Rendering Strategies

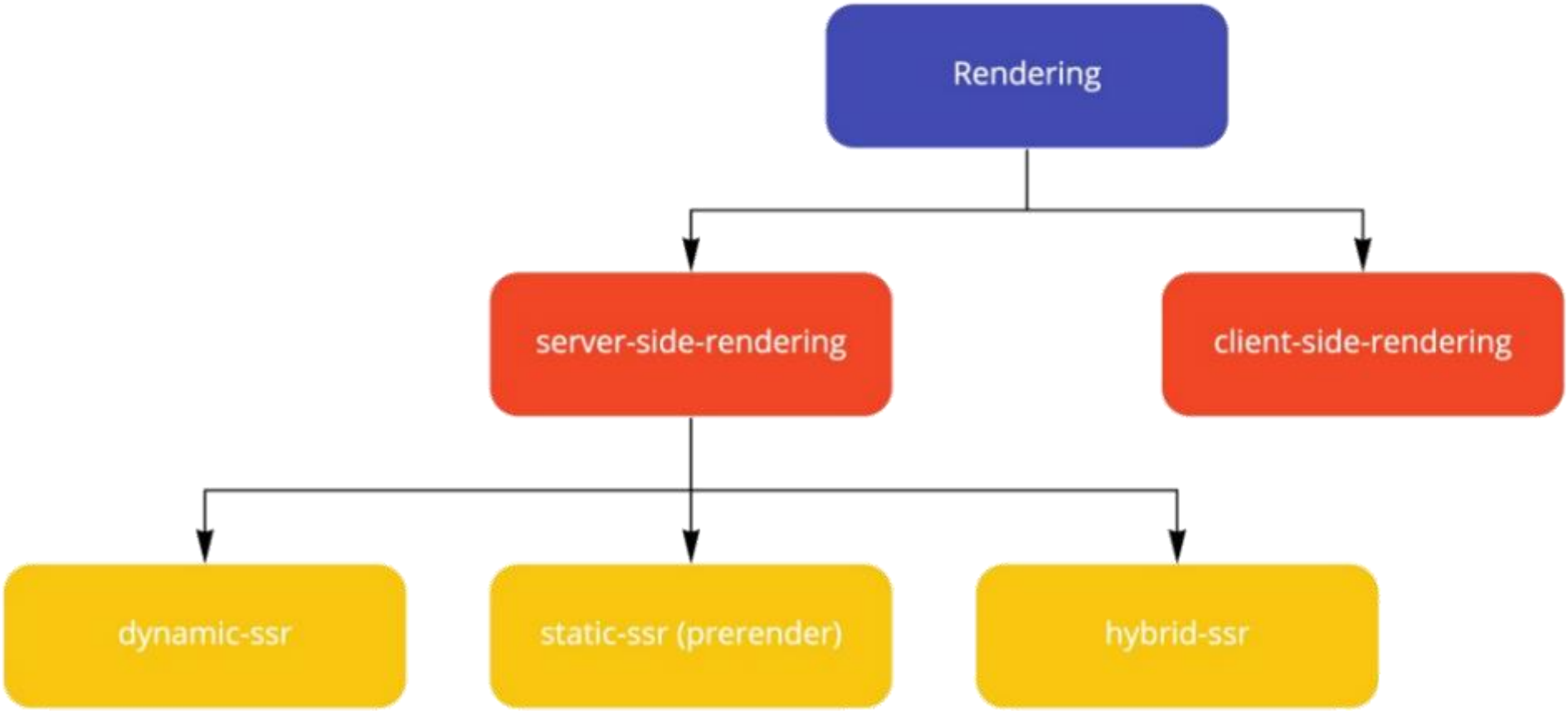
Outline

1. Static site generation (SSG)
2. Server-side rendering (SSR)
3. Client-side rendering (CSR)
4. Incremental Static Regeneration (ISR)

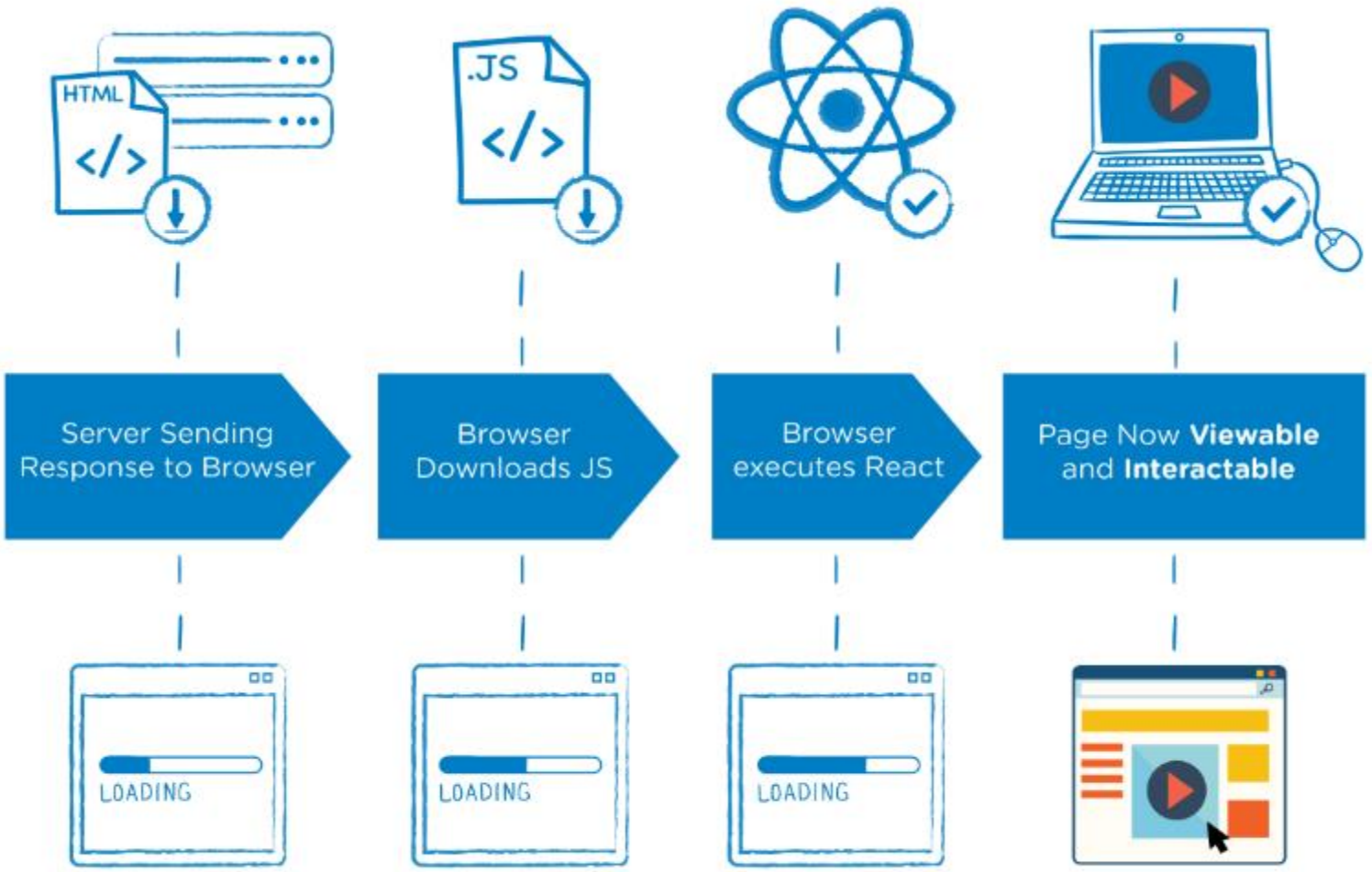
Rendering Strategies

- Next.js let you decide the desired rendering strategy per page:
  - **Static site generation (SSG)**: generating static pages at build time
  - **Server-side rendering (SSR)**: dynamically render a page for each request using
  - **Client-side rendering (CSR)** for certain components only
  - **Incremental Static Regeneration (ISR)**: regenerate static pages in production without needing a full rebuild of the site

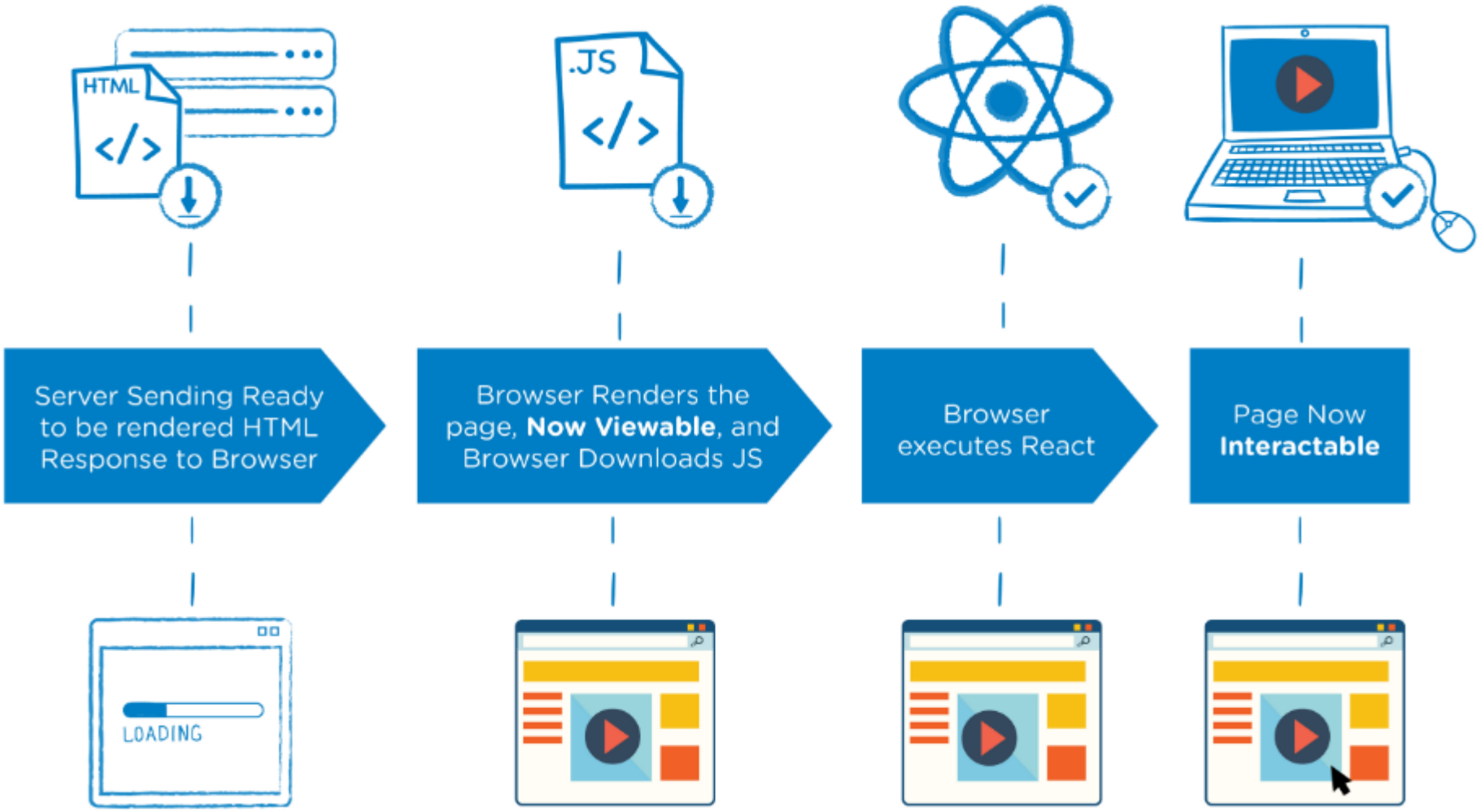
Rendering Strategies



CSR


HTML
.JS
Server Sending Response to Browser
Browser Downloads JS
Browser executes React
Page Now Viewable and Interactable
LOADING
LOADING
LOADING

SSR


HTML
.JS
Server Sending Ready to be rendered HTML Response to Browser
Browser Renders the page, **Now Viewable**, and Browser Downloads JS
Browser executes React
Page Now **Interactable**
LOADING

SSR

Server-side rendering consists of the following steps:

1. **Client's HTTP request** – sends the server a request for the HTML document
2. **Data fetching** – The server fetches any required data from the database or third-party APIs
3. **Server-side pre-rendering** – The server compiles the JavaScript components into static HTML and sends it to the client
4. **Page load and rendering**– The client downloads the HTML file and displays the static components on the page
5. **Hydration** – The client downloads the JavaScript file(s) embedded into the HTML, processes the code, and attaches event listeners to the components. This process is also called hydration or rehydration

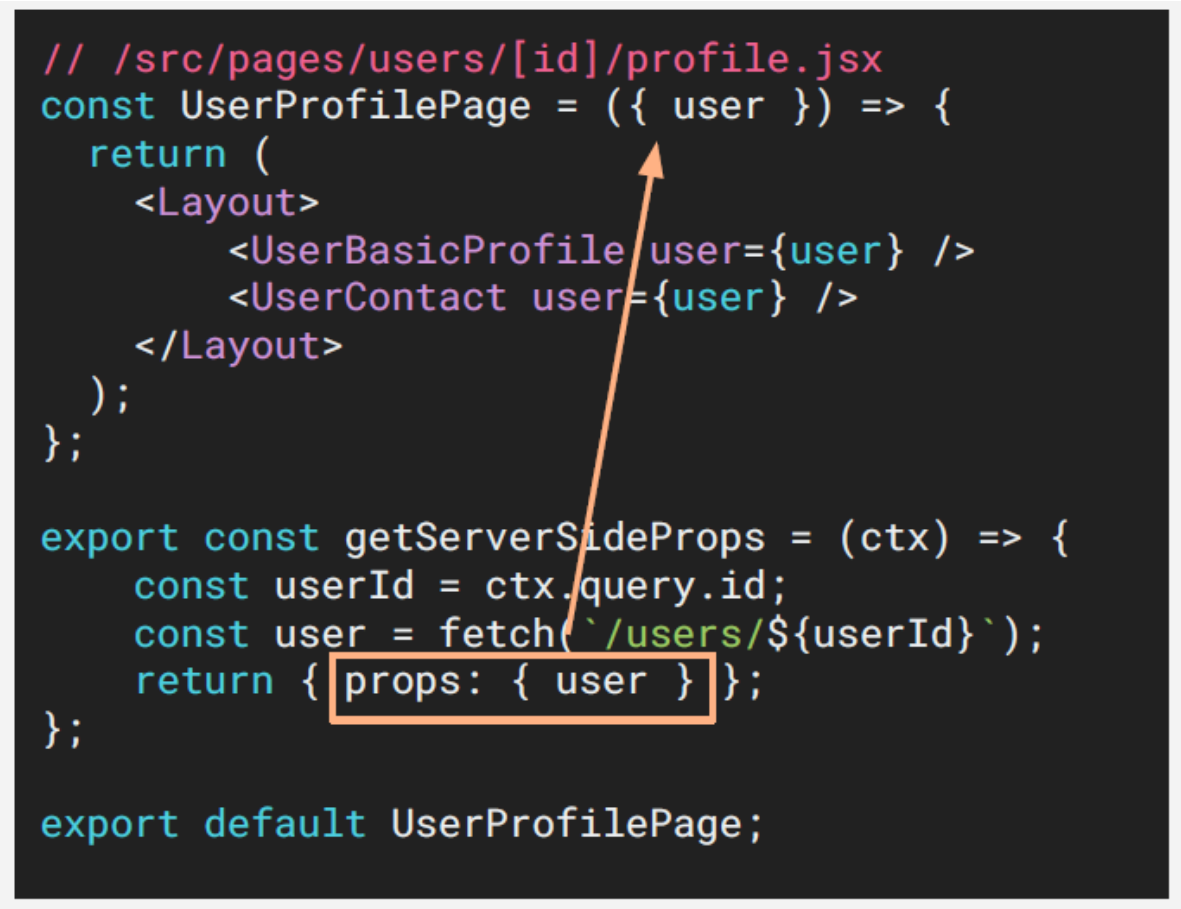
getServerSideProps

If you export a function called **getServerSideProps** (Server-Side Rendering) from a page, Next.js will render this page on the server on each request using the data returned by **getServerSideProps**

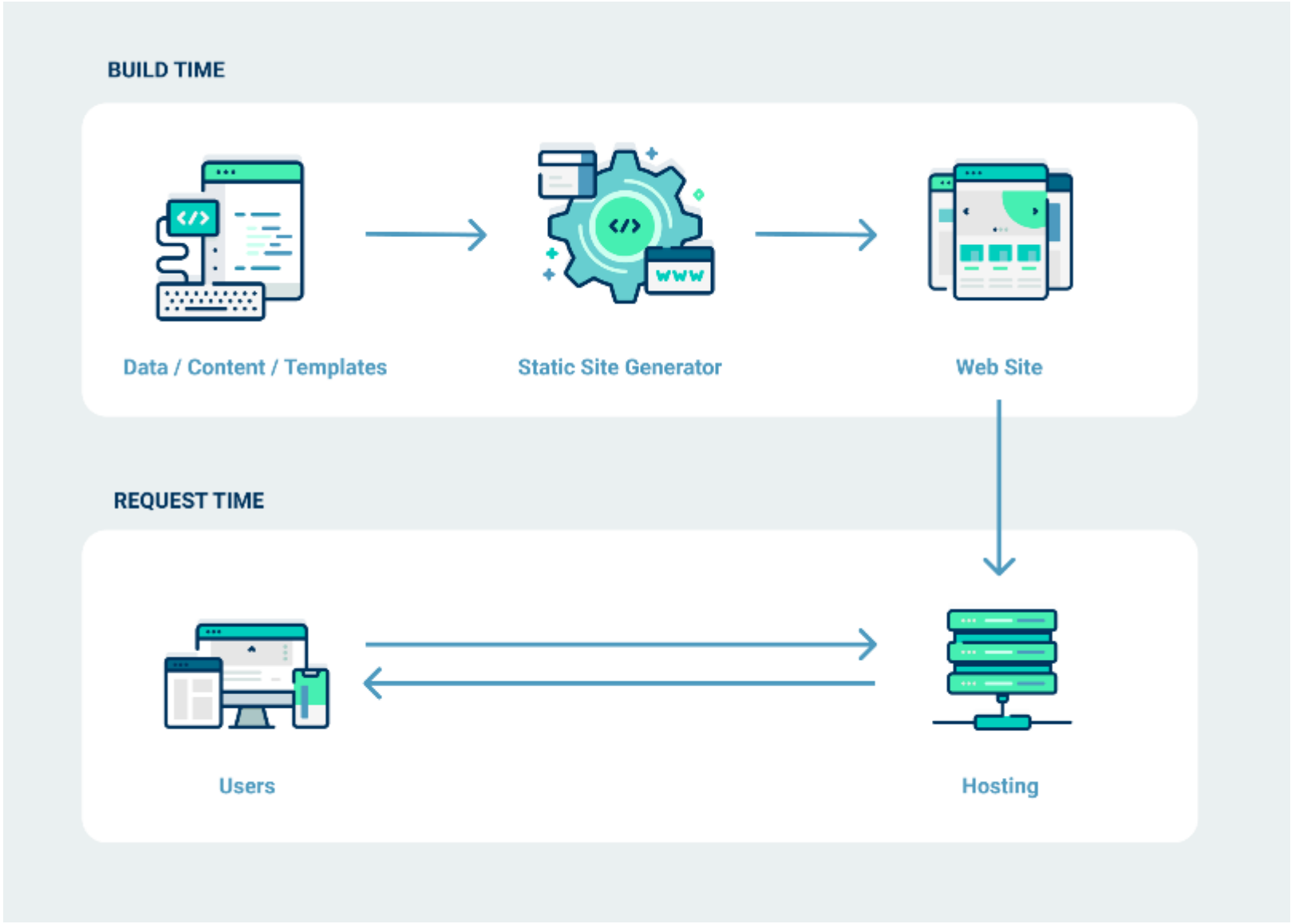
```
function Page({ data }) {
  // Render data...
}

// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  // Pass data to the page via props
  return { props: { data } }
}

export default Page
```

getServerSideProps - Example

- **getServerSideProps** is called on the server-side to get props for page component



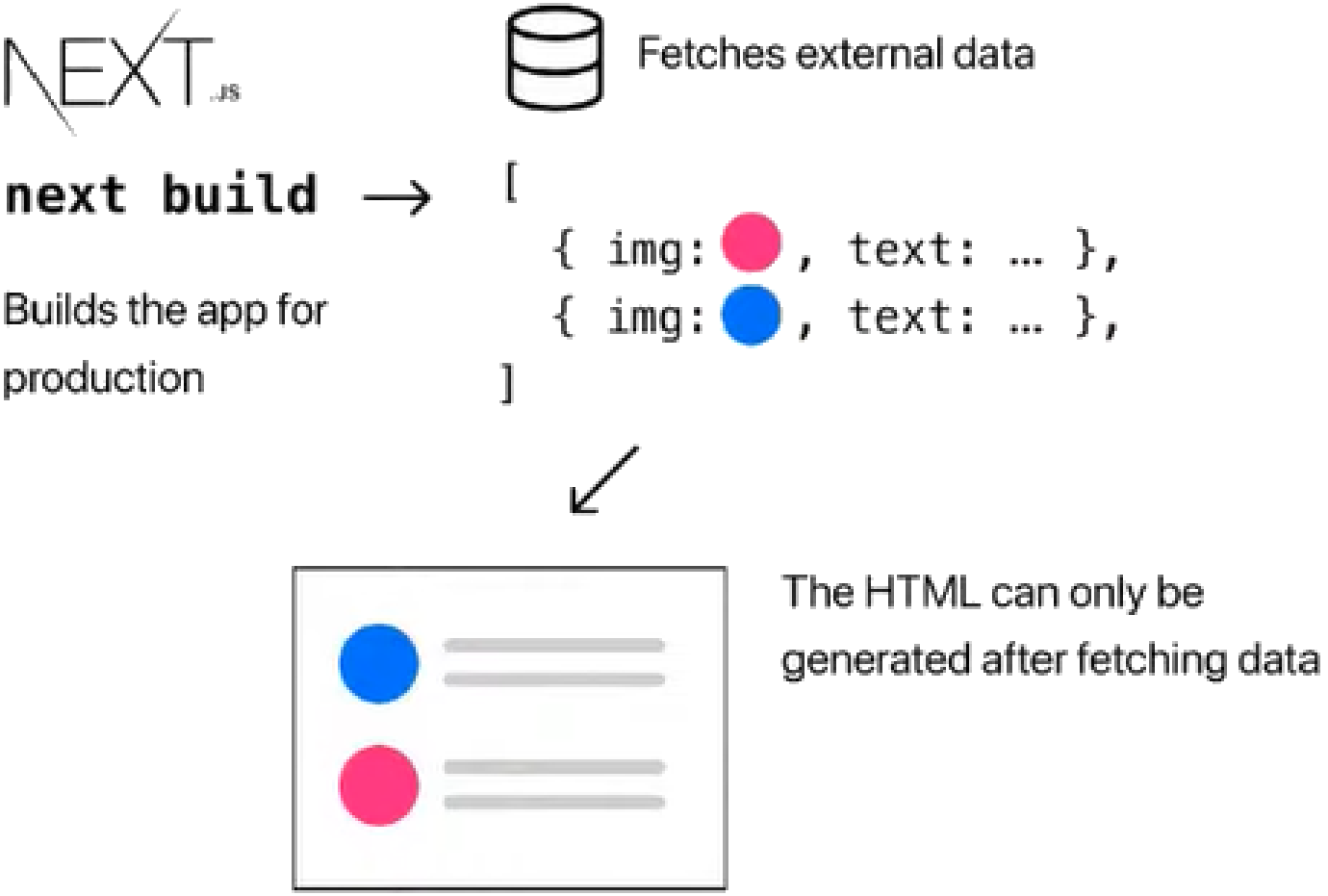
SSG



https://www.netlify.com/blog/2020/04/14/what-is-a-static-site-generator-and-3-ways-to-find-the-best-one/

Static Generation with Data

- For pages that can only be generated after fetching external data at build time



getStaticProps

If you export a function called **getStaticProps** (Static Site Generation) from a page, Next.js will pre-render this page at build time using the props returned by **getStaticProps**

```
// posts will be populated at build time by getStaticProps()
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}

// This function gets called at build time on server-side.
export async function getStaticProps() {
  // Call an external API endpoint to get posts.
  // You can use any data fetching library
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}

export default Blog
```

Statically Generate Pages with Dynamic Routes

If you want to statically generate a page at a path called `/posts/<id>` where **`<id>`** can be dynamic, then...

Create a page at **`/pages/posts/[id].js`**

The page file must contain:

1. A React component to render this page
2. **`getStaticPaths`** which returns an array of possible values for **`id`**
3. **`getStaticProps`** which fetches necessary data for the post with **`id`**

getStaticPaths

When you export a function called **getStaticPaths** (Static Site Generation) from a page that uses dynamic routes, Next.js will statically pre-render all the paths specified by **getStaticPaths**

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

// Generates `/posts/1` and `/posts/2`
export async function getStaticPaths() {
  return {
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
    fallback: false, // can also be true or 'blocking'
  }
}

// `getStaticPaths` requires using `getStaticProps`
export async function getStaticProps(context) {
  return {
    // Passed to the page component as props
    props: { post: {} },
  }
}

export default function Post({ post }) {
  // Render post...
}
```

```
getStaticPaths() {
  return  {
      paths : [.....] ,
      fallback : true / false
  }
}
```

Fallback

- fallback: false

only pages that are generated during next build (i.e. returned from the paths property of getStaticPaths) will be visible

E.g., if a user creates a new blog page at `/post/[post-id]`, it will not be immediately visible afterwards, and visiting that URL will lead to a 404.

Fallback

fallback property can accept 3 values:

- false : new paths will result in a 404 page
- true : new path will be statically generated (getStaticProps is called) - loading state is shown while generating page (via router.isFallback and showing fallback page)
- page is rendered with required props after generating
- new path will be cached in CDN (later requests will result in cached page) - crawler Bots may index fallback page (not good for SEO)

fallback: true

```
function Video({ videoId }) {
  const router = useRouter()
  // If the page is getting generated
  if (router.isFallback) {
    return <div>Loading...</div>
  }
  // then  return the main component
}
```

Fallback

- "blocking" : new path will be waiting for HTML to be generated (via SSR)

- there will be no loading state(no fallback page)

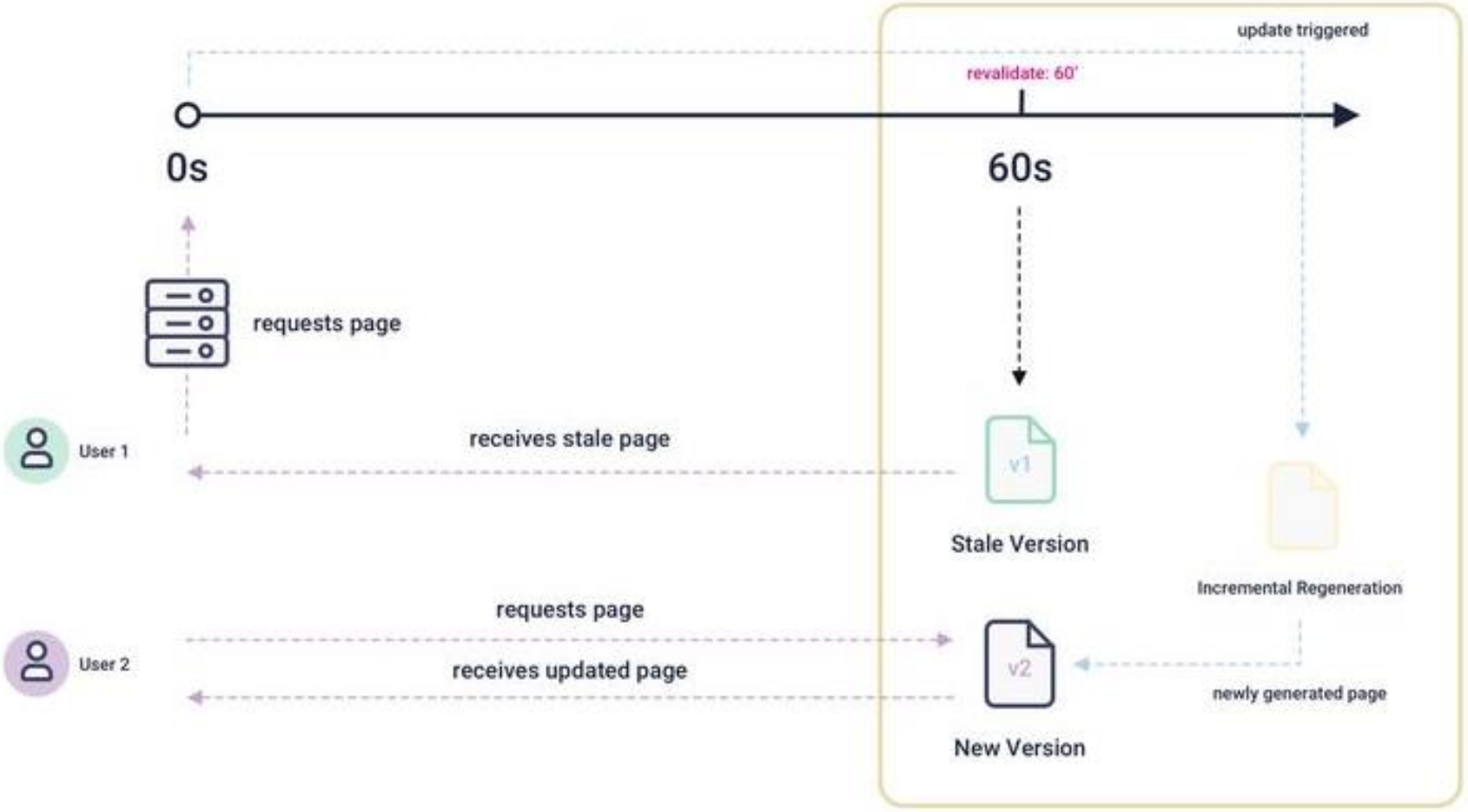
- new path will be cached (later requests will result in cached page)

SSG

- SSG: 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 your 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

Incremental Static Regeneration (ISR)

- ISR enables developers to use static generation on a per-page basis, without having to rebuild the entire site



ISR Example

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

export async function getStaticProps({ params }) {
  return {
    props: {
      product: await getProductFromDatabase(params.id)
    },
    revalidate: 60
  }
}
```

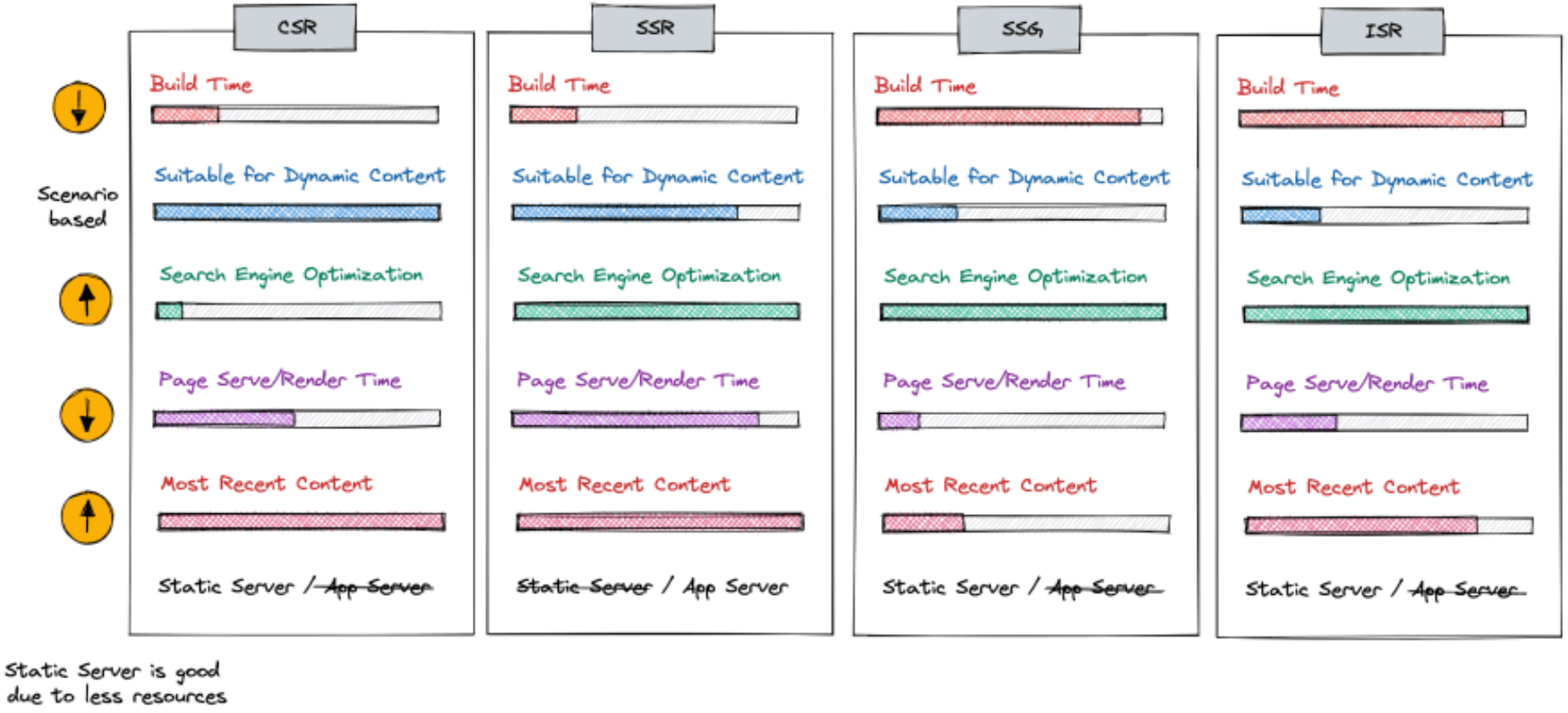
ISR

- Avoids long builds with unnecessary computation
  - IRS allows using static-generation on a per-page basis, without needing to rebuild the entire site.
  - Static pages can be generated at runtime (on-demand) instead of at build-time
  - E.g., When products changes only incrementally update those pages without needing a full rebuild

On-Demand Revalidation

https://nextjs.org/docs/basic-features/data-fetching/incremental-static-regeneration

Comparison



https://dev.to/pahanperera/visual-explanation-and-comparison-of-csr-ssr-ssg-and-isr-34ea

Summary

- Next.js has two forms of pre-rendering: **Static Generation** and **Server-side Rendering.** The difference is in **when** it generates the HTML for a page
- **Static Generation** is the pre-rendering method that generates the HTML at **build time.** The pre-rendered HTML is then *reused* on each request
- **Server-side Rendering** is the pre-rendering method that generates the HTML on **each request**
- Importantly, Next.js lets you **choose** which pre-rendering form to use for each page. You can create a "hybrid" Next.js app by using Static Generation for most pages and using Server-side Rendering for others

Resources

- Learn Next.js

http://nextjs.org/learn

- E-commerce Demo

https://nextjs.org/commerce

- Useful list of resources

https://github.com/unicodeveloper/awesome-nextjs