1. Nextjs data fetching

On the server, with fetch

On the server, with third-party libraries

On the client, via a Route handler

On the client, with third-party libraries

1. Caching

Next.js improves our application performance and reduces costs by caching rendering work and data requests.

By default, Next.js will cache as much as possible to improve performance and reduce cost. This means routes are statically rendered and data requests are cached unless you opt out.

**“Opting out means choosing not to follow this default approach.”**

Caching mechanism

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mechanism | What | Where | Purpose | duration |
| Request memoization | Return values of function | server | Re use data in react component tree | Pre-request life |
| Data cache | data | server | Store data across user requests and deployments | Persistent(can be revalidate) |
| Full route cache | Html and RSC payload | server | Reduce rendering cost and improve performance | Persistent(can be revalidate) |
| Route cache | RSC payload | client | Reduce server requests on navigation | User session or time-based |

1. Nextjs rendering

**Static rendering (default)**

By default, nextjs statically renders routes to improve performance. At build tim, server and client component are rendered on the server, and the result of the work is cached and reused on subsequent requests.

**Dynamic rendering**

In dynamic rendering both server and client components for a route are rendered on the server at request tim.

During rendering if a dynamic function or uncached data request is discovered, Nextjs will switch to dynamically rendering the whole route.

Next.js has four rendering strategies

**Server-side rendering (SSR)**

When a user navigates to a specific route in our next application their browser sends http request to the server.

If the pages uses `getServerSideProps`, nextjs will execute function at each request

Once the data is fetched it is passed as props to the page component.

The page component is then rendered on the server with fetched data injected into it

Finally the server returns rendered html to the users browser as the response.

The js file is sent to the server. When the user (browser) request, the server will run the **fetch (url, {cache: “no-store”}).** After the data is fetched, it will be built on the server (including the data from the API).The server will send the HTML to the user’s browser.

Use-case

If our site data is updated frequently.

While at the same time, SEO is an essential factor as well.

Pros

No loading, real-time data, good for SEO, It can be used for personalized content

Cons

The user needs to wait for the HTML to be built on the server-side

Too much burden on the server => every user’s requests, the server need to rebuild.

**Client-side rendering (CSR)**

If the data fetching is done on the client side using `useEffect` hook / libraries like SWR

When the users browser loads the page, javascript code in the clien-side bundle is executed. Client side code then fetches the data asynchronously from the data source.

Build process => The HTML file is sent to the server.

When the user request, the server will send the HTML file, then the client requests the data to the API server.

In next 13 we have to add “use client” at the top to let Next.js know its rendered on the client side.

AT the time the client is requesting data from the API server, the browser will display the loading-state.

After the data is fetched from the API server, then the loading state will be turned off and the screen will be updated with the data from the API.

Use-case

if our site data is updated frequently.

seo is not a essential factor.

Pros

Real-time data, it can be used for personalized content, The burden is not too big for the server.

Cons

There will be a loading state

Not good for seo => since the HTML built from the server is not including the data from the API.

**Static-site generation (SSG)**

If the page uses `getStaicProps` next.js will generate the page at build time

Inside `getStaicProps`, we fetch data from the data source.

The fetched data is then used to pre-render the page.

When a user request the page, next.js servers the pre-rendered html directly from the cache without needing to re-render the page on server. Once data is fetched, page is updated/reacted accordingly to display the fetched data to user.

The build and fetch data process happened at the same time=> **fetch ()**(by default in Next 13) is running at build time.

After the HTNL+JSON is built (the data from API is included). It will be sent to the server.

When the user (browser) request, the server will send the HTML+JSON file, so the user doesn’t need to wait.

Use-case

If our site data is definite(fetch once and that’s it)

Pros

Overall the fastest method, no loading, the burden is not too big for the server, good for SEO.

Cons

Don’t have trigger to update the data from API unless it’s redeployed.

Can’t be used as personalized content=> because this method doesn’t have any way to update the built HTML file from the server.

**Incremental-static regeneration (ISR)**

Pretty much the same with SSG, but with the capability to update the data.

If we set the revalidated data on the **fetch (url, {next :{ revalidate: 10}}),** the server will revalidate the data from the API and see whether there is any change.

If there is any update from the API, then the built HTNL file will be updated.

But, it will only be updated after the set time in revalidate props time is passed.

Use-case

Best practice for most static sites that don’t need a real-time update.

Pros

Basically, it is SSG but with a additional tweak to be able to update the site data.

Cons

Can’t be used as personalized content.

Can’t be used if we have a real-time feature on our site.

**SEO factors**

**Clean and logical website structure**

Organize the website content using a clear hierarchy like relevant headings, breadcrumbs, and internal linking to aid search engines in crawling and understanding content

**Fast page loading speed**

Prioritize optimizing website loading speed for both desktop and mobile devices, as search engines consider it a significant ranking factor. Utilize techniques like image optimization, code minification, and efficient caching mechanism.

**Mobile friendliness**

Ensure website is responsive and adapts seamlessly to different screen sizes, providing optimal experience on mobile devices. Search engine prioritize mobile-friendly websites in their search results.

**Semantic html**

Use semantic html elements like <h1> for main headings, <h2> for subheadings, <p> for paragraphs to convey the meaning and structure of content accurately. This helps search engines understand the context and relevance of you page

**XML sitemap**

Create and submit a XML sitemap to search engines, acting as a map that guides them to all the important pages on website. This can accelerate the indexing process.

**Structured data**

Implement structured data markup to provide additional information about content in a format that search engines can easily understand. This can enhance the richness of search result