm**
- **Next.js Blog Post Application Requirements**

Installing Node.js and npm

> **Node.js** includes **npm** (Node Package Manager), which you'll use to install dependencies like Next.js.

- **1. Windows Setup**
- **2. macOS Setup**
- **3. Linux Setup**

---

1. Windows Setup

Official Installer

1. Visit the official Node.js website:

- 👉 https://nodejs.org
- Download the **LTS version** (recommended for most users).

2. Run the installer:

- Follow the prompts (use default settings).
- Installs both **Node.js** and **npm**.

3. Verify installation in Command Prompt or PowerShell:

```
node -v
npm -v
```

Using nvm-windows (Node Version Manager for Windows)

> ⚠ This is different from the nvm used on Linux/macOS.

1. Go to:

- 👉 https://github.com/coreybutler/nvm-windows/releases

2. Download and run `nvm-setup.exe`.

3. In Command Prompt or PowerShell:

```
nvm install lts
nvm use lts
```

---

2. macOS Setup

Official Installer

1. Visit https://nodejs.org and download the **macOS Installer (LTS)**.

2. Run the `.pkg` file and complete the installation.

3. Verify installation:

```
node -v
npm -v
```

Using Homebrew + NVM

> NVM lets you manage multiple Node.js versions.

1. Install Homebrew (if not already installed):

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Install nvm:

```
brew install nvm
mkdir ~/.nvm
```

3. Add to your shell config (`~/.zshrc`, `~/.bash_profile`, or `~/.zprofile`):

```
export NVM_DIR="$HOME/.nvm"
[ -s "$HOMEBREW_PREFIX/opt/nvm/nvm.sh" ] && \. "$HOMEBREW_PREFIX/opt/nvm/nvm.sh"
```

4. Apply changes:

```
source ~/.zshrc  # or ~/.bash_profile, depending on your shell
```

5. Install Node.js:

```
nvm install --lts
nvm use --lts
nvm alias default node
```

6. Verify installation:

```

```

```
node -v
npm -v
```

---

3. Linux Setup

Official NodeSource Installer

1. Run the following (replace `18` with your desired version):

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install -y nodejs
```

2. Verify installation:

```
node -v
npm -v
```

Using NVM

1. Install NVM:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

2. Load NVM (add to `~/.bashrc` or `~/.zshrc`):

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
```

3. Apply changes:

```
source ~/.bashrc  # or source ~/.zshrc
```

4. Install Node.js:

```
nvm install --lts
nvm use --lts
nvm alias default node
```

5. Verify installation:

```
node -v
npm -v
```

Next.js Blog Post Application Requirements

- **Objective**
- **Functional Requirements**
- **Technical Requirements**
- **Data Models**
- **Non-Functional Requirements**
- **Optional Enhancements**
- **Tech Stack**

Objective

To develop a **basic blog post application** using **Next.js 15** that showcases all major features of the framework, including the **App Router**, **server components**, **client components**, **dynamic routing**, **API routes**, **authentication**, **SEO optimization**, **image optimization**, **static generation**, **server-side rendering**, and more. The app will serve as a learning and demo tool.

Functional Requirements

- 1. **Homepage**
- 2. **Blog Post Page**
- 3. **Create/Edit Blog Post Page**
- 4. **Authentication**
- 5. **Comments Section**
- 6. **Admin Dashboard**
- 7. **Search and Filtering**
- 8. **Tag and Category Pages**

1. Homepage

- Lists latest blog posts with title, excerpt, author, date, and thumbnail.
- Uses `fetch()` from a server component.
- Paginated or infinite scrolling.

2. Blog Post Page

- Dynamic route: `/blog/[slug]`
- Displays full blog content.

- Rendered statically (SSG) if possible or SSR otherwise.
- Includes metadata (title, og tags) using `generateMetadata`.

3. Create/Edit Blog Post Page

- Protected via authentication (admin-only).
- Rich text editor (Markdown or WYSIWYG).
- Uses client component + API call to submit.
- Form validation with Zod or Yup.

4. Authentication

- Uses `next-auth` (auth.js).
- GitHub, Google OAuth login.
- Session-based access control.
- Logged-in users can like posts, add comments.

5. Comments Section

- Comment form (client component).
- Displays threaded comments (SSR or ISR).
- Likes or reactions on comments (optional).

6. Admin Dashboard

- Route: `/admin`
- List, edit, delete blog posts.
- Add new categories/tags.
- Protected route using middleware and session checking.

7. Search and Filtering

- Client-side filtering by tag/category.
- Full-text search using server-side route or client search.

8. Tag and Category Pages

- Dynamic routes: `/tag/[tag]`, `/category/[category]`
- Lists all posts under a tag or category.
- Uses `generateStaticParams` and `generateMetadata`.

Technical Requirements

Next.js Features to Demonstrate

Feature	Description
App Router	Use `/app` directory for routing

Feature	Description
Server Components	Fetch blog data from server components
Client Components	Rich text editor, search bar, comment form
Dynamic Routing	Blog posts, categories, tags
`generateMetadata()`	Dynamic SEO on each page
API Routes (`app/api`)	For CRUD operations on posts/comments
Authentication (`next-auth`)	Secure login, protected routes
Layouts & Templates	Global layout, nested layouts
Static Generation (SSG)	For most blog and category pages
Server-Side Rendering (SSR)	For search and user-specific pages
Image Optimization	Blog post thumbnails with `<Image>`
Middleware	Session handling and route protection
ISR (Incremental Static Regeneration)	Revalidate content after changes
Error & Loading UI	`error.js`, `loading.js` per route
Environment Variables	For secure API keys
SEO/OG Tags	Using dynamic metadata
Deployment	Vercel or Docker deployment

Data Models

- **Blog Post**
- **Comment**

Blog Post

```
{
  id: string;
  title: string;
  slug: string;
  content: string;
  thumbnailUrl: string;
  authorId: string;
  createdAt: Date;
  updatedAt: Date;
  tags: string[];
  category: string;
}
```

Comment

```
{
  id: string;
  postId: string;
  author: string;
  message: string;
  parentId?: string;
  createdAt: Date;
}
```

Non-Functional Requirements

- **Responsive Design**: Mobile-first using Tailwind CSS.
- **Accessibility**: Authentication and Authorization.
- **Performance**: Lazy loading images, server components.
- **Security**: Auth, sanitization of user input.
- **Code Quality**: TypeScript, ESLint, Prettier.
- **Testing**: Unit tests (Jest), E2E (Playwright or Cypress).

Optional Enhancements

- Dark Mode toggle.
- Markdown editor with preview.
- Author profiles and bios.
- RSS Feed and sitemap generation.
- Email notifications on comments.
- PWA support.

Tech Stack

- **Frontend**: Next.js 15 (App Router), TypeScript, Tailwind CSS
- **Backend**: Next.js API routes or external backend
- **Database**: SQLite, PostgreSQL, or MongoDB (via Prisma or Mongoose)
- **Auth**: `next-auth`
- **Deployment**: Vercel / Docker + Render / Railway
