Basic Overview

1. Server-Side Rendering (SSR)

- **Definition**: SSR is the technique of rendering a page on the server on each request. When a user requests a page, Next.js pre-renders it on the server, and the complete HTML is sent to the browser. This allows the content to be visible to the user faster and improves SEO because search engines can index the fully rendered HTML.
- **Function**: In Next.js, SSR is achieved with `getServerSideProps`.
- **Example**:

```
export async function getServerSideProps(context) {
  // Fetch data from an API
  const res = await fetch(`https://api.example.com/data`);
  const data = await res.json();

  return {
    props: { data }, // Passed to the page component as props
  };
}
```

- **Use Case**: SSR is useful for pages with dynamic data that changes often or needs to be fetched from an API with every request.

2. Client-Side Rendering (CSR)

- **Definition**: CSR refers to rendering pages on the client side (browser) after the initial load. The server sends a minimal HTML file, and JavaScript takes over to render the content.
- **Hydration**: CSR pages go through a hydration process, where JavaScript enhances the static HTML, making the page interactive.
- **Use Case**: CSR is typically used for single-page applications where SEO is less of a priority, or the content doesn't need to be pre-rendered on the server.

3. Hydration

- **Definition**: Hydration is the process by which a static HTML file sent from the server becomes interactive in the browser. In Next.js, this is crucial for pages using SSR or static generation to enable client-side interactivity after the HTML loads.
- **Example**: In Next.js, you don't manually handle hydration, as it's handled automatically during SSR or static generation.

4. Static Generation (SSG)

- **Definition**: SSG is a technique where the HTML for a page is generated at build time and reused for each request. This makes loading very fast because the same pre-rendered HTML is served to all users.
- **Function**: SSG is achieved in Next.js using `getStaticProps`.
- **Example**:

```
export async function getStaticProps() {
  // Fetch data at build time
  const res = await fetch(`https://api.example.com/data`);
  const data = await res.json();

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

- **Use Case**: SSG is ideal for content that doesn't change frequently, such as blog posts, product pages, and other mostly static content.

5. Incremental Static Regeneration (ISR)

- **Definition**: ISR allows Next.js to re-generate static pages after a certain interval, even after the app has been deployed. This enables static generation for content that might need occasional updating without rebuilding the entire app.
- **Function**: ISR uses the `revalidate` property in `getStaticProps`.
- **Example**:

```
export async function getStaticProps() {
  const res = await fetch(`https://api.example.com/data`);
  const data = await res.json();

  return {
    props: { data },
    revalidate: 10, // Re-generate the page every 10 seconds if there are requests
  };
}
```

- **Use Case**: ISR is useful for content that is mostly static but may need periodic updates, like blog posts, news articles, or product listings.

6. Fallback

- **Definition**: `fallback` is used with `getStaticPaths` to handle dynamic routes. When a path is not generated at build time, `fallback` allows Next.js to serve a fallback page or dynamically create the page on request.
- **Options**:
  - **`fallback: false`**: Only the paths returned by `getStaticPaths` are generated at build time, and any other path will show a 404 page.
  - **`fallback: true`**: The page is rendered as a fallback initially and filled with data once it's loaded.
  - **`fallback: 'blocking'`**: The page is generated on the server, and the user waits until it's ready (no fallback loading state).
- **Example**:

```
export async function getStaticPaths() {

  return {
    paths: [{ params: { id: '1' } }],
    fallback: true,
  };
}
```

7. getStaticProps and getStaticPaths

- **getStaticProps**: Fetches data at build time for static generation.
- **getStaticPaths**: Defines dynamic routes to pre-render during build time.
- **Example**:

```
export async function getStaticPaths() {
  const res = await fetch(`https://api.example.com/items`);
  const items = await res.json();

  const paths = items.map((item) => ({
    params: { id: item.id.toString() },
  }));

  return { paths, fallback: false };
}
```

8. useSWR

- **Definition**: useSWR is a React hook for data fetching, commonly used for CSR in Next.js. It allows caching, revalidation, and automatic re-fetching.
- **Example**:

```
import useSWR from 'swr';

const fetcher = (url) => fetch(url).then((res) => res.json());

export default function Component() {
  const { data, error } = useSWR('/api/data', fetcher);

  if (error) return <div>Failed to load</div>;
  if (!data) return <div>Loading...</div>;

  return <div>{data.name}</div>;
}
```

- **Use Case**: Ideal for client-side data fetching where you want to cache and automatically update the data.

9. Routing in Next.js

- **File-Based Routing**: Next.js automatically generates routes based on the file structure in the pages directory.
- **Dynamic Routing**: Dynamic routes are created by wrapping the file name in square brackets, like [id].js, to handle parameters.
- **Nested Routing**: Simply create nested folders within pages to create a nested route.
- **API Routes**: These are serverless functions stored in the pages/api directory that allow you to build backend endpoints.

10. Navigation in Next.js

- **Linking Pages**: Next.js provides the <Link> component to enable client-side navigation.

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

function HomePage() {
  return (
    <Link href="/about">
      <a>Go to About</a>
    </Link>
  );
}
```

- **Client-Side Navigation**: Using <Link> or useRouter for navigation prevents full page reloads, making transitions faster.
- **useRouter**: A hook that provides access to the router object.

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

function Component() {
  const router = useRouter();

  const goToAbout = () => {
    router.push('/about');
  };

  return <button onClick={goToAbout}>Go to About</button>;
}
```

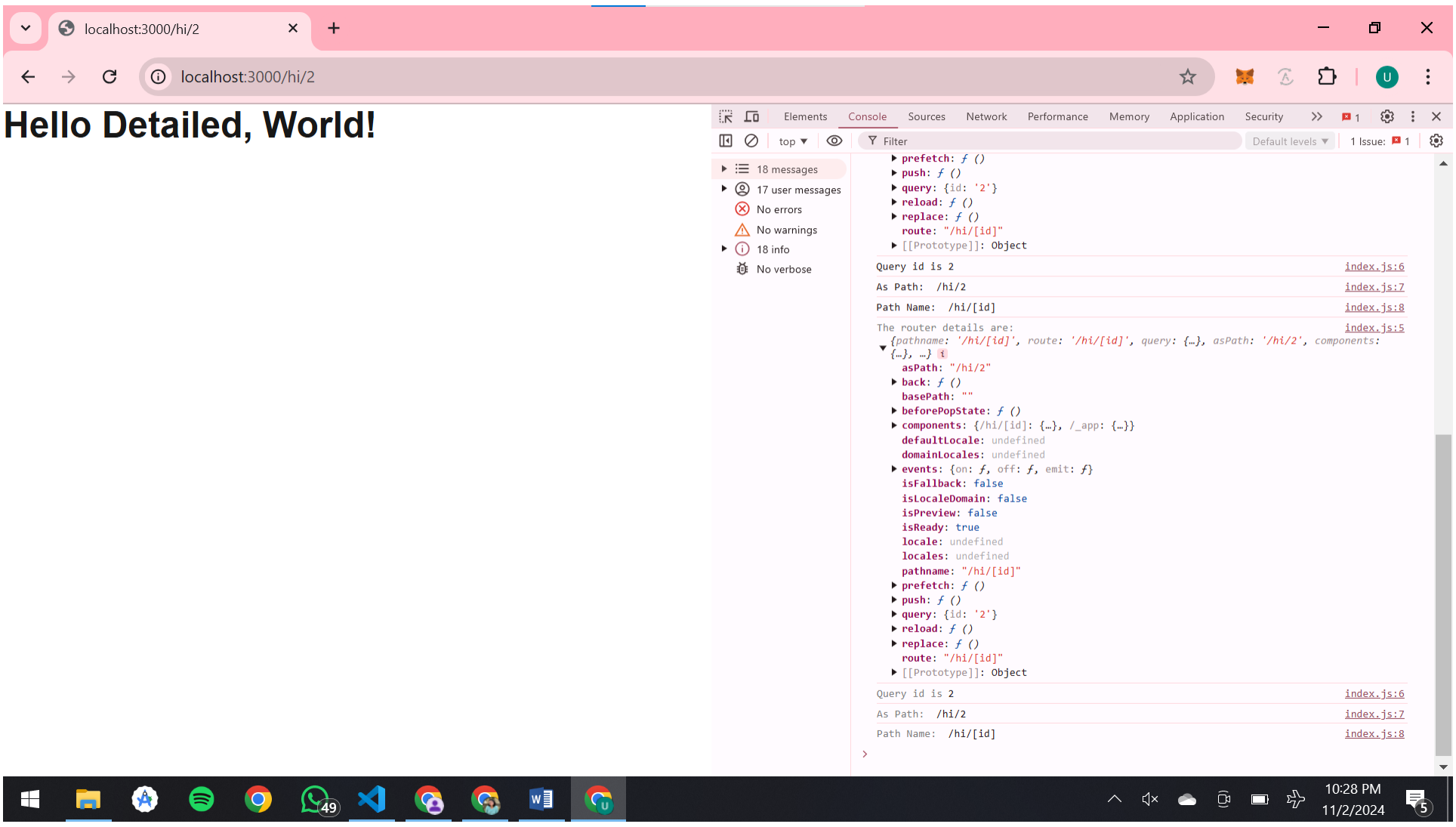
Summary Table

Topic	Purpose	Usage
SSR	Server-Side Rendering	getServerSideProps
CSR	Client-Side Rendering	Client-side JavaScript

Topic	Purpose	Usage
Hydration	Make page interactive	Automatic in SSR/SSG
SEO	Improve indexing	Better in SSR/SSG
ISR	Incremental Static Regeneration	revalidate in getStaticProps
fallback	Dynamic paths handling	getStaticPaths
getStaticProps	Fetch data at build time	Static Generation
getStaticPaths	Define dynamic routes for static pages	Dynamic Static Generation
useSWR	Client-side data fetching	useSWR hook
Link and useRouter	Client-side navigation	<Link> and useRouter

---

UseRouter:



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

- **Purpose**: Navigates to a different route.
- **Usage**: **Typically used for client-side navigation without reloading the page.**
- **Parameters**:
  - **url**: The path you want to navigate to, e.g., /about.

Example:

```
router.push('/about');                    // Navigates to /about
```

2. router.replace(url, as, options)

- **Purpose**: Similar to router.push, but instead of adding a new entry in the browser's history stack, it replaces the current entry. This means users cannot navigate back to the previous page.

Example:

```
router.replace('/login');                 // Navigates to /login and replaces the current entry
```

3. router.reload()

- **Purpose**: Reloads the current page. This is useful if you need to reset the page state or fetch new data.
- **Example**:

```
router.reload();
```

4. router.back()

- **Purpose**: Navigates back in the history stack (similar to pressing the browser's back button).
- **Example**:

```
router.back();
```

6. router.query

- **Purpose**: Accesses the query parameters in the URL. This is particularly useful for dynamic routes.
- **Example**:

```
const { id } = router.query;                    // Accesses id if the route is something like /post?id=123
```

7. router.pathname

- **Purpose**: Returns the path of the current route. It can be useful for checking the current route without the query string.
- **Example**:

```
console.log(router.pathname);                   // e.g., /about or /posts/[id]
```

8. router.asPath

- **Purpose**: Returns the actual path in the browser's address bar, including the query string. It's helpful for tracking the full path including dynamic segments and query parameters.
- **Example**:

```
console.log(router.asPath);                     // e.g., /posts/123
```

10. router.isFallback

- **Purpose**: Indicates whether the page is in fallback mode. It's mainly used with dynamic paths in static generation (SSG) where getStaticPaths uses fallback: true.

Example:

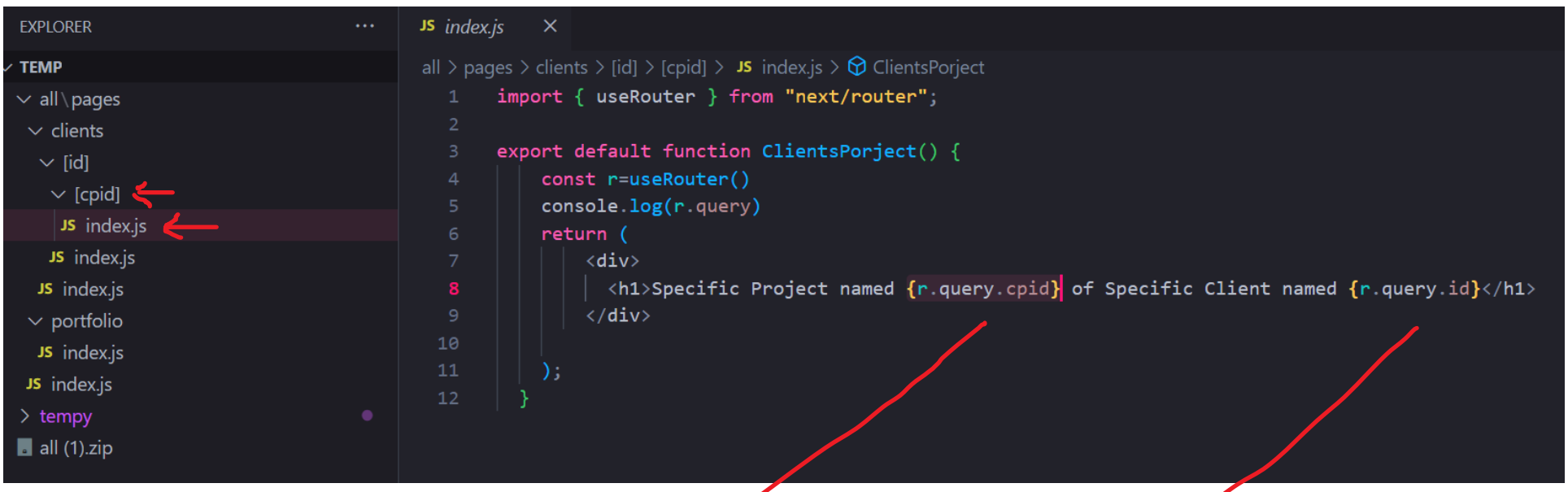
```
if (router.isFallback) {
  return <div>Loading...</div>;
}
```

➢ **router.push(url, as, options)**:

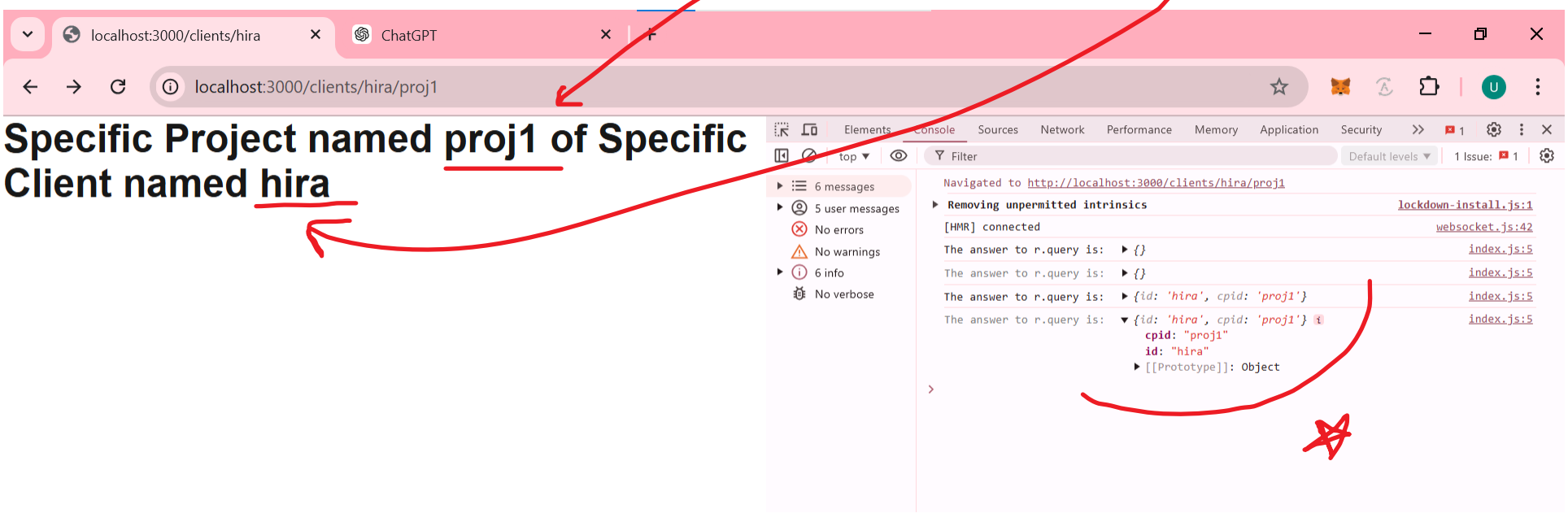
- Adds a new entry to the browser's history stack. This means when you navigate to a new page using router.push, the user can go back to the previous page by clicking the browser's back button.
- Use router.push when you want to allow users to navigate back to the page they came from.

➢ **router.replace (url, as, options)**:

- Replaces the current entry in the history stack with the new one. This means navigating to a new page with router.replace will not allow the user to go back to the previous page with the browser's back button.
- Use router.replace when you don't want the user to return to the previous page, like when redirecting after a form submission or login.



Whatever is the folder name, that will be the ID in r.query



Static Props:

```
export async function getStaticProps() {
    //running server-side code & using the FileSystem module of Node server
    const p = path.join(process.cwd(), "data", "dummy-backend.json");
    const datajson = await fs.readFile(p);
    const data = JSON.parse(datajson);
    console.log("re-rendering it");
    if (!data) {
        return {
            redirect: {
                destination: "/no-data",
            },
        };
    }
    if (data.products.length === 0) {
        return {
            notFound: true,
        };
    }
    return {
        props: {
            products: data.products,
        },
        //Incremental Static Regeneration
        revalidate: 10,
    };
}

export default Home;
```

```
export async function getStaticProps ( context )
{
    const p=path.join(process.cwd(),'data','dummy-backend.json');
    const datajson =await fs.readFile(p);
    const data=JSON.parse(datajson);

    const p1=data.products.find(o=>o.id===context.params.pid)
    //there is no such product (it means you have given wrong pid in URL)
    if(!p1)
    {
        return{
            redirect:{
                destination:'/no-data'
            }
        }
    }

    return {
        props:{
            loadedProduct:p1
        }
    }
}
//define all paths in this func for which you want your server to pre-generate this dynamic page.
//It will then pre-generate all such pages during build time
export async function getStaticPaths(){
    const p=path.join(process.cwd(),'data','dummy-backend.json');
    const datajson =await fs.readFile(p);
    const data=JSON.parse(datajson);

    const ids=data.products.map(obj=>obj.id)

    const pathss=ids.map(obj=>
        (
            { params: {pid:obj} }
        )
    )

    return{
        paths:pathss,
        fallback:true
    }
}
export default ProductDetailPage;
```

getStaticProps is a special function in Next.js that enables **Static Site Generation (SSG)** by fetching data at build time. This function runs on the server side during the build process, allowing you to pre-render pages with static content before the page is served to the user.

Purpose of getStaticProps

- **Data Fetching at Build Time**: getStaticProps is used to fetch data that does not change frequently, such as product listings, blog posts, or static content. The data is fetched at build time and embedded into the static HTML, making the page load faster.
- **Static Site Generation (SSG)**: The function pre-generates HTML pages for all paths using the data returned by getStaticProps.
- **SEO Benefits**: Since content is pre-rendered, it's indexed easily by search engines, improving SEO.

Key Concepts of getStaticProps

1. **When getStaticProps Runs**
    - **At Build Time**: getStaticProps runs once at build time to generate static pages for every route that uses it.
    - **On Revalidation** (if ISR is used): If the page is accessed after the revalidation period (10 seconds here), Next.js re-generates the page in the background with the latest data, allowing content updates without needing to rebuild the entire site.

Example Scenarios for getStaticProps

1. **Blog Posts**: Fetching a list of blog posts from a CMS and displaying them on a static blog page.
2. **E-commerce Product Listings**: Fetching product data to display on a static product listing page with periodic updates using ISR.
3. **Marketing Pages**: Fetching content for static landing pages or marketing pages that rarely change.

Advantages of Using getStaticProps

- **Improved Performance**: Since the page is pre-rendered as static HTML, it's highly performant and loads quickly.
- **SEO Benefits**: Pre-rendered content is indexed better by search engines.
- **Simplified Data Handling**: With getStaticProps, data is available at build time, reducing the need for client-side fetching.
- **Partial Dynamic Content with ISR**: revalidate lets you update content without rebuilding the entire site.

In summary, getStaticProps is a powerful way to pre-render data-driven pages with Next.js, enabling fast, SEO-friendly pages and giving flexibility with Incremental Static Regeneration for content that needs occasional updates.

Static Paths:

`getStaticPaths` is another special function in Next.js used with **Dynamic Routes** to pre-generate pages for all the specific paths listed during the build process. It's essential for creating **Static Site Generation (SSG)** for dynamic routes, where the page content is based on parameters, such as product IDs or blog post slugs.

Purpose of `getStaticPaths`

`getStaticPaths` is needed when you want to pre-render multiple instances of a dynamic page using specific parameters. For instance, in an e-commerce site with dynamic routes for each product, `getStaticPaths` allows you to pre-generate pages for specific products so they are available immediately.

Code Example

Here's the code from your example:

```
export async function getStaticPaths() {
    const p = path.join(process.cwd(), 'data', 'dummy-backend.json');
    const datajson = await fs.readFile(p);
    const data = JSON.parse(datajson);

    // Getting all IDs from products
    const ids = data.products.map(o => o.id);
    const pathss = ids.map(o => ({ params: { pid: o } }));

    return {
        paths: pathss,
        fallback: true,
    };
}
```

Key Concepts in `getStaticPaths`

1. **Paths Definition**
    - `getStaticPaths` must return an object containing a `paths` array, which defines the dynamic routes that Next.js should pre-render at build time.
    - In this example:

      ```
      const pathss = ids.map(o => ({ params: { pid: o } }));
      ```

        - The `paths` array here is created by mapping through `data.products`, and for each product ID, a route parameter object `{ params: { pid: o } }` is defined.
        - This tells Next.js to pre-render a static page for each `pid` path (e.g., `/products/1`, `/products/2`).
2. **Returned Object Properties**
    - **`paths`**: An array of objects representing the parameters needed to generate each page. Each object inside `paths` represents one route with parameters specified.

      ```
      return {
          paths: pathss,
          fallback: true,
      };
      ```

- **`fallback`**: Specifies what should happen when a requested path is not present in `paths`.

3. **Fallback Options** `fallback` defines how Next.js should handle paths not included in the `paths` array:
   - **`fallback: false`**:
     - Only pre-generated paths will be available. Any other paths will show a 404 page.
     - Use this when you only want to support the specific paths you've pre-rendered.
   - **`fallback: true`**:
     - When a path is not pre-rendered, Next.js will render the page on the server and cache it.
     - This is useful when you have many potential paths, and pre-rendering all of them is not practical.
     - For pages with `fallback: true`, you should handle the loading state in the component, as the content may initially be empty while the page is generated.
     - For example, in `ProductDetailPage`, you used:

```
if (!props.loadedProduct) {
    return <p>Loading...</p>;
}
```

       - This ensures that users see a loading message until the server-generated page content is ready.
   - **`fallback: "blocking"`**:
     - When a path is not pre-rendered, Next.js will render the page on the server like with `fallback: true`, but it waits until the page is generated to show anything.
     - Users don't see a loading state; instead, they only see the page when it's fully generated.

Example Scenarios for `getStaticPaths`

1. **Blog Posts**: A blog site with dynamic routes like `/posts/[id]` where each post has a unique ID. You'd use `getStaticPaths` to pre-render pages for the most popular blog posts and use `fallback: true` or `"blocking"` for the rest.
2. **Product Pages**: In an e-commerce app, dynamic routes like `/products/[pid]` can be pre-rendered for popular items with `getStaticPaths` and `fallback: true` to allow server-side generation for less common items.

Workflow of `getStaticPaths`

1. **At Build Time**: `getStaticPaths` runs only at build time to determine which pages need to be pre-rendered.
2. **On Page Request (if `fallback` is enabled)**: When a user requests a page not in `paths` and `fallback` is `true` or `"blocking"`, Next.js will dynamically generate the page on the server, cache it, and serve it to the user.
3. **Future Requests**: Once a page has been generated and cached, it's treated as a static page, so future requests are served quickly.

Full Flow of `getStaticProps` and `getStaticPaths`

1. **`getStaticPaths`** provides the paths that should be pre-rendered during build.
2. **`getStaticProps`** then fetches data specific to each path and returns it as props.
3. **`fallback`** determines what happens if a user requests a page not generated by `getStaticPaths`.

In summary, `getStaticPaths` is crucial for defining paths for pre-rendered pages in Next.js when working with dynamic routes, and it gives you control over handling pages that may or may not have been pre-generated.

Server Side Props:

`getServerSideProps` is a function in Next.js that enables **Server-Side Rendering (SSR)**. When you use `getServerSideProps`, a page's data is fetched on the server for each request, ensuring the page is always rendered with the latest data. This is different from **Static Generation** methods (`getStaticProps` and `getStaticPaths`), which generate pages at build time or in advance.

Purpose of `getServerSideProps`

`getServerSideProps` is used when you need data that changes frequently or is personalized based on each request, such as:

- User-specific data (e.g., user dashboard)
- Frequently updated data (e.g., stock prices, sports scores)
- Content that changes with each request (e.g., personalized recommendations)

By using SSR with `getServerSideProps`, you ensure that the data rendered is always up-to-date.

How `getServerSideProps` Works

- **Runs on Every Request**: Unlike static generation functions, `getServerSideProps` executes on the server every time a page is requested. It provides real-time data for each request, which may impact performance slightly if the data fetching is complex or takes time.
- **Data Fetching on the Server**: Any data-fetching code you write within `getServerSideProps` is run only on the server. This means sensitive information, like database credentials or private API keys, stays secure and never reaches the client.

Basic Example of `getServerSideProps`

Here's a simple example:

```
export async function getServerSideProps(context) {
    const res = await fetch('https://api.example.com/data');
    const data = await res.json();

    return {
        props: {
            data,
        },
    };
}

function Page({ data }) {
```

```
    return <div>{JSON.stringify(data)}</div>;
}

export default Page;
```

Key Concepts in `getServerSideProps`

1. **Context Parameter**
   - `context` is an object with several useful properties, including:
     - **`params`**: Contains route parameters (useful for dynamic routes).
     - **`req` and `res`**: Access the HTTP request and response objects. These allow you to handle cookies, headers, and other request details.
     - **`query`**: Query parameters from the URL.
     - **`resolvedUrl`**: The actual path requested by the user.

   Example:

   ```
   export async function getServerSideProps(context) {
       console.log(context.params);  // Dynamic route params
       console.log(context.query);   // URL query parameters
       console.log(context.req);     // Request object
       console.log(context.res);     // Response object
   }
   ```

2. **Returning `props`**
   - `getServerSideProps` must return an object with a `props` key, which contains data passed to the page component as props.
   - These props are accessible in the component's function, allowing you to render data directly from the server.
3. **Redirecting Users**
   - You can use `redirect` to navigate the user to another page based on certain conditions.
   - Example:

   ```
   export async function getServerSideProps(context) {
       if (!context.query.valid) {
           return {
               redirect: {
                   destination: '/error-page',
                   permanent: false,
               },
           };
       }
       return { props: {} };
   }
   ```

   - **`permanent: true`** or **`false`** defines the type of redirect (301 for permanent, 302 for temporary).
4. **Handling Errors**
   - If there's a problem fetching data, you can return a `notFound` object, which will show a 404 page.
   - Example:

   ```
   export async function getServerSideProps() {
       const data = await fetchData();
       if (!data) {
           return {
               notFound: true,
           };
       }
       return { props: { data } };
   }
   ```

Use Cases for `getServerSideProps`

1. **User Authentication**: If you want to render a page only for authenticated users, you can use `getServerSideProps` to check user authentication status before rendering.

   ```
   export async function getServerSideProps(context) {
       const { req } = context;
       const user = await verifyAuth(req);

       if (!user) {
           return {
               redirect: {
                   destination: '/login',
                   permanent: false,
               },
           };
       }

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

2. **Frequently Updated Content**: Pages that need frequent updates, such as a live scoreboard, can use `getServerSideProps` to fetch the latest data on each request.
3. **Geolocation or IP-based Content**: `getServerSideProps` can determine user location based on `req` information, allowing you to show region-specific content.

Workflow of `getServerSideProps`

1. **Request Initiation**: Each time a user accesses the page, `getServerSideProps` is called on the server.
2. **Data Fetching**: The function retrieves necessary data, possibly from an API, database, or third-party service.
3. **Data Return**: The data is returned as `props`, which are passed into the page component.
4. **Rendering**: The page is rendered on the server with the fetched data, and the HTML is sent to the client.

Differences Between `getStaticProps`, `getServerSideProps`, and Client-Side Data Fetching

- **`getStaticProps`**: Runs at build time, creates static pages, and is ideal for content that doesn't need frequent updates.
- **`getServerSideProps`**: Runs on every request, generating HTML on the server each time. It's useful for dynamic or user-specific data.
- **Client-Side Data Fetching** (e.g., `useEffect` in React): Fetches data on the client after the initial render, suitable for data that doesn't need SSR or isn't essential for SEO.

Performance Considerations

While `getServerSideProps` guarantees fresh data for each request, it can impact load times, especially if data fetching is slow. This approach is best used for pages that truly require fresh, real-time content.

In summary, `getServerSideProps` is perfect for pages where data needs to be up-to-date with each request. It allows you to access server-side information, manage redirections, and control the user experience based on real-time conditions.