Table of Contents

[NEXT JS 2](#_Toc132705662)

[KEY FEATURES 2](#_Toc132705663)

[CREATING NEXT APP 2](#_Toc132705664)

[FILE BASED ROUTING 2](#_Toc132705665)

[ACCESSING ROUTES 3](#_Toc132705666)

[SLUGS 4](#_Toc132705667)

[ROUTING - LINKS AND NAVIGATION 5](#_Toc132705668)

[PRERENDERING AND DATA FETCHING 6](#_Toc132705669)

[STYLING 6](#_Toc132705670)

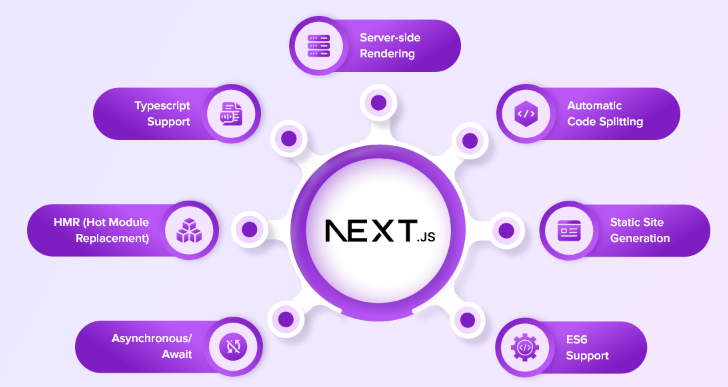
[STATIC GENERATION 7](#_Toc132705671)

[getServerStaticProps() 22](#_Toc132705672)

[API ROUTES 23](#_Toc132705673)

# NEXT JS

## KEY FEATURES

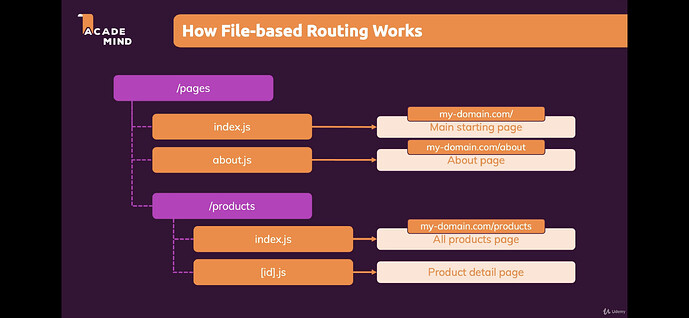


* **SERVER-SIDE PRE-RENDERING**
* **FILE BASED ROUTING**

## CREATING NEXT APP

|  |  |
| --- | --- |
| CREATING A NEXT APP | npx create-next-app |
|  | * pages - This folder is used for file based routing |
|  |  |

## FILE BASED ROUTING



|  |  |
| --- | --- |
|  | File Based Routing (Static Paths)   1. index.js - <http://localhost:3000/> 2. about.js- <http://localhost:3000/about> 3. /portfolio/index.js - <http://localhost:3000/portfolio> 4. /portfolio/list.js - <http://localhost:3000/portfolio/list> |
|  | File Based Routing (Dynamic Paths)   * /portfolio/[portfolio].js - [http://localhost:3000/portfolio/<anything](http://localhost:3000/portfolio/%3canything)> |

### ACCESSING ROUTES

|  |  |
| --- | --- |
| import { useRouter } from "next/router";  export default function PortfolioProject() {  const router = useRouter();  console.log("portfolio query", router.query['portfolio']);  console.log("hello query string", router.query['hello']);  return <h1>Portfolio Project Page</h1>;  }  URL : <http://localhost:3000/portfolio/1?hello=word> |  |
| * URL :<http://localhost:3000/clients/12> * Note – [id] is a folder |  |

|  |  |
| --- | --- |
| import { useRouter } from "next/router";  export default function ClientProjects() {  const router = useRouter();  return <h1>ClientProjects {router.query['clientproject']}</h1>;  }  F***older Structure***  pages  --[id]  --[clientproject]  Index.js | * URL -- <http://localhost:3000/clients/1/p1> |

### SLUGS

|  |  |  |
| --- | --- | --- |
|  | URL - <http://localhost:3000/blogs/sports/2022?type=new> | import { useRouter } from "next/router";  export default function Blog() {  const router = useRouter();  console.log(router.query);  return <h1>Blog Page</h1>;  } |

* The path parameters are available as an array in route.query. These slugs take variable path params

|  |  |
| --- | --- |
| <http://localhost:3000/blogs/sports/2022?type=new> | Fetch “new” “sports” blogs of “2022” |
| <http://localhost:3000/blogs/sports/> | Fetch “all” “sports” blogs |

### ROUTING - LINKS AND NAVIGATION

|  |  |
| --- | --- |
| function Home() {  return (  <ul>  <li>  <Link href="/">Home</Link>  </li>  <li>  <Link href="/about">About Us</Link>  </li>  <li>  <Link href="/blog/hello-world">Blog Post</Link>  </li>  </ul>  )  }  export default Home | * The Next.js router allows us to do client-side route transitions between pages, similar to a single-page application. * **ADVANCED OPTIMIZATION - Any <Link /> in the viewport (initially or through scroll) will be prefetched by default** (including the corresponding data) for pages using [Static Generation](https://nextjs.org/docs/basic-features/data-fetching/get-static-props). The corresponding data for [server-rendered](https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props) routes is fetched only when the <Link /> is clicked. |

#### DYNAMIC ROUTING

|  |  |
| --- | --- |
| METHOD – 1  File Path - /clients/index.js  import Link from "next/link";  export default function ClientDetails() {  const clients =[  { id:'c1', name:'Client1'},  { id:'c2', name:'Client2'}  ];  return <div> <h1>ClientDetails Page</h1>  <ul>  {clients.map(client =>(  <li key={client.id}>  **<Link href={`/clients/${client.id}`}> {client.name}</Link>**  </li>  ))}  </ul>  </div>;  } | URL - <http://localhost:3000/clients>    File Path - /client/[id]/index.js  import { useRouter } from "next/router";  export default function Client() {  const router = useRouter();  return <div>  <h1>{router.query['id']}</h1>  </div>;  } |
| METHOD -2  export default function ClientDetails() {  const clients =[  { id:'c1', name:'Client1'},{ id:'c2', name:'Client2'}  ];  return <div> <h1>ClientDetails Page</h1>  <ul>  {clients.map(client =>(  <li key={client.id}>  **<Link href={**  **{ pathname : "/clients/[id]",**  **query: {id:client.id}**  **}}> {client.name}</Link>**  </li>  ))}  </ul>  </div>;  } | |

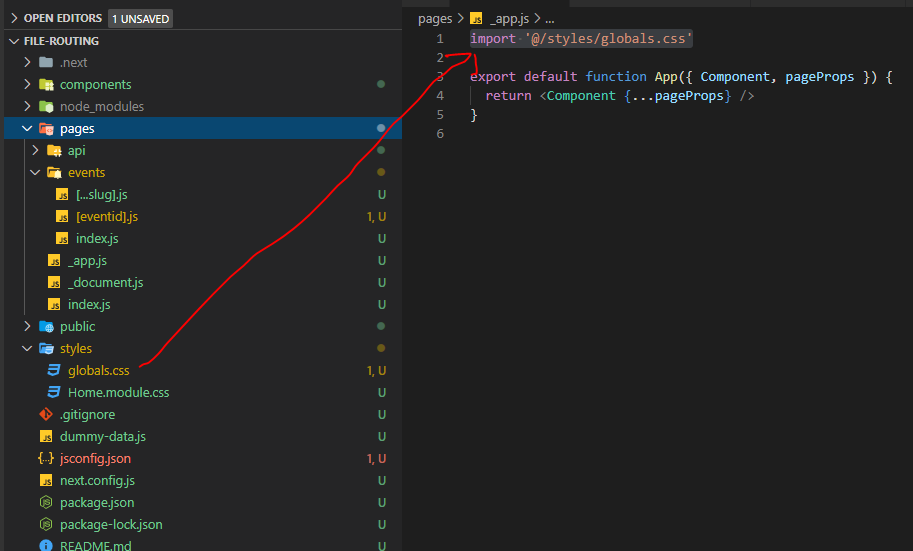
#### NAVIGATING PROGRAMMATICALLY

|  |  |
| --- | --- |
| import { useRouter } from "next/router";  export default function Client() {  const router = useRouter();  const clientId = router.query['id'];  const loadProjectHandler = ()=>{  router.push({  pathname:'/clients/[id]/[clientprojectid]',  query: {  id: 'c1',  clientprojectid:'projectA'  }  })  }  return <div>  <h1>{clientId}</h1>  <button onClick={loadProjectHandler}>Navigate to Project A</button>  </div>;  } |  |

## STYLING

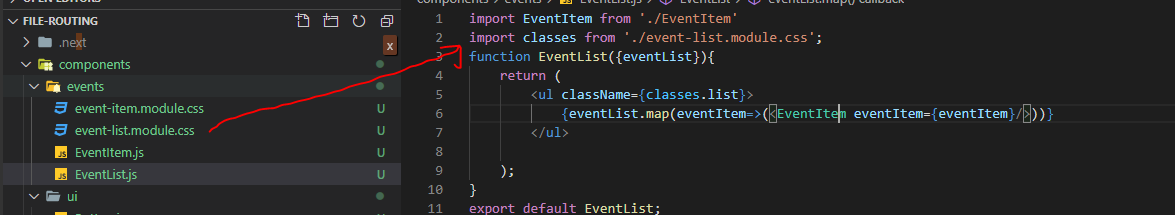
Styling can be added at global level or scoped with the component

* GLOBAL CSS – The global css can be added in **\_app.js**



|  |  |
| --- | --- |
| EXAMPLE | import EventItem from './EventItem'  import classes from './event-list.module.css';  function EventList({eventList}){  return (  <ul **className={classes.list}**>  {eventList.map(eventItem=>(<EventItem eventItem={eventItem}/>))}  </ul>  );  }  export default EventList; |

* SCOPED CSS (Component Level)



## PRERENDERING AND DATA FETCHING

Data fetching in Next.js allows us to render our content in different ways, depending on application's use case. These include ***pre-rendering with Server-side Rendering or Static Generation and updating or creating content at runtime with Incremental Static Regeneration.***

1. **STATIC GENERATION** – All the pages are pre generated during the build time
2. **SERVER SIDE RENDERING** – Pages are created just in time when it is request from the server.

### STATIC GENERATION

* Pages are pre-generated during the build time itself. These pages can be cached on CDN level.
* Note – In the static generated page are hydrated after rendering – and the React code takes over for any further dynamic functionalities (known as Hydration)

#### getStaticProps()

* If we export a function called getStaticProps (Static Site Generation) from a page, Next.js will pre-render this page at build time using the props returned by getStaticProps.

export async function getStaticProps(context) {

return {

props: {}, // will be passed to the page component as props

}

}

|  |
| --- |
| **Question -** [**When should I use getStaticProps?**](https://nextjs.org/docs/basic-features/data-fetching/get-static-props#when-should-i-use-getstaticprops)  We should use getStaticProps if:   * The data required to render the page is available at build time ahead of a user’s request * The data comes from a headless CMS * The page must be pre-rendered (for SEO) and be very fast — getStaticProps generates HTML and JSON files, both of which can be cached by a CDN for performance * The data can be publicly cached (not user-specific). This condition can be bypassed in certain specific situation by using a Middleware to rewrite the path. |
| **Question -** [**When does getStaticProps run**](https://nextjs.org/docs/basic-features/data-fetching/get-static-props#when-does-getstaticprops-run) **?**  **getStaticProps always runs on the server and never on the client.**   * getStaticProps always runs during ***next build*** * getStaticProps runs in the background when using [**fallback: true**](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-true) * getStaticProps is called before initial render when using [**fallback: blocking**](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-blocking) * getStaticProps runs in the background when using revalidate * getStaticProps runs on-demand in the background when using [revalidate()](https://nextjs.org/docs/basic-features/data-fetching/incremental-static-regeneration#on-demand-revalidation) * Note - getStaticProps does not have access to the incoming request (such as query parameters or HTTP headers) as it generates static HTML. If we need access to the request for our page, consider using [Middleware](https://nextjs.org/docs/middleware) in addition to getStaticProps. |

EXAMPLE : [Using getStaticProps to fetch data from a CMS](https://nextjs.org/docs/basic-features/data-fetching/get-static-props#using-getstaticprops-to-fetch-data-from-a-cms)

The following example shows how you can fetch a list of blog posts from a CMS.

// posts will be populated at build time by getStaticProps()

function Blog({ posts }) {

return (

<ul>

{posts.map((post) => (

<li>{post.title}</li>

))}

</ul>

)

}

// This function gets called at build time on server-side.

// It won't be called on client-side, so you can even do

// direct database queries.

export async function getStaticProps() {

// Call an external API endpoint to get posts.

// You can use any data fetching library

const res = await fetch('https://.../posts')

const posts = await res.json()

// By returning { props: { posts } }, the Blog component

// will receive `posts` as a prop at build time

return {

props: {

posts,

},

}

}

export default Blog

#### [WRITE SERVER-SIDE CODE DIRECTLY](https://nextjs.org/docs/basic-features/data-fetching/get-static-props#write-server-side-code-directly)

* As getStaticProps runs only on the server-side, it will never run on the client-side. It won’t even be included in the JS bundle for the browser, so we can write direct database queries without them being sent to browsers.
* This means that instead of fetching an **API route** from getStaticProps (that itself fetches data from an external source), we can write the server-side code directly in getStaticProps.

EXAMPLE

|  |  |
| --- | --- |
| **import fs from "fs/promises";**  **import path from "path";**  export default function Home(props) {  const { products } = props;  return (  <div>  <h1>Product List</h1>  <ul>  {products.map((product) => ( <li key={product.id}>{product.title}</li> ))}  </ul>  </div>  );  }  export async function getStaticProps() {  const filePath = path.join(process.cwd(), "data", "dummy-backend.json");  const fileData = await fs.readFile(filePath);  const data = JSON.parse(fileData);  return {  **props: {**  **products: data.products,**  **},**  };  } |  |

|  |
| --- |
|  |
| * The build output shows the legends – to show how files are generated SSG, static or Server Side generated |
| * Next JS keep the props(dynamic data) in the source code – which can then be used rehydration (once React kicks in for dynamic behavior on the page) |

EXAMPLE

* For example - An API route is used to fetch some data from a CMS. That API route is then called directly from getStaticProps. This produces an additional call, reducing performance. Instead, the logic for fetching the data from the CMS can be shared by using a lib/ directory. Then it can be shared with getStaticProps.

// lib/load-posts.js

// The following function is shared

// with getStaticProps and API routes

// from a `lib/` directory

export async function loadPosts() {

// Call an external API endpoint to get posts

const res = await fetch('https://.../posts/')

const data = await res.json()

return data

}

// pages/blog.js

import { loadPosts } from '../lib/load-posts'

// This function runs only on the server side

export async function getStaticProps() {

// Instead of fetching your `/api` route you can call the same

// function directly in `getStaticProps`

const posts = await loadPosts()

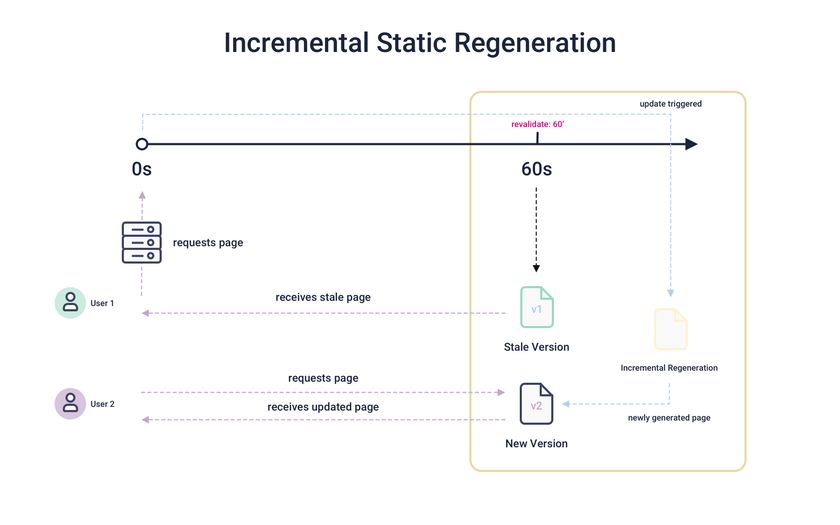
// Props returned will be passed to the page component

return { props: { posts } }

}

#### INCREMENTAL STATIC GENERATION

* The static generation of HTML files happens when the application is built (***npm run build***)
* Problem with static generation is – if the data changes (backend)- it will never show up in the view as page are generated during build. To get the updated data – we must build the application again
* Hence for the application, where the data is updating frequently - We must make use of ***Incremental Static Generation(ISR)* feature of NextJS**



* In ISR, the page statically generated once at build time, but that it's continuously updated even after deployment without re-deploying it.
* We can configure NextJs – to tell NextJS- in how much interval the pages must be regenerated. For example – if we configure it for 60 sec - That means that if a request is made for a certain page -let's say less than 60 seconds since it was last re-generated, the existing page would be served to the visitor. But if it's past those 60 seconds, then this page would be pre-generated on the server instead.
* During development (***npm run dev***), the page will be re-generated for every request, no matter what we configure in revalidate. So, with the development server, we will always see the latest page with the latest data.

|  |
| --- |
| export async function getStaticProps() {  console.log("(Re) - Generating")  const filePath = path.join(process.cwd(), "data", "dummy-backend.json");  const fileData = await fs.readFile(filePath);  const data = JSON.parse(fileData);  return {  props: {  products: data.products,  },  revalidate:5  };  } |

#### getStaticProps() CONFIGURATION

|  |  |
| --- | --- |
| export async function getStaticProps() {  ……  return {  props: {  ……  },  notFound:true  };  } | * This will return 404 Page (not found) page. * Use Case – We can this property to true in case no data available for the current page |
| export async function getStaticProps() {  …..  return {  props: {  ……  },  redirect:{  destination: '/nodata'  }  };  } | * This will redirect to a given route. |

#### DYNAMIC PARAMETERS

|  |  |
| --- | --- |
|  | * URL: <http://localhost:3002/p1> * For dynamic parameters – NextJS does not create a Statically generated pages because NextJS will have no clue how many pages has to be generated. * If a page has [Dynamic Routes](https://nextjs.org/docs/routing/dynamic-routes) and uses getStaticProps, it needs to define a list of paths to be statically generated. * When we export a function called getStaticPaths (Static Site Generation) from a page that uses dynamic routes, Next.js will statically pre-render all the paths specified by getStaticPaths. Refer below example   **export async function getStaticPaths()**   * getStaticPaths will only run during build in production, it will not be called during runtime * In development (next dev), getStaticPaths will be called on every request. |

|  |  |
| --- | --- |
| import fs from "fs/promises";  import path from "path";  export default function ProductDetailPage(props){  const {loadedProduct} = props;  return(  <div>  <h1>{loadedProduct.title}</h1>  <p>{loadedProduct.description}</p>  </div>  )  }  export async function getStaticProps(context) {  const {params} = context  const productId = params.pid;  const filePath = path.join(process.cwd(), "data", "dummy-backend.json");  const fileData = await fs.readFile(filePath);  const data = JSON.parse(fileData);  const product = data.products.find(p => p.id == productId);  return {  props: {  loadedProduct: product,  },  };  }  **export async function getStaticPaths() {**  **return {**  **paths: [**  **{ params: { pid: "p1" } },**  **{ params: { pid: "p2" } },**  **{ params: { pid: "p3" } },**  **],**  **fallback:false**  **};**  **}** | STATICALLY GENERATED PAGE BASED ON THE LIST OF PATHS |

1. [**When should I use getStaticPaths?**](https://nextjs.org/docs/basic-features/data-fetching/get-static-paths#when-should-i-use-getstaticpaths)

You should use getStaticPaths if you’re statically pre-rendering pages that use dynamic routes and:

* + The data comes from a headless CMS
  + The data comes from a database
  + The data comes from the filesystem
  + The data can be publicly cached (not user-specific)
  + The page must be pre-rendered (for SEO) and be very fast — getStaticProps generates HTML and JSON files, both of which can be cached by a CDN for performance

1. [**When does getStaticPaths run**](https://nextjs.org/docs/basic-features/data-fetching/get-static-paths#when-does-getstaticpaths-run) **?**
   * getStaticPaths will only run during build in production, it will not be called during runtime.

|  |  |
| --- | --- |
|  | **npm run build**   * The pages (p1, p2, p3) are pre generated during the build. These are the paths provided in getStaticPaths() |

1. [**How does getStaticProps run with regards to getStaticPaths**](https://nextjs.org/docs/basic-features/data-fetching/get-static-paths#how-does-getstaticprops-run-with-regards-to-getstaticpaths)
   * getStaticProps runs during next build for any paths returned during build
   * getStaticProps runs in the background when using fallback: true
   * getStaticProps is called before initial render when using fallback: blocking
2. [**Where can I use getStaticPaths**](https://nextjs.org/docs/basic-features/data-fetching/get-static-paths#where-can-i-use-getstaticpaths)
   * getStaticPaths **must** be used with getStaticProps
   * You **cannot** use getStaticPaths with [getServerSideProps](https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props)
   * You can export getStaticPaths from a [Dynamic Route](https://nextjs.org/docs/routing/dynamic-routes) that also uses getStaticProps
   * You **cannot** export getStaticPaths from non-page file (e.g. your components folder)
   * You must export getStaticPaths as a standalone function, and not a property of the page component

#### getStaticPaths()

When exporting a function called getStaticPaths from a page that uses [Dynamic Routes](https://nextjs.org/docs/routing/dynamic-routes), Next.js will statically pre-render all the paths specified by getStaticPaths.

export async function getStaticPaths() {

return {

paths: [

{ params: { ... } } // See the "paths" section below

],

fallback: true, false or "blocking" // See the "fallback" section below

};

}

##### [getStaticPaths return values](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#getstaticpaths-return-values)

The getStaticPaths function should return an object with the following **required** properties:

###### [paths](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#paths)

The paths key determines which paths will be pre-rendered. For example, suppose that you have a page that uses [Dynamic Routes](https://nextjs.org/docs/routing/dynamic-routes) named pages/posts/[id].js. If you export getStaticPaths from this page and return the following for paths:

return {

paths: [

{ params: { id: '1' }},

{

params: { id: '2' },

// with i18n configured the locale for the path can be returned as well

locale: "en",

},

],

fallback: ...

}

Then, Next.js will statically generate */posts/1 and /posts/2* during **next build** using the page component in pages/posts/[id].js.

The value for each params object must match the parameters used in the page name:

* If the page name is pages/posts/[postId]/[commentId], then params should contain postId and commentId.
* If the page name uses [catch-all routes](https://nextjs.org/docs/routing/dynamic-routes#catch-all-routes) like pages/[...slug], then params should contain slug (which is an array). If this array is ['hello', 'world'], then Next.js will statically generate the page at /hello/world.
* If the page uses an [optional catch-all route](https://nextjs.org/docs/routing/dynamic-routes#optional-catch-all-routes), use null, [], undefined or false to render the root-most route. For example, if you supply slug: false for pages/[[...slug]], Next.js will statically generate the page /.
* The params strings are **case-sensitive** and ideally should be normalized to ensure the paths are generated correctly. For example, if WoRLD is returned for a param it will only match if WoRLD is the actual path visited, not world or World.
* Separate of the params object a locale field can be returned when [i18n is configured](https://nextjs.org/docs/advanced-features/i18n-routing), which configures the locale for the path being generated.

DYNAMICALLLY LOADING PATHS

import fs from "fs/promises";

import path from "path";

export default function ProductDetailPage(props){

    const {loadedProduct} = props;

     return(

        <div>

            <h1>{loadedProduct.title}</h1>

            <p>{loadedProduct.description}</p>

        </div>)

}

async function getData(){

    const filePath = path.join(process.cwd(), "data", "dummy-backend.json");

    const fileData = await fs.readFile(filePath);

    const data = JSON.parse(fileData);

    return data;

}

export async function getStaticProps(context) {

    const {params} = context

    const productId = params.pid;

    const data = await getData();

    const product = data.products.find(p => p.id == productId);

    return {

      props: {

        loadedProduct: product,

      },

    };

  }

export async function getStaticPaths() {

    const data = await getData();

    const ids = data.products.map(d=> d.id);

    const pathParams =  ids.map(id=>({params:{pid:id}}));

    return {

      paths: pathParams,

      fallback:'blocking'

    };

  }

###### FALLBACK

<https://dev.to/tomdohnal/blocking-fallback-for-getstaticpaths-new-next-js-10-feature-1727>

[fallback: false](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-false)

* **If fallback is false, then any paths not returned by getStaticPaths will result in a 404 page.**
* When next build is run, Next.js will check if getStaticPaths returned fallback: false, it will then build **only** the paths returned by getStaticPaths. This option is useful if we have a small number of paths to create, or new page data is not added often. If you find that you need to add more paths, and you have fallback: false, you will need to run next build again so that the new paths can be generated.
* The following example pre-renders one blog post per page called pages/posts/[id].js. The list of blog posts will be fetched from a CMS and returned by getStaticPaths. Then, for each page, it fetches the post data from a CMS using [getStaticProps](https://nextjs.org/docs/api-reference/data-fetching/get-static-props).

// pages/posts/[id].js

function Post({ post }) {

// Render post...

}

// This function gets called at build time

export async function getStaticPaths() {

// Call an external API endpoint to get posts

const res = await fetch('https://.../posts')

const posts = await res.json()

// Get the paths we want to pre-render based on posts

const paths = posts.map((post) => ({

params: { id: post.id },

}))

// We'll pre-render only these paths at build time.

// { fallback: false } means other routes should 404.

return { paths, fallback: false }

}

// This also gets called at build time

export async function getStaticProps({ params }) {

// params contains the post `id`.

// If the route is like /posts/1, then params.id is 1

const res = await fetch(`https://.../posts/${params.id}`)

const post = await res.json()

// Pass post data to the page via props

return { props: { post } }

}

export default Post

[fallback: true](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-true)

If fallback is **true**, then the behavior of getStaticProps changes in the following ways:

* The paths returned from getStaticPaths will be rendered to HTML at build time by getStaticProps.
* **The paths that have not been generated at build time will not result in a 404 page. Instead, Next.js will serve a**[**“fallback”**](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-pages)**version of the page on the first request to such a path**.

**EXAMPLE**

|  |
| --- |
| import fs from "fs/promises";  import path from "path";  export default function ProductDetailPage(props){      const {loadedProduct} = props;      if(!loadedProduct){          return(<div>Loading..</div>)      }      return(          <div>              <h1>{loadedProduct.title}</h1>              <p>{loadedProduct.description}</p>          </div>      )  }  export async function getStaticProps(context) {      const {params} = context      const productId  = params.pid;      console.log("(Re) - Generating")      const filePath = path.join(process.cwd(), "data", "dummy-backend.json");      const fileData = await fs.readFile(filePath);      const data = JSON.parse(fileData);      const product = data.products.find(p => p.id == productId);      return {        props: {          loadedProduct: product,        },      };    }    export async function getStaticPaths() {      return {        paths: [          { params: { pid: "p1" } },          { params: { pid: "p2" } },          { params: { pid: "p3" } },        ],        fallback:true      };    } |

* When a page with fallback: true is navigated to through next/link or next/router (client-side) Next.js will not serve a fallback and instead the page will behave as [fallback: 'blocking'](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-blocking).
* In the background, Next.js will statically generate the requested path HTML and JSON. This includes running getStaticProps.
* When complete, the browser receives the JSON for the generated path. This will be used to automatically render the page with the required props. From the user’s perspective, the page will be swapped from the fallback page to the full page.
* At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, like other pages pre-rendered at build time.
* **Note: fallback: true is not supported when using**[**output: 'export'**](https://nextjs.org/docs/advanced-features/static-html-export)**.**

[**When is fallback: true useful?**](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#when-is-fallback-true-useful)

* fallback: true is useful if your app has a very large number of static pages that depend on data (such as a very large e-commerce site). If you want to pre-render all product pages, the builds would take a very long time.
* **Instead, you may statically generate a small subset of pages and use fallback: true for the rest**. When someone requests a page that is not generated yet, the user will see the page with a loading indicator or skeleton component.
* Shortly after, getStaticProps finishes and the page will be rendered with the requested data. From now on, everyone who requests the same page will get the statically pre-rendered page.
* This ensures that users always have a fast experience while preserving fast builds and the benefits of Static Generation.
* fallback: true will not update generated pages, for that we have to implement [Incremental Static Regeneration](https://nextjs.org/docs/basic-features/data-fetching/incremental-static-regeneration).

[fallback: 'blocking'](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-blocking)

**If fallback is 'blocking', new paths not returned by getStaticPaths will wait for the HTML to be generated, identical to SSR (hence why blocking), and then be cached for future requests so it only happens once per path.**

getStaticProps will behave as follows:

* The paths returned from getStaticPaths will be rendered to HTML at build time by getStaticProps.
* **The paths that have not been generated at build time will not result in a 404 page.** Instead, Next.js will SSR on the first request and return the generated HTML. i.e Next JS will wait for the HTML to be generated.
* When complete, the browser receives the HTML for the generated path. From the user’s perspective, it will transition from "the browser is requesting the page" to "the full page is loaded". There is no flash of loading/fallback state.
* At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, like other pages pre-rendered at build time.
* fallback: 'blocking' will not update generated pages by default. To update generated pages, use [Incremental Static Regeneration](https://nextjs.org/docs/basic-features/data-fetching/incremental-static-regeneration) in conjunction with fallback: 'blocking'.
* **Note:** fallback: 'blocking' is not supported when using [output: 'export'](https://nextjs.org/docs/advanced-features/static-html-export).

EXAMPLE

import fs from "fs/promises";

import path from "path";

export default function ProductDetailPage(props){

    const {loadedProduct} = props;

    return(

        <div>

            <h1>{loadedProduct.title}</h1>

            <p>{loadedProduct.description}</p>

        </div> )

}

export async function getStaticProps(context) {

    const {params} = context

    const productId  = params.pid;

    const filePath = path.join(process.cwd(), "data", "dummy-backend.json");

    const fileData = await fs.readFile(filePath);

    const data = JSON.parse(fileData);

    const product = data.products.find(p => p.id == productId);

    return {

      props: {

        loadedProduct: product,

      },

    };

  }

  export async function getStaticPaths() {

    return {

      paths: [

        { params: { pid: "p1" } },

        { params: { pid: "p2" } },

        { params: { pid: "p3" } },

      ],

      fallback:'blocking'

    };

  }

###### [FALLBACK PAGES](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-pages)

In the “fallback” version of a page:

* The page’s props will be empty.
* Using the [router](https://nextjs.org/docs/api-reference/next/router), you can detect if the fallback is being rendered, router.isFallback will be true.
* The following example showcases using isFallback:

// pages/posts/[id].js

import { useRouter } from 'next/router'

function Post({ post }) {

const router = useRouter()

// If the page is not yet generated, this will be displayed

// initially until getStaticProps() finishes running

if (router.isFallback) {

return <div>Loading...</div>

}

// Render post...

}

// This function gets called at build time

export async function getStaticPaths() {

return {

// Only `/posts/1` and `/posts/2` are generated at build time

paths: [{ params: { id: '1' } }, { params: { id: '2' } }],

// Enable statically generating additional pages

// For example: `/posts/3`

fallback: true,

}

}

// This also gets called at build time

export async function getStaticProps({ params }) {

// params contains the post `id`.

// If the route is like /posts/1, then params.id is 1

const res = await fetch(`https://.../posts/${params.id}`)

const post = await res.json()

// Pass post data to the page via props

return {

props: { post },

// Re-generate the post at most once per second

// if a request comes in

revalidate: 1,

}

}

export default Post

#### [getStaticPaths with TypeScript](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#getstaticpaths-with-typescript)

For TypeScript, you can use the GetStaticPaths type from next:

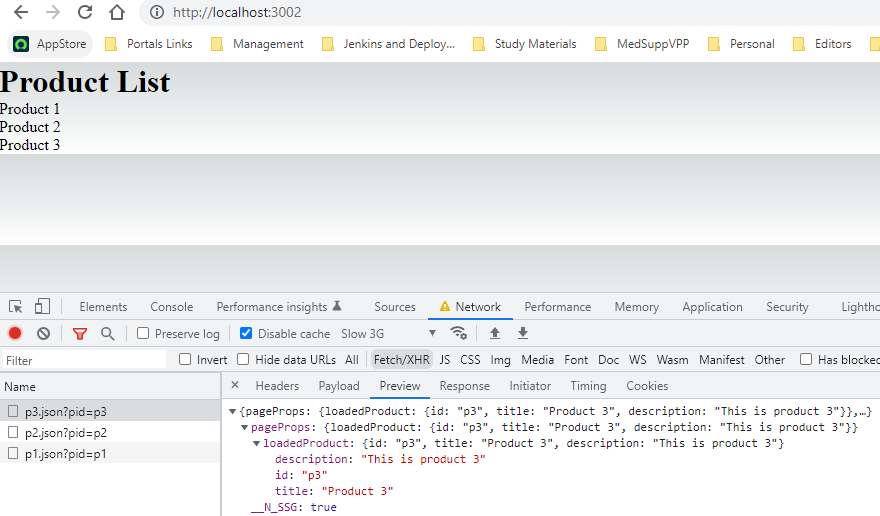
import { GetStaticPaths } from 'next'

export const getStaticPaths: GetStaticPaths = async () => {

// ...

}

#### LINK PRE-FETCHING in getStaticPaths()



* For a production build (npm run build). It creates statically generated pages (for all different parameters provided in getStaticPaths().
* It also creates corresponding the **json** files. These json files helps in prefetching the props (data), In the above example – the props /data of all the links are prefetched

##### [GENERATING PATHS ON-DEMAND](https://nextjs.org/docs/basic-features/data-fetching/get-static-paths#generating-paths-on-demand)

getStaticPaths allows you to control which pages are generated during the build instead of on-demand with [fallback](https://nextjs.org/docs/api-reference/data-fetching/get-static-paths#fallback-blocking). Generating more pages during a build will cause slower builds.

You can defer generating all pages on-demand by returning an empty array for paths. This can be especially helpful when deploying your Next.js application to multiple environments. For example, you can have faster builds by generating all pages on-demand for previews (but not production builds). This is helpful for sites with hundreds/thousands of static pages.

// pages/posts/[id].js

export async function getStaticPaths() {

// When this is true (in preview environments) don't

// prerender any static pages

// (faster builds, but slower initial page load)

if (process.env.SKIP\_BUILD\_STATIC\_GENERATION) {

return {

paths: [],

fallback: 'blocking',

}

}

// Call an external API endpoint to get posts

const res = await fetch('https://.../posts')

const posts = await res.json()

// Get the paths we want to prerender based on posts

// In production environments, prerender all pages

// (slower builds, but faster initial page load)

const paths = posts.map((post) => ({

params: { id: post.id },

}))

// { fallback: false } means other routes should 404

return { paths, fallback: false }

}

#### NOT FOUND

import fs from "fs/promises";

import path from "path";

export default function ProductDetailPage(props){

    const {loadedProduct} = props;

    return(

        <div>

            <h1>{loadedProduct.title}</h1>

            <p>{loadedProduct.description}</p>

        </div>)

}

async function getData(){

    const filePath = path.join(process.cwd(), "data", "dummy-backend.json");

    const fileData = await fs.readFile(filePath);

    const data = JSON.parse(fileData);

    return data;

}

export async function getStaticProps(context) {

    const {params} = context

    const productId = params.pid;

    const data = await getData();

    const product = data.products.find(p => p.id == productId);

    if(!product){

        return {notFound:true}

    }

    return {

      props: {

        loadedProduct: product,

      },

    };

  }

  export async function getStaticPaths() {

    const data = await getData();

    const ids = data.products.map(d=> d.id);

    const pathParams =  ids.map(id=>({params:{pid:id}}));

    return {

      paths: pathParams,

      fallback:'blocking'

    };

  }

### SERVER SIDE GENERATION - getServerStaticProps()

* If we export a function called getServerSideProps (Server-Side Rendering) from a page, Next.js will pre-render this page on each request using the data returned by getServerSideProps.
* The “context” object has hold of “request”,”response” and “params” objects.

export async function getServerSideProps(context) {

return {

props: {}, // will be passed to the page component as props

}

}

getServerSideProps only runs on server-side and never runs on the browser. If a page uses getServerSideProps, then:

* **When we request this page directly, getServerSideProps runs at request time, and this page will be pre-rendered with the returned props**
* **When we request this page on client-side page transitions through**[**next/link**](https://nextjs.org/docs/api-reference/next/link)**or**[**next/router**](https://nextjs.org/docs/api-reference/next/router)**, Next.js sends an API request to the server, which runs getServerSideProps**
* getServerSideProps returns JSON which will be used to render the page. All this work will be handled automatically by Next.js, so you don’t need to do anything extra as long as you have getServerSideProps defined.
* getServerSideProps can only be exported from a **page**. You can’t export it from non-page files.
* Note that you must export getServerSideProps as a standalone function — it will **not** work if you add getServerSideProps as a property of the page component.

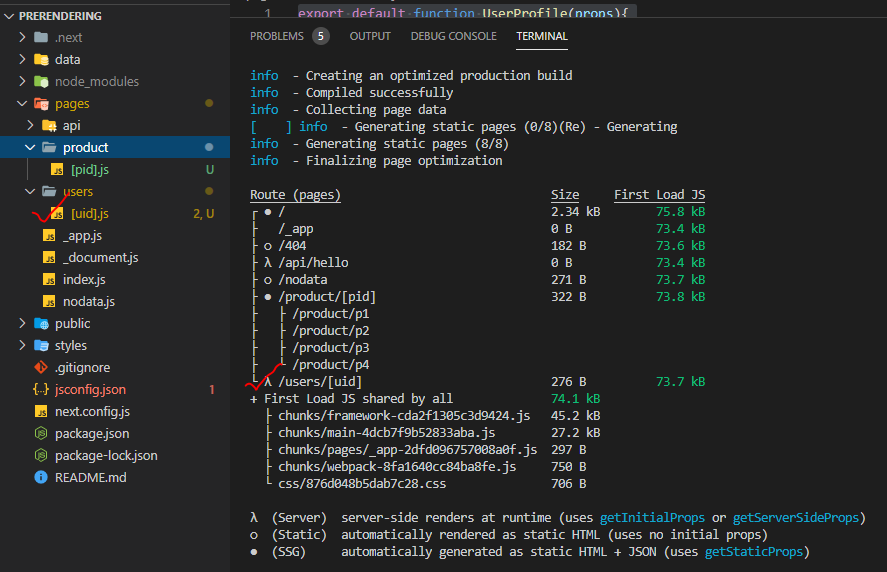
#### EXAMPLE

* Let’s say we have - *UserProfilePage* component where we want to get some users specific data and show that on the screen. Here we cannot pre-render because we need to know for which user, we are rendering this. Instead, we want to identify the user making the request (e.g -user name).
* For that - we need to get access to the request object which carries the cookies and the headers to find out which users sent this request.That would be a typical use case for getServerSideProps.

|  |
| --- |
| export default function UserProfile(props){      return(  <h1>{props.id}</h1> )  }  export async function getServerSideProps(context){      const {params,req, res} = context;      const userId = params.uid;      return {          props: {              id:'userId -'+userId          }      }  } |

BUILDING THE APP : **npm run build**

* The /users/[uid] route is server side generated



## CLIENT-SIDE DATA FETCHING

The client side data fetching usecases:

1. Data changing with high frequency (stock data)
2. High user-specific data (last orders of online shopping)

In these use case we cannot make use of server side rendering or static site generation

## COMBINING PREFETCHING WITH CLIENT-SIDE DATA FETCHING

## OPTIMIZING NEXTJS APP

### CONFIGURING HEAD CONTENT

## API ROUTES