Rendering Strategies

Outline

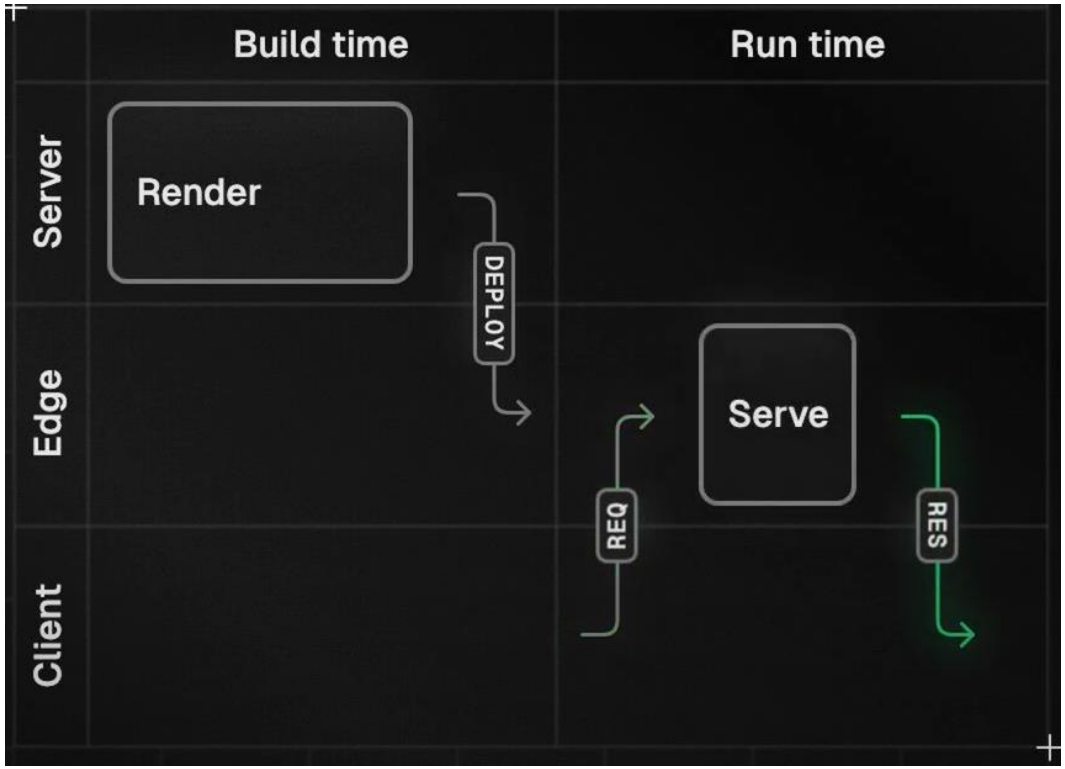
1. Static Rendering
2. Dynamic Rendering
3. Incremental Static Regeneration (ISR)
4. Partial Prerendering (PPR)

Rendering Strategies

Next.js let you decide the desired rendering strategy per page:

- **Static Rendering (Pre-rendering)**: generating static pages at build time
- **Dynamic Rendering**: dynamically render a page for each request
- **Incremental Static Regeneration (ISR)**: Update static content without a full rebuild of the entire site
- **Partial Prerendering (PPR)**: Combines the benefits of static and dynamic rendering in the same route

Static Rendering

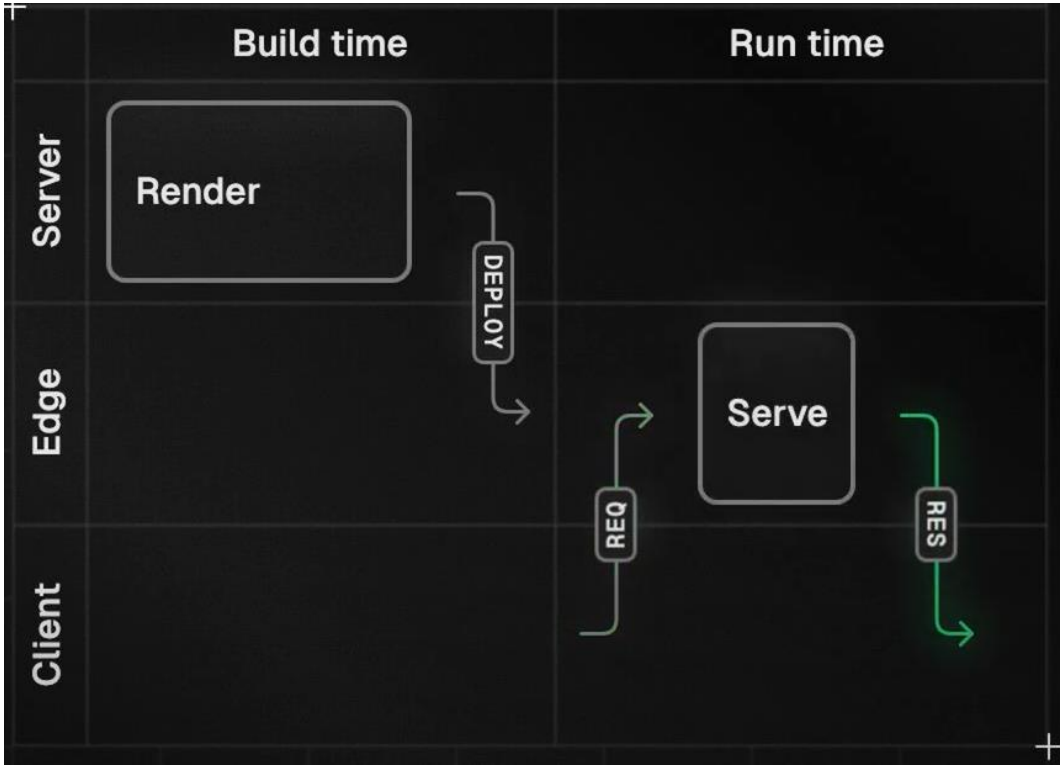


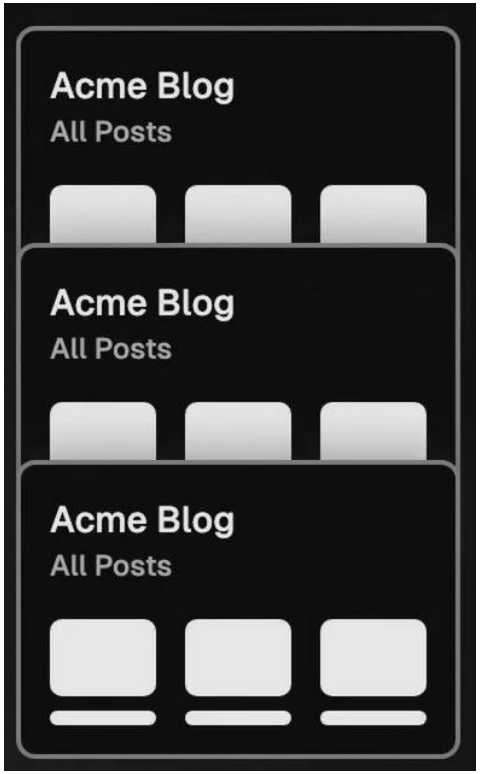
Static Rendering (Pre-rendering)

- **What**: Pages are rendered on the server at build time
  - The resulting HTML (+ optimized JS/CSS) is static and can be efficiently served from a Content Delivery Network (CDN) or Edge server
  - The pre-rendered static result is shared across all users and server requests, bypassing the need for the origin server to render it again at runtime
  - This is the default behavior unless dynamic functions (such as cookies(), headers(), searchParams) or fetching with no cache are used
  - Offers the best performance and SEO
- Best suited for:
  - Data not personalized to the user
  - Content that is known at build time and unchanging
  - E.g., static blog post or a product page

Static Rendering (Pre-rendering)

- Pre-rendered static result can be deployed to a CDN or Edge server
  - When a user requests the page, the static file is served directly and quickly from the Edge location closest to them, bypassing the need for the origin server to render it again at runtime
  - E.g., company blog posts





Source: YouTube video

Static Rendering Example

- This component will fetch and display a list of heroes
  - When not using any Dynamic APIs (headers, cookies or the URL's search params) anywhere in this route, it will be prerendered during next build to a static page

```
export default async function Page() {
    const data = await fetch('http://localhost:8080/api/heroes')
    const heroes = await data.json()
    console.log("Heroes count: ", heroes.length)
    return (
      <ul>
        {heroes.map((hero) => (
          <li key={hero.id}>{hero.name}</li>
        ))}
      </ul>
    )
  }
```

generateStaticParams Example

```
export const dynamicParams = true // or false, to 404 on unknown paths

// Generate Pages with Dynamic Routes
// generateStaticParams returns an array of possible values for post id
export async function generateStaticParams() {
  const posts = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )
  return posts.map((post) => ({
    id: String(post.id),
  }))
}
export default async function Page({ params }) {
  const { id } = await params
  const post = await fetch(`https://api.vercel.app/blog/${id}`).then((res) =>
    res.json()
  )
  return (
    <main>
      <h1>{post.title}</h1><p>{post.content}</p>
    </main>
  )
}
```

If a blog post created after build-time is requested, Next.js will generate and cache this page on-demand

When you export a function called **generateStaticParams** from a page that uses dynamic routes, Next.js will statically pre-render all the paths for the Ids returned by **generateStaticParams**

How this example works?

- The `generateStaticParams` function can be used in combination with dynamic route segments to define the list of route segment parameters that will be statically generated at build time
- During next build, all known blog posts are generated (25 posts for example)
- All requests made to these pages (e.g. /blog/1) are cached and instantaneous
- Using `export const dynamicParams = true`

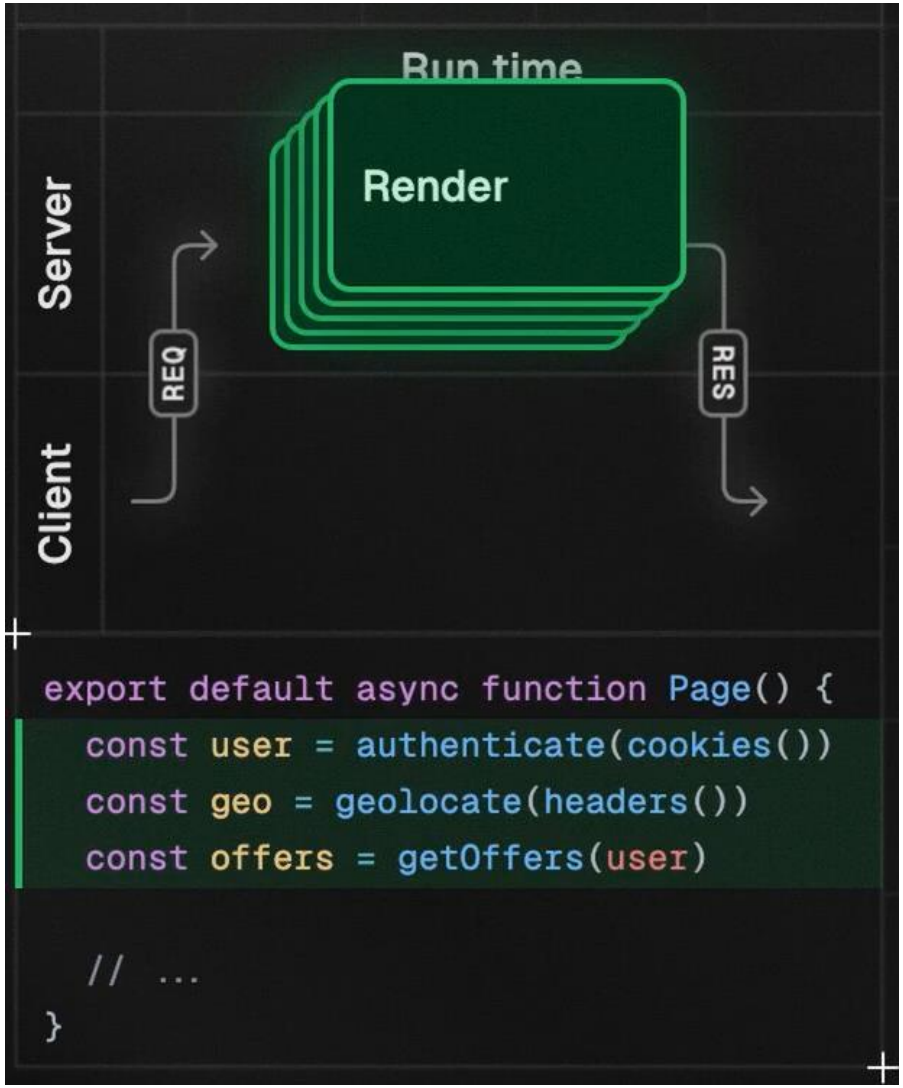
If a blog post created after build-time is requested (e.g., /blog/26), Next.js will **generate** and **cache** this page on-demand

  - If set to false, new paths will result in a 404 page

Static Rendering - Advantages and limitations

- ✓ Pre-rendered static pages which can be pushed to a CDN to for global and scalable access
- ✓ Static content is fast, resilient to downtime, and immediately indexed by crawlers
- 👎 For building a large-scale static site, it may take hours for the site to build
- 👎 Consider an e-commerce store with 100,000 products. Product prices change frequently. When changing the price of headphones from $100 to $75 as part of a promotion, the entire site need to be rebuild
    - It's not feasible to wait hours for the new price to be reflected

Dynamic Rendering

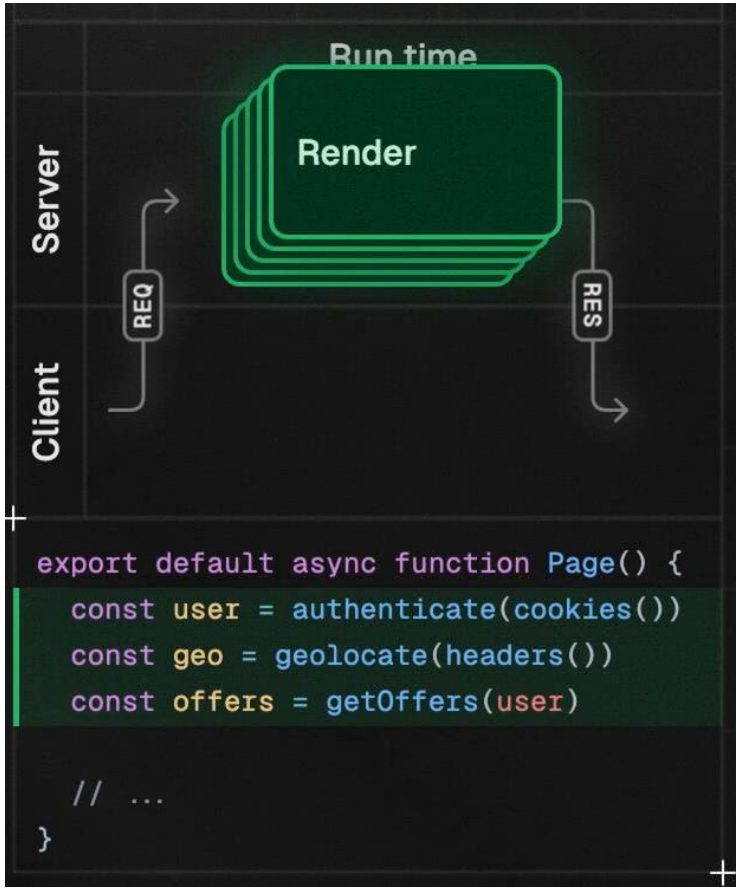


Dynamic Rendering

- Pages are rendered on the server at request time
  - The server renders the page every time it receives a request from a client
  - The process involves: Request -> Server Renders -> Response
  - Happens automatically if the page uses dynamic API (such as cookies(), headers(), searchParams) or fetches data dynamically (fetch with cache: 'no-store' or revalidate: 0)
- Essential for scenarios involving:
  - Personalized content (like user-specific recommendations) or frequently changing data, as each render might produce a unique result
  - Information that is only available at the time of the user request, such as cookies or the URL's search params

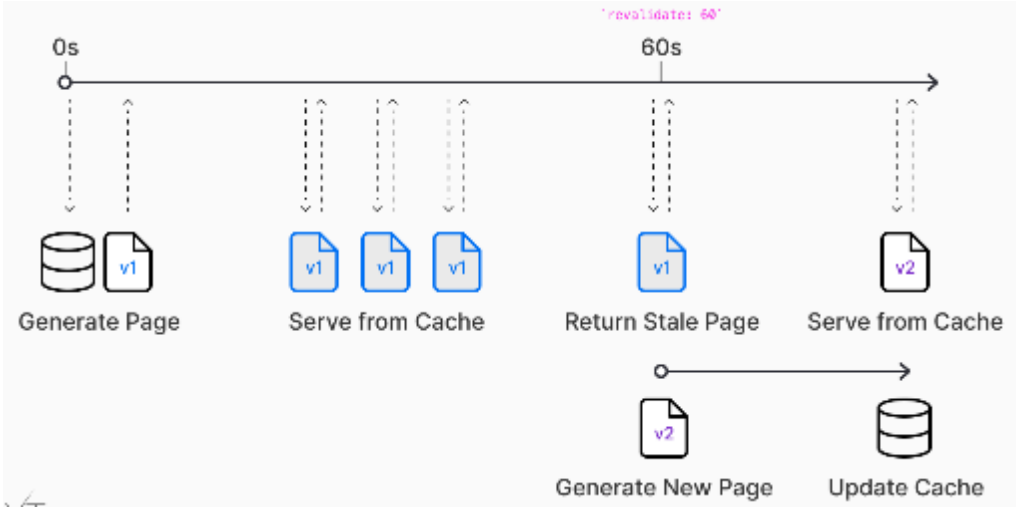
Dynamic Rendering

- Produced Personalized content
  - Request -> Server Renders -> Response
  - E.g., Recommendations **personalized** to each user



Source: YouTube video

Incremental Static Regeneration (ISR)



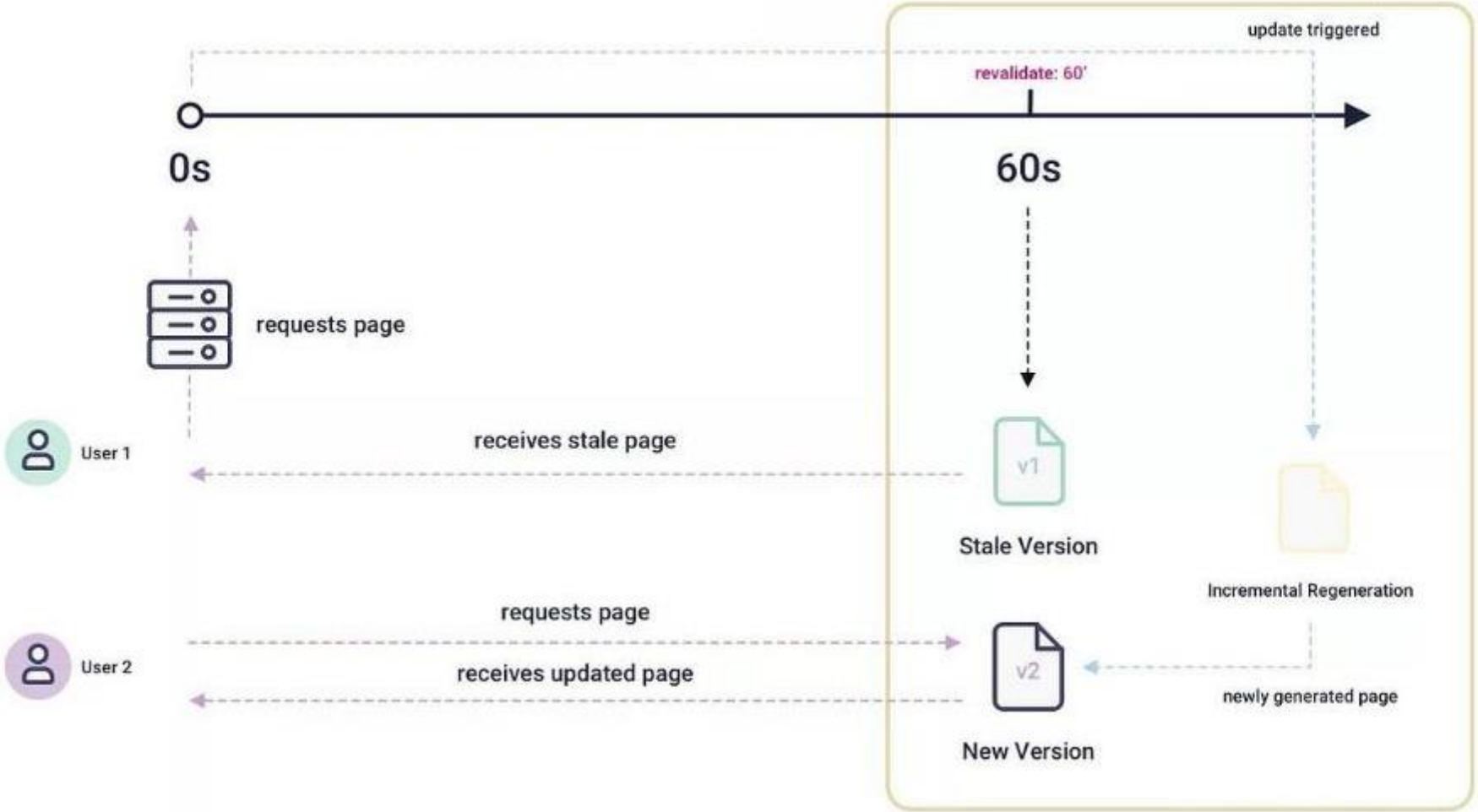
Incremental Static Regeneration (ISR)

Incremental Static Regeneration (ISR) enables:

- Update static content without a full rebuild of the entire site
- Reduce server load by serving prerendered, static pages for most requests
- Handle **dynamic content serving** without full site re-build
  - **Incremental** as only the pages that need updating are regenerated, not the entire site
  - ISR reduces build times significantly, making it easier to scale large apps

ISR

- ISR enables periodic content refresh



ISR with revalidate 10s

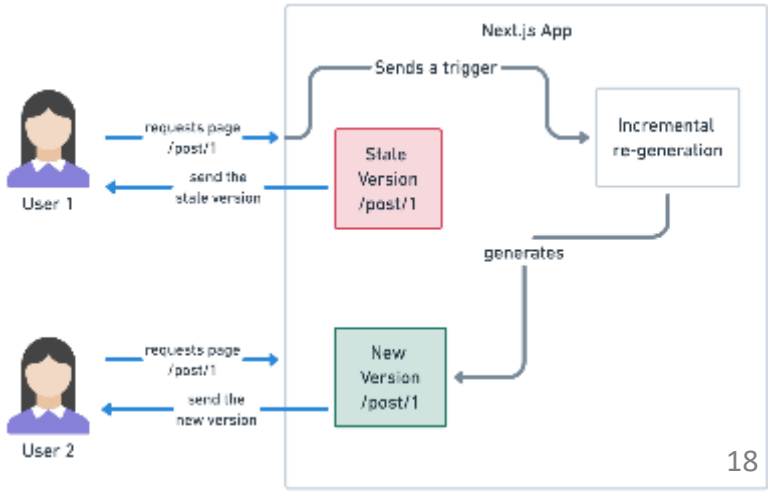
Source: [Article]

Time-based revalidation

- This fetches and displays a list of blog posts on /blogs. After an hour, the cache for this page is invalidated on the next visit to the page
  - Then, in the background, a new version of the page is generated with the latest blog posts

```
export const revalidate = 3600 // invalidate every hour

export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts = await data.json()
  return (
    <main>
      <h1>Blog Posts</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </main>
  )
}
```

- Revalidate defines a time interval (in seconds) after which a page revalidated
- After 1h has passed, the next request will still show the cached (stale) page **AND** the page will be regenerated and cashed in the background



On-demand revalidation with revalidatePath

- Invalidate pages on-demand with the `revalidatePath` function
  - For example, this Server Action would get called after adding a new post
  - This will clear the cache for the entire route and allow the Server Component to fetch fresh data

```
'use server'

import { revalidatePath } from 'next/cache'

export async function createPost() {
  // Invalidate the /posts route in the cache
  revalidatePath('/posts')
}
```

On-demand revalidation with revalidateTag

- For most use cases, prefer revalidating entire paths
- If you need more granular control, you can use the `revalidateTag` function

```tsx
export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog', {
    next: { tags: ['posts'] },
  })
  const posts = await data.json()
  // ...
}
```

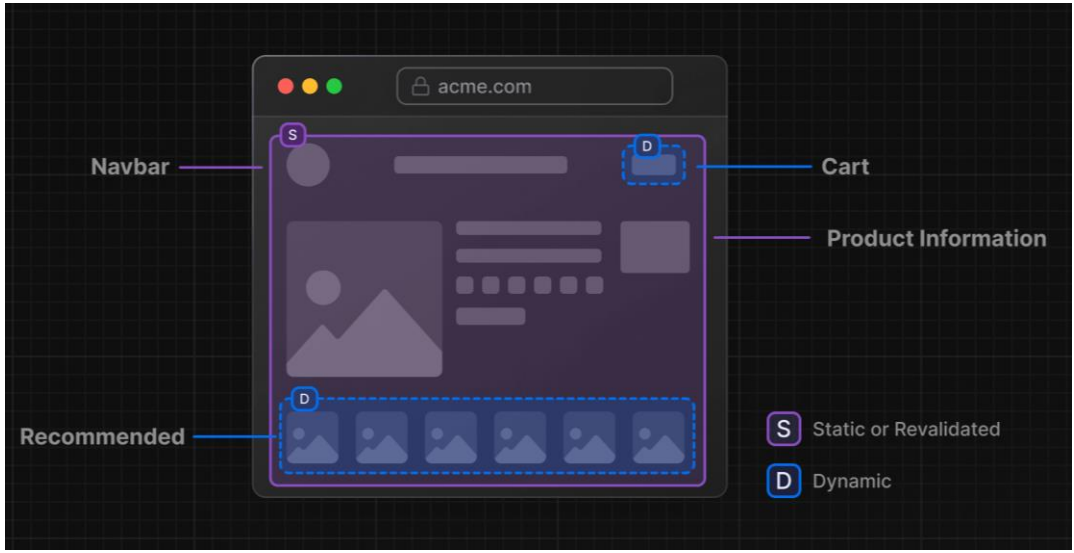
```tsx
'use server'

import { revalidateTag } from 'next/cache'

export async function createPost() {
  // Invalidate all data tagged with 'posts' in the cache
  revalidateTag('posts')
}
```

Caching data from a Database or a File

```
import { unstable_cache } from 'next/cache'
import { db, posts } from '@/lib/db'

const getPosts = unstable_cache(
  async () => {
    return await db.select().from(posts)
  },
  ['posts'],
  { revalidate: 3600, tags: ['posts'] }
)

export default async function Page() {
  const allPosts = await getPosts()

  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

- You can use the `unstable_cache` API to cache the response when running next build

Partial Prerendering (PPR)

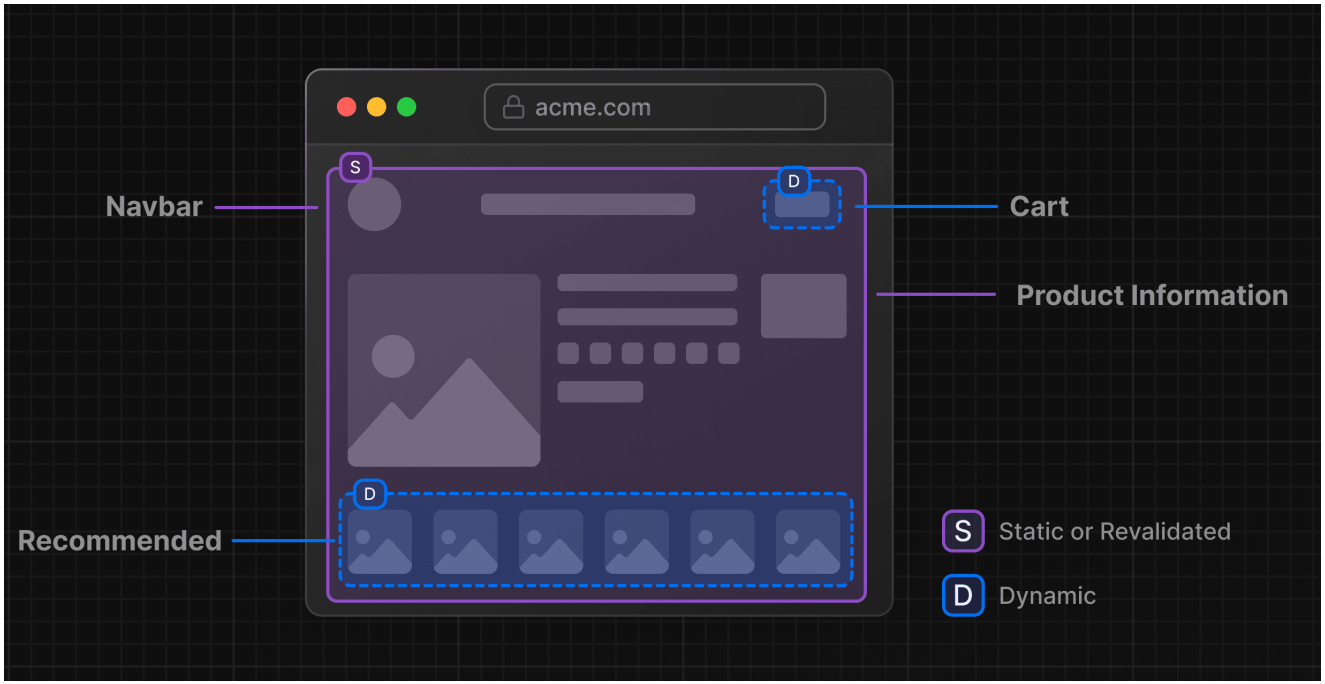


Partial Prerendering (PPR) - Experimental

- Goal: Get the best of both worlds – the fast initial load of static rendering plus the flexibility of dynamic content on the same page
- Mechanism: Leverages React Suspense
- Build Time: Renders a static "shell" of the page. Any components wrapped in <Suspense> that depend on dynamic APIs or uncached data have their rendering postponed and are replaced with their specified fallback UI (like a loading skeleton)

Partial Prerendering (PPR) – Example 1

Live Demo

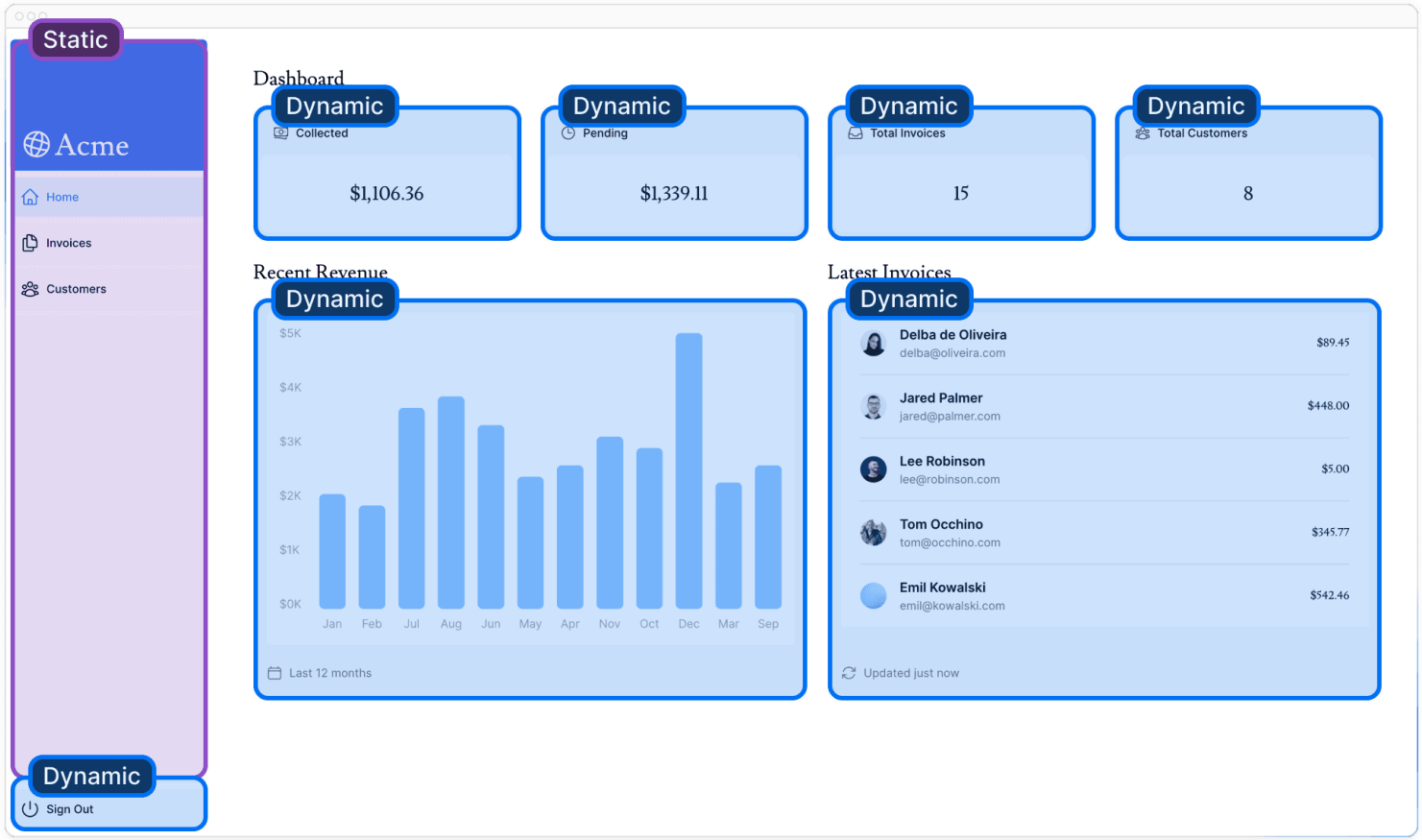


When a user visits a route:

- A static route shell that includes the navbar and product information is served, ensuring a fast initial load
- The shell leaves holes where dynamic content like the cart and recommended products will load in asynchronously
- The async holes are streamed in parallel, reducing the overall load time of the page

Source: Next.js Article

Partial Prerendering (PPR) – Example 2



- The **<SideNav>** Component doesn't rely on data and is not personalized to the user, so it can be static
- The components in **<Page>** rely on data that changes often and will be personalized to the user, so they can be dynamic

Source: [Next.js Article]

Suspense Component

- Partial Prerendering uses React's Suspense to defer rendering parts of the until the data is loaded
- The Suspense fallback is embedded into the initial HTML file along with the static content
  - At build time (or during revalidation), the static content is prerendered to create a static shell. The rendering of dynamic content is postponed until the user requests the route
- Suspense is used as a boundary between your static and dynamic code

Suspense - Example

```
export default function Page() {
  return (
    <div>
      <SingleProduct />

      <Suspense fallback={<RecommendedProductsSkeleton />}>
        <RecommendedProducts />
      </Suspense>

      <Suspense fallback={<ReviewsSkeleton />}>
        <Reviews />
      </Suspense>
    </div>
  );
}
```

Partial Prerendering - at Runtime

- When a user requests the page, the fast static shell (with fallbacks) is served immediately from the Edge
- The browser starts painting this shell and loading essential assets (JS, CSS)
- Simultaneously, requests are made to the server to render the dynamic parts (the "holes" left by Suspense)
- The server computes these dynamic parts and streams the resulting HTML back to the client
- The client seamlessly patches this streamed HTML into the existing static shell, replacing the fallbacks

Partial Prerendering - Benefits

- **Fast Initial Load**: Users get the near-instant response of a static page
- **Dynamic Content**: Still supports personalized or real-time data where needed
- **Unified Model**: Developers can write code as if the page is dynamic, using dynamic APIs where necessary, and Next.js/React optimize automatically
- **Simple to Use**: Relies on existing React Suspense boundaries; no new APIs are required to enable it

Summary

- **Static Rendering (Pre-rendering)**: Page is rendered once at build time and served quickly from a CDN/Edge. Fast initial load but unsuitable for dynamic content
- **Dynamic Rendering**: Server renders the page on every request. Necessary for personalized or frequently updated content but can be slower initially as the user waits for the server computation
- **Incremental Static Regeneration (ISR)**: Update static content without a full rebuild of the entire site
- **Partial Prerendering (PPR)**: Hybrid approach combining static and dynamic rending (e.g., static nav bar with dynamic signed-in user info)