Introduction to Next.js

A modern JavaScript library for building fullstack React applications

Next.js

- Next.js is a *fullstack* React framework for building React applications.
- Next.js allows you to work within a single React codebase across the front and back end of your application.
- Next.js can be rendered fully on the server, statically built, rendered fully on the client or (more typically) hybrid rendered on both the front and back end.
- Why it’s popular: performance, SEO (avoid SPA pitfalls), developer experience (single React application, blurs the distinction between FE/BE code).

Key Features

- **File-based routing** (pages directory, app directory in Next.js 13+).
- **Pre-rendering** (SSG – Static Site Generation, SSR – Server-Side Rendering).
- **API Routes** (creating backend endpoints inside Next.js).
- **Image Optimization** (`next/image`).
- **Built-in CSS & styling support** (CSS modules, Tailwind, styled-jsx, etc.).
- **Middleware** (running code before requests are processed).
- **App Router & React Server Components** (from Next.js 13+).

Rendering in Next.js

Next.js Rendering Strategies

Unlike traditional React applications which rely on frontend only rendering, Next.js supports a variety rendering strategies.

- **CSR (Client-Side Rendering)** – standard React behavior (single page application, SPA).
- **SSR (Server-Side Rendering)** – `getServerSideProps`.
- **SSG (Static Site Generation)** – `getStaticProps` + `getStaticPaths`.
- **ISR (Incremental Static Regeneration)** - statical generate site but conditional regenerate based on condition (like cache expiration)
- **PPR (Partial Prerendering)** – render on the server what is static, render on the client what is dynamic

Benefits of hybrid rendering

- **SEO** - Traditional client side rendering can result in worse SEO for a variety of reasons including:
  - Slow to first paint (when browser loads actual pixels to the canvas) - typically SPAs must make additional network requests before painting the canvas
  - Head elements are often not rendered properly in SPAs, which makes indexing metadata more difficulty
  - Client side routing can result in poorly indexed site pages
- Quicker page load - when static content is rendered on the server, it can be cached and delivered via a CDN for quick page loads.
- Hybrid rendering allows the static parts of the site to be rendered and cached on the server while the dynamic, app portions of the site can still use React on the client side and allow dynamic interactivity.

Hydration

A key concept in Next is that of *hydration*. Hydration is the process where a React application which was rendered initially on the server and served as HTML gets reattached to a React application (binding event handlers and managing data) on the client side.

```
import { useState, useEffect } from "react";

export default function Home({ initialCount }) {
  // This state will be hydrated on the client
  const [count, setCount] = useState(initialCount);

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

// SSR: fetch initial count
export async function getServerSideProps() {
  return {
```

Routing in Next.js

Routing Systems

Next.js uses files and directories to define application routes. There are two routing systems provided (pages and app router). Both systems provide ability to handle dynamic router paths (for instance for a dynamic ID).

Pages router is not the preferred way to develop new Next applications though many legacy libraries still rely on it. *You should use App Router for new applications*.

Pages Router

Pages router was the original way to route pages in Next and uses the `pages` directory to defined file based routes. I.e.: `pages/index.js` for the landing pages, `pages/post/[id].js` for a dynamic route handling requests for a specific blog post.

Pages router uses `getStaticProps` and `getServerSideProps` to fetch data during static site generation or server side requests. Pages router uses `getStaticPaths` to get data

Pages router uses three primary methods related to back end rendering:

- `getStaticProps` - used for get data when the site is built statically
- `getServerSideProps` - used when site is rendered initially on the server
- `getStaticPaths` - used to get data for generating *paths* for a static build. I.e. all blog post ids if you are building a statically generated blog.

Pages Router - `getStaticProps`

- runs at *build time*
- used to fetch data to statically build pages
- to use, simply export an `async` function called `getStaticProps` from the `/pages` route file.

```
export default function Home({ message }) {
  return <h1>{message}</h1>;
}

export async function getStaticProps() {
  // normally this would contain an async data request, i.e. to a headless CMS
  //  or for a static asset (i.e. a JSON or markdown file)
  return {
    props: {
      message: "Hello from getStaticProps!",
    },
  };
}
```

Pages Router - `getServerSideProps`

- runs at *runtime* during a page request
- used to fetch data for a dynamic request
- to use, simply export an `async` function called `getServerSideProps` from the `/pages` route file.

```
// pages/index.js
export default function Home({ time }) {
  return <h1>Current time: {time}</h1>;
}

export async function getServerSideProps() {
  // normally this would be a call to a dynamic changing data source
  //  database, API, key/value store, etc.
  return {
    props: {
      time: new Date().toISOString(),
    },
```

Pages Router - `getStaticProps`

- runs *pre-build* for *static builds* to get data needed to determine which pages should be pre-built.
- can *not* be used with `getServerSideProps`, only used for static build

Pages Router - `getStaticProps` example

```
// pages/posts/[id].js
import { useRouter } from "next/router";

export default function Post({ post }) {
  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </div>
  );
}

export async function getStaticPaths() {
  // Normally you'd fetch these from an API or database
  const posts = [{ id: "1" }, { id: "2" }, { id: "3" }];

  const paths = posts.map((post) => ({
    params: { id: post.id },
  }));

  return {
    paths,
  };
}

export async function getStaticProps({ params }) {
  // Simulated data fetch
  const posts = {
    1: { title: "First Post", content: "Hello World!" },
    2: { title: "Second Post", content: "Next.js is awesome." },
    3: { title: "Third Post", content: "Static generation rocks." },
  };

  const post = posts[params.id];

  if (!post) {
    return { notFound: true };
  }

  return {
    props: { post },
  };
}
```

App Router

App Router is the new and preferred routing system to use with Next.

App Router routing files reside in the `/app` directory, instead of the `pages` directory.

Instead of `getStaticProps` and `getServerSideProps`, to fetch data at buildtime, App Router uses async functions directly inside server components.

App Router also supports nested layouts, streaming and loading states.

App Router - Data Fetching

Instead of fetching data via a separate method (`getStaticProps` or `getServerSideProps`), App Router uses an async function directly in an async component.

```
export default async function Dashboard() {
  const res = await fetch("https://api.example.com/live-data", {
    cache: "no-store", // always fetch fresh data
  });
  const data = await res.json();

  return <pre>{JSON.stringify(data, null, 2)}</pre>;
}
```

App Router - Dynamic Routes

`generateStaticParams` is used instead of `getStaticPaths` to get data for static builds using App Router.

```tsx
// app/posts/[id]/page.tsx
export async function generateStaticParams() {
  const res = await fetch("https://api.example.com/posts");
  const posts = await res.json();

  return posts.map((post: any) => ({
    id: post.id.toString(),
  }));
}

export default async function Post({ params }: { params: { id: string } }) {
  const res = await fetch(`https://api.example.com/posts/${params.id}`);
  const post = await res.json();

  return <h1>{post.title}</h1>;
}
```

App Router - Caching Options

App router has various cache options to specify when data should be refetched.

App Router Option	Equivalent Pages Router Concept
`cache: 'no-store'`	`getServerSideProps`
`cache: 'force-cache'`	`getStaticProps`
`next: { revalidate: X }`	`getStaticProps` with `revalidate` (ISR)

Incremental Static Regeneration (ISR)

Next supports ISR to allow fine-grained control over page caching.

With ISR, you can specify a simple expiry how long a static built page should be served or custom custom validation logic. You can also invalidate caches.

For App Router Next apps, simply exporting a `revalidate` variable will define a page expiry.

```
interface Post {
  id: string
  title: string
  content: string
}

// Next.js will invalidate the cache when a
// request comes in, at most once every 60 seconds.
export const revalidate = 60

export async function generateStaticParams() {
  const posts: Post[] = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )
  return posts.map((post) => ({
    id: String(post.id),
  }))
}
```

React Server Components

App Router relies on Server Components to handle server side build and rendering.

Server Components are a new type of React components that allow React components to be built on the server, either at buildtime in CI/CD environment or during a live server request.

Server Components are typically used within the context of a framework (typically Next).

The following two slides show a comparison between a client side rendered component (CSR) and with the use of Server Component (SCR) solution in a React application to render Markdown content.

Example Implementation Without Server Component

```
// bundle.js
import marked from "marked"; // 35.9K (11.2K gzipped)
import sanitizeHtml from "sanitize-html"; // 206K (63.3K gzipped)

function Page({ page }) {
  const [content, setContent] = useState("");
  // NOTE: loads *after* first page render.
  useEffect(() => {
    fetch(`/api/content/${page}`).then((data) => {
      setContent(data.content);
    });
  }, [page]);

  return <div>{sanitizeHtml(marked(content))}</div>;
}
```

```
// api.js
app.get(`/api/content/:page`, async (req, res) => {
  const page = req.params.page;
```

Example Implementation with Server Component

```
import marked from "marked"; // Not included in bundle
import sanitizeHtml from "sanitize-html"; // Not included in bundle

async function Page({ page }) {
  // NOTE: loads *during* render, when the app is built.
  const content = await file.readFile(`${page}.md`);

  return <div>{sanitizeHtml(marked(content))}</div>;
}
```

Implementation Comparison 1/2

What are the tradeoffs to these solutions?

When might you use one solution over the other?

Implementation Comparison 2/2

Among the tradeoffs between CSR (client side rendered) and RSC (react server component) are:

- **Faster page load for RSC** - the CSR example requires two request/response cycles, one for the app, the second for the data that app needs. With RSC, both app and data are returned at the same time.
- **Smaller bundle size for RSC** - packages in the RSC component are not bundled with the app, resulting in a smaller bundle size.
- **Better SEO for RSC** - RSC is returning actual HTML which tends to perform much better for SEO purposes than a simple JS bundle.

In general RSC has advantages over CSR but, if you are building a true web app, like a dashboard, utility or game, where SEO is less important, CSR can be preferable to simplify development (only a single front end context).

Introducing Client-Side Interactivity into a RSC components

To add client-side interactivity, child components should use the "use client" directive.

```
// Client Component
"use client";

export default function Expandable({ children }) {
  const [expanded, setExpanded] = useState(false);
  return (
    <div>
      <button onClick={() => setExpanded(!expanded)}>Toggle</button>
      {expanded && children}
    </div>
  );
}
```

`<Suspense>`

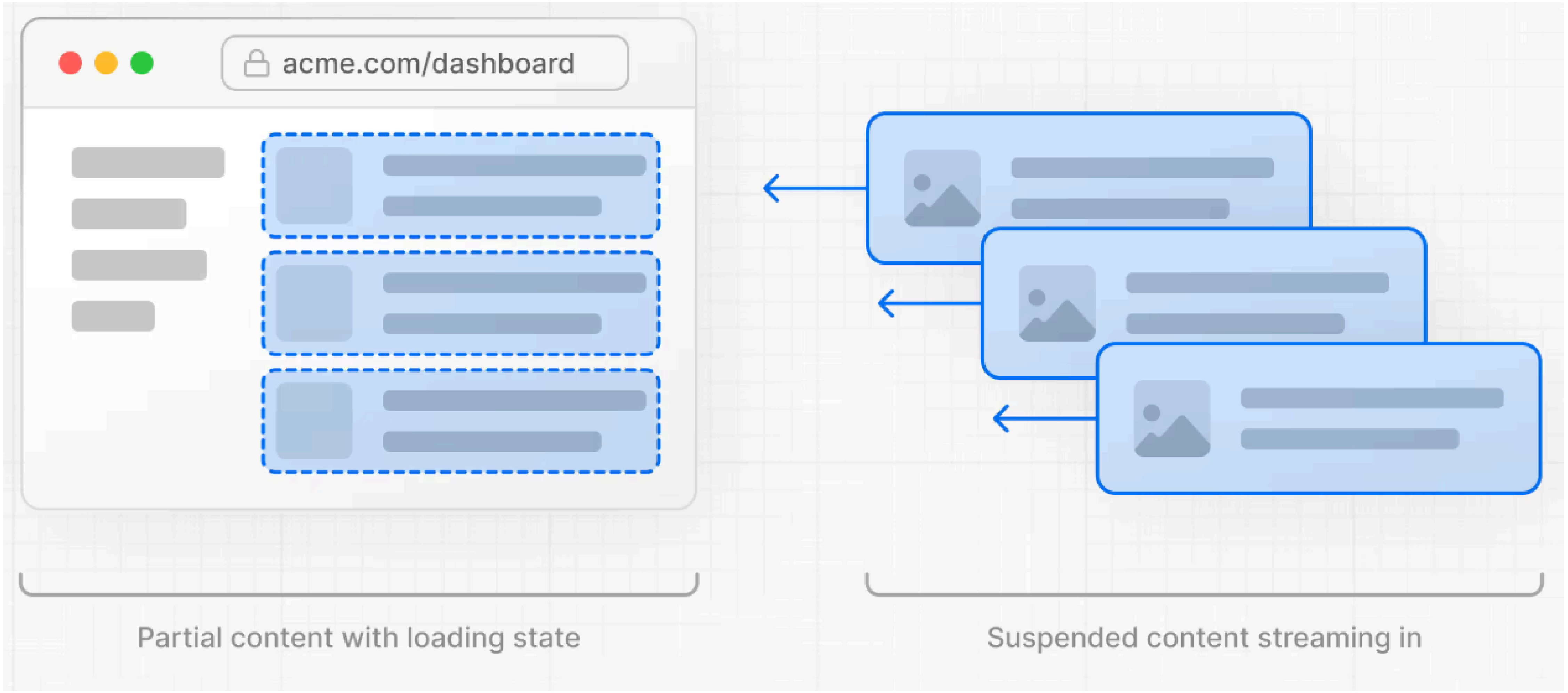
`<Suspense>` is a React feature that allows you to display a fallback Component if an asynchronous component is in a `pending`.

Suspense can be used for lazy loading components. In these cases, if, for instance conditional logic hides then shows a component, `Suspense` will be displayed while the lazy loaded component renders. Lazy components can also be code-split and the component itself may be fetched and rendered as needed.

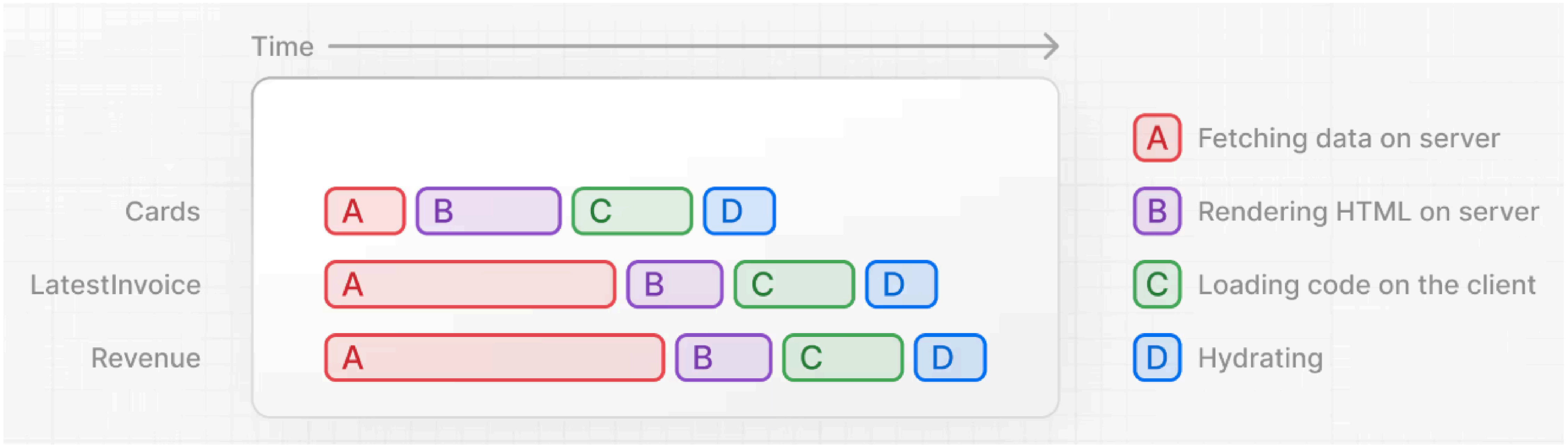
In the case of Server Components, `Suspense` can be used to allow **streaming components**. In this case, the page can initially load and paint while the server is still asynchronously streaming slower loading components wrapped in `<Suspense>`.

`Suspense` is a very powerful feature for hybrid React apps that improves performance, SEO and interactivity for web sites and applications. It prevents slow requests that only affect a section of the page from blocking the rendering of the rest of the page.

Suspense Visualized 1/2


acme.com/dashboard
Partial content with loading state
Suspended content streaming in

Suspense Visualized 2/2


Time
Cards
LatestInvoice
Revenue
A
B
C
D
A Fetching data on server
B Rendering HTML on server
C Loading code on the client
D Hydrating

Next Components

Next has a few components that provide various optimization over their raw HTML alternatives.

We will focus on three of the most important components:

- Image
- Link
- Script

Generally, these should always be used over their HTML equivalents.

Next/Image

The Next Image component provides optimizations, resizing images, delivering scalable alternatives and building these to the CDN for efficient delivery to the client.

Import:

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

Use:

```
<Image src="/hero.png" alt="Hero banner" width={1200} height={600} priority />
```

Next/Link

The Next Link component provides optimized client-side routing in Next applications. Without using the Link component, navigation can result in very inefficient full-page refreshes while navigating through an application.

Import:

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

Use:

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

Next/Script

The Next Script component allows you to include third party scripts within your application and specify when those scripts are loaded.

Example GA (Google Analytics) script inclusion:

```
import Script from "next/script";

export default function Home() {
  return (
    <div>
      <h1>My Page</h1>

      {/* Loads Google Analytics after the page is interactive */}
      <Script
        src="https://www.googletagmanager.com/gtag/js?id=GA_MEASUREMENT_ID"
        strategy="afterInteractive"
      />

      <Script id="google-analytics" strategy="afterInteractive">
        {`
          window.dataLayer = window.dataLayer || [];
          function gtag(){dataLayer.push(arguments);}
          gtag('js', new Date());
```

Getting Started

To get started with Next, run the following:

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

Select your configuration options. Make sure to enable TS, use App Router. Otherwise, the defaults are fine.

```
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 your code inside a `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to use Turbopack? (recommended) No / Yes
Would you like to customize the import alias (`@/*` by default)? No / Yes
What import alias would you like configured? @/*
```

After installation, `cd` to the new directory, and run `npm run dev`.

Deployment - TK

Summary - TK