**Routing Mechanism**

1- nested routes e.g: product/review

2- dynamic routes e.g: product/1234

3- nested dynamic routes e.g: /product/2314/review/234

4- catch all routes e.g: [...abc]

5- optional catch-all-routes e.g: [[...abc]]

6- navigation e.g: <Link href={`/product/${productId}`}><a>product 3</a></Link>

===> 'replace' props in <Link href='/product/1' replace> ===> 'scroll' is another prop

7- navigating programmatically e.g: router.push() / router.replace()

8- custom 404 page e.g: create '404.js' in pages folder

**Pre-rendering and Data Fetching**

1- Pre-rendering: advance generation of html page a) it is fast b) better SEO

2- Types of Pre-rendering

a) Static Generation e.g: Blog pages, e-commerce product pages, documentation, marketing pages

b) Server-side rendering

3- **Static Generation**:- all the data/pages are rendered at build time by default.

It is a recommended method to pre-render pages whenever possible.

Page can be built at once, cached by a CDN and served to the client almost instantly

a) without data (without fetching external data) - this is a default scenario of Nextjs

b) with data (with external data) e.g: getStatisProps()

c) incremental static generation

d) dynamic parameters when fetching data

4- **getStaticProps()**

a) - this function runs only on the server side

- The function will never run client side

- the code you write inside getStaticProps won't even be included in the JS bundle that is sent to the browser

b) - You can run server-side code directly inside getStaticProps

- Accessing the file system using the fs module or querying a database can be done inside getStaticProps

- You also don't have to worry about including API keys in getStaticProps as that won't make it to the browser

c) - getStaticProps is allowed only in a page and cannot be run from a regular component file

- It is used only for pre-rendering and not client side data fetching

d) - getStaticProps should return an object and object should contain a props key which is an object

e) - getStaticProps will run at build time

- During development, getStaticProps runs on every request

5- Link Pre-fetching:-

1. Any <Link /> component in the viewport (initially or through scroll) will be pre-fetched by default (including the corresponding data) for pages using Static Generation.
2. When a page with getStaticProps is pre-rendered at build time, in addition to the page HTML file, Next.js generates a JSON file holding the result of running getStaticProps.
3. The JSON file will be used in client-side routing through next/link, or next/router
4. When you navigate to a page that's pre-rendered using getStaticProps, Next.js fetches the JSON file (pre-computed) at build time) and uses it as the props to create the page component client-side.
5. Client-side page transitions will not call getStaticProps as only the exported JSON is used.

6- **getStaticPaths()**

a) **fallback: false**

1. The paths returned from getStaticPaths will be rendered to HTML at build time by getStaticProps
2. If fallback is set to false, then any paths not returned by getStaticPaths will result in 404 page

When to use?

1. if you have an application with a smaller number of paths to pre-render
2. When new pages are not added often
3. A blog site with a few articles is a good example for fallback set to false

b) **fallback: true**

* + 1. The paths returned from getStaticPaths will be rendered to HTML at build time by getStaticProps
    2. The paths that have not been generated at build time will not result in 404 page. Instead, Next.js will serve a "fallback" version of the page on the first request to such a path.
    3. In the background, Next.js will statically generate the requested path HTML and JSON. This includes running getStaticProps
    4. When that's done, 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 prospective, the page will be swapped from the fallback page to the full page.
    5. At the same time, Next.js keeps track of the new list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, just like other pages pre-rendered at build time.

When to use?

1. The true value is most suitable when your app has a very large number of static pages that depend on data
2. A large e-commerce site
3. You want all the product pages to be pre-rendered but if you have a few thousand products, builds can take a really long time
4. You may statically generate a small subset of products that are popular and use "fallback: true" for the rest
5. When someone requests a page that's not generated yet, the user will see the page with a loading indicator
6. Shortly after, getStaticProps finishes and the page will be rendered with the requested data. From then onwards, everyone who requests the same page will get the statically pre-rendered page.
7. This ensures that users always have a fast experience while preserving fast builds and the benefits of static generation

c) **fallback: blocking**

1. The paths returned from getStaticPaths will be rendered to HTML at build time by getStaticProps
2. The paths that have not been generated at build time will not result in 404 page. Instead, on the first request, Next.js will render the page on the server and return the generated HTML.
3. When that's done, the browser receives the HTML for the generated path. From the user's prospective, it will transition from "the browser is requesting the page" to "the full page is loaded". There is no flash of loading/fallback state
4. At the same time, Next.js keeps track of the new list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, just like other pages pre-rendered at build time.

When to use?

1. On a UX level, sometimes, people refer the page to be loaded without a loading indicator if the wait time is a few milli seconds. This helps avoid the layout shift.
2. Some crawlers did not support JavaScript. The landing page will be rendered and then the full was loaded which was causing the problem.

**Issues with Static Generation**

1) The build time is proportional to the number of pages in the application.

e.g; a page takes 100ms to build, then E-commerce app with 100,000 products takes > 2.5 hours to build.

2) A page, once generated, can contain stale data till the time you rebuild the application.

e.g; product prices vary regularly but the entire app needs to re-built for displaying the fresh data

* 1. One Solution:- Pre-render only few pages at build time and rest of the pages can be pre-rendered on request
  2. problem:- If your application has 90% static pages and 10% dynamic pages, getStaticPaths will not help.

**Incremental Static Regeneration (ISR)**

1. There was a need to update only those pages which needed a change without having to rebuild the entire app.
2. With ISR, Next.js allows you to update static pages after you've built your application.
3. You can statically generate individual pages without needing to rebuild the entire site, effectivly solving the issue of dealing with stale data

How?

1. In the getStaticProps function, apart from the props key, we can specify a revalidate key.
2. The value for revalidate is the number of seconds after which a page re-generation can occur.

**Problems with Static Generation:**

1. We cannot fetch data at request time.
2. With not being able to fetch data per request, we run into the problem of stale data
3. Let's say we are building a news website
4. The content is very dynamic in the sense that new articles can be published almost every second
5. getStaticProps will fetch the news at build time which is not suitable at all
6. getStaticPaths will help fetch the data on initial request but it is then cached for subsequent requests
7. ISR can help but we might not able to see the most up-to-date news always
8. Rather, fetch data on the client side by making a get request from the component. But no SEO.

b) We don't get access to the incoming requests

1. Problem when the data that needs to be fetched is specific to user.
2. Let's say we are building a website similar to twitter
3. As user, I should see tweets that are personalized based on my interests
4. The tweets that I see also need to SEO friendly as it is public content that anyone in the world can see
5. To fetch tweets specific to the user, we need the userId. And that can only be obtained only if we access to the incoming request
6. You could do it client side in useEffect for example but that means you again miss on SEO

7- **Server-Side Rendering:-**

1. SSR is a form of pre-rendering where the HTML is generated at request time.
2. SSR is required when you need to fetch data per request and also when you need to fetch personalized data keeping in mind SEO

=> **getServerSideProps()**

1. - this function runs only on the server side
   1. The function will never run client side
   2. the code you write inside getServerSideProps won't even be included in the JS bundle that is sent to the browser
2. - You can run server-side code directly inside getServerSideProps
   1. Accessing the file system using the fs module or querying a database can be done inside getServerSideProps
   2. You also don't have to worry about including API keys in getServerSideProps as that won't make it to the browser
3. - getServerSideProps is allowed only in a page and cannot be run from a regular component file
   1. It is used only for pre-rendering and not client side data fetching
4. - getServerSideProps should return an object and object should contain a props key which is an object
5. - getServerSideProps will run at request time

=> **getServerSideProps(context)**:- here in context argument includes: params, req, res, query

=> **Client-side Data Fetching**

a) You might not always need to pre-render the data

b) e.g: User Dashboard Page

c) It is private, that is behind the login screen

d) Highly user-specific and SEO is not relevant

e) No need to pre-render the data

**Pre-rendering and Data-fetching Summary**

* Pre-rendering refers to the process of generating HTML in advance which results in better performance and SEO.
* Next JS supports two forms of pre-rendering – Static Generation and Server-side Rendering
* **Static Generation**
* A method of pre-rendering where the HTML pages are generated at build time.
* Pages can be built at once, cached by a CDN and served to clients almost instantly
* Example: Marketing or Blogging site
* For a normal page use getStaticProps function to fetch the data ahead of time
* For a dynamic page, you also need the getStaticPaths function
* Fallback: false | true | `blocking`
* Pages cannot be updated without a full re-build
* Incremental Static Regeneration. This strategy is suitable perhaps for a documentation site OR an e-commerce site
* **Server-side Rendering**
* Fetch data at request time
* Personalize data based on user information in the incoming request
* Example: News listing page
* getServerSideProps function helps with SSR data fetching
* Combining pre-rendering with client-side data fetching
* Shallow routing – Routing without calling getStaticSideProps/getServerSideProps

**API ROUTES**

* API routing mechanism is similar to page based routing mechanism
* APIs are associated with a route based on their file name
* Every API route exports a default function typically named as handler function
* The handler function receives the *request* and *response* as parameters
* Cater to different request types like GET and POST using *req.method*
* Dynamic API routes
* Catch all API routes
* How to handle a DELETE request
* We should not call our own API routes for pre-rendering content

**Styling Intro**

* **Global** – In our application, we need to import the CSS file within *pages/\_app.js*
* **Component Level** – Next.js supports CSS modules using *[name].module.css* naming convention
* **SASS Support** – Install the sass package
* **CSS-in-JS Solutions** – Inline styles and Styled Components

**Miscellaneous Section Intro**

* App layout in \_app.js file
* Head component which helps you dynamically manage a document’s head section
* Image component optimization
* Configure absolute imports and configure path aliases with jsconfig.json
* *next export* command which exports your app into static HTML
* Setup support for Typescript
* Preview mode feature is very helpful when working with a CMS
* Next configuration file and configuring redirects
* Environment variables

**Authentication in Nextjs**

* Identity and Access
* Identity: Authentication (who the user is)
* Access: Authorization (verifies what permission the user has)
* **Types**
  + Client-side authentication
  + Server-side authentication
  + API routes authentication
* User Data
* No need to persist? Auth services like Github, Facebook, Google etc. to ensure the user is authenticated
* Need to persist? Database