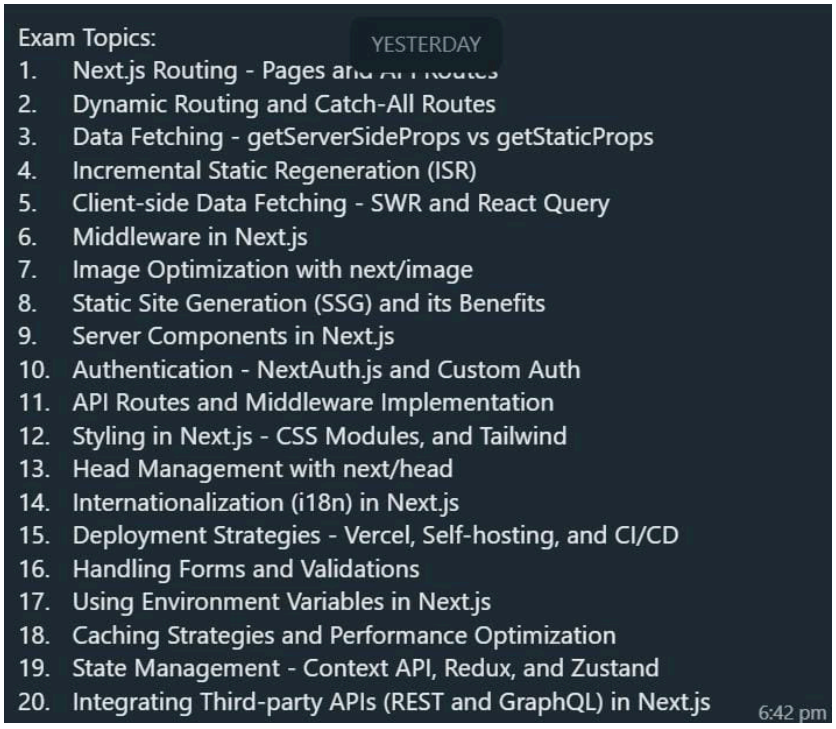
MASTERING NEXT.JS ADVANCED TECHNIQUES

An In-Depth Guide to Routing, Data Management, and Performance Optimization

Table of Contents



1. Introduction to Next.js 15 Routing: App Router vs Pages Router

What is App Router in Next.js 15?

The App Router is a modern routing approach introduced in Next.js 15, designed to efficiently handle complex, large-scale applications. It supports powerful features like layouts, nested

routes, and enhanced server-side component handling, making it ideal for applications with dynamic content and modular architectures.

Key Features of App Router:

- **Layouts:** Group pages into layouts to create reusable UI components across multiple routes.
- **Nested Routing:** Organize routes in a hierarchical structure for cleaner, more scalable applications.
- **Server-Side Components:** Efficiently load server-rendered components, enhancing performance.
- **Error Boundaries:** Advanced error handling and fallback UI for better resilience.

What is Pages Router?

The Pages Router (legacy) uses a simpler, file-based routing system, mapping files inside the pages/ directory to routes. While it's easy to implement, the Pages Router is limited for large, complex applications that require dynamic or nested routes.

Key Features of Pages Router:

- **File-Based Routing:** Automatically maps files in the pages/ directory to URL routes.
- **Simple Structure:** Suitable for smaller applications or when dynamic routing is not necessary.

Folder Structure Comparison:

App Router:

```
app/
  ├── layout.js         // Shared layout for pages
  ├── page.js           // Maps to '/'
  ├── about/
  │    └── page.js      // Maps to '/about'
  ├── blog/
  │    └── page.js      // Maps to '/blog'
  └── dashboard/
       └── page.js      // Maps to '/dashboard'
```

Pages Router:

```
pages/
  ├── index.js          // Maps to '/'
  ├── about.js          // Maps to '/about'
  ├── blog.js           // Maps to '/blog'
  └── dashboard.js      // Maps to '/dashboard'
```

Key Differences Between App Router and Pages Router:

Feature	Pages Router (Traditional)	App Router (Latest, Next.js 13+)
Folder for Routes	`pages/`	`app/`
Dynamic Routes	`pages/[id].js` for dynamic routes	`app/[id]/page.js` for dynamic routes
Layouts	No built-in layout system	Supports layouts, which can be nested
API Routes	`pages/api/` for API routes	`app/api/` for API routes
File Structure	Simpler, one-to-one file-to-route mapping	More flexible, supports layouts and nesting
React Server Components	Not supported natively	Supports React Server Components
Server-Side Rendering	Limited, based on page-level SSR	Full support for SSR, streaming, and suspense

2. Dynamic Routing and Catch-All Routes in App Router and Pages Router

Dynamic Routing in App Router

In the App Router, dynamic routes are defined by wrapping the dynamic part of the URL in square brackets ([param]). This feature helps create pages for dynamic content like product pages, user profiles, etc.

Example:

```
// app/[id]/page.js
export default function Product({ params }) {
  return <div>Product ID: {params.id}</div>;
}
```

Dynamic Routing in Pages Router

In the Pages Router, dynamic routes are also defined using square brackets, but the structure is simpler.

Example:

```
// pages/[id].js
export default function Product({ params }) {
  return <div>Product ID: {params.id}</div>;
}
```

Catch-All Routes in App Router

Catch-all routes capture multiple or optional parameters using [[...param]] (optional) or [...param] (required).

Example:

```
// app/[...slug]/page.js
export default function SlugPage({ params }) {
  return <div>Slug: {params.slug.join('/')}</div>;
}
```

Catch-All Routes in Pages Router

In Pages Router, catch-all routes work similarly with [...param] to capture any additional path segments.

Example:

```
// pages/[...slug].js
export default function SlugPage({ params }) {
  return <div>Slug: {params.slug.join('/')}</div>;
}
```

3. Data Fetching in Next.js 15: getServerSideProps vs getStaticProps

getServerSideProps (SSR - Server-Side Rendering)

getServerSideProps enables data fetching on every request, ensuring that the data displayed is always fresh. It's ideal for dynamic content that changes frequently.

Example:

```
// app/[id]/page.js
export async function getServerSideProps({ params }) {
  const res = await fetch(`https://api.example.com/product/${params.id}`);
  const data = await res.json();
  return { props: { data } };
}
```

getStaticProps (SSG - Static Site Generation)

getStaticProps generates static content at build time, offering faster page loads for content that doesn't change frequently.

Example:

```
// app/product/page.js
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/products');
  const data = await res.json();
  return { props: { data } };
}
```

Comparison Table:

Feature	getServerSideProps	getStaticProps
Fetching Time	On each request	At build time
Best for	Dynamic content	Static content
Data Freshness	Always up-to-date	Can become outdated
Rebuild Behavior	Runs on every request	Runs only during build/re-deploy
Use Case	Frequent data changes	Content that doesn't change often
Performance	Slower	Faster

Feature	getServerSideProps	getStaticProps	ISR
Rendering Time	Server	Build Time	Build + Runtime
Data Fetch	Har Request	Ek Martaba Build Mein	Scheduled Revalidations
Use Case	Dynamic Data	Static Data	Mixed Data

4. Incremental Static Regeneration (ISR) in Next.js 15

What is ISR?

ISR allows you to statically generate pages at build time and then update them after deployment without rebuilding the entire site. This feature enables static pages with up-to-date data, improving performance and scalability.

Example:

```
// app/[id]/page.js
export async function getStaticProps() {
  const res = await fetch(`https://api.example.com/product/1`);
  const data = await res.json();
  return {
    props: { data },
    revalidate: 60, // Revalidate after 60 seconds
  };
}
```

Flowchart for ISR:

1. Static Page is Generated
2. Revalidation After Set Time (e.g., 60s)
3. Content Update (when new data is available).

Aspect	getServerSideProps	getStaticProps	Incremental Static Regeneration (ISR)
Definition	Runs on every request to the server	Runs at build time	Updates static content without rebuilding the entire site
Use Cases	Real-time data, personalized content	Content that doesn't change frequently	Large sites with frequently updated content
Example Use Case	Fetching products from an API	Fetching articles from a CMS	Updating blog posts
Data Fetching	Server-side	Build-time	Build-time with background updates
Performance	Slower, as it runs on each request	Faster, as it is pre-rendered	Combines benefits of static and server-side rendering
SEO Benefits	Good for real-time data	Great for static content	Combines benefits of both static and dynamic content
Code Example	`getServerSideProps`	`getStaticProps`	`getStaticPaths`, `getStaticProps` with `revalidate`
When to Use	When you need fresh data on every request	When data is static and doesn't change often	When you need to update static content without a full rebuild

5. Client-Side Data Fetching: SWR vs React Query

SWR (Stale-While-Revalidate)

SWR is a React hook for client-side data fetching, which caches data and revalidates it in the background, ensuring that the app always shows up-to-date information.

Example:

import useSWR from 'swr';

```
const fetcher = (url) => fetch(url).then((res) => res.json());
const { data, error } = useSWR('https://api.example.com/data', fetcher);
```

React Query

React Query provides more advanced features, like caching, pagination, and background synchronization, making it suitable for complex client-side data fetching.

Example:

```
import { useQuery } from 'react-query';

const fetchData = async () => {
  const res = await fetch('https://api.example.com/data');
  return res.json();
};

const { data, error } = useQuery('data', fetchData);
```

Comparison Table:

Feature	SWR	React Query
Caching	Yes, with background revalidation	Advanced caching and background syncing
Pagination	Not built-in	Built-in pagination support
Query Invalidation	Manual control	Automatic query invalidation
Use Case	Simple data fetching	Complex data fetching with caching and pagination
Performance	Lightweight and fast	More features, heavier than SWR

6. Detailed Comparison: Pages Router vs App Router

Feature Comparison:

Feature	Pages Router	App Router
Folder Structure	pages/ directory	app/ directory
Routing	File-based routing	Flexible routing with layouts and nested routes
Layouts	Not available	Available
Error Handling	Basic error handling	Advanced error handling and boundaries
Use Case	Small to medium projects	Large, complex applications with nested layouts
SSR/SSG Support	Yes	Yes
Performance	Fast but limited for complex apps	Highly optimized for large apps

7. Best Practices in Next.js 15: Nested Routing, Dynamic Routes, and Data Fetching Strategies

- **Nested Routing:** Use the App Router's layout system to organize your app into nested routes for better scalability.
- **Dynamic Routes:** Leverage dynamic routes for handling user-specific content, such as product details.
- **Data Fetching Strategies:** Combine SSR, SSG, and ISR to ensure optimal performance and data freshness based on your page requirements.

This document provides a deep understanding of Next.js 15's App Router, including its advanced features, data fetching strategies, and client-side options like SWR and React Query. It also offers insights into how the legacy Pages Router compares and when to use each approach.