In a Frontend System Design interview, **Incremental Static Regeneration (ISR)** is your answer to the question: *"How do you handle a site with millions of pages without 10-hour build times or slow SSR?"*

ISR allows you to use static generation (SSG) on a per-page basis **without needing to rebuild the entire site.**

---

1. The Theoretical Concept

ISR is the "Middle Ground" between SSG and SSR.

- **SSG:** Build once, stays the same until the next deploy. (Fast, but stale).
- **SSR:** Build on every request. (Fresh, but slow).
- **ISR:** Build once, then **silently update** in the background after a specific amount of time.

The "Stale-While-Revalidate" Logic:

1. **Request 1:** User visits a page. They get the **cached/stale** static HTML immediately.
2. **Trigger:** If the "revalidate" timer has expired, Next.js triggers a background re-render.
3. **Background:** The server fetches fresh data and generates a new HTML file.
4. **Request 2:** The next user still gets the old page (the update is still happening).
5. **Request 3:** Once the background build finishes, the cache is updated. The next user gets the **fresh** HTML.

---

2. ISR Implementation: Pages Router

In the Pages Router, you add a revalidate property to the object returned by getStaticProps.

```
// pages/products/[id].js

export async function getStaticProps({ params }) {
  const res = await fetch(`https://api.example.com/products/${params.id}`);
  const data = await res.json();

  return {
    props: { data },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every 60 seconds
    revalidate: 60,
  };
}
```

```
export async function getStaticPaths() {
  return {
    // Only pre-build the most popular items at build-time
    paths: [{ params: { id: '1' } }],
    // fallback: 'blocking' ensures that pages NOT pre-built
    // will be SSR'd on the first hit and then cached via ISR
    fallback: 'blocking',
  };
}

export default function Product({ data }) {
  return <div>{data.name} - {data.price}</div>;
}
```

---

3. ISR Implementation: App Router

In the App Router, ISR is even more powerful because you can control it via the fetch API directly or segment-level configs.

Method A: Time-based Revalidation

```
// app/products/[id]/page.tsx

export default async function Page({ params }: { params: { id: string } }) {
  // This fetch will be cached and revalidated every 60 seconds
  const res = await fetch(`https://api.example.com/products/${params.id}`, {
    next: { revalidate: 60 }
  });
  const data = await res.json();

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

Method B: On-Demand Revalidation (The "Expert" Way)

Instead of waiting for a timer, you can update the page **immediately** when your data changes (e.g., via a CMS Webhook).

- **Tag the fetch:**
  TypeScript
  `fetch('...', { next: { tags: ['products'] } })`

- **Trigger the update (in an API route or Server Action):**
  TypeScript
  ```
  import { revalidateTag } from 'next/cache';

  // Call this when a product is updated in your DB
  revalidateTag('products');
  ```

---

4. Hydration in ISR

Hydration in ISR behaves exactly like **SSG Hydration**.

- The browser receives the static HTML (which might be a few minutes "stale").
- The user sees content immediately.
- React hydrates the interactive parts.
- **Key Detail:** Hydration happens on the client based on whatever version of the HTML was sent. It doesn't "know" a newer version is being built in the background until the user refreshes or navigates back.

---

5. Why use ISR? (Interview Strategy)

- **Scalability:** You don't have to build 100,000 pages at build-time. You build the top 100 and let ISR handle the rest "on-demand."
- **Performance:** Users always get a static file from the CDN (Fastest possible load).
- **Cost:** Much cheaper than SSR because the server only works once every x minutes, rather than once every x requests.

6. Interview Q&A

Q: What is the difference between fallback: 'blocking' and fallback: true?
A: blocking waits for the server to render the HTML (like SSR) on the first visit, then caches it. true immediately serves a "Loading" state/shell and fetches data on the client, then caches the result. blocking is generally better for SEO.

Q: Does ISR work on a simple CDN like S3?
A: No. ISR requires a Node.js server (or Edge Functions) to handle the background regeneration. Pure SSG works on S3; ISR needs a "smart" host like Vercel, Netlify, or a custom Node server.

Q: How do you handle a "Flash of Stale Content"?
A: ISR is designed for "eventual consistency." If you need the user to see fresh data immediately (like a user's bank balance), do not use ISR. Use SSR or Client-side fetching instead. Use ISR for things like product reviews or blog posts.

Q: Is ISR specific to Next.js?
A: While Next.js popularized it, the concept is a generic caching strategy called Stale-While-Revalidate. Other frameworks like Nuxt (Vue) or Remix have similar features, but Next.js has the most mature implementation.

---

1. Does the page update automatically for the active user?

No. If a user is currently looking at a page, the content will **not** magically change in front of them when the ISR background re-generation completes.

- **The Experience:** The user stays on the "stale" version of the page they loaded.
- **The Update:** The updated HTML is only served on the **next** request.
- **The Reason:** Automatically swapping out the DOM while a user is reading or interacting could be jarring (and would require a web-socket or polling mechanism, which ISR does not use).
- **How they see it:** The user would need to **refresh** the page or navigate away and back to the route to trigger the browser to fetch the now-updated version from the CDN/Cache.

2. If no one is using the site, does the CDN update?

No. This is a common misconception. ISR is **Lazy**. It does not run on a "cron job" or a fixed schedule in the background if there is no traffic.

6. **The Trigger:** Re-generation is only triggered by a **user request**.
7. **The Scenario:** If you set revalidate: 60 (1 minute), but no one visits your site for 5 hours, **nothing happens** for 5 hours. The server stays idle, and the old HTML stays on the CDN.
8. **The First Visitor:** After 5 hours, the first person to visit the site will still see the **stale** 5-hour-old page. However, their visit "wakes up" the server and tells it: *"Hey, this is old, start building a new one in the background."*
9. **The Second Visitor:** Only the person who arrives *after* that background build finishes will see the new content.

---

3. The "Expert" Fix: On-Demand Revalidation

If you need the CDN to update **even if no one is visiting**, or you want it to update **immediately** after a database change without waiting for a timer, you use **On-Demand Revalidation** (available in both Routers).

Instead of a timer (revalidate: 60), you create a **Webhook** in your CMS or Database.

- You change a price in your database.
- Your database sends a POST request to your Next.js "revalidate" API route.
- Next.js forces a re-generation of that specific page **immediately**.
- The CDN is updated **before** the next user even arrives.

Summary for Interviews

Feature	**Time-based ISR**	**On-Demand ISR (Webhooks)**
Trigger	User Request + Timer	API Call (External Event)
Active User	Needs Refresh	Needs Refresh
Zero Traffic	No update happens	**Update happens immediately**
Best For	High-traffic blogs/news	E-commerce prices, CMS edits

Interview Q&A

Q: "If ISR is lazy, isn't that bad for the very first user after a long break?"
A: "Yes, that user gets stale content. However, in a system design context, we prioritize Availability and Latency. Serving a stale static page instantly is usually better than making that user wait 5 seconds for a slow SSR render, especially if the 'stale' content is still 99% accurate."