1. CSR (Client-Side Rendering)

- **How it works**:
    - The browser first loads a minimal HTML file with almost no content.
    - Then, JavaScript runs in the browser to fetch data (usually from an API) and render the UI.
    - Rendering happens **entirely on the client side** (in the browser).
- **When to use**:
    - Apps with heavy client-side interactivity.
    - Dashboards, Single Page Apps (SPAs), user-specific data (e.g., logged-in views).
- **Pros**:
    - Smooth user experience (no full page reloads).
    - Great for highly interactive apps.
    - Scales well with dynamic data.
- **Cons**:
    - Slower initial load (blank page → fetch data → render).
    - Poor SEO (search engines may not index content well if it's loaded via JS).

---

2. SSR (Server-Side Rendering)

- **How it works**:
    - The page is rendered **on the server** for each request.
    - The server fetches the data, builds the HTML, and sends it to the browser.
    - The browser then hydrates the page with React for interactivity.
- **When to use**:
    - Dynamic content that changes frequently (e.g., news websites, e-commerce product pages).
    - Good for SEO because the HTML is fully rendered on load.

- **Pros**:
    - Better SEO (search engines see full HTML).
    - Faster first page load (HTML arrives pre-built).
    - Always up-to-date data since it's rendered on request.
- **Cons**:
    - Slower response time (server must render for each request).
    - Heavier server load.
    - Not as fast as SSG for repeat visitors.

---

3. SSG (Static Site Generation)

- **How it works**:
    - HTML pages are **pre-rendered at build time** (not per request).
    - Pages are generated once and served via CDN or static hosting.
    - Next.js supports **Incremental Static Regeneration (ISR)**, which can rebuild specific pages after deployment.
- **When to use**:
    - Websites with content that doesn't change often (blogs, documentation, portfolios, marketing sites).
    - Huge boost for performance and SEO.
- **Pros**:
    - Very fast (served as static HTML from CDN).
    - Best for SEO (fully rendered content).
    - Scales extremely well (no server-side computation per request).
- **Cons**:
    - Not suitable for highly dynamic content (unless ISR is used).
    - Rebuilding the site takes time if content changes frequently.

🔹 **ISR (Incremental Static Regeneration)**

- **How it works**:
    - ISR is an **extension of SSG**.
    - Pages are **pre-rendered at build time** like SSG, **but** they can be **updated/rebuilt in the background** after the site is deployed.
    - When a request comes in after the revalidation time, Next.js regenerates that page **in the background** and serves the new version for future requests.

✅ **ISR Pros**

- Keeps the **speed of SSG** (static pages served via CDN).
- Keeps content **fresh automatically** without rebuilding the whole site.
- Best balance between **performance** and **data freshness**.

❌ **ISR Cons**

- Content is not **real-time** (users may see slightly stale content until revalidation triggers).
- Requires **Next.js server or Vercel-like infrastructure** (not plain static hosting).

🔹 Older Next.js (Pages Router)

- You explicitly chose the rendering method:
  - getServerSideProps → **SSR**
  - getStaticProps → **SSG / ISR**
  - useEffect (client fetching) → **CSR**
- By default, a page with no data fetching methods was **Static (SSG)**.

🔹 Latest Next.js (App Router, v13/v14)

- **By default, pages are Server Components → rendered on the server.**
- This is similar to SSR, but with optimizations:
  - Server Components **don't send JavaScript for the component itself** to the client, only the serialized result.
  - Only components marked with "use client" run on the client.

So technically:
✅ Yes, **SSR (server rendering) is the default in Next.js App Router**.
But it's not exactly the same as old SSR — it's **Server Components rendering**, which is even more optimized.

⚙️ Data Fetching in Next.js 13+ (App Router)

- **fetch in a Server Component** → runs on the server, data is fetched before sending HTML.
- You can also control caching:
  - fetch(url, { cache: 'no-store' }) → always fetch fresh data (**SSR-like**).
  - fetch(url, { next: { revalidate: 60 } }) → ISR (revalidate every 60 seconds).
  - Default fetch → cached and static (**SSG**).

🔎 Summary

- **Next.js App Router (latest)** → Server rendering (SSR) is default.
- But you can easily switch to **SSG/ISR/CSR** using caching strategies and "use client".

Got it 👍 You can absolutely use **Axios** instead of fetch in **Next.js 14 App Router**. The caching behavior (cache: 'no-store', revalidate) only works with **fetch**, but you can still mimic **CSR, SSR, SSG, ISR** with Axios.

Let's go case by case 👇

🔹 1. CSR with Axios

- Run on client ("use client").
- Use useEffect + Axios.

Example:

```
"use client";

import { useEffect, useState } from "react";

import axios from "axios";


export default function CSRPage() {

 const [data, setData] = useState<any>(null);


 useEffect(() => {

  axios.get("https://api.example.com/data").then((res) => {

   setData(res.data);

  });

 }, []);


 return <div>{data ? data.message : "Loading..."}</div>;
```

}

✅ Always fresh, ❌ SEO poor.

🔹 **2. SSR with Axios**

- Run in **Server Component**.
- Axios fetch happens on server **for each request**.

Example

```
import axios from "axios";


export default async function SSRPage() {
 const res = await axios.get("https://api.example.com/data");
 const data = res.data;


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

✅ SEO friendly, always fresh.
❌ Slower than SSG/ISR.

🔹 **3. SSG with Axios**

- Axios runs at **build time**.
- Use cache() helper from React to memoize results so it doesn't re-fetch every request.

Example:

```
import axios from "axios";
import { cache } from "react";
```

```
const getData = cache(async () => {
 const res = await axios.get("https://api.example.com/data");
 return res.data;
});

export default async function SSGPage() {
 const data = await getData();
 return <div>{data.message}</div>;
}
```

✅ Very fast, cached.
❌ Stale until redeploy.

🔹 **4. ISR with Axios**

- Next.js ISR depends on fetch revalidate, but with Axios we can **simulate ISR** by:
    - Creating an **API Route** (/api/data) that uses fetch with revalidate.
    - Calling that API with Axios.

Example:

/app/api/data/route.ts

```
export async function GET() {
 const res = await fetch("https://api.example.com/data", {
  next: { revalidate: 60 }, // ISR: regenerate every 60s
 });
 const data = await res.json();

 return Response.json(data);
}
```

/app/isr/page.tsx

```
import axios from "axios";

export default async function ISRPage() {
 const res = await axios.get("http://localhost:3000/api/data");
 const data = res.data;

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

✅ Fast like SSG, but updates every 60s.
❌ Needs extra API layer.

🔹 **CSR (Client-Side Rendering)**

- Rendering happens **in the browser** after JavaScript loads.
- Use "use client" + useEffect with Axios.
- Data is **always fresh** (fetched from client).
- ❌ Poor SEO (search engines see empty HTML at first).
- ❌ Slower first load (blank page → then data loads).
- ✅ Best for dashboards, SPAs, user-specific apps.

🔹 **SSR (Server-Side Rendering)**

- Rendering happens **on the server** for every request.
- Directly call Axios in a **Server Component**.
- Data is **always fresh** (new fetch for every request).
- ✅ SEO friendly (HTML is fully rendered).

- ⚡ Medium performance (slower than static but always up-to-date).
- ❌ Heavy load on server if many users.
- ✅ Best for news sites, e-commerce product pages, real-time data.

🔹 **SSG (Static Site Generation)**

- Rendering happens **once at build time**.
- Use cache() helper with Axios so the result is reused.
- Data is **stale until next build**.
- ✅ Extremely fast (served as static HTML from CDN).
- ✅ SEO friendly.
- ❌ Content doesn't update automatically.
- ✅ Best for blogs, documentation, portfolios, marketing sites.

🔹 **ISR (Incremental Static Regeneration)**

- Rendering happens **at build time**, then **regenerates in background** after a set time.
- With Axios → need an **API route** that uses fetch(..., { revalidate: N }), then call it via Axios.
- Data is **updated automatically after N seconds**.
- ✅ Fast like SSG but not stale for long.
- ✅ SEO friendly.
- ❌ Not real-time (slightly stale between revalidations).
- ✅ Best for e-commerce catalogs, blogs with frequent updates.

👉 **Summary in one line each:**

- **CSR** → Always fresh, poor SEO, slower first load.

- **SSR** → Always fresh, great SEO, heavier on server.
- **SSG** → Fastest, SEO friendly, but stale until rebuild.
- **ISR** → Fast + SEO friendly + updates automatically after time.

🔑 **Key Points in Next.js 14 + Axios**

- **CSR** → Client Components with Axios in useEffect.
- **SSR** → Server Components directly call Axios.
- **SSG** → Use cache() from React to memoize Axios calls at build time.
- **ISR** → Needs an **API Route** that uses fetch(..., { next: { revalidate } }), then Axios consumes that route.

👉 This means in **Next.js 14 App Router**, the real control of caching & revalidation is tied to **fetch**, not Axios. But you can still achieve all four methods with Axios by combining it with **React's cache()** and **API Routes**.