BASICS

Convention

A Dynamic Segment can be created by wrapping a file or folder name in square brackets: `[segmentName]`. For example, `[id]` or `[slug]`.

Dynamic Segments can be accessed from `useRouter`.

Example

For example, a blog could include the following route `pages/blog/[slug].js` where `[slug]` is the Dynamic Segment for blog posts.

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()
  return <p>Post: {router.query.slug}</p>
}
```

Route	Example URL	`params`
`pages/blog/[slug].js`	`/blog/a`	`{ slug: 'a' }`
`pages/blog/[slug].js`	`/blog/b`	`{ slug: 'b' }`
`pages/blog/[slug].js`	`/blog/c`	`{ slug: 'c' }`

Catch-all Segments

Dynamic Segments can be extended to **catch-all** subsequent segments by adding an ellipsis inside the brackets `[...segmentName]`.

For example, `pages/shop/[...slug].js` will match `/shop/clothes`, but also `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`, and so on.

Route	Example URL	`params`
`pages/shop/[...slug].js`	`/shop/a`	`{ slug: ['a'] }`
`pages/shop/[...slug].js`	`/shop/a/b`	`{ slug: ['a', 'b'] }`
`pages/shop/[...slug].js`	`/shop/a/b/c`	`{ slug: ['a', 'b', 'c'] }`

index.js × | index.jsx ×

```jsx
import { useRouter } from "next/router";

const Index = () => {
  const router = useRouter();

  // NOTE: http://localhost:3000/1/2/3
  console.log(router.query.slug); // HACK: [1,2,3]

  return <>hehehe</>;
};

export default Index;
```

router.push

Handles client-side transitions, this method is useful for cases where `next/link` is not enough.

```
router.push(url, as, options)
```

- `url`: `UrlObject | String` - The URL to navigate to (see Node.JS URL module documentation ↗ for `UrlObject` properties).
- `as`: `UrlObject | String` - Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes.

Customizing The 404 Page

To create a custom 404 page you can create a `pages/404.js` file. This file is statically generated at build time.

pages/404.js

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

Rendering

By default, Next.js **pre-renders** every page. This means that Next.js generates HTML for each page in advance, instead of having it all done by client-side JavaScript. Pre-rendering can result in better performance and SEO.

Each generated HTML is associated with minimal JavaScript code necessary for that page. When a page is loaded by the browser, its JavaScript code runs and makes the page fully interactive (this process is called hydration↗ in React).

Pre-rendering

Next.js has two forms of pre-rendering: **Static Generation** and **Server-side Rendering**. The difference is in **when** it generates the HTML for a page.

- Static Generation: The HTML is generated at **build time** and will be reused on each request.
- Server-side Rendering: The HTML is generated on **each request**.

SSG

Static Site Generation

WHEN: Static Content, Data change is negligible.

(Will run on Server)

```
export async function getStaticProps(context) {
  console.log(context);
  const id = context.params.id;
  const event = await getEventById(id);
  return {
    props: { event: event },
  };
}
```

can be used with getStaticPaths.

→ can be accessed by component being rendered

```
export async function getStaticPaths() {
  const events = await getAllEvents();
  const ids = events.map((e) => e.id);
  const paths = ids.map((i) => ({
    params: {
      id: i,
    },
  }));
  return {
    paths: paths,
    fallback: false,
  };
}
```

generate static pages for all dynamic paths

it means there will be an error if dynamic path passed was not found in dB at Build!

IN-case you want to build pages on DEMAND!

```
export async function getStaticProps(context) {
  console.log(context);
  const id = context.params.id;
  const event = await getEventById(id);
  if (!event) {
    return {
      redirect: {
        destination: "/no-data",
      },
    };
  }
  return {
    props: { event: event },
  };
}

export async function getStaticPaths() {
  const events = await getAllEvents();
  const ids = events.map((e) => e.id);
  const paths = ids.map((i) => ({
    params: {
      id: i,
    },
  }));
  return {
    paths: paths,
    fallback: true,
  };
}
```

if no event (e4) was found redirect to 404

if e4 found build it.

ids = [e1, e2, e3]

if req for e4 comes, e4 page will be tried to build.

FALLBACKS:-

- **False:** Only paths specified will be built. All other paths to 404.
- **True:** Pages might build on demand.

  Components can access `router.isFallback` to show react Suspense or loading gifs.

  If you want to show user the component is loading use this.
- **BLOCKING:** Pages might build on demand.

  No Loading state can be shown

  User will see a blank page until the page is fully built by server.

When should I use Static Generation?

We recommend using **Static Generation** (with and without data) whenever possible because your page can be built once and served by CDN, which makes it much faster than having a server render the page on every request.

You can use Static Generation for many types of pages, including:

- Marketing pages
- Blog posts and portfolios
- E-commerce product listings
- Help and documentation

You should ask yourself: "Can I pre-render this page **ahead** of a user's request?" If the answer is yes, then you should choose Static Generation.

On the other hand, Static Generation is **not** a good idea if you cannot pre-render a page ahead of a user's request. Maybe your page shows frequently updated data, and the page content changes on every request.

In cases like this, you can do one of the following:

- Use Static Generation with **Client-side data fetching:** You can skip pre-rendering some parts of a page and then use client-side JavaScript to populate them. To learn more about this approach, check out the Data Fetching documentation.
- Use **Server-Side Rendering:** Next.js pre-renders a page on each request. It will be slower because the page cannot be cached by a CDN, but the pre-rendered page will always be up-to-date. We'll talk about this approach below.

ISR

Incremental Static Regeneration

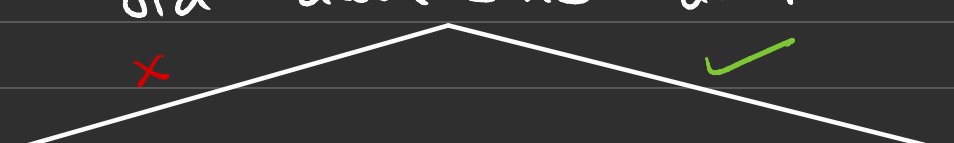
WHEN: datan changes are very less frequent.

(Will Run on Server only)

```
export async function getStaticProps() {
  console.log("Home Page Rendering");
  const arr = await getFeaturedEvents();
  return {
    props: {
      fevents: arr,
    },
    revalidate: 10,
  };
}
```

After build, when revalidate expires, (if client requests,) server will try to rebuild the page by hitting the API. 1

old data = new data



Resend old page.

Build a new page with new data

Same as SSG
no revalidation.

revalidation time
in seconds

SSR

Server Side Rendering

WHEN: data changes very frequently.

```jsx
import React from "react";

const Index = (props) => {
  return <div>{props.data}</div>;
};

export async function getServerSideProps() {
  const res = await fetch("http://localhost:4000/api");
  const data = await res.json();
  return {
    props: {
      data: data,
    },
  };
}
export default Index;
```

1- Client requests.
2- Server hits API / executes getServerSideProps
3- Server builds the page with taza data
4- Client recieves the page.

WHEN APP BUSINESS RELIES ON FRESH DATA USE THIS!

CSR

Client-side Rendering (CSR)

In Client-Side Rendering (CSR) with React, the browser downloads a minimal HTML page and the JavaScript needed for the page. The JavaScript is then used to update the DOM and render the page. When the application is first loaded, the user may notice a slight delay before they can see the full page, this is because the page isn't fully rendered until all the JavaScript is downloaded, parsed, and executed.

After the page has been loaded for the first time, navigating to other pages on the same website is typically faster, as only necessary data needs to be fetched, and JavaScript can re-render parts of the page without requiring a full page refresh.

In Next.js, there are two ways you can implement client-side rendering:

1. Using React's `useEffect()` hook inside your pages instead of the server-side rendering methods (`getStaticProps` and `getServerSideProps`).
2. Using a data fetching library like SWR ↗ or TanStack Query ↗ to fetch data on the client (recommended).

Here's an example of using `useEffect()` inside a Next.js page:

Here's an example of using `useEffect()` inside a Next.js page:

pages/index.js

```
import React, { useState, useEffect } from 'react'

export function Page() {
  const [data, setData] = useState(null)

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://api.example.com/data')
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`)
      }
      const result = await response.json()
      setData(result)
    }

    fetchData().catch((e) => {
      // handle the error as needed
      console.error('An error occurred while fetching the data: ', e)
    })
  }, [])

  return <p>{data ? `Your data: ${data}` : 'Loading...'}</p>
}
```

In the example above, the component starts by rendering `Loading...`. Then, once the data is fetched, it re-renders and displays the data.

Although fetching data in a `useEffect` is a pattern you may see in older React Applications, we recommend using a data-fetching library for better performance, caching, optimistic updates, and more. Here's a minimum example using SWR ↗ to fetch data on the client:

pages/index.js

```
import useSWR from 'swr'

export function Page() {
  const { data, error, isLoading } = useSWR(
    'https://api.example.com/data',
    fetcher
  )

  if (error) return <p>Failed to load.</p>
  if (isLoading) return <p>Loading...</p>

  return <p>Your Data: {data}</p>
}
```

Good to know:

Keep in mind that CSR can impact SEO. Some search engine crawlers might not execute JavaScript and therefore only see the initial empty or loading state of your application. It can also lead to performance issues for users with slower internet connections or devices, as they need to wait for all the JavaScript to load and run before they can see the full page. Next.js promotes a hybrid approach that allows you to use a combination of server-side rendering, static site generation, and client-side rendering, **depending on the needs of each page** in your application. In the App Router, you can also use Loading UI with Suspense to show a loading indicator while the page is being rendered.

SSG + CSR

1. Combines benifits of both SSG, CSR.

2. SEO, friendley.

3. SEO important data can be prerendered using ssg.

4. Less important or fresh data can be loaded again to display.

```
5   csr/index.jsx ×   Index.jsx ×   ssgcsr/index.jsx
15 import React, { useEffect } from "react";
14
13 const getData = async () => {
12   const res = await fetch("http://localhost:4000/api");
11   const data = await res.json();
10   return data;
 9 };
 8
 7 const Index = (props) => {
 6   const [data, setData] = useState(props.data);
 5
 4   useEffect(() => {
 3     const newData = getData();
 2     //! WARNING: Check if newData is actually new than Server Data
 1     setData(newData);
16   }, []);
 1
 2   return <div>{data}</div>;
 3 };
 4
 5 export async function getStaticProps() {
 6   const newData = getData();
 7   return {
 8     props: {
 9       data: newData,
10     },
11   };
12 }
13
14 export default Index;
```