Next.js Rendering Methods in App Router

Next.js provides several rendering methods to optimize performance and user experience. With the **App Router**, introduced in Next.js 13, these methods are seamlessly integrated into the framework. Here's an overview of the key rendering methods:

1. Server-Side Rendering (SSR)

- **Description**: The page is rendered on the server for every request, ensuring the user gets the most up-to-date content.
- **Use Case**: Ideal for dynamic content that changes frequently or requires authentication.
- **Implementation**: Use the `async` function in a server component or `fetch` with `cache: 'no-store'`.

```
// Example in a server component
export default async function Page() {
  const data = await fetch('https://api.example.com/data', { cache: 'no-store' });
  return <div>{data.title}</div>;
}
```

2. Static Site Generation (SSG)

- **Description**: The page is pre-rendered at build time and served as static HTML.
- **Use Case**: Best for content that doesn't change often, like blogs or marketing pages.
- **Implementation**: Use `fetch` with `cache: 'force-cache'` or default caching in server components.

```
// Example in a server component
export default async function Page() {
  const data = await fetch('https://api.example.com/data', { cache: 'force-cache' });
```

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

3. Incremental Static Regeneration (ISR)

- **Description**: Combines SSG with the ability to update static pages at runtime. Pages are regenerated in the background after a specified time.
- **Use Case**: Suitable for content that updates periodically, like news or product listings.
- **Implementation**: Use `revalidate` in the `fetch` function.

```
// Example in a server component
export default async function Page() {
  const data = await fetch('https://api.example.com/data', { next: {
revalidate: 60 } });
  return <div>{data.title}</div>;
}
```

4. Client-Side Rendering (CSR)

- **Description**: The page is rendered entirely on the client side after fetching data via APIs.
- **Use Case**: Useful for highly interactive pages or when SEO is not a priority.
- **Implementation**: Use React hooks like `useEffect` to fetch data.

```
// Example in a client component
'use client';

import { useEffect, useState } from 'react';

export default function Page() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
```

```
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  return <div>{data?.title}</div>;
}
```

5. Streaming and Suspense

- **Description**: Allows streaming of server-rendered content to the client and progressively rendering parts of the UI.
- **Use Case**: Enhances performance for large or complex pages.
- **Implementation**: Use React's `Suspense` and server components.

```
// Example with Suspense
import { Suspense } from 'react';

export default function Page() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Content />
    </Suspense>
  );
}

async function Content() {
  const data = await fetch('https://api.example.com/data', { cache: 'no-store'
});
  return <div>{data.title}</div>;
}
```

By combining these methods, you can create highly optimized and scalable applications with the **Next.js App Router**. Each method serves a specific purpose, so choose based on your application's needs!