NextJS

What is NextJS?

- NextJS is a react framework for building web applications.
- Uses React for building user interfaces.
- Provides features that enables us to build production-ready applications. Like routing, optimized rendering, data fetching, bundling, compiling etc.
- Do not need to install additional packages.

Benefits of NextJS

- Simplifies the process of building Web Applications.
  - Routing - File based routing.
  - API Routes - Can be created as a full stack framework.
  - Rendering - Supports Server Side Rendering and Client Side Rendering.
  - Data fetching - Async await support data fetching.
  - Styling - CSS Modules and Tailwind CSS.
  - Optimization - images and fonts.
  - Dev and Prod build system.

React Server Components (RSC)

There are two component types in react from React version 18.

- Server Components
  - All components are server components in default.
  - Run tasks like reading files or fetching data from a database.
  - Do not have the ability to use hooks or handle user interactions.

- Client Components
  - Add “use client” at the top of the component.
  - Cannot perform task like reading files but the can use hooks to manage interactions.

We can setup Type Script when we creating the NextJS application. In the terminal it will be asked if we want to integrate Type Script or not.

CSS Modules

- Next.js has built-in support for CSS Modules using the `.module.css` extension.
- CSS Modules locally scope CSS by automatically creating a unique class name.
- Allows you to use the same class name in different files without worrying about collisions.
- CSS Modules the ideal way to include component-level CSS.
- CSS Modules can be imported into any file inside the `app` directory.
- CSS Modules are **only enabled for files with the `.module.css` and `.module.sass` extensions**.
- In production, all CSS Module files will be automatically concatenated into **many minified and code-split** `.css` files.

App Router vs Pages Router

Next.js uses file-system routing, which means the routes in our application are determined by how you structure your files

Pages Router

By adding React components to the `pages` directory, routes are automatically established.

▼ pages

about.tsx

index.tsx

team.tsx

Features of Pages Router

- Automatic Routes
- SEO Optimization
- Data Fetching Methods
- Custom Document & App Component

App Router

The App Router is Next.js's answer to advanced routing needs, allowing for more complex patterns and setups.

- All routs must be in /app folder.
- Every file that corresponds to a route must be named page.tsx.
- Every folder correspons to a path segment in the browser url.
- Example :

▼ app

▼ about

page.tsx

globals.css

layout.tsx

▼ login

page.tsx

page.tsx

- Routes :
  - http://localhost:3000/
  - http://localhost:3000/aboutus
  - http://localhost:3000/contactus
  - http://localhost:3000/blog
  - http://localhost:3000/products

Nested Routing

NextJS allows to create folders in folders for nested routing.

Example :

▼ app

▼ blog

page.tsx

▼ first

page.tsx

▼ second

page.tsx

layout.tsx

page.tsx

- Routes :
  - http://localhost:3000/blog/first
  - http://localhost:3000/blog/second

Dynamic Routes

We can use square brackets “[]” to define dynamic routes in NextJS.

Example :

▼ app

▼ products

▼ [productId]

page.tsx

page.tsx

- Routes :
  - http://localhost:3000/products/2

Nested Dynamic Routes

We can use square brackets inside square brackets in NextJS.

Example :

▼ app

▼ products

▼ [productId]

▼ [reviews]

page.tsx

page.tsx

page.tsx

- Routes :
  - http://localhost:3000/products/2/reviews/4

Pages Router vs App Router

Feature	App Router	Pages Router
Routing type	Server-centric	Client-side
Support for Server Components	Yes	No
Complexity	More complex	Simpler
Performance	Better	Worse
Flexibility	More flexible	Less flexible

Feature	App Router	Page Router
File-based routing	Uses nested folders to define routes	Files directly represent routes
Components	Server Components by default	Client Components by default
Data fetching	fetch function for data fetching	getServerSideProps, getStaticProps, getInitialProps
Layouts	Layouts can be nested and dynamic	Layouts are static
Dynamic routes	Supported, but syntax differs	Supported
Client-side navigation	Supported with router.push	Supported with Link component
Priority	Takes precedence over Page Router	Fallback if no matching route in App Router

Catch All Segments

- Open the page if any of the URL starts with the first folder name.
- Dynamic Segments can be extended to **catch-all** subsequent segments by adding an ellipsis inside the brackets `[...segmentName]`.
- In here “params” return an array of URL segments.

▼ app

▼ docs

▼ [...slug]

page.tsx

▼ optional-docs

▼ [[...slug]]

page.tsx

layout.tsx

page.tsx

- Routes :
  - http://localhost:3000/docs/any-segment
  - http://localhost:3000/docs/any-segment/another-segment

- http://localhost:3000/docs - go to 404 page

Optional Catch All Segments

- Catch-all Segments can be made **optional** by including the parameter in double square brackets: `[[...segmentName]]`.
- **Optional catch-all** segments is that with optional, the route without the parameter is also matched.
- Routes :
  - http://localhost:3000/optional-docs
  - http://localhost:3000/optional-docs/optional-segment-1

- When we navigate to an unknown URL, it will redirect us to default 404 not found page. So in order to customize this,
- Create a “not-found.tsx” file in “/src/app/” folder.
- This is more likely to file base routing in NextJS.
- When we want to invoke not-found page in a custom function, we can invoke the notFound() function from next/navigation.
- We can define individual not-found files related to each route in specific folders when using notFound() function.

More about Routing

- NextJS uses a file system based routing.
- Each folder represents a route segment add to a corresponding segment in the URL path.
- We must add page.tsx file in the respective folder to publicly access. Only the content that return from the page.tsx file send to the client.

Private Folders

- A private folder indicates that it is a private implementation detail and should not be considered by the routing system.
- The folder and all its subfolders are excluded from routing.
- Prefix the folder name with an underscore “_”.
- If we want to include an underscore segment in URL segments, we can prefix the folder name with “%5F,” which is the URL encoded form of an underscore.

Benefits of Private Folders

- For separating UI logic from routing logic.
- For consistently organizing internal files across a project.
- For sorting a grouping files.
- Avoiding naming conflicts with future NextJS file conventions.

Route Groups

- Allows us to logically group our routes and project files without affecting the URL path structure.
- We can mark a folder as a route group to exclude it from the URL paths.
- A route group can be created by wrapping a folder's name in parenthesis: `(folderName)`.

▼ app

▼ (auth)

▼ register

page.tsx

▼ login

page.tsx

layout.tsx

page.tsx

Layouts

- A page is a UI that is unique to a route.
- A Layout is a UI that is shared between multiple pages in the application.
- We can define a layout by default exporting a React components from a layout.tsx file.
- That component should accept a children prop that will be populated with a child page during rendering.
- The root layout is a mandatory layout for every NextJS application.
- This root layout file will be regenerated if you delete if when running the application.

▼ app

▼ aboutus

page.tsx

layout.tsx

page.tsx

Nested Layouts

- Also layouts can be nested.
- We can create a layout.tsx file in the folder we want and implement that layout file.
- Nested layouts allows us to create layouts that applies only to specific areas of our application.

▼ app

▼ products

▼ [productId]

layout.tsx

page.tsx

page.tsx

layout.tsx

page.tsx

- We can apply layouts to route groups to selectively apply a layout to certain segments while leaving others unchanged.
- We can add layout.tsx file in the root of the route group folder.

▼ app

▼ (auth)

▼ (with-layout-auth)

▼ register

page.tsx

▼ login

page.tsx

layout.tsx

layout.tsx

page.tsx

- This feature allows us to opt specific segments into a layout without altering the URL.
- The routes outside of the group do not share the layout.

Routing Metadata

- Ensuring proper Search Engine Optimization (SEO) is crucial for increasing visibility and attracting users.
- NextJS introduced the Metadata API which allows us to define metadata for each page.
- Metadata ensures accurate and relavant information is displayed when our pages are shared or indexed.
- There are 02 methods to configure metadata in layout.tsx or page.tsx file.

1. Export a static metadata object.
2. Export a dynamic generateMetadata function.

Metadata Rules

- Both layout.tsx and page.tsx files can export metadata. If defined in a layout, it applies to all pages in that layout, but if defined in a page, it applies only to that page.
- Metadata is read in order, from the root level down to the final page level.
- When there's metadata in multiple places for the same route, they get combined, but page metadata will replace layout metadata if they have the same properties.
- Both layout and page can have metadata but page metadata takes precedence if both are present.
- When multiple segments export metadata object the properties are merged to form the final metadata object. During merging the deepest segment takes priority.
- Dynamic metadata depends on the dynamic data such as the current route parameters, external data or metadata in parent segment.
- To define dynamic metadata, we export a generateMetadata function that returns a metadata object from a layout.tsx or page.tsx file.
- generateMetadata function can also be defined as an Async function.
- We cannot export both the metadata object and generateMetadata function from the same route segment.

title field in metadata

- The title field's primary purpose is to define the document title.
- It can be either a string or an object.
- When we using the title as an object, there are three keys,
    - absolute - title that completely ignores template set in the parent segment we can use this. When the child component has this, it replaces template metadata completely.
    - default - usefull when we want to provide a fallback title for child route segments that do not explicitly specify a title. If a child segment does not have a title defined it will fall back to the default title.

    - template - to create dynamic titles by adding a prefix or suffix we can use this. This property applies to child route segments and not the segment which it is defined. This affects when there are metadata exports in the child routes.

Navigation

- We can manually enter the URLs in the browser's address bar to navigate to the different routes.
- We can use UI elements like links to navigate. Clicking on them or by programmatic navigation after completing and action.

Link Component

- To enable client side navigation NextJS provides us with the link component.
- The "<Link>" component is a React component that extends the HTML "<a>" element, and it's the primary way to navigate between routes in NextJS.
- We can use this component to navigate any UI.
- To use it, we need to import it from "next/link".
- We can provide the desired route with "href".
- The "replace" prop of the Link component replaces the current history state instead of adding a new URL to the stack.

Navigating Programmatically

- For automatic redirections.
- We need to use useRouter hook from "next/navigation" for implementing the programmatical navigation. For this we must set the component as a client component.
- We need to create a variable assigning useRouter function and then we can use the push function in the router to navigate where we want. We can pass the route we want to navigate as in the href attribute.
- Also we can use replace("/") function to replace the history instead of navigating to a specific route.

- We can use back() function to navigate back in the browser history.
- There is a function called forward to navigate forward in the application.

Template File

- Layouts only mount part representing the content of the newly loaded page but keep all the common elements untouched.
- Layouts do not remount shared components resulting in better performance.
- When we want to create new instances for each of their children on navigation.
- Templates are similar to layouts in that they wrap each child layout or page.
- But, with templates, when a user navigates between routes that share a template, a new instance of the component is mounted, DOM elements are recreated, state is not preserved, and effects are re-synchronized.
- A template can be defined by exporting a default react component from a template.tsx file.
- Similar to layouts, templates also should accept a children prop which will render the nested segments in the route.
- We can replace layout with template file.

▼ app

▼ (auth)

▼ aboutus

page.tsx

▼ contactus

page.tsx

▼ services

page.tsx

template.tsx

layout.tsx

page.tsx

Loading File

- This file allows us to create loading states that are displayed to users while a specific route segment's content is loading.
- The loading state appears immediately upon navigation, giving users the assurance that the application is responsive and actively loading.
- We need to add loading.tsx file to designated folder to add a loading file.
- This file automatically wrap page.tsx file and all its nested children within a react suspense boundary.

Benefits of using loading.tsx file

- We can display the loading state as soon as a user navigates to a new route.
- NextJS allows the creation of shared layouts that remain interactive while new route segments are loading.

Error Handling

- There is a file named error.tsx for handling errors.
- We export a functional component named ErrorBoundary in this file.
- This is a client component so we need to add "use client" at the top of the component. Error boundaries have to be client components in NextJS.
- This component can also receieve the error object as a prop to display more information from the message property.

Benefits of error.tsx file

- Automatically wrap a route segment and its nested children in a React error boundary.
- Create error UI tailored to specific segments using the file system hierarchy to adjust granularity.

- Isolate the errors to affected segments while keeping the rest of the application functional.
- Add functionality to attempt to recover from an error without a full page reload.

Error Recovery

- Error object in the ErrorBoundary component comes with a property named “reset”. We can use that property through a callback function to retry rendering the component again.
- We need to convert our page.tsx file into client component as well when using this feature.
- Executing the reset function attempt to re-render the error boundary contents if successful the fallback error component is replace with the re-rendering component from page.tsx.

Handling Errors in Nested Routes

- Errors bubble up to the closest parent error boundary.
- An error.tsx file will cater to errors for all its nested child segments.
- By positioning error.tsx files at different levels in the nested folders of a route, we can achieve a more granular level of error handling.
- The placement of error.tsx file plays a crucial role determining the scope of error handling allowing for more precise control which part of the UI is affected when errors occur.

Handling Errors in Layouts

- The Layout is above the ErrorBoundary in hierarchy. So, the error boundary does not catch errors thrown here because it’s nested inside the layouts component.
- The error boundary will not handle errors thrown in a layout.tsx component within the same segment.

- We need to place the error.tsx file in the layout parent segment. We should move error.tsx file into the parent folder.
- The placement of the error.tsx file plays a pivotal role managing errors efficiently across different segments of our application.

Parallel Routes

- Parallel routes are an advanced routing mechanism that allows for the simultaneous rendering of multiple pages within the same layout.
- Parallel routes in NextJS are defined using a feature known as slots.
- Slots help structure our content in a modular fashion.
- To define a slot, we use the “@folder” naming convention.
- Each slot is then passed as a prop to its corresponding “layout.tsx” file.

Benefits of Parallel Routes

- A clear benefit of parallel routes is their ability to split a single layout into various slots, making the code more manageable.
- Independent route handling.
    - Each slot of our layout, such as about or contact, an have its own loading and error states.
    - This granular control is particularly beneficial in scenarios where different sections of the page loading at varying speeds or encounter unique errors.
- Sub-navigation.
    - Each slot of our dashboard can essentially function as a mini-application, complete with its own navigation and state management.
    - This is especially useful in a complex application such as our dashboard where different sections serve distinct purposes.

Unmatched Routes

Navigating from the UI

- In the case of navigation with the UI, NextJS retains the previously active state of a slot regardless of changes in the URL. It means the other slots remain unchanged with the previous state.

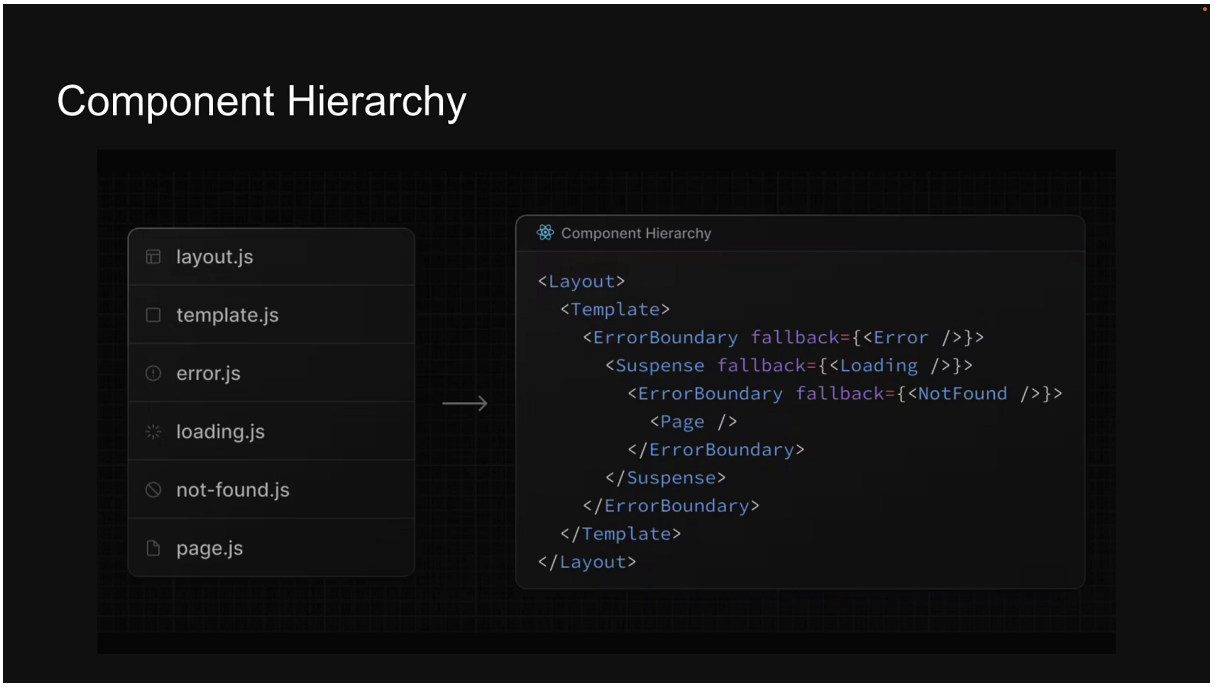
Page Reload

- NextJS immediately searches for a default.tsx file within each unmatched slot.
- The presence of this file is critical, as it provides the default content that NextJS will render in the user interface.
- If the default.tsx file is missing in any of the unmatched slots for the current route, NextJS will render a 404 error.
- We need to include a default content in the initial load for avoiding that when refreshing a recently loaded parallel route.
- The “default.tsx” file in NextJS serves as a fallback to render content when the framework cannot retrieve a slot’s active state from the current URL.
- We have entirely freedom to define the UI for unmatched routes. We can either mirror the content found in page.tsx or craft an entirely custom view.

Conditional Routes

- Parallel routing is a way for implementing conditional routing.
- This is like a fully separated code in the same URL in different environments. Like when user is authenticated the URL goes to Dashboard and if not authenticated it goes to Login Page.

Component Hierarchy of files in NextJS

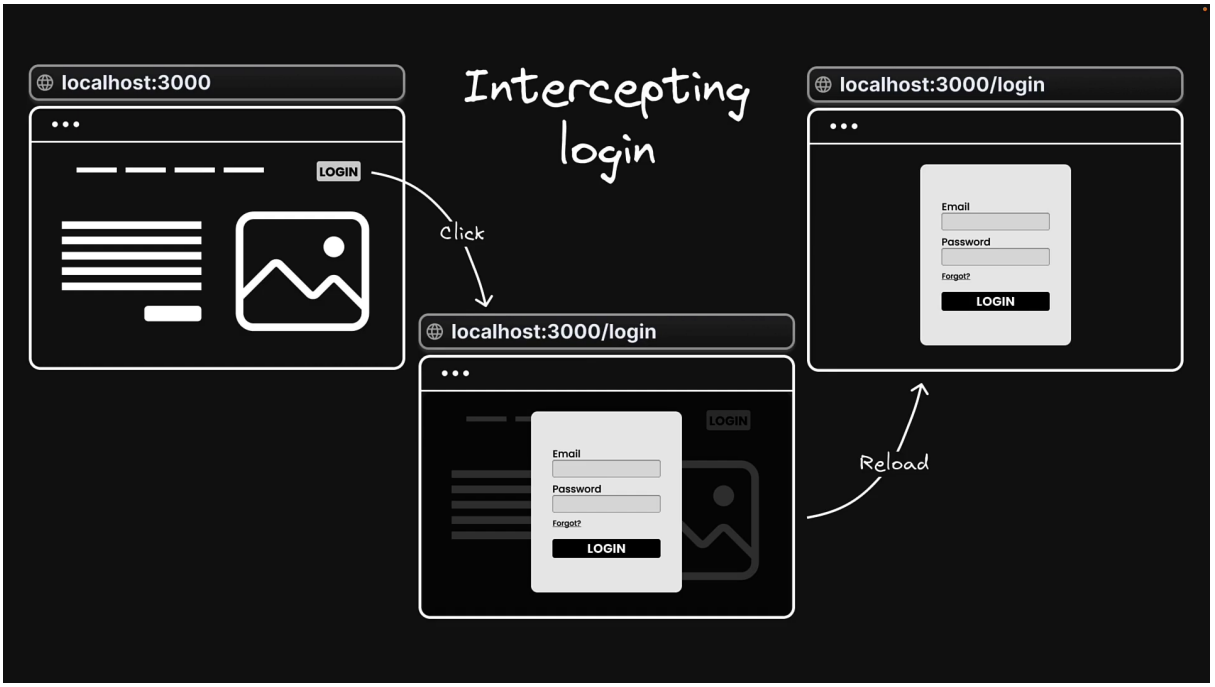


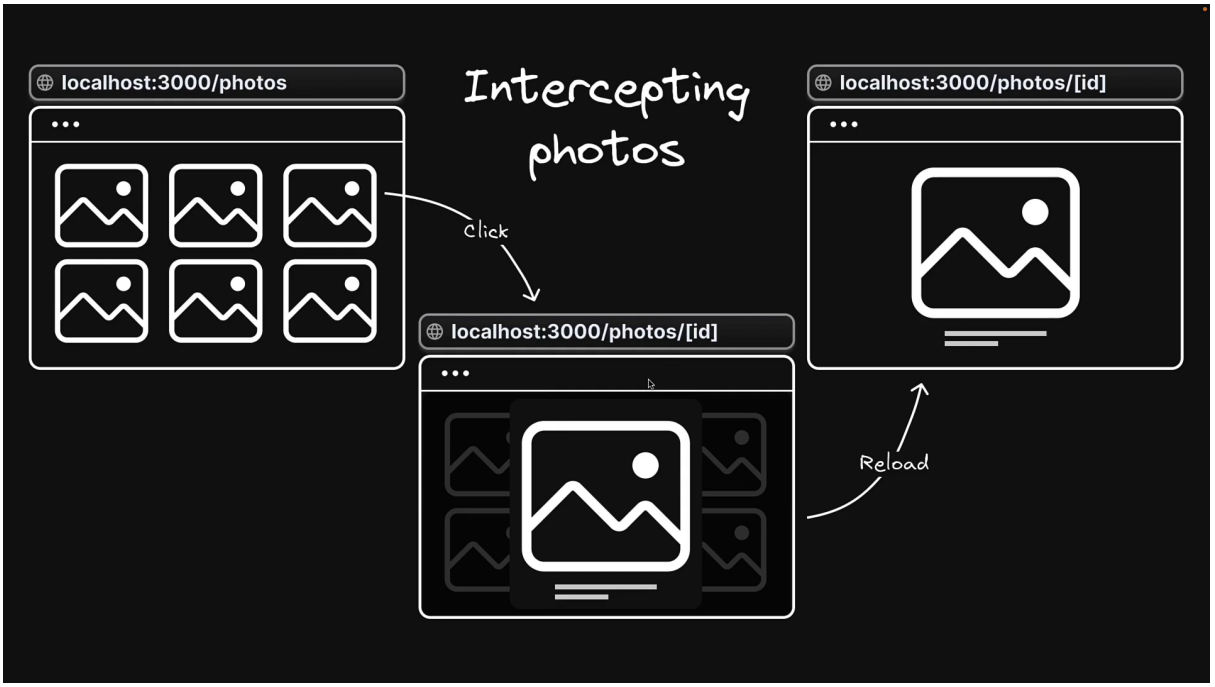
Advanced Routing Patterns

- Parallel Routes
- Intercepting Routes

Intercepting Routes

- Intercepting routs allow us to intercept or stop default behavior to present an alternate view or component when navigating through the UI, while still preserving the intended route for scenarios like page reloads.
- This can be useful if we want to show a route while keeping the context of the current page.





- We can click on some content like a button or a image it will show a modal of the route without effecting the ongoing context but if we reload or access the direct URL it will direct to the route directly.

- To create a intercepting route at the same level, we use a “.” within parentheses notation in a folder name. Like (.)folderName.
- When clicking on the link will go to the intercepted page and if we reload the page again, it will redirect to the original page.
- To intercept above one level above of the level, we have to use “(..)folderName”.
- To intercept above two levels above of the level, we have to use “(..)(..)folderName”. This convention has an issue and opened a bug ticket in there.
- To intercept segments from the root app directory, we have to use “(...)folderName”.
- Intercepting routes allow us to load a route from another part of our application within the current layout.

Route Handlers

- We can create custom request handlers for our routes using a feature called route handlers.
- Unlike page routes, which respond with HTML content, route handlers allow us to create RESTful endpoints, giving us full control over the response.
- There is no overhead of having to create and configure a separate server.
- Route handlers are also great for making external API requests.
- Route handlers run server-side, ensuring that sensitive information like private keys remains secure and never gets shipped to the browser.
- Route handlers are equivalent of API routes in Pages router.
- Route handlers must place within the “app” folder.
- File name in the Route handler must be “route.ts”.
- We can define an export a function corresponding to the HTTP verb. This must be a HTTP verb for executing.
- Similar to page routes, route handlers can be organized in folders and nested within sub folders.

- We must be mindful about potential conflicts between page routes and route handlers. We can move APIs to directory named as “api” to avoid these conflicts.
- We can define dynamic routes with square parenthesis “[]” as page routing.
- Handler function receives two parameters.
    - request object
    - context

URL Query Parameters

- We need request parameter to handle the query parameter effectively.
- We are actually dealing with NextRequest from “next/server”. This method provide convenient methods form managing query parameters with ease.
- We can use NextRequest..nextUrl.searchParams to get all parameters and then filter our query with get(”query”) method passing the query as argument.
- Query parameters are optional but they are incredibly useful for implementing search, sort and pagination functionalities for our data.

Redirects in Route Handlers

- We can use redirect function in “next/navigation” to do this.
- It give a status code as 307, a temporary redirect.

Headers in Route Handlers

- HTTP headers represent the metadata associated with an API request and response.
- These metadata can be classified into two categories.
    - Request Headers - these are sent by the client, such as a web browser, to the server. They contain essential information about the request, which helps the server understand and process it correctly.
        - User-Agent : identifies the browser and operating system to the server.

        - Accept : indicates the content types like text, video, or image formats that the client can process.
        - Authorization : used by the client to authenticate itself to the server.
    - Response Headers - these are sent back from the server to the client. They provide information about the server and the data being sent in the response.
        - Content-Type : indicates the media type of the response. It tells the client what the data type of the returned content is, such as text/html for HTML documents, application/json for JSON data, etc.
- By default request has two headers Accept and User-Agent.
- There are two ways to read headers in request.
    - request parameters - we can use Headers web API to extract the headers in the request. We must pass request.headers to this as an argument. Then we can pass the header name as a string to the get() function through request headers.
    - Using the headers function that NextJS provides - in there we can invoke headers() function to get headers and use get() function to get the specific header value by passing the header name as a string.
- Headers return from the headers() function are read only.
- To set headers, we need to return a new response with headers with setHeaders() method.
- The client renders the content differently based on the response header.

Cookies in Route Handlers

- Cookies are small pieces of data that a server sends to a user’s web browser.
- The browser may store the cookie and send it back to the same server with later requests.
- Cookies are mainly used for three purposes.
    - Session management like loggings and shopping carts.
    - Personalization like user preferences and themes.

    - Tracking like recording and analyzing user behavior.
- There are two options for setting cookies.
    - We can user Set-Cookie header to set a cookie in a response with a key value pair.
    - Using the cookies() function provided by NextJS. This cookies function supports many function like has, delete, getAll etc.

Caching in Route Handlers

- Route Handlers are cached by default when using the GET method with the response object in NextJS.
- When we reload the request in development mode, the time route will return the updated time but when it is in production mode returns the same time that we sent the first request. This is the caching behavior.
- How to inform NextJS that we do not want the response to be cached.
    - dynamic mode in Segment Config option. The default value of "dynamic" is "auto" by default which attempts to cache as much as possible but we set it to "force-dynamic" ensuring that the the handler is executed for each user request.
- There are three other ways to opt out of caching.
    - Using the request object with the GET method.
    - Employing dynamic functions like headers() and cookies().
    - Using any HTTP method other than GET.

MiddleWares

- Middleware in NextJS is a powerful feature that offers a robust way to intercept and control the flow of requests and responses within our applications.
- It does this at a global level significantly enhancing features like redirection, URL rewrites, authentication, headers and cookies management, and more.
- To create middleware in NextJS start by adding middleware.ts file in the src folder.

- Middleware allows us to specify paths where it will be active. There are two main approaches.
    - Custom matcher config
    - Conditional statements
- URL rewriting means change the response content by keeping the URL same.
- We can effectively use middlewares to manipulate headers and cookies also.

Rendering In NextJS

- Rendering is the process that transforms the code we write into user interfaces.
- In NextJS, choosing the right time and place to do this rendering is vital for building a performant application.
- React being the go-to library for developing Single Page Applications (SPAs).

Evolution Of Rendering in React

Client Side Rendering (CSR)

The method of rendering, where the component code is transformed into a user interface directly within the browser (the client), is known as client side rendering (CSR).

Drawbacks of CSR

- SEO - generating HTML that mainly contains a single div tag is not optimal for SEO, as it provides little content for search engines to index. Reliance on Java Script for rendering content on the client can significantly hurt SEO, as search engines might struggle to index the content properly.
- Performance - having the browser (the client) handle all the work, such as fetching data, computing the UI, and making the HTML interactive, can slow

things down. Users might see a blank screen or a loading spinner while the page loads. The user experience can suffer from slow load times, as the browser has to download, parse and execute Java Script before the user sees any meaningful content on the page.

- Each new feature added to the application increases the size of the Java Script bundle, prolonging the wait time for users to see the UI.

Server Side Solutions

- This significantly improves SEO because search engines can easily index the server-rendered content.
- Users can immediately see the page HTML content, instead of a blank screen or loading spinner.

Hydration

- Static HTML page initially served by the server is brought to life.
- During hydration, React takes control in the browser, reconstructing the component tree in memory based on the static HTML that was served.
- It carefully plans the placement of interactive elements within this tree. Then, React proceeds to bind the necessary Java Script logic to these elements.
- This involves initializing the application state, attaching event handlers for actions such as clicks and mouseovers, and setting up any other dynamic functionalities required for a fully interactive user experience.

- Server Side Solutions can be categorized into two strategies.
  - Static Site Generation - SSG
  - Server Side Rendering - SSR
- SSG occurs at build time, when the application is deployed on the server. This results in pages that are already rendered and ready to serve. It id ideal for content that does not change often, like blog posts.

- SSR on the other hand, renders pages on-demand in response to user requests. It is suitable for personalized content like social media feeds, where the HTML depends on the logged in user.
- Server-Side Rendering (SSR) was a significant improvement over Client-Side Rendering (CSR), providing a faster initial page loads and better SEO.

Drawbacks of SSR

- We have to fetch everything before we can show anything - components cannot start rendering and then pause or “wait” while data is still being loaded.
    - If a component needs to fetch data from a database or another source (like an API), this fetching must be completed before the server can begin rendering the page.
    - This can delay the server’s response time to the browser, as the server must finish collecting all necessary data before any part of the page can be sent to the client.
- We have to load everything before we can hydrate anything.
    - For successful hydration, where React adds interactivity to the server-rendered HTML, the component tree in the browser must exactly match the server-generated component tree.
    - This means that all the Java Script for the components must be loaded on the client before you can start hydrating any of them.
- We have to hydrate everything before we can interact with anything.
    - React hydrates the component tree in a single pass, meaning once it starts hydrating, it will not stop until it is finished with the entire tree.
    - As a consequence, all components must be hydrated before you can interact with any of them.
- This is all or nothing waterfall.
    - Having to load the data for the entire page.
    - Load the Java Script for the entire page, and
    - hydrate the entire page

create an “all or nothing” waterfall problem that spans from the server to the client, where each issue must be resolved before moving to the next one.

- This is inefficient if some parts of your application are slower than others, as is often the case is real-world applications

Drawbacks of SSR briefly

- Data fetching must be completed before the server can begin rendering HTML.
- The Java Script required for the components needs to be fully loaded on the client side before hydration process can start.
- All components have to be hydrated before they become interactive.

React Suspense SSR Architecture

- React Suspense SSR Architecture to address the above issues.
- Use the <Suspense> component to unlock two major features.
  - HTML streaming on the server
  - Selective hydration on the client.
- By wrapping a part of the page within <Suspense> component we instruct React, it does not need to wait main section data to be fetched to start streaming the HTML of the rest of the page. React will send a placeholder like a loading spinner instead of the complete component. Once the server is ready for the data for that section React sends additional HTML through the ongoing stream accompanied by inline script tag containing the minimal Java Script committed to correctly position that HTML. So **we do not have to fetch everything before we can show anything**.
- If a particular section delays the initial HTML, it can be seamlessly integrated into the stream later.
- Code splitting allows us to mark specific code segments as not immediately necessary for loading, signaling our bundler to segregate them into separate “<script>” tags.
- Using “React.lazy” for code splitting enables us to separate the main section’s code from the primary Java Script bundle.

- The Java Script containing React and the code for the entire application, excluding the main section, can now be downloaded independently by the client, without having to wait for the main section's code.
- By wrapping the main section within "<Suspense>", you've indicated to React that it should not prevent the rest of the page from not just streaming but also from hydrating.
- This feature, called **selective hydration** allows for the hydration of sections as they become available, before the rest of the HTML and the Java Script code are fully downloaded.
- Thanks to Selective Hydration, a heavy piece of Java Script does not prevent the rest of the page from becoming interactive.
- In scenarios where multiple components are awaiting hydration, React prioritizes hydration based on user interactions.

Drawbacks of Suspense SSR

- Even though Java Script code is streamed to the browser asynchronously, eventually, the entire code for a web page must be downloaded by the user.
- The current approach requires that all React components undergo hydration on the client-side, irrespective of their actual need for interactivity.
- In spite of servers' superior capacity for handling intensive processing tasks, the bulk of Java Script execution still takes place on the user's device.

Suspense for SSR Challenges

- Increased bundle size leading to excessive downloads for users.
- Unnecessary hydrating delaying interactivity.
- Extensive client-side processing that could result in poor performance.

React Server Components (RSC)

- React Server Components (RSC) represent a new architecture designed by the React team.

- This approach aims to leverage the strengths of both server and client environments, optimizing for efficiency, load times, and interactivity.
- The architecture introduces a dual-component model.
  - Client Components
  - Server Components
- This distinction is not based on the functionality of the components but rather on where they execute and the specific environments they are designed to interact with.

Client Components

- Typically rendered on the client-side (CSR) but, they can also be rendered to HTML on the server (SSR), allowing users to immediately see the page's HTML content rather than a blank screen.
- Components that primarily run on the client but can (and should) also be executed once on the server as an optimization strategy.
- Client components have access to the client environment, such as the browser, allowing them to use state, effects and event listeners to handle interactivity and also access browser-exclusive APIs like geolocation or localStorage, allowing us to build UI for specific use cases.

Server Components

- Server components represent a new type of React component specifically designed to operate exclusively on the server.
- Unlike Client components, their code stays on the server and never downloaded to the client.

Benefits of Server Components

- Reduce Bundle Size - server components do not send code to the client, allowing large dependencies to remain server-side. This benefits users with slower internet connections or less capable devices by eliminating the need to download, parse and execute Java Script for these components.

Additionally, this removes hydration step, speeding up app loading and interaction.

- Direct access to server-side resources - by having direct access to server-side resources like databases and file systems, Server Components enable efficient data fetching and rendering without needing additional client side processing. Leveraging the server's computational power and proximity to data sources, they manage compute-intensive rendering tasks and send only interactive pieces of code to the client.
- Enhanced Security - Server Components' exclusive server-side execution enhances security by keeping sensitive data and logic, including tokens and API keys, away from the client-side.
- Improved data fetching - server components enhance data fetching efficiency. Typically, when fetching data on the client-side using useEffect, a child component cannot begin loading its data until the parent component has finished loading its own data. This can cause for poor performance. The main issue is not the round trips themselves, but that these round trips are made from the client to the server. Server Components enable applications to shift these sequential round trips to the server side. By moving this logic to the server, request latency is reduced, and overall performance is improved, eliminating client-server "waterfalls".
- Caching - Rendering on the server enables caching of the results, which can be reused in subsequent requests and across different users. This approach can significantly improve performance and reduce costs by minimizing the amount of rendering and data fetching required for each request.
- Faster initial page load and first contentful paint - Initial page load and first contentful paint (FCP) are significantly improved with server components. By generating HTML on the server, pages become immediately visible to users without the delay of downloading, parsing and executing Java Script.
- Improved SEO - regarding Search Engine Optimization (SEO), the server-rendered HTML fully accessible to search engine bots, enhancing the indexability of our pages.
- Efficient streaming - server components allow the rendering process to be divided into manageable chunks, which are then streamed to the client as soon as they are ready. This approach allows users to start seeing parts of

the page earlier, eliminating the need to wait for the entire page to finish rendering on the server.

- Server Components take charge of data fetching and static rendering, while Client Components are tasked with rendering the interactive elements of the application.
- The RSC architecture enables react applications to leverage the best aspects of both server and client rendering, all while using a single language, a single framework, and a cohesive set of APIs.

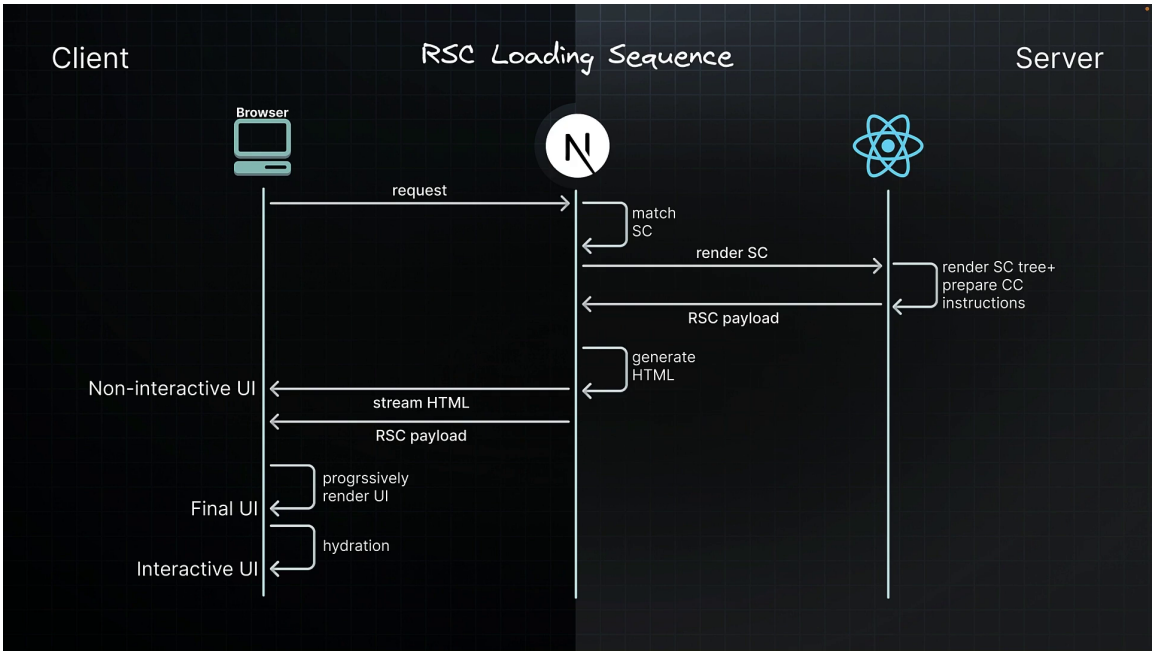
RSC and NextJS

- The App Route in NextJS is built around the RSC architecture.
- All the RSC features can be seen in NextJS.
- By default, every component in a NextJS application considered a Server Component.
- We must use “use client” directive at the top of the Client Component.
- Client Components are primarily executed in the client and have access to browser APIs but they are also pre-rendered once on the server to allow the user to immediately see pages HTML content rather than a blank screen.

RSC Rendering Lifecycle

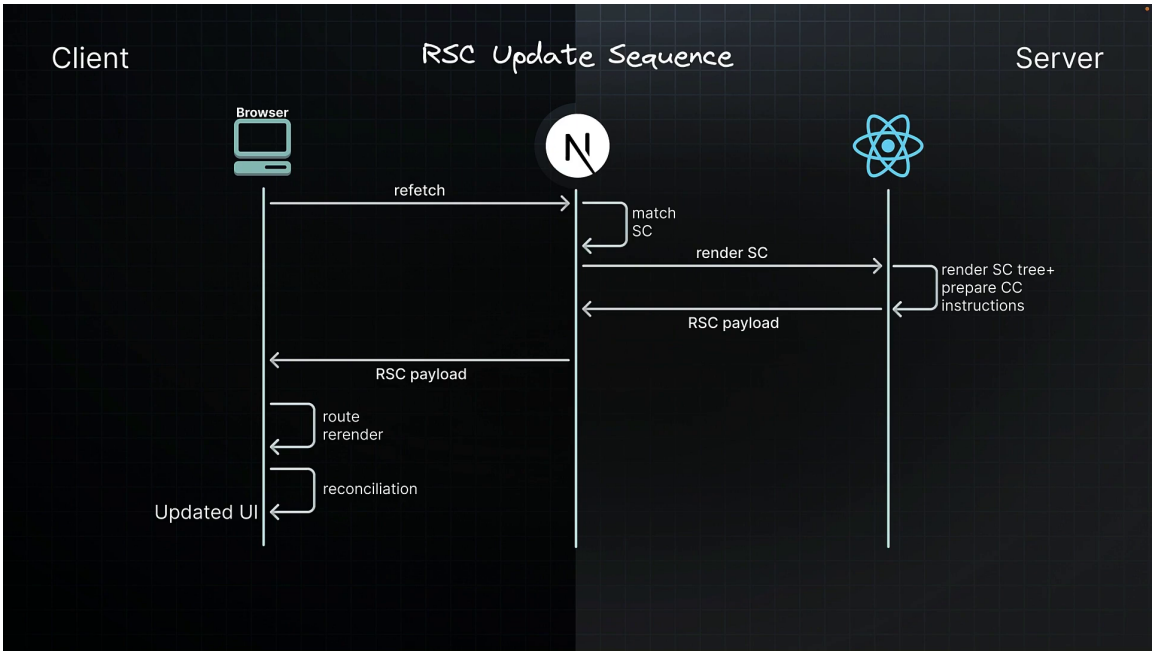
- For React Server Components (RSC), it is important to consider three elements. Our browser (the client), and on the server side, NextJS (the framework) and React (the library).
- The steps of NextJS and React Initial Loading Sequence.
    - When the browser request a page, the NextJS App Router matches the requested URL to a Server Component.
    - NextJS instruct React to render that Server Component.
    - React renders the Server Component and any child components also server components converting them to a special JSON format known as the RSC payload.
    - During this rendering if any server component suspense react pauses rending of that sub directory and sends a placeholder value instead.

- Meanwhile Client Components are prepared with instructions for later in the lifecycle.
- NextJS Uses the RSC payload which includes the client component instructions to generate HTML on the server. This HTML is streamed to our browser to immediately show a fast non-interactive preview of the route.
- Alongside NextJS stream the RSC payload as React renders each unit of UI.
- In the browser NextJS processes streamed React response. React uses the RSC payload and client component instructions to progressively render the UI.
- Once all the components and Server components would have been loaded, the final UI state is presented to the user.
- Client Components undergo hydration transforming our application from static into an interactive experience.



- The steps of NextJS and React Updating Sequence for refreshing parts of the application.
  - The browser request a refetch of a specific UI such as a full route, NextJS processes the request and matches it to the requested server component.

- NextJS instructs React to render the component tree.
- React renders the components similar to the initial loading. But unlike the initial sequence there is no HTML generation for updates.
- NextJS progressively stream the response data back to the client.
- On receiving the streamed response NextJS trigger a re-render of the new route using the new output.
- React reconciles or merges the new rendered output with the existing components on screen.
- Since the UI description is an special JSON format and not HTML react can update the DOM while preserving crucial UI updates such as focus or input values.



- We have three server rendering strategies.
  - Static rendering
  - Dynamic rendering
  - streaming

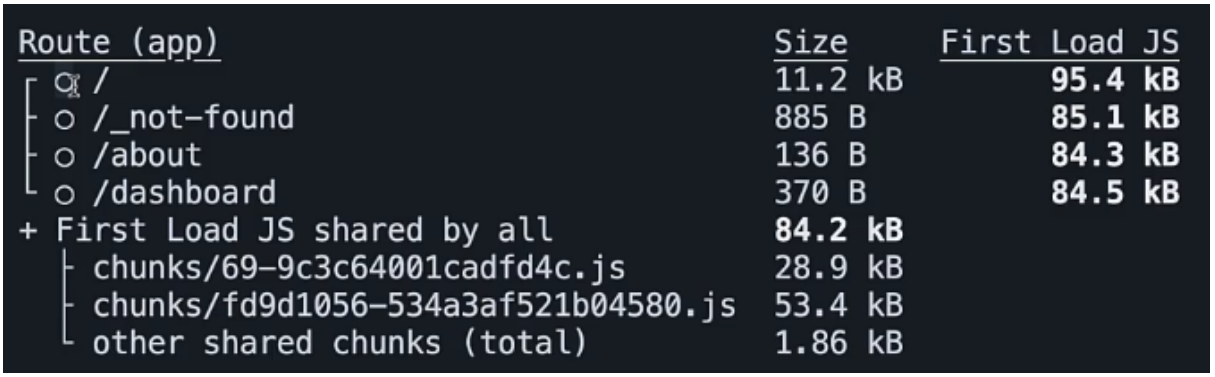
Static Rendering

- Static Rendering is a server rendering strategy where we generates HTML pages at the time of building our application.
- This approach allows the page to be built once, cached by a CDN, and served to the client almost instantly.
- This optimization also enables us to share the results of the rendering work among different users, resulting in a significant performance boost for our application.
- Static rendering is particularly useful for blog pages, e-commerce product pages, documentation and marketing pages.
- Static Rendering is the default rendering process of the app router.
- All routes are automatically prepared at built time without additional setup.

Difference between Production Server vs Development Server

- For production, an optimized build is created once, and we deploy that build.
- A development server, on the other hand, focuses on the developer experience.
- We cannot afford to build our app once, make changes, rebuild, and so on.
- For production builds, a page will be pre-rendered once when we run the build command.
- In development mode, a page will be pre-rendered for every request.
- We do not have to care about static rendering in development mode because it renders in every request. We can see it when we using a date in our application. It updates in every request in development server but not in production server.

- NextJS provides a legend for the type of routes generated.



- .next folder contains the content which need to serve the application in production server. It contains folders like server, static and etc. In server we can see the app folder and there are specific files.
    - HTML pages - Client Components also pre-rendered and it is an optimization step so there are Server Component and Client Component HTML files.
    - rsc files for RSC payload for each route - this special JSON format generated by react for each route is a compact string representation of the virtual DOM. For a Server Component the payload includes render result of the server component. For a Client Component the payload includes placeholders or instructions where client components should be rendered along with references to their Java Script files. The rsc file contains references for Java Script files for reconciliation and hydration.

Prefetching In NextJS

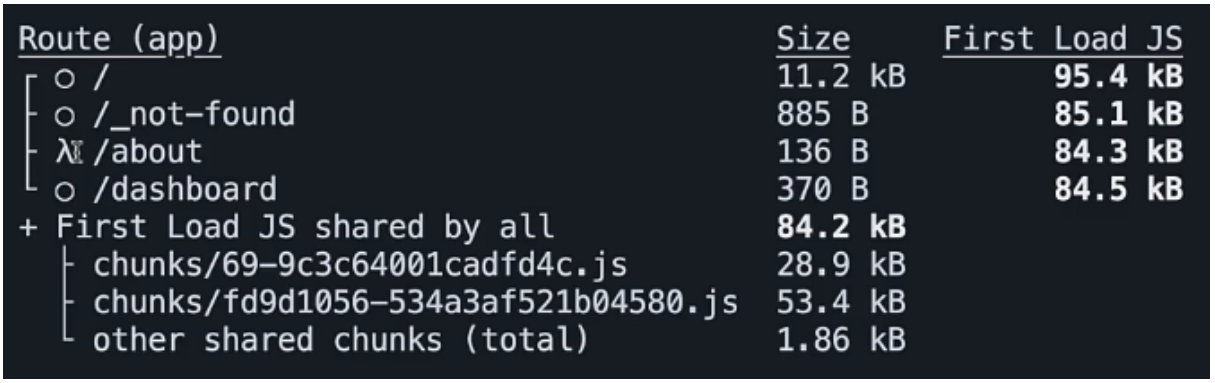
- Prefetching is a technique used to preload a route in the background before the user navigates to it.
- Routes are automatically prefetched as they become visible in the user's viewport, either when the page first loads or as it comes into view through scrolling.
- For static routes, the entire route is prefetched and cached by default.
- If we directly render the route then it will download the HTML file.

Static Rendering Summary

- Static Rendering is a strategy where the HTML is generated at build time.
- Along with the HTML, the RSC payload is created for each component, and Java Script chunks are produced for client-side component hydration in the browser.
- If we navigate directly to a page route, the corresponding HTML file is served.
- If we navigate to a route from a different one, the route is created on the client side using the RSC payload and Java Script chunks, without any additional requests to the server.
- Static rendering is great for performance and use cases include blogs, documentation and marketing pages etc.

Dynamic Rendering

- Dynamic Rendering is a server rendering strategy where routes are rendered for each user at request time.
- It is useful when a route has data that is personalized to the user or contain information that can only be known at request time, such as cookies or URL's search parameters.
- News websites, personalized e-commerce pages and social media feeds are some examples where dynamic rendering is beneficial.
- During rendering, if a dynamic function is discovered, NextJS will switch to dynamically rendering the whole route.
- In NextJS, these dynamic functions are cookies(), headers() and searchParams. Using any of these will opt the whole route into dynamic rendering at request time.
- In this example, about page has the dynamic rendering and it has the lambda symbol with it.



- Dynamically rendered pages are not statically rendered in build time. So we cannot see it's HTML in the ".next/server/app" folder.

Dynamic Rendering Summary

- Dynamic Rendering is a strategy where HTML is generated at request time.
- NextJS automatically switches to dynamic rendering when it comes across a dynamic function in the component, such as cookies(), headers() or the searchParam object.
- This form of rendering is great for when we need to render HTML personalized to a user, such as social media feed.
- As a developer, we do not need to choose between static and dynamic rendering. NextJS will automatically choose the best rendering strategy for each route based on the features and APIs used.

Streaming

- Streaming is a strategy that allows for progressive UI rendering from the server.
- Work is divided into chunks and streamed to the client as soon as it is ready.
- This enables users to see parts of the page immediately, before the entire content has finished rendering.
- Streaming significantly improves both the initial page loading performance and the rendering of UI elements that rely on slower data fetches, which

would otherwise block the rendering of the entire route.

- Streaming is integrated into the NextJS App Router by default.
- With the App Router, we can use async-await with react components.
- We can wrap components with <Suspense> component from react to stream a component that delays to load.
- We can give a loading component in fallback prop in the <Suspense> component.

Server and Client Composition Patterns

Server Components

- Fetching data.
- Directly accessing backend resources.
- Protecting sensitive information (like access tokens and API keys) on the server.
- Keeping large dependencies server-side, which helps in reducing client-side Java Script.

Client Components

- Adding interactivity.
- Handling event listeners (such as onClick(), onChange(), etc.).
- Managing state and lifecycle effects (using hooks like useState(), useReducer(), useEffect()).
- Using browser-exclusive APIs.
- Using custom hooks.
- Using React Class components.

Server-only code

- Certain code is intended to execute only on the server.

- We might have modules or functions that use multiple libraries, use environment variables, interact directly with a database or process confidential information.
- Since Java Script modules can be shared, it is possible for code that is meant only for the server to unintentionally end up in the client.
- If server-side code gets bundled into the client-side Java Script, it could lead to a extended bundle size, expose secret keys, database queries and sensitive business logic.
- It is crucial to separate server-only code from client-side code to protect the application's security and integrity.
- We can use **server-only package** to provide a build-time error if developers accidently import one of these modules into a client component.
- To use this package.
  - Install Package.

```
npm i server-only
```

  - Import package.

```
import “server-only”
```

Third-party Packages

- Third party packages in the ecosystem are gradually adapting, beginning to add the “use client” directive to the components that rely on client-only features, marking a clear distinction in their execution environment.
- Many components from npm packages, which traditionally leverage client-side features, have not yet integrated this directive.
- The absence of “use client” means that while these components will function correctly in Client Components, they may encounter issues or might not work at all within Server Components.
- To address this, we can wrap third-party components that rely on client-only features in our own Client Components.

- To solve this we can apply “use client” directive on the top of the component. So in that case we cannot use that component as a server-side component.
- To resolve this, we must encapsulate third-party components that depend on client only features within our own Client Components. So we can simply invoke that component in the Server Component without converting it to a Client Component by adding “use client” directive.

Context Providers

- Context Providers are typically rendered near the root of an application to share global application state and logic. (For example, Application theme)
- However, since React context is not supported in Server Components, attempting to create a context at the root of our application will result in an error.
- To address this, we can create a context and render its provider inside a separate Client Component. Even though we wrap the rest of the application within a client component server components down the tree will remain server components.

Client-only code

- Just as it is important to restrict certain operations to the server, it is equally important to confine some functionality to the client side.
- Client-only code typically interacts with browser-specific features like the DOM, the window object, localStorage etc which are not available on the server.
- Ensuring that such code is executed only on client-side prevents errors during server-side rendering.
- To prevent unintended server-side usage of client-side code, we can use a package called client-only.
- To install client-only package.
    - Install the package

```
npm install client-on
```

- Import the package.

Client Component Placement

- To compensate for server components not being able to manage state and handle interactivity, we need to create client components.
- It is recommended to place the client components lower in our component tree.
- When we add “use client” to a component. It not only make that component a client component but also affect every child component in the component tree below it.

Supported and Unsupported patterns of interleaving Server and Client Components

- Server Component can have another Server Component in it.
- Client Component can have another Client Component in it.
- A Server Component can have a Client Component in it.
- A Client Component cannot have a Server Component in it. Because any component such as a Server Component nested inside a Client Component is automatically converted into a Client Component. Since Client Components are rendering after Server Components, we cannot import a Server Component into a Client Component module as it will require a new request back to the server. This pattern is not supported in NextJS.
- Instead of nesting a Server Component in the Client Component, we can pass it as a prop to the Client Component. A common pattern as a React children prop placing in a slot.

Data Fetching in NextJS

- App Router uses the React Server Components (RSC) architecture, which allows us to fetch data using either server components or client components.
- Its advantageous to fetch data using server components, as they have direct access to server-side resources such as databases or file systems.
- This not only taps into server's computational power and proximity to data sources for efficient data fetching and rendering but also minimizes the need for client-side processing.
- Server components supports various configurations for caching, revalidating and optimizing data fetching.
- On the client side, data fetching is typically managed through third-party libraries such as TanStack Query offers its own robust APIs.

Fetching Data with Server Components

- The RSC architecture in the app router introduces support for async and await keywords in server components.
- This allows us to use the familiar Java Script await syntax by defining our component as an asynchronous function.
- We do not need to use useState() to fetch data in Server Components.

Loading and Error States

- In React we manage these states by creating separate variables and conditionally rendering UI based on their values.
- To implement Error state, define and export a React component in loading.tsx.
- For handling errors, define and export a React component in error.tsx. Error handling components are client components.

Caching Data in NextJS

- NextJS extends the native fetch API and automatically caches the return values of fetch. This caching improves performance and reduces costs.

- The initial data fetched from the server is stored that is called data cache on the server and reused for every subsequent request.
- This eliminates the need to repeatedly query our JSON server.

What is the Data Cache?

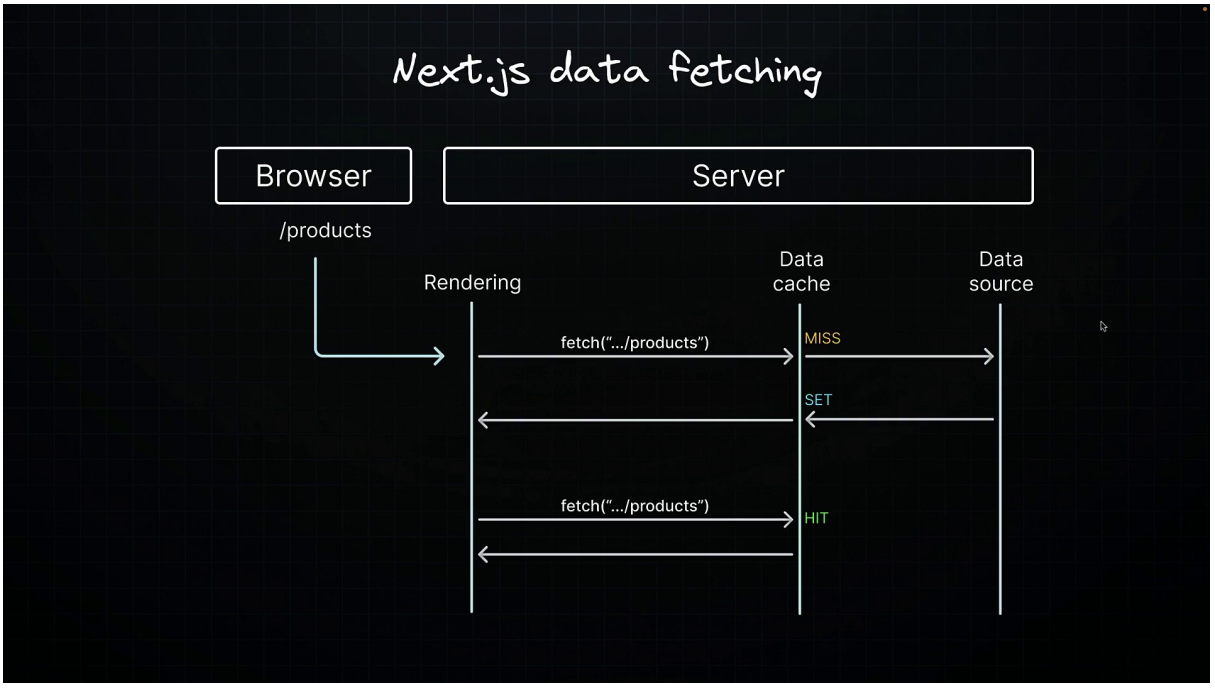
It is a server-side cache that persists the results of data fetches across incoming server requests and deployments.

Why it is required?

The data cache improves app performance and reduces the costs by eliminating the need to re-fetch data from our data source with every request.

How does it work?

First time request is made during rendering NextJS checks a data cache for a cached response if a cached response is found it is returned immediately. If a cached response is not found the result is made to the data source and the result is stored in the data cache. For subsequent fetch request with the same URL and options, the cached value is returned by passing the need to contact the data source.



- We can found this cached fetch requests in .next/fetch-cache folder.

- This is a server side persistent cache not the browser cache.

Opt Out Caching

- For individual data fetches, we can opt out of caching by setting the **cache** option to **no-store**.
- This ensures data is directly fetched from the data source every time fetch is called.
- Once we specify the **no-store** option for a fetch request, subsequent fetch requests will also not be cached.
- When we place two fetch request one above one in the code, if the first request is a normal fetch and the second is a no-store one, only second request will be fetched again but if we write the no-store request first and write normal request in second, then both request data from the data source.
- The route segment configuration can be apply by adding “export const fetchCache = “default-cache”” on top of the component. This can be followed to avoid above behavior of fetch requests.
- By default, NextJS will cache fetch() requests that occur before any dynamic functions (cookies(), headers(), searchParams) are used and will not cache requests found after dynamic functions.
- NextJS will not cache any request after dynamic function (cookies(), headers(), searchParams) has been invoked.

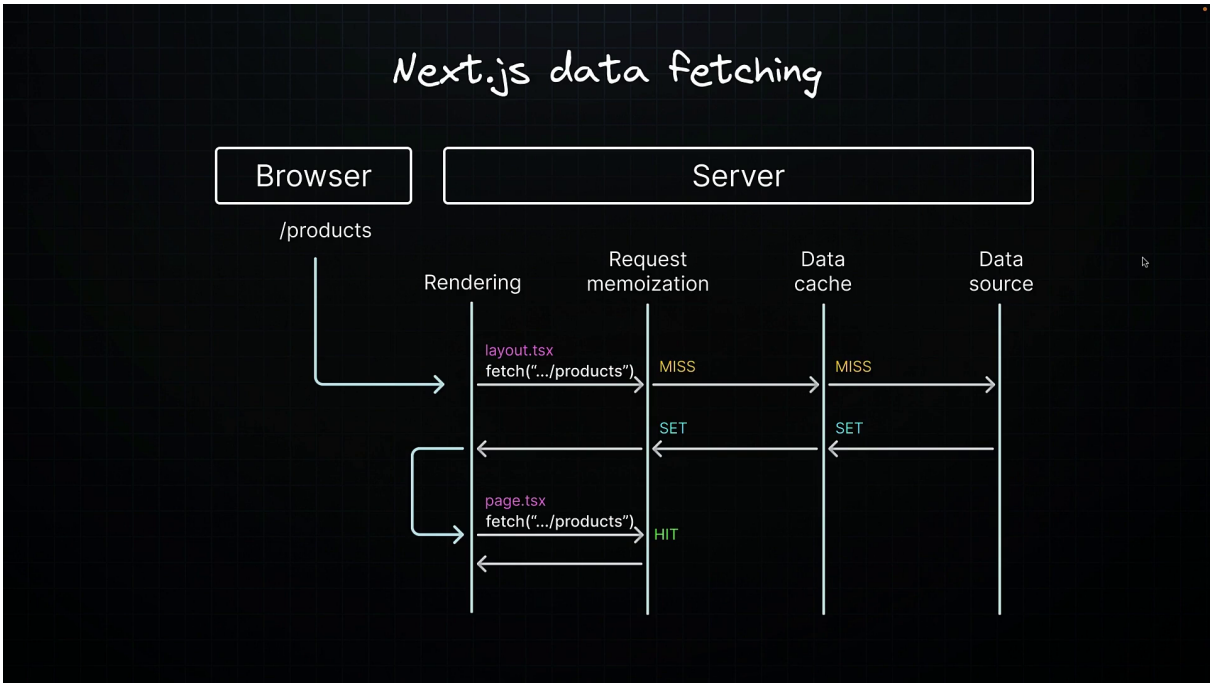
Request Memoization

- Request memoization is a technique that deduplicates requests for the same data within a single render pass.
- This approach allows for re-use of data in a React Component tree, prevents redundant network calls and enhances performance.
- For the initial request, data is fetched from an external source and the result is stored in memory.
- Subsequent request for the same data within the same render pass retrieve the result from memory, bypassing the need to make the request again.

- This optimization not only enhances performance but also simplifies data fetching within a component tree.
- When the same data is needed across different components in a route (e.g. in a Layout, Page and multiple components), it eliminates the need to fetch data at the top of the tree and pass props between components.
- Instead, data can be fetched directly within the components that require it, without concerns about the performance implications of multiple network requests for the same data.

Steps of Request Memoization

When we navigate to /products in the browser, the layout component initiate a fetch, it checks in memory to see a request with the same URL and options has already been made. Finding none it then checks the data cache which also shows no result. It then fetches the data from the JSON server stores the result in the data cache and in-memory and return it into the layout component. When the layout renders it proceeds to render the page component nested inside. This is still in the same render phase and this is not a new page reload. The page component initiates a fetch request, since the URL and options are the same as the one already in-memory the result of that request return to the page component. There is no need to check the data cache nor making additional requests to the data source improving overall performance.



- Request memoization is a React feature, not specifically a NextJS feature.
- Memoization only applies to the GET method in fetch requests.
- Memoization only applies within the React Component tree. It does not extend to fetch requests in Route Handlers as they are not part of the React component tree.
- For cases where fetch is not suitable (e.g. some database clients, CMS clients or GraphQL clients), you can use the React cache function to memoize functions.

Caching in NextJS

- By default, NextJS caches all fetch requests in the data cache, which is a persisitent HTTP cache on the server.
- This optimize pages such as a blog post where the content rarely changes.
- We also know that we can opt out of caching
  - by using the cache: "no-store" option in a fetch request
  - by using a dynamic function before making the fetch request
  - by using a route segment config like fetch-cache or dynamic
- A news website is a great example where we want to make sure we are fetching the latest data at all times.
- This approach seems binary. Either caching or no caching.
- In real-world applications, there are scenarios where a middle ground is required.
- For example, an event listings page might have event details such as schedule or venue information that change occasionally
- In this case, it is acceptable to fetch updated data once every hour as freshness is not critical.
- For such scenarios, NextJS allows us to revalidate the cache.

Revalidation

- Revalidation is the process of purging the Data Cache and re-fetching the latest data.
- Time-based revalidation - NextJS automatically revalidates data after a certain amount of time has passed.
- We can set the revalidate route segment configuration to establish the default revalidation time for a layout or page by adding “export const revalidate = 10” to the top of the component code. This option will not override the revalidate value set by individual requests.
- Regarding the revalidation frequency, the lowest revalidate time across each layout and page of a single route will determine the revalidation frequency of the entire route. This ensures that the child pages are revalidated as frequently as their parent layouts.

Data Fetching in Client Components

- We can also use Client Components for the data fetching but we cannot get features like request memoization, caching and revalidation. For that we need to rely on a library like TanStack Query.
- We can also call route handlers for external APIs.

NextJS 15 Changes

- Caching Strategy.
- Request specific APIs are now asynchronous.
- Smooth upgrades with @next/codemode CLI.
- React 19 support.
- Static route indicator.
- Support for next.config.ts.
- <Form> component for handling forms.