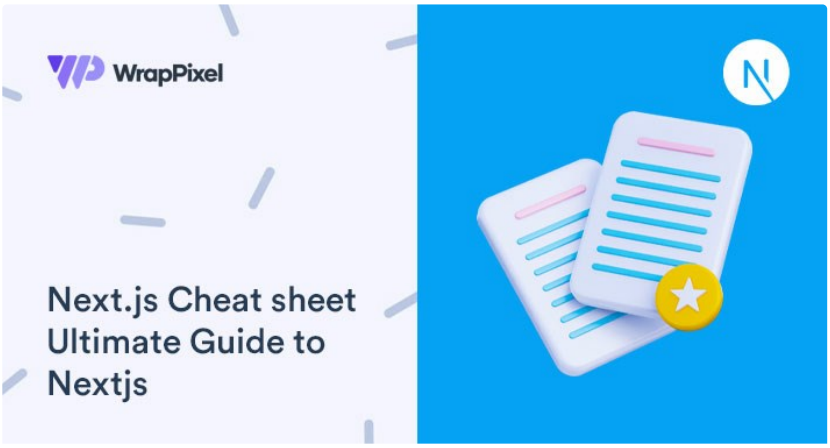
NextJs Cheat Sheet: Ultimate Guide to NextJs

By Sunil Joshi On July 12, 2023 Cheat Sheet, Nextjs



With ever so increasing rise in popularity of NextJs, it has become quite a standard to use it for server-side React web applications. So, that means the NextJs cheat sheet will be your main asset going forward.

It gets all the features of React.js – the JavaScript's UI library to build components and adds so many additional features that sometimes as a NextJs developer it's hard to catch up on the different code snippets, commands to run, packages to install.

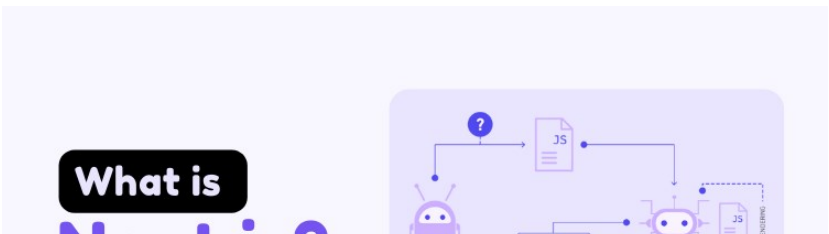
Whether you're a beginner or a seasoned pro, this cheat sheet will serve as a valuable resource. Now, let's explore the specifics that can greatly enhance your NextJs projects. This Nextjs Cheat sheet will help you if you are working on a built in NextJs Templates. This NextJs Cheat sheet will be solution to all your next js coding problems.

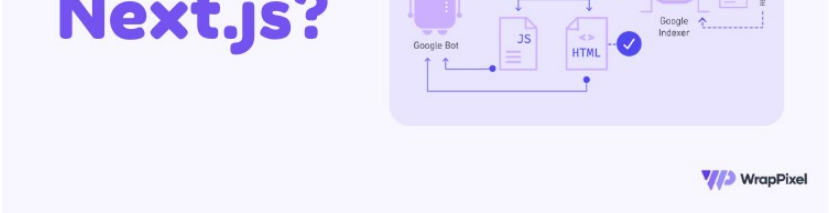
To solve this issue, we have created a to-the-point NextJs cheat sheet of all the features that will benefit all developers – be they a beginner or a pro. Let's dive in!

What is NextJs?

If are don't know what is nextjs. Then let me tell you. According to its definition on the official docs:

- **NextJs** is a flexible **React Framework** that gives you building blocks to create fast **Web Applications**.





Basically, it provides some of the crucial features and building blocks on top of a standard React.js application to make a modern website and apps.

Here are some of those important features (amongst others) that we will discuss:

Table of Contents

NextJs Cheat Sheet

Modernize Nuxt Js Admin Dashboard

⭐ 4.88 / 5.00

Categories

Given below is the NextJs cheat sheet that you can use in your next project.

Create App Using Nextjs Cheat Sheet

In this first step of the NextJs cheat sheet, we will create a NextJs app, the recommended process is to use the official create-next-app command which sets up all the necessary files, folders, and configuration automatically.

```
npx create-next-app@latest
# OR
yarn create next-app
```

Then run npm run dev or yarn dev to start the local development server on http://localhost:3000.

Alternatively, if you manually want to install NextJs, then first you should install next, react, and react-dom in your project as:

```
npm install next react react-dom
# OR
yarn add next react react-dom
```

Inside your *package.json* file, add the following scripts:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "lint": "next lint"
}
```

Use TypeScript, ESLint and npm.

```
npx create-next-app --typeScript --eslint --use-npm
```

Create Pages

To create a simple static page, under the pages directory, create a file named *demo.js* which exports a React component:

```
function Demo() {
  return <h1>Demo</h1>
}
export default Demo
```

This page will be available at http://localhost:3000/demo of your local environment.

App Router

A key component of Next.js has been file-system-based routing. We provided this illustration of building a route from a single React component in our

initial post:

```
// Pages Router

import React from 'react';
export default () => <h1>About us</h1>;
```

Nothing more needed to be configured. Place a file within the pages and click Next. The remainder would be handled by js router. We continue to adore the simplicity of the route. However, as the framework's popularity increased, so did the kinds of interfaces that programmers wanted to create with it.

Developers have requested better layout definition support, the ability to layer UI components as layouts, and increased flexibility when specifying the loading and error states.

The router must be the focal point of the entire framework. Page changes, data retrieval, caching, data modification and revalidation, streaming, content style, and more.

```
// New: App Router 
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

```
export default function Page() {
  return <h1>Hello, Next.js!</h1>;
}
```

```
// Pages Router

// This "global layout" wraps all routes. There's no way to
// compose other layout components, and you cannot fetch global
// data from this file.
export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />;
}
```

```
// New: App Router 
// The root layout is shared for the entire application
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
```

```
    </html>
  );
}
```

```
// Layouts can be nested and composed
export default function DashboardLayout({ children }) {
  return (
    <section>
      <h1>Dashboard</h1>
      {children}
    </section>
  );
}
```

```
// Pages Router

// This file allows you to customize the <html> and <body> tags
// for the server request, but adds framework-specific features
// rather than writing HTML elements.
import { Html, Head, Main, NextScript } from 'next/document';

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  );
}
```

```
// New: App Router 
// The root layout is shared for the entire application
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

A lot of other relevant feature requests for our routing system could be addressed at the same time as we had the chance to design a new file-system router. For instance:

In the past, _app.js could only import global stylesheets from outside npm packages (such as component libraries). It was not the best development

experience. Any CSS file can be imported (and placed) in any component using the App Router.

Previously, using your application was restricted until the full page was hydrated if you choose to use server-side rendering with Next.js (through getServerSideProps). We have refactored the architecture with the App Router to be fully integrated with React Suspense, allowing us to selectively hydrate portions of the page without preventing other UI components from being interactive.

Turbopack

Through *next dev –turbo* and soon your production builds (*next build –turbo*), Turbopack, our new bundler that we are testing and stabilising through Next.js, helps speed up local iterations while working on your Next.js application.

We have witnessed a steady increase in popularity since the alpha release of Next.js 13 as we have worked to fix problems and add support for lacking functionality. To get feedback and increase stability, we have been testing Turbopack on Vercel.com and with numerous Vercel clients running substantial Next.js websites. We appreciate the community's assistance in testing and alerting our team to bugs.

We are now prepared to advance into the beta phase after six months.

Webpack and Next.js currently have more features than Turbopack does. Support for those functionalities is being tracked in this issue. The majority of use cases ought to be supported today, though. In order to continue addressing bugs from greater adoption and get ready for stability in a future version, we are releasing this beta.

Turbopack's incremental engine and cache layer are being improved, and this will eventually speed up not only local development but also production builds. Keep an eye out for a future release of Next.js that will allow you to run *next build –turbo* for instant builds.

Use *next dev –turbo* to test out the Turbopack beta in Next.js 13.4.

File-Based Metadata API

By exporting a Metadata object from a layout or page, you can define metadata (such as *title*, *meta*, and *link* tags inside your HTML *head* element) using the new Metadata API that was introduced in Next.js 13.2.

```
// either Static metadata
export const metadata = {
  title: 'Home',
};
// Output:
// <head>
//  <title>Home</title>
```

```
// </head>

// or Dynamic metadata
export async function generateMetadata({ params, searchParams })
  const product = await getProduct(params.id);
  return { title: product.title };
}
// Output:
// <head>
//  <title>My Unique Product</title>
// </head>

export default function Page() {}
```

The Metadata API now supports new file conventions in addition to configuration-based metadata, making it simple to quickly alter your pages for better SEO and online sharing:

opengraph-image. (jpg | png | svg)
twitter-image. (jpg | png | svg)
favicon.ico icon. (ico | jpg | png | svg)
sitemap. (xml | js | jsx | ts | tsx)
robots. (txt | js | jsx | ts | tsx)
manifest. (json | js | jsx | ts | tsx)

Dynamic Open Graph Images

At the Next.js Conference, @vercel/og was put to the test by creating over 100,000 dynamic ticket pictures for each guest. We are thrilled to introduce dynamically generated images to all Next.js applications without the requirement for an external package thanks to widespread adoption among Vercel clients and over 900,000 downloads since the release.

To create images, you may now import ImageResponse from next/server:

```
import { ImageResponse } from 'next/server';

export const size = { width: 1200, height: 600 };
export const alt = 'About Acme';
export const contentType = 'image/png';
export const runtime = 'edge';

export default function og() {
  return new ImageResponse();
  // ...
}
```

Route Handlers and file-based Metadata are only two of the Next.js APIs that work well with *ImageResponse*. For instance, you can create Open Graph and Twitter pictures at build time or dynamically at request time using

ImageResponse in an *opengraph-image.tsx* file.

Static Export for App Router

Fully static exports are now supported by the Next.js App Router.

A static website or Single-Page Application (SPA) can be used as a starting point, and you can later upgrade to employ Next.js features that demand a server.

As part of the next build process, Next.js creates an HTML file for each route. A strict SPA can be broken down into separate HTML files with the help of Next.js to prevent extra JavaScript code from being loaded on the client-side, hence lowering the bundle size and enabling quicker page loads.

```
/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  output: 'export',
};

module.exports = nextConfig;
```

The *app* router's new features, like static Route Handlers, Open Graph images, and React Server Components, are compatible with Static Export.

Like traditional static-site generation, Server Components, for instance, will execute throughout the build and render the components into static HTML for the initial page load and a static payload for client movement across routes.

Before, you had to perform the next export before using Static Export in the *pages* directory. However, when output: '*export*' is set, the next build will produce an out directory thanks to the *next.config.js* option. The app router and pages directory can both be configured in the same way. Therefore, the subsequent export is no longer necessary.

Parallel Routes and Interception

Parallel Routes and Intercepting Routes are two brand-new dynamic conventions that Next.js 13.3 introduces to help you create complex routing scenarios. With the help of these features, you can display multiple pages in the same view, such as with intricate dashboards or modals.

You can simultaneously render one or more pages in the same view that can be accessed separately using Parallel Routes. It can also be used to render pages conditionally.

Named "slots" are used to build parallel routes. According to the @folder convention, slots are defined:

```
dashboard
```

```
├── @user
│   └── page.js
├── @team
│   └── page.js
├── layout.js
└── page.js
```

```
export default async function Layout({ children, user, team }) {
  const userType = getCurrentUserType();

  return (
    <>
      {userType === 'user' ? user : team}
      {children}
    </>
  );
}
```

The *@user* and *@team* parallel route slots (explicit) in the aforementioned example are conditionally produced based on your reasoning. The implicit route slot children does not require mapping to a *@folder*. Dashboard/page.js, for instance, is identical to Dashboard/*@children*/page.js.

By "masking" the browser URL, you can load a new route within the existing layout by intercepting routes. When the context of the current page must be preserved, such as when expanding a photo in a feed through a modal while the feed is kept in the modal's background, this is helpful.

Style Your NextJs app

There are many ways to style your apps in this step of the NextJs cheat sheet, some of the common methods are:

- **Global styling**: import the global *styles.css* in your *pages/_app.js* where these styles will apply to all pages and components in your app as:

```
import '../styles.css'
export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

- **Component-Level CSS**: NextJs supports CSS Modules where you can name the files as *[name].module.css* and then import it on a specific component. Here's an example:

```
// Button.module.css
.error {
  color: white;
  background-color: red;
}
```

```
// Button.jsx
import styles from './Button.module.css'
export function Button() {
  return (
    <button
      type="button"
      className={styles.error}
    >
      Cancel
    </button>
  )
}
```

- **Using SASS**: first, you need to install the SASS package on your NextJs app:

```
npm install --save-dev sass
```

Then you can configure the SASS compiler options in next.config.js file:

```
const path = require('path')
module.exports = {
  sassOptions: {
    includePaths: [path.join(__dirname, 'styles')],
  },
}
```

Optimize Images & Fonts

In this step of the NextJs cheat sheet, to optimize images you should use the built-in Image component. Install it in your project as:

```
import Image from 'next/image'
```

Then give it an src attribute as shown in the following example:

```
import Image from 'next/image'
const myLoader = ({ src, width, quality }) => {
  return `https://example.com/${src}?w=${width}&q=${quality || 7
}
const MyImage = (props) => {
  return (
    <Image
      loader={myLoader}
      src="me.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
```

```
}
```

Custom 404 Pages

To create custom 404 page, create a `404.js` file in the `pages` folder

```
export default function Custom404() {
  return <h1>404 - Page Not Found</h1>
}
```

You can also create a `500.js` file for the server error 500 page

SWR

It is a React Hooks library for remote data fetching on the client

You can use it inplace of `useEffect`

```
import useSWR from 'swr'

export default function Home() {
  const { data, error } = useSWR('api/user', fetch)

  if (error) return <div>failed to load</div>
  if (!data) return <div>loading...</div>

  return (
    <>
      {data.map((post) => (
        <h3 key={post.id}>{post.title}</h3>
      ))}
    </>
  )
}
```

Fetch Data

In the NextJs cheat sheet, there are many ways to fetch data from external sources to your NextJs app, here are some:

- **getServerSideProps**: if you want your app to pre-render a page on each request, then the getServerSideProps function should be exported as so:

```
export async function getServerSideProps(context) {
  return {
    props: {},
  }
}
```

Here's an example to fetch data at request time which pre-renders the result it gets back from the data source:

```
function Page({ data }) {
  // Code to render the `data`
}
export async function getServerSideProps() {
  const res = await fetch(`https://.../data`)
  const data = await res.json()
  return { props: { data } }
}
export default Page
```

- **getStaticPaths**: if you want to dynamically generate routes on your app alongside with getStaticProps then, getStaticPaths will pre-render all the paths provided to it as:

```
export async function getStaticPaths() {
  return {
    paths: [
      { params: { ... } }
    ],
    fallback: true
  };
}
```

- **getStaticProps**: if you want NextJs to generate a page at build time using the props passed to it, then getStaticProps should be exported as:

```
export async function getStaticProps(context) {
  return {
    props: {},
  }
}
```

Note that the props here must be passed to the page component as props. An example of its usage when you want the data to fetch from a CMS is as follows:

```
function BlogPosts ({ posts }) {
  return (
    <>
      {posts.map((post) => (
        <h1>{post.title}</h1>
        <p>{post.summary}</p>
      ))}
    </>
  )
}
export async function getStaticProps() {
```

```
  const res = await fetch('https://.../posts')
  const posts = await res.json()
  return {
    props: {
      posts,
    },
  }
}
export default BlogPosts
```

- **Incremental Static Regeneration(ISR)**: if you want to create or update existing static pages *after* you've built your site, then ISR allows you to statically generate on a per-page basis. This means that now you don't need to rebuild the entire site from scratch.

For this to work, you just need to add the revalidate prop to the getStaticProps method:

```
export async function getStaticProps(context) {
  return {
    props: {},
    revalidate: 5 // this means the request to re-generate the p
  }
}
```

- **Fetch data on client-side**: this can be done in two different ways — either via the useEffect hook as:

```
function User() {
  const [data, setData] = useState(null)
  const [isLoading, setLoading] = useState(false)
  useEffect(() => {
    setLoading(true)
    fetch('api/user-data')
      .then((res) => res.json())
      .then((data) => {
        setData(data)
        setLoading(false)
      })
  }, [])
  if (isLoading) return <p>Loading user data...</p>
  if (!data) return <p>No user data found</p>
  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

or via the SWR library which handles caching, revalidation, focus tracking, re-fetching on intervals, and more as:

```
import useSWR from 'swr'
const fetcher = (...args) => fetch(...args).then((res) => res.js
function User() {
  const { data, error } = useSWR('/api/user-data', fetcher)
  if (error) return <div>Failed to load user data</div>
  if (!data) return <div>Loading user data...</div>
  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Linting Your Code

You can use ESLint out-of-the-box for linting. Simply add the following script to the *package.json* file:

```
"scripts": {
  "lint": "next lint"
}
```

Now you can run npm run lint or yarn lint to start the linter. If you are using ESLint in a monorepo where NextJs isn't installed in your root directory, you just simply add the rootDir to your *.eslintrc* file:

```
{
  "extends": "next",
  "settings": {
    "next": {
      "rootDir": "packages/my-app/"
    }
  }
}
```

To use Prettier with ESLint settings, first install the dependency:

```
npm install --save-dev eslint-config-prettier
# OR
yarn add --dev eslint-config-prettier
```

And then add prettier to your existing ESLint configuration file:

```
{
  "extends": ["next", "prettier"]
```

```
}
```

TypeScript Support

TypeScript support is also one of the NextJs cheat sheets. To use TypeScript with NextJs when you start an app, use the create-next-app command along with the –ts or –typescript flag:

```
npx create-next-app@latest --ts
# or
yarn create next-app --typescript
```

This will spin up a new NextJs project with all the Typescript files and components without any extra configuration.

But if you want to integrate TypeScript in an existing project then, create a new *tsconfig.json* file at the root of the project directory. Then run npm run dev or yarn dev, with this NextJs will guide you through the installation of the required packages to finish setting up TypeScript integration.

Using Scripts

We are going to use scripts in this step of the NextJs cheat sheet, the native HTML <script> element is replaced by next/script component in NextJs. Here's an example of loading a Google Analytics script:

```
import Script from 'next/script'
export default function Home() {
  return (
    <>
      <Script src="https://www.google-analytics.com/analytics.js
    </>
  )
}
```

First, you import the script component:

```
import Script from 'next/script'
```

Next, there are different ways to handle scripts with this component which can be set by the strategy property with one of the following three values:

1. beforeInteractive: load the script before the page is interactive.
2. afterInteractive: load the script immediately after the page becomes interactive.
3. lazyOnload: load the script during idle time.

Here's an example:

```
<script
  src="https://cdn.jsdelivr.net/npm/cookieconsent@3/build/cookie
  strategy="beforeInteractive"
```

```
/>
```

App-Level Routing

In the next step of the NextJs cheat sheet, NextJs has a file-system-based router that works on the concept of pages. You can either have index routes like *pages/index.js* which map to / and *pages/blog/index.js* which map to /blog.

Or you can have nested routes where it supports nested files. For example, a file located on *pages/blog/my-post.js* will route to /blog/my-post.

For dynamic routes, you need to use the bracket syntax so that it can match named parameters. For example, *pages/[username]/settings.js* will map to /johndoe/settings.

The <Link> component is used to do client-side route transitions. First, you need to import it as:

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

Then, use it in a component:

```
import Link from 'next/link'
function Home() {
  return (
    <ul>
      <li>
        <Link href="/">
          <a>Home</a>
        </Link>
      </li>
      <li>
        <Link href="/about">
          <a>About Us</a>
        </Link>
      </li>
      <li>
        <Link href="/blog/hello-world">
          <a>Blog Post</a>
        </Link>
      </li>
    </ul>
  )
}
export default Home
```

For dynamic paths, you can use string interpolation to create the desired path:

```
import Link from 'next/link'
function Posts({ posts }) {
```

```
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${encodeURIComponent(post.slug)}`}>
            <a>{post.title}</a>
          </Link>
        </li>
      ))}
    </ul>
  )
}
export default Posts
```

API Routing

Any file inside the *pages/api* folder is mapped to /api/* which will be treated as an API endpoint. For example, to return a JSON response with an OK status code of 200, you can export a handler function with req and res passed as parameters:

```
export default function handler(req, res) {
  res.status(200).json({ name: 'John Doe' })
}
```

To handle different HTTP methods, you can use the req.method in your request handler:

```
export default function handler(req, res) {
  if (req.method === 'POST') {
    // Process a POST request
  } else {
    // Handle any other HTTP method
  }
}
```

Middlewares

1. To use middlewares in NextJs, first install the latest version of Next:

```
npm install next@latest
```

2. Then create a _middleware.ts file inside your */pages* directory
3. Finally, export a middleware function form the same file:

```
import type { NextFetchEvent, NextRequest } from 'next/server'
export function middleware(req: NextRequest, ev: NextFetchEvent)
  return new Response('Hello, world!')
}
```

For example, here is an example where middleware is used for logging:

```
import { NextRequest } from 'next/server'
// Regex for public files
const PUBLIC_FILE = /\.(.*)$/
export default function middleware(req: NextRequest) {
  // Only log for visited pages
  if (!PUBLIC_FILE.test(req.nextUrl.pathname)) {
    // We fire and forget this request to avoid blocking the req
    // and let logging occur in the background
    fetch('https://in.logtail.com', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${process.env.LOGTAIL_TOKEN}`,
      },
      body: JSON.stringify({
        message: 'Log from the edge',
        nested: {
          page: req.nextUrl.href,
          referrer: req.referrer,
          ua: req.ua?.ua,
          geo: req.geo,
        },
      }),
    })
  }
}
```

Authentication

There are many different ways to authenticate a user in a NextJs app. Some of the common ones are:

1. **Authenticating statically generated pages**: here, your page can render a loading state from the server after which it will fetch the user data from the client side. In the following example, the page renders a loading skeleton state and once the request is met, it shows the user's name:

```
import useUser from '../lib/useUser'
import Layout from '../components/Layout'
const Profile = () => {
  // Fetch the user client-side
  const { user } = useUser({ redirectTo: '/login' })
  // Server-render loading state
  if (!user || user.isLoggedIn === false) {
    return <Layout>Loading...</Layout>
```

```
  }
  // Once the user request finishes, show the user
  return (
    <Layout>
      <h1>Your Profile</h1>
      <pre>{JSON.stringify(user, null, 2)}</pre>
    </Layout>
  )
}
export default Profile
```

2. **Authenticating server-rendered pages**: here you need to export an async getServerSideProps() function from a page by which NextJs will pre-render this page on each request. Here's an example where if there is a session, then the user is returned as a prop to the Profile component:

```
import withSession from '../lib/session'
import Layout from '../components/Layout'
export const getServerSideProps = withSession(async function ({
  const { user } = req.session
  if (!user) {
    return {
      redirect: {
        destination: '/login',
        permanent: false,
      },
    }
  }
  return {
    props: { user },
  }
})
const Profile = ({ user }) => {
  // Show the user. No loading state is required
  return (
    <Layout>
      <h1>Your Profile</h1>
      <pre>{JSON.stringify(user, null, 2)}</pre>
    </Layout>
  )
}
export default Profile
```

3. **Authenticating with third-party providers**: for common authentication providers like Auth0, Firebase, Supabase, etc, you can take a look at the official GitHub repository for examples of how to set up and configure your own NextJs app.

Testing

The last step of the NextJs cheat sheet, Just like with authentication, testing can be done in a lot of different ways and with different testing tools. Here's how to setup testing with common tools:

1. **Testing with Cypress**: start off with the with-cypress example to quickly start a NextJs app with Cypress as:

```
npx create-next-app@latest --example with-cypress with-cypress-a
```

Or manually, install the cypress package:

```
npm install --save-dev cypress
```

Then add it to the scripts field of your *package.json* file:

```
"scripts": {
  ...
  "cypress": "cypress open",
}
```

Finally, run Cypress with the following command:

```
npm run cypress
```

To create a Cypress test file. Simply create a file under *cypress/integration/app.spec.js* as:

```
describe('Navigation', () => {
  it('should navigate to the about page', () => {
    // Start from the index page
    cy.visit('http://localhost:3000/')
    // Find a link with an href attribute containing "about" and
    cy.get('a[href*="about"]').click()
    // The new url should include "/about"
    cy.url().should('include', '/about')
    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About Page')
  })
})
```

2. **Testing with Jest and React Testing Library**: again you can use it quickly with the community provided with-jest example while you spin off a new Next project:

```
npx create-next-app@latest --example with-jest with-jest-app
```

Or manually, you can install Jest, React Testing Library, and Jest DOM packages:

```
npm install --save-dev jest @testing-library/react @testing-libr
```

Then create a *jest.config.js* file in the project's root directory:

```
const nextJest = require('next/jest')
const createJestConfig = nextJest({
  // Provide the path to your NextJs app to load next.config.js
  dir: './',
})
// Add any custom config to be passed to Jest
const customJestConfig = {
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
  // if using TypeScript with a baseUrl set to the root director
  moduleDirectories: ['node_modules', '<rootDir>/'],
  testEnvironment: 'jest-environment-jsdom',
}
// createJestConfig is exported this way to ensure that next/jes
module.exports = createJestConfig(customJestConfig)
// Add a test script to the package.json file:
"scripts": {
  ...
  "test": "jest --watch"
}
// Then create a Jest test file under __tests__/index.test.jsx a
import { render, screen } from '@testing-library/react'
import Home from '../pages/index'
describe('Home', () => {
  it('renders a heading', () => {
    render(<Home />)
    const heading = screen.getByRole('heading', {
      name: /welcome to next\.js!/i,
    })
    expect(heading).toBeInTheDocument()
  })
})
```

To run the test simply execute the npm run test command.

In this article, you got to know about the NextJs Cheat Sheet , and how it helps in making modern React-based websites and apps. Then you saw how to set up a NextJs project, how to fetch data, optimize assets, add TypeScript support, use linters, integrate testing tools, and more!

For those who prioritize built in Templates (Modernize free Nextjs Admin Template) over the custom templates. This cheat sheet is for you. Given below is the Picture Modernize free Nextjs Admin Template:

Conclusion

In conclusion, having a NextJs Cheat Sheet can greatly benefit developers working with this powerful framework. Providing a quick reference guide to the most commonly used features and functionalities enables developers to work more efficiently and effectively.

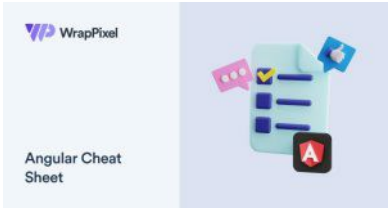
A NextJs Cheat Sheet serves as a handy tool for both beginners and experienced developers. It helps reduce the time spent searching for syntax or configuration details, allowing developers to focus more on writing clean and optimized code.

Additionally, a cheat sheet can also aid in improving code quality by promoting best practices and highlighting common pitfalls to avoid. It acts as a reliable companion ensuring developers follow the recommended guidelines while developing their NextJs applications.

Overall, having access to a comprehensive NextJs Cheat Sheet is invaluable for any developer looking to streamline their workflow and enhance their productivity. Whether you're building small projects or large-scale applications, this resource will undoubtedly prove to be an essential asset in your development toolkit.

Related Posts

Nuxt Cheat Sheet & Nuxt.js Essentials



Angular Cheat Sheet 2023

Author

Sunil Joshi
Co-Founder at WrapPixel

Sunil Joshi is an avid designer cum developer who is passionate about solving complex UX challenges across digital businesses. He is a trendsetter in the field of data visualization and dashboard design and has been lauded for his clean and minimalist design aesthetic. He co-founded WrapPixel, the design marketplace in 2016 with an aim to bring great design and clean code within easy reach of everyone.

Leave a Reply

Your email address will not be published. Required fields are marked *

Name*

Enter your full name here

Email*

Enter email here

Save my name, email, and website in this browser for the next time I comment.

Message

Write message here

Post Comment