

Types of rendering



Understand how data is fetched and rendered on the server/client.

♦ 1. Types of Rendering

Static Site Generation (SSG) — pre-rendered at build time

Server-Side Rendering (SSR) — rendered at request time

Client-Side Rendering (CSR) — rendered in browser

Perfect — this is one of the **core concepts of Next.js and web performance** in general.
Let's break it down clearly 🙋

♦ 1. Types of Rendering

1 Static Site Generation (SSG)

- **When:** At **build time** (before deployment).
- **How:** The HTML pages are **pre-rendered once** and reused for every request.
- **Where:** On the **server during build**, not on every user visit.
- **Data fetching:** Happens using `getStaticProps()` or `getStaticPaths()` in Next.js.
- **Use case:** Blogs, documentation, marketing pages — content doesn't change often.
- **Pros:**
 - ✓ Super fast — served from CDN.
 - ✓ SEO friendly.
 - ✓ Reduces server load.
- **Cons:**
 - ✗ Needs rebuild to update data.

2 Server-Side Rendering (SSR)

- **When:** At **request time** — every time a user visits the page.
- **How:** The server fetches data and renders HTML dynamically for that request.
- **Where:** On the **server at runtime**.
- **Data fetching:** Happens using `getServerSideProps()` in Next.js.
- **Use case:** Dashboards, personalized pages (user-specific data).

- **Pros:**
 - ✓ Always fresh data.
 - ✓ SEO friendly.
- **Cons:**
 - ✗ Slower than SSG (since it runs for every request).
 - ✗ More server cost.

3 Client-Side Rendering (CSR)

- **When:** In the **browser after page load**.
- **How:** The page loads a basic HTML + JS bundle, then fetches data via API calls (like `fetch()` or `axios`) and renders dynamically.
- **Where:** On the **client (browser)**.
- **Data fetching:** Usually done in `useEffect()` in React.
- **Use case:** Single Page Apps (SPAs), dashboards, or parts of pages that update frequently.
- **Pros:**
 - ✓ Great for dynamic, interactive experiences.
 - ✓ No need to rebuild or reload the whole page.
- **Cons:**
 - ✗ Not SEO friendly (HTML is initially empty).
 - ✗ Slightly slower first load since data comes after JS runs.

♦ Summary Table

Rendering Type	Rendered On	Data Fetched	Best For	Example Hook/Method
SSG	Build Time	Once (static)	Blogs, Docs	<code>getStaticProps</code> , <code>getStaticPaths</code>
SSR	Request Time	Per Request	Dashboards, Auth Pages	<code>getServerSideProps</code>
CSR	Client Side	On Browser	SPAs, Dynamic UIs	<code>useEffect</code> , API calls

Would you like me to explain **how to use each one in Next.js with small code examples** (SSG, SSR, CSR)?



Ok

Perfect 🍌

Let's go through **each type of rendering in Next.js** with **simple code examples** — so you'll understand *where the data is fetched and where the rendering happens*.

⚙️ 1 Static Site Generation (SSG)

- ✓ Data is fetched **at build time** (before deployment).
- ✓ Page is static and reused for every request.

```
js

// app/blogs/[id]/page.js (Next.js 13+ App Router)

export async function generateStaticParams() {
  // Predefine which blog pages to build
  const res = await fetch("https://jsonplaceholder.typicode.com/posts");
  const posts = await res.json();
  return posts.map((post) => ({ id: post.id.toString() }));
}

export default async function BlogPage({ params }) {
  const res = await fetch(`https://jsonplaceholder.typicode.com/posts/${params.id}`);
  const post = await res.json();

  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.body}</p>
    </div>
  );
}
```

🧠 What happens:

- Next.js fetches data **once during build**.
- The pages are **pre-rendered into static HTML**.
- When users visit `/blogs/1`, it's instantly served from cache/CDN.

⚙️ 2 Server-Side Rendering (SSR)

- ⚡ Data is fetched **for each request**.
- ✓ Always fresh data from the server.

```
js

// app/users/page.js

export const dynamic = "force-dynamic"; // forces SSR rendering

export default async function UsersPage() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users", {
    cache: "no-store", // ensures data is not cached
  });
  const users = await res.json();

  return (
    <div>
      <h1>Server-Side Rendered Users</h1>
      {users.map((user) => (
        <p key={user.id}>{user.name}</p>
      ))}
    </div>
  );
}
```

🧠 What happens:

- Every time someone visits `/users`, the **server fetches the latest data** → renders HTML → sends to browser.
- Good for **real-time or user-specific data** (like dashboards).

⚙️ 3 Client-Side Rendering (CSR)

🧩 Data is fetched **in the browser** after page loads.

```
jsx

"use client";

import { useEffect, useState } from "react";

export default function ProductsPage() {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    fetch("https://fakestoreapi.com/products")
      .then((res) => res.json())
      .then((data) => setProducts(data));
  }, []);

  return (
    <div>
      <h1>Client-Side Rendered Products</h1>
      {products.map((p) => (
        <p key={p.id}>{p.title}</p>
      ))}
    </div>
  );
}
```

🧠 What happens:

- Page first loads with a **loading state or empty HTML**.
- Then JavaScript runs → `fetch()` gets data → UI updates.
- Common for **dashboards or app sections that update frequently**.

📄 Quick Recap

Rendering Type	Data Fetched	Rendering Location	Example
SSG	At build time	Server (build)	<code>generateStaticParams()</code>
SSR	At request time	Server (runtime)	<code>dynamic = "force-dynamic"</code>
CSR	After load	Browser (client)	<code>useEffect()</code>

Would you like me to show **how to combine SSR + CSR on the same page** (e.g., render initial data on server, then update in browser)? That's a real-world pattern.





ould you like me to show ****how to combine SSR + CSR on the same page**** (e.g., render initial data on server, then update in browser)? That's a real-world pattern.
ok

Perfect 🍌

Let's look at a **real-world hybrid rendering pattern** — combining **SSR + CSR** on the same page.

This is very common in dashboards, feeds, or any page where you want **fast initial load (SSR) + live updates (CSR)**.

⚙️ Example: SSR + CSR (Hybrid Rendering)

🧩 File: `app/news/page.js`

```
jsx

"use client";

import { useEffect, useState } from "react";

// 📌 Next.js 13+ allows SSR + CSR mix using server + client components
export default function NewsPage({ initialNews }) {
  const [news, setNews] = useState(initialNews);

  // 🔄 Client-side update (refresh data after page load)
  useEffect(() => {
    async function fetchLatestNews() {
      const res = await fetch("https://newsapi.org/api/latest");
      const data = await res.json();
      setNews(data.articles);
    }

    fetchLatestNews();
  }, []);

  return (
    <div>
      <h1>📰 Latest News</h1>
      <ul>
        {news.map((article, i) => (
          <li key={i}>{article.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

🧠 Step 2: Pass SSR Data to the Client Component

```
js

// app/news/page.server.js or layout file

import NewsPage from "./page";

// 🌐 Server-side: fetches data at request time (SSR)
export default async function NewsPageWrapper() {
  const res = await fetch("https://newsapi.org/api/latest", {
    cache: "no-store",
  });
  const data = await res.json();
  return <NewsPage initialNews={data.articles} />;
}
```

```
});
const data = await res.json();

// Pass initial SSR data to client
return <NewsPage initialNews={data.articles} />;
}
```

How It Works

Step	Where It Happens	Description
1	Server (SSR)	Fetch initial data → render HTML → fast first paint + SEO friendly
2	Client (CSR)	After page loads, useEffect runs → fetch latest data → update UI dynamically

Why Use Hybrid Rendering?

✓ Best of both worlds:

- Fast **initial render** (SSR)
- **Up-to-date content** via client refresh
- **SEO-friendly** + **dynamic updates**

✓ Real-world examples:

- Dashboards with live stats
- News feeds
- Product pages that update stock info
- Any app where data changes frequently

Would you like me to show the **same pattern but in the App Router format** (using `server` and `client` components cleanly in `/app` folder)? That's the modern Next.js 14+ style.



Incremental Static Regeneration (ISR) — revalidate pages after deployment

Excellent 🌟 — you're diving into one of **Next.js's most powerful features!**

Let's break down **Incremental Static Regeneration (ISR)** step-by-step so you fully understand it 🙌

Incremental Static Regeneration (ISR)

◆ Definition

ISR allows you to:

- ✓ Pre-render static pages (like SSG)
- ✓ And **automatically re-generate** them **after deployment**, without rebuilding the whole app.

Simple Explanation

Think of ISR as an **upgradable SSG**.

Feature	SSG	ISR
Build time	Once during build	Once + automatically updated later
When page updates	Needs full rebuild	Automatically revalidated in background
Performance	Very fast	Still very fast
Use case	Blogs, docs	Blogs, product pages, dashboards with periodic updates

◆ How ISR Works

- 1 The page is **pre-rendered at build time** (like SSG).
- 2 You define a **revalidate interval** (in seconds).
- 3 When a user visits after the interval expires:
 - The **old page is served immediately** (no delay).
 - Meanwhile, **Next.js rebuilds the page in the background**.
 - Once done, the **new version replaces the old one** automatically.

Example Code (Next.js App Router)

```
js

// app/products/page.js

export const revalidate = 60; // ⌚ Regenerate every 60 seconds

export default async function ProductsPage() {
  const res = await fetch("https://fakestoreapi.com/products");
  const products = await res.json();

  return (
    <div>
      <h1>🛒 Products (ISR Enabled)</h1>
      <ul>
        {products.map((p) => (
          <li key={p.id}>{p.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

What happens:

- This page is built once at deployment.
- Every **60 seconds**, Next.js checks if the page is outdated.
- If someone visits after 60 seconds:
 - They get the **old (cached)** version instantly.
 - The server **regenerates a fresh version in the background**.
 - Future visitors see the **new updated page** — no downtime!

ISR vs SSG vs SSR

Feature	SSG	ISR	SSR
Rendered at	Build time	Build time + revalidation	Every request
Rebuild needed?	Yes	No	No
Speed	⚡ Fastest	⚡ Fast	🚀 Slower
Fresh data	❌ Only after rebuild	✅ Periodically updated	✅ Always fresh
SEO Friendly	✅	✅	✅
Use Case	Static blogs	Frequently updated blogs/products	Live dashboards

✅ When to Use ISR

Use ISR when:

- You want **fast static performance** ⚡
- But also need **periodic updates** 🕒
- Example:
 - News site (updates every 5 min)
 - eCommerce product list
 - Blog that adds new posts daily

Would you like me to show how **ISR + dynamic routes (like `/products/[id]`)** works with `generateStaticParams()` and `revalidate` together? That's the real-world combo most apps use.



♦ 2. Data Fetching Methods

In app/ router:

`fetch()` with async components

Perfect 🍌 — now you're learning the **core of how data fetching works in the Next.js app/ router** (the modern system introduced in Next.js 13+).

Let's break it down clearly 🙌

♦ 2. Data Fetching Methods (in app/ Router)

In the **App Router**, you can fetch data directly **inside async components** using the built-in `fetch()` function.

This is **different from the Pages Router**, where we used `getStaticProps`, `getServerSideProps`, etc.

⚙️ 1 Fetching with async Server Components

✅ **By default, components in app/ are Server Components**, so you can write **async** directly and use `fetch()` inside.

Example:

```
js

// app/users/page.js

export default async function UsersPage() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  const users = await res.json();

  return (
    <div>
      <h1>👤 Users List (Server Component)</h1>
      {users.map((user) => (
        <p key={user.id}>{user.name}</p>
      ))}
    </div>
  );
}
```

🧠 **What happens:**

- The data is fetched **on the server** before rendering the page.
- The page is **pre-rendered** into HTML.
- The browser receives fully rendered content → **SEO-friendly** + fast.

⚙️ 2 `fetch()` Caching Behavior

In the App Router, `fetch()` has **built-in caching** and **revalidation control**.

♦ **Default behavior:**

- Responses are **cached and reused** (like SSG).
- Perfect for static pages that don't change often.

Example 1: No Caching (SSR)

```
js

export default async function ProductsPage() {
  const res = await fetch("https://fakestoreapi.com/products", {
    cache: "no-store", // disables cache -> fetches fresh data every request
  });
  const products = await res.json();

  return (
    <div>
      <h1>Server-Side Rendering (No Cache)</h1>
      {products.map((p) => (
        <p key={p.id}>{p.title}</p>
      ))}
    </div>
  );
}
```

- ◆ `cache: "no-store"` → behaves like **Server-Side Rendering (SSR)**.

Example 2: Cached with Revalidation (ISR)

```
js

export const revalidate = 60; // Regenerate every 60s

export default async function BlogsPage() {
  const res = await fetch("https://api.example.com/blogs", {
    next: { revalidate: 60 }, // alternative way
  });
  const blogs = await res.json();

  return (
    <div>
      <h1>📝 Blogs (ISR Enabled)</h1>
      {blogs.map((b) => (
        <p key={b.id}>{b.title}</p>
      ))}
    </div>
  );
}
```

- ◆ `next: { revalidate: 60 }` OR `export const revalidate = 60`
→ enables **Incremental Static Regeneration**.

Example 3: Force Dynamic (Always SSR)

```
js

export const dynamic = "force-dynamic";

export default async function WeatherPage() {
  const res = await fetch("https://api.weatherapi.com/v1/current.json?q=London");
  const data = await res.json();

  return (
    <div>
      <h1>🌤️ Weather (Always Server-Side)</h1>
      <p>City: {data.location.name}</p>
      <p>Temp: {data.current.temp_c}°C</p>
    </div>
  );
}
```

```

    </div>
  );
}

```

♦ `dynamic = "force-dynamic"` → disables all caching → pure SSR.

Summary Table

Mode	Option	Type	Description
Static	<i>(default)</i>	SSG	Cached at build time
Dynamic (SSR)	<code>cache: "no-store"</code> or <code>dynamic = "force-dynamic"</code>	SSR	Fetches on every request
Revalidated (ISR)	<code>revalidate: <seconds></code>	ISR	Cached + periodic refresh

3 When to Use Which

Case	Use	Fetch Option
Data never changes	Static	Default fetch
Data changes often	SSR	<code>cache: "no-store"</code>
Data changes sometimes	ISR	<code>revalidate: <seconds></code>

Would you like me to show how **to fetch data in a client component** (like using `useEffect`) and how it differs from this server-side fetching?



`getStaticProps`, `getServerSideProps` (legacy pages)

Perfect, Soumalya 🙌 — this is where we connect the **old Pages Router (before App Router)** concepts like `getStaticProps` and `getServerSideProps` with what you're learning now (`fetch()` in App Router).

Let's clearly understand both and how they differ 🙌

Data Fetching in the Pages Router (Legacy Method)

Before Next.js 13 introduced the **App Router** (`app/` folder), data fetching was done in the **Pages Router** (`pages/` folder) using **special functions** exported from each page.

♦ 1 `getStaticProps()` — Static Site Generation (SSG)

- ✓ Runs **at build time** (when you deploy).
- ✓ The page is **pre-rendered** as static HTML.
- ✓ Fast and SEO-friendly.

Example:

```
js

// pages/blogs.js

export async function getStaticProps() {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts");
  const posts = await res.json();

  return {
    props: {
      posts, // passes to the component below
    },
    revalidate: 60, // ISR (optional) – regenerate every 60 sec
  };
}

export default function Blogs({ posts }) {
  return (
    <div>
      <h1>📝 Blogs (SSG)</h1>
      {posts.map((p) => (
        <p key={p.id}>{p.title}</p>
      ))}
    </div>
  );
}
```

🧠 What happens:

- Runs only **once during build**.
- The HTML is generated and served fast from CDN.
- If **revalidate** is added → it becomes **ISR** (auto-regenerated).

♦ 2 `getServerSideProps()` — Server-Side Rendering (SSR)

- ⚡ Runs **on every request**.
- ✓ Always serves fresh data.
- ✓ Good for dashboards, auth pages, etc.

Example:

```
js

// pages/users.js

export async function getServerSideProps() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  const users = await res.json();

  return {
    props: {
      users,
    },
  };
}
```

```

    };
  }

  export default function Users({ users }) {
    return (
      <div>
        <h1>👤 Users (SSR)</h1>
        {users.map((u) => (
          <p key={u.id}>{u.name}</p>
        ))}
      </div>
    );
  }

```

🧠 What happens:

- Runs **every time** a user visits `/users`.
- Next.js fetches fresh data → renders HTML → sends to browser.

💡 3 `getStaticPaths()` — Dynamic SSG for Dynamic Routes

Used with `getStaticProps()` when you have **dynamic pages** like `/blog/[id]`.

Example:

```

js

// pages/blog/[id].js

export async function getStaticPaths() {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts");
  const posts = await res.json();

  const paths = posts.map((p) => ({
    params: { id: p.id.toString() },
  }));

  return { paths, fallback: "blocking" };
}

export async function getStaticProps({ params }) {
  const res = await fetch(
    `https://jsonplaceholder.typicode.com/posts/${params.id}`
  );
  const post = await res.json();

  return { props: { post } };
}

export default function Blog({ post }) {
  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.body}</p>
    </div>
  );
}

```

🧠 What happens:

- Next.js builds all pages listed in **paths** at **build time**.
- If **fallback: "blocking"** → new pages can be built on-demand (**ISR** style).

Summary Table

Method	Where It Runs	When It Runs	Use Case	Replaced By (App Router)
<code>getStaticProps</code>	Server (build time)	Before deployment	Static pages	<code>fetch()</code> (default)
<code>getStaticPaths</code>	Server (build time)	Before deployment	Dynamic SSG pages	<code>generateStaticParams()</code>
<code>getServerSideProps</code>	Server (runtime)	Every request	Dynamic pages	<code>fetch()</code> with <code>cache: "no-store"</code> or <code>dynamic = "force-dynamic"</code>

✓ Migration Tip

If you ever move a project from `pages/` → `app/`, this is how they map:

Old Pages Router	App Router Equivalent
<code>getStaticProps()</code>	<code>fetch()</code> (default static)
<code>getServerSideProps()</code>	<code>fetch()</code> with <code>{ cache: "no-store" }</code>
<code>getStaticPaths()</code>	<code>generateStaticParams()</code>
ISR (<code>revalidate</code>)	<code>export const revalidate = <seconds></code>

Would you like me to show one side-by-side example — `pages/` (legacy) vs `app/` (modern) — for the **same API data** so you can clearly see the difference in structure?



Would you like me to show one side-by-side example — ``pages/`` (legacy) vs ``app/`` (modern) — for the ****same API data**** so you can clearly see the difference in structure?
ok'

Perfect, Soumalya 🙌

Let's see a **side-by-side comparison** — how the **same page** (fetching user data from an API) looks in the **old Pages Router** and in the **modern App Router**.

🧩 Example Goal

We'll fetch a list of users from this API:

👉 <https://jsonplaceholder.typicode.com/users>

🧱 1 Legacy Pages Router (Before Next.js 13)

File: `pages/users.js`

```
js

// ✅ This runs on the server – before the page loads
export async function getServerSideProps() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  const users = await res.json();

  return {
    props: {
      users, // passed to the component below
    },
  };
}

// ✅ This runs on the client – after server sends props
export default function UsersPage({ users }) {
  return (
    <div>
      <h1>👤 Users (Pages Router - SSR)</h1>
      {users.map((u) => (
        <p key={u.id}>{u.name}</p>
      ))}
    </div>
  );
}
```

🧠 What happens:

- `getServerSideProps()` runs **on every request** (server-side).
- The HTML is fully rendered and sent to the browser.
- The browser **does not call the API** itself — it only gets HTML.

⚙️ 2 Modern App Router (Next.js 13+)

File: `app/users/page.js`

```
js

// ✅ This component runs on the server by default
export const dynamic = "force-dynamic"; // same as SSR (fetch every time)

export default async function UsersPage() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users", {
    cache: "no-store", // disables caching → same as SSR
  });
  const users = await res.json();

  return (
    <div>
      <h1>👤 Users (App Router - SSR)</h1>
      {users.map((u) => (
        <p key={u.id}>{u.name}</p>
      ))}
    </div>
  );
}
```

🧠 What happens:

- The function runs on the **server** automatically (no need for `getServerSideProps`).

- `fetch()` with `cache: "no-store"` ensures **fresh data** on each request.
- Cleaner and more flexible structure.

Summary of Differences

Feature	Pages Router	App Router
Folder	<code>pages/</code>	<code>app/</code>
Data Fetching Method	<code>getStaticProps / getServerSideProps</code>	Direct <code>fetch()</code> inside async component
SSR Example	<code>getServerSideProps()</code>	<code>fetch() + cache: "no-store"</code> or <code>dynamic = "force-dynamic"</code>
SSG Example	<code>getStaticProps()</code>	<code>fetch()</code> (default, cached)
ISR Example	<code>getStaticProps() + revalidate</code>	<code>export const revalidate = <seconds></code>
Dynamic Paths	<code>getStaticPaths()</code>	<code>generateStaticParams()</code>
Simplicity	More boilerplate	Cleaner & unified

✓ In short:

The App Router makes server-side fetching easier — no need for special functions, just use `async` components + `fetch()`.

Would you like me to show the **SSG + ISR version** of this same example (so you can compare all three: SSR, SSG, ISR) using both routers?



 Learn: How to cache, revalidate, and stream data.

Excellent, Soumalya 🙌 — this is a **very powerful part of Next.js App Router**.

Now that you already understand **SSG**, **SSR**, and **ISR**, let's go one level deeper into **how caching, revalidation, and streaming** work in Next.js data fetching.

Learn: How to Cache, Revalidate, and Stream Data

Next.js gives you full control over **how your data is fetched, stored, and updated**. Let's break it down clearly 🙌

1 Caching — Control When Data Is Stored or Reused

◆ What is caching?

Caching means **saving fetched data** so that it can be reused for future requests — instead of fetching it again and again.



Example 1 — Default Cache (Static Fetch)

```
js

export default async function ProductsPage() {
  const res = await fetch("https://fakestoreapi.com/products");
  const products = await res.json();

  return (
    <div>
      <h1>🛒 Cached Data (Default SSG)</h1>
      {products.map((p) => <p key={p.id}>{p.title}</p>)}
    </div>
  );
}
```

What happens:

- By default, `fetch()` caches data at **build time** (Static Site Generation).
- The same HTML is reused until you **rebuild** the app.
-  Super fast, but  not updated until rebuild.



Example 2 — No Cache (SSR)

```
js

export default async function LiveProducts() {
  const res = await fetch("https://fakestoreapi.com/products", {
    cache: "no-store", // disables cache
  });
  const data = await res.json();

  return (
    <div>
      <h1>< No Cache (SSR)</h1>
      {data.map((item) => <p key={item.id}>{item.title}</p>)}
    </div>
  );
}
```

What happens:

- Data is fetched **on every request**.
- Same as `getServerSideProps()` (SSR).
-  Always fresh,  but a bit slower.

2 Revalidation — Control When Cache Refreshes (ISR)

You can re-fetch and rebuild data automatically after a set time interval.

Example 3 — Revalidate Every 60 Seconds (ISR)

```
js

export const revalidate = 60; // ⌚ Rebuild every 60 seconds

export default async function NewsPage() {
  const res = await fetch("https://newsapi.org/api/latest", {
    next: { revalidate: 60 }, // same as export const revalidate
  });
  const news = await res.json();

  return (
    <div>
      <h1>📰 News (Revalidate Every 60s)</h1>
      {news.map((n, i) => <p key={i}>{n.title}</p>)}
    </div>
  );
}
```

🧠 What happens:

- Data is cached initially.
- After **60 seconds**, when a user visits again:
 - Old data is served immediately.
 - Next.js **re-fetches in background** → updates cache.
- ✅ Combines speed of SSG + freshness of SSR.

⚙️ 3 Streaming — Render Page in Chunks

Streaming lets your page **render progressively** instead of waiting for all data to load. Users see part of the page immediately (like skeletons or partial content).

🧩 Example 4 — Streaming with Suspense

```
jsx

import { Suspense } from "react";

async function PostsList() {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts");
  const posts = await res.json();

  return (
    <ul>
      {posts.slice(0, 5).map((p) => (
        <li key={p.id}>{p.title}</li>
      ))}
    </ul>
  );
}

export default function StreamedPage() {
  return (
    <div>
      <h1>🕒 Streaming Example</h1>
      <Suspense fallback=<p>Loading posts...</p>>
        { /* This section loads while data is fetched */ }
      <PostsList />
    </Suspense>
  </div>
  );
}
```

```
};  
}
```

🧠 What happens:

- Page starts rendering immediately.
- **Suspense** shows fallback (like a loader) until data is fetched.
- Once data arrives, the posts are injected — **no full reload needed**.

Summary Table

Feature	Description	Example Option	Type
Cache	Store fetched data for reuse	<code>fetch(url)</code>	Static
No Cache (SSR)	Always fetch new data	<code>{ cache: "no-store" }</code>	Dynamic
Revalidate (ISR)	Rebuild cached data after interval	<code>{ next: { revalidate: 60 } }</code>	Hybrid
Streaming	Render parts of UI progressively	<code><Suspense fallback={...}></code>	User Experience

✅ When to Use What

Goal	Use	Code Example
Page rarely changes	Cache (SSG)	<code>fetch(url)</code>
Page updates frequently	No Cache (SSR)	<code>fetch(url, { cache: "no-store" })</code>
Page updates sometimes	Revalidate (ISR)	<code>fetch(url, { next: { revalidate: 60 } })</code>
Speed up UI while waiting for data	Streaming	<code><Suspense fallback="Loading..." /></code>

Would you like me to show a **real-world example combining all three** — like a dashboard that uses caching for products, revalidation for analytics, and streaming for live updates?