

~~NEXT~~.JS

Basics

Outline

1. Introduction
2. Next.js routing
3. Built-In Components
4. Fetching data on both the server and client side
5. Rendering Strategies
6. Managing Local and Global States
7. CSS and Built-In Styling Methods

Next.js vs React

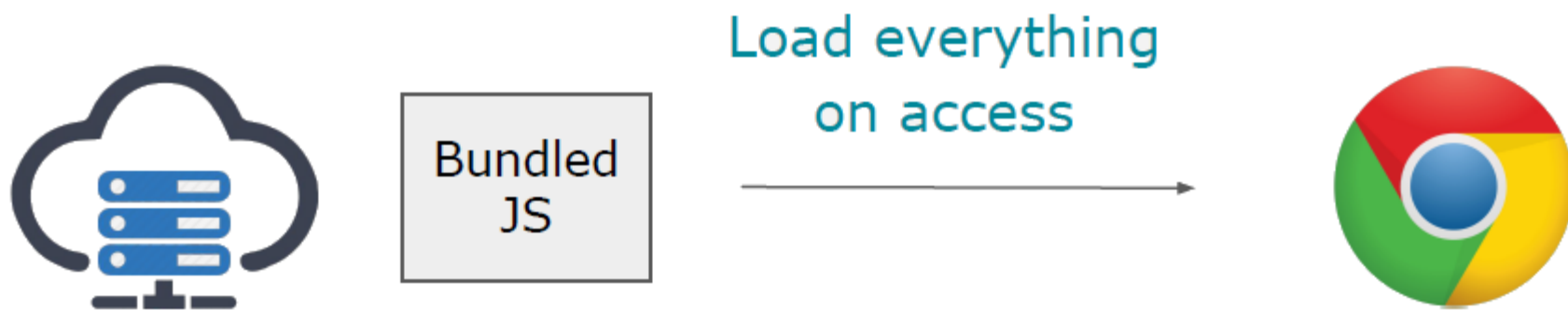
- React is just a **client-side JavaScript** library, Next.js is a framework for building rich and complete Web App **both on the client and server sides**
- React runs on the client side.
 - Could negatively affect Search Engine Optimization (SEO) and
 - Initial load performance. To display the complete web app, the browser had to download the entire application bundle, parse its content, then execute it and render the result in the browser, which could take up to a few seconds for a large applications

What is Next.js?

- Next.js = React framework that allows creating user interfaces, static pages, and server-side rendered pages
- It provides a large set of features out of the box, such as:
 - Automatic code-splitting
 - File system-based routing systems
 - Route prefetching
 - API Routes (Serverless Functions)
 - Automatic image optimization
 - Server-side rendering
 - Static site generation
 - Incremental static generation
 - Support for internationalization
 - Fast refresh on develop environment

Code splitting

- In SPA, a huge bundled file will be loaded as default



- With Next.js , code will be split on per page base as default



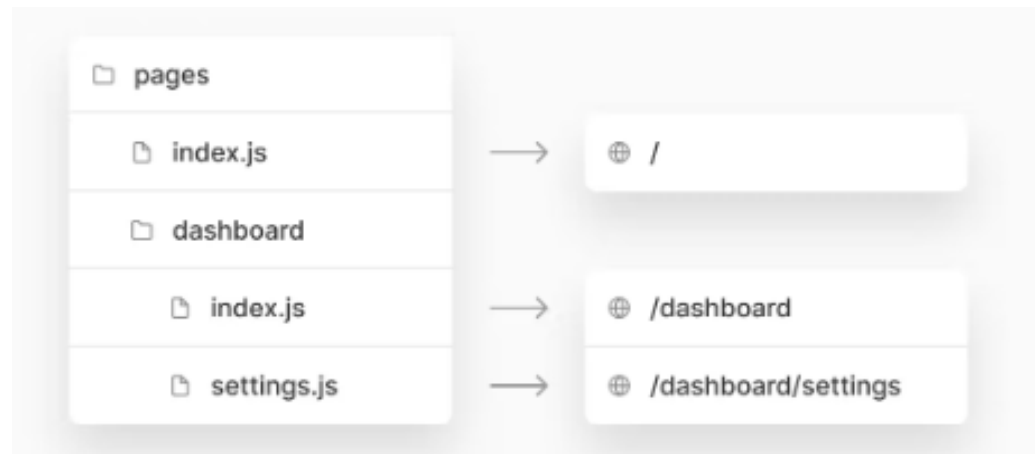
Getting started

- Install latest **Node.js** <https://nodejs.org/en/>
- Download **VS Code** <https://code.visualstudio.com/>
- Create an empty folder (with no space in the name use **dash** - instead)
- Create a react app
npx create-next-app .
- Run the app in dev mode: **npm run dev**
- Build the app: **npm run build**
- Run the optimized build: **npm run start**

Project Folder Structure

- Next.js uses **pages/** folder navigation, every JavaScript file inside it will be a public page
 - the **pages/** directory is a container for the app public pages
- The **public/** folder contains all the public and static assets such as images, fonts, etc.
- **public/** and **pages** are mandatory and reserved directories so make sure not to delete or use them for different purposes
- **styles/** optional folder for organizing stylesheets

Routing



Routing

- Next.js provides file system-based router
- One .js file -> one page

src

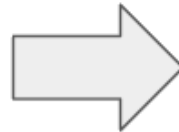
└ pages

├ index.jsx

└ users

└ [id]

└ profile.js



/

/users

└ [id]

└ profile

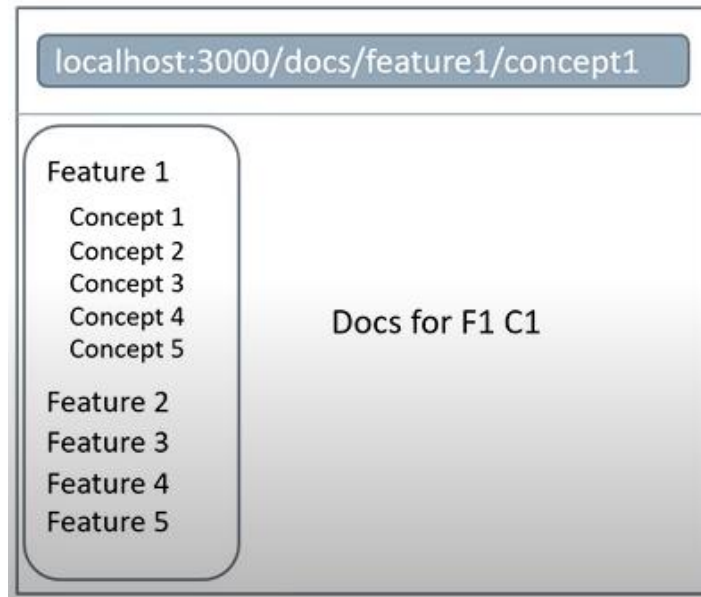
File paths in project

Routing paths

Routing

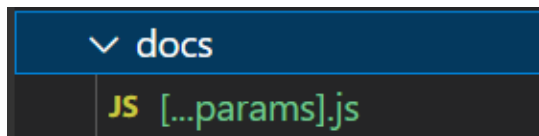
- Page based routing - Pages are associated with a route based on their file name
- Nested routes - Nested folder structure, files will be automatically routed in the same way in the URL
- Dynamic routes - Can be created by adding square brackets to a page name
- Catch all routes - Add three dots inside square brackets to create a catch all route. Helpful when you want different URLs for the same page layout or even when you're working with pages where some of the route parameters are optional
- Link component to navigate on click of an element
- useRouter hook's routerpush method to navigate programmatically
- Custom 404 page

Catch-All Route



```
const router = useRouter()
const { params } = router.query
console.log(params)

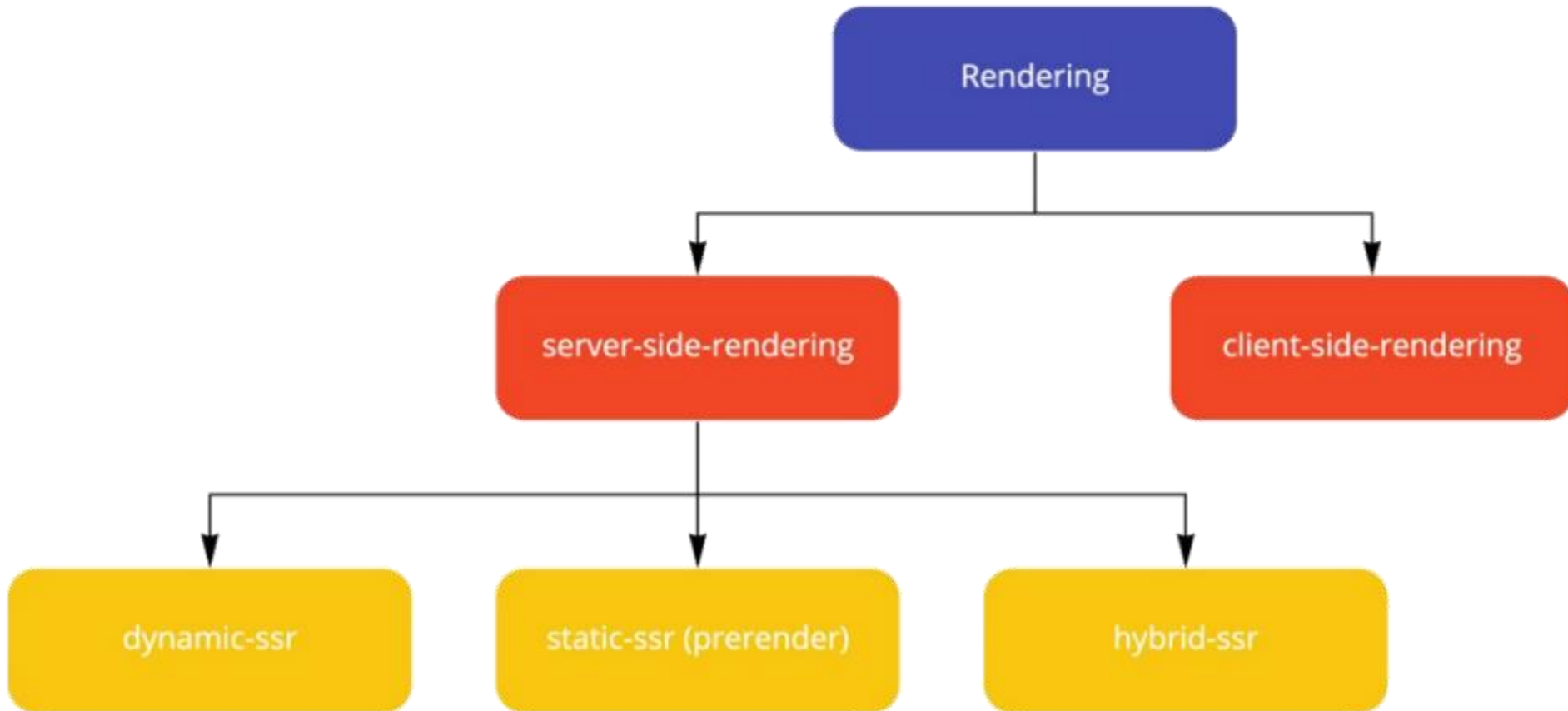
if (params.length === 2) {
  return (
    <h1>
      Viewing docs for feature {params[0]} and concept {params[1]}
    </h1>
  )
} else if (params.length === 1) {
  return <h1>Viewing docs for feature {params[0]}</h1>
}
```



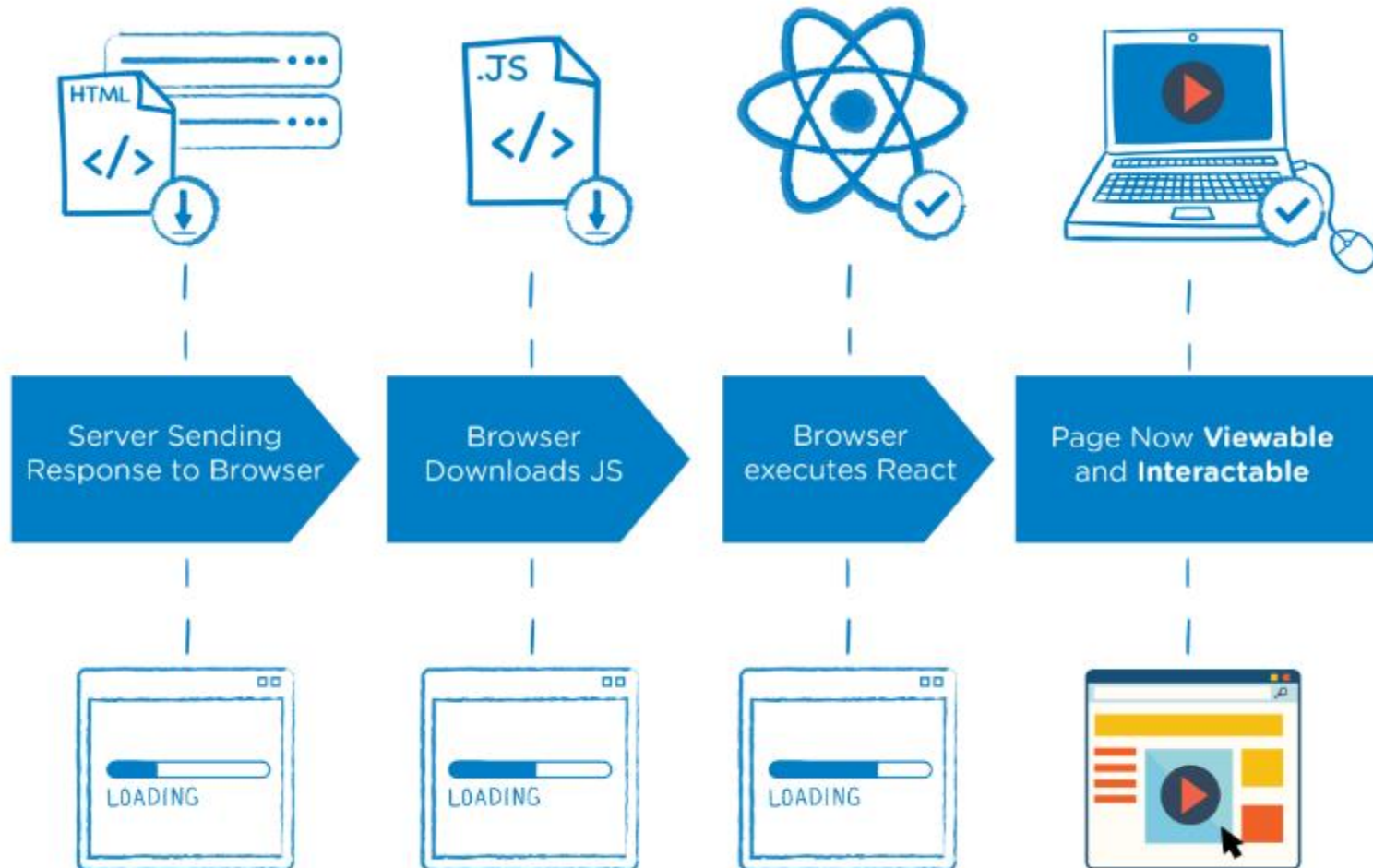
Rendering Strategies

- 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

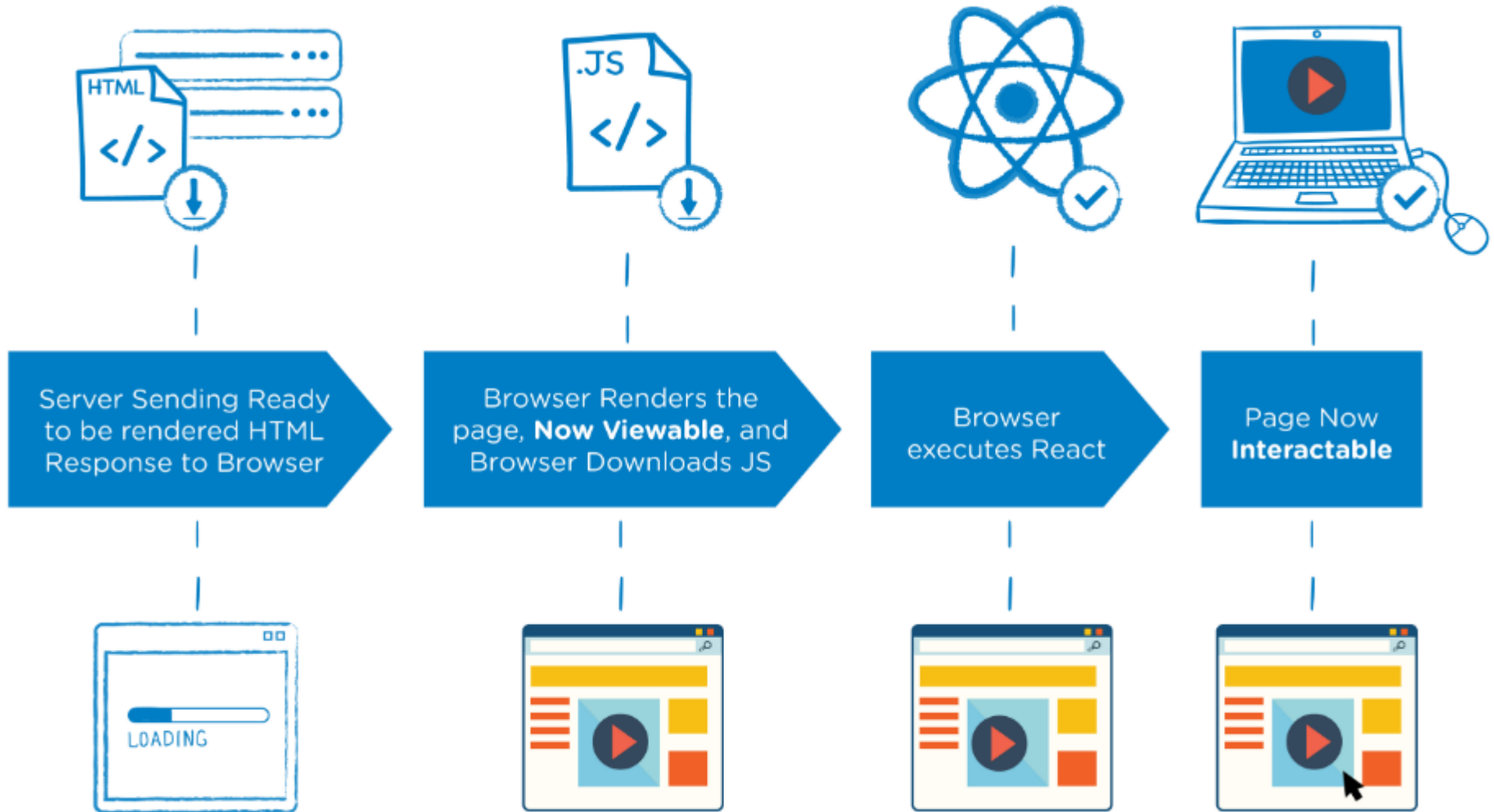
Rendering Strategies



CSR



SSR



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

```
// /src/pages/users/[id]/profile.jsx
const UserProfilePage = ({ user }) => {
  return (
    <Layout>
      <UserBasicProfile user={user} />
      <UserContact user={user} />
    </Layout>
  );
};

export const getServerSideProps = (ctx) => {
  const userId = ctx.query.id;
  const user = fetch(`/users/${userId}`);
  return { props: { user } };
};

export default UserProfilePage;
```

An orange arrow originates from the 'props' object in the return statement of the 'getServerSideProps' function and points to the 'user' prop in the 'UserBasicProfile' and 'UserContact' components within the 'UserProfilePage' function. A light orange box highlights the 'props: { user }' object in the return statement.

SSG

BUILD TIME

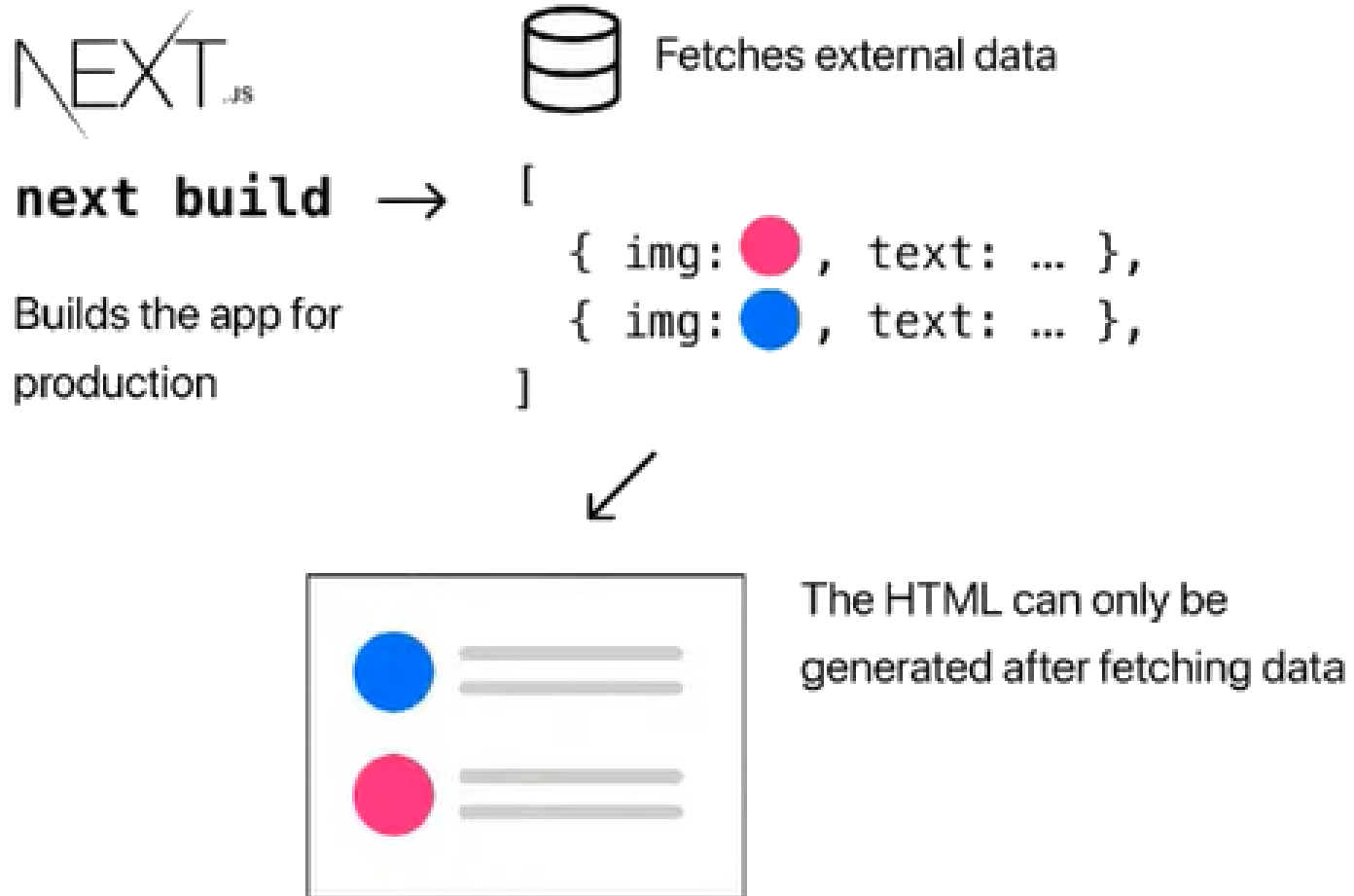


REQUEST TIME



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
```

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

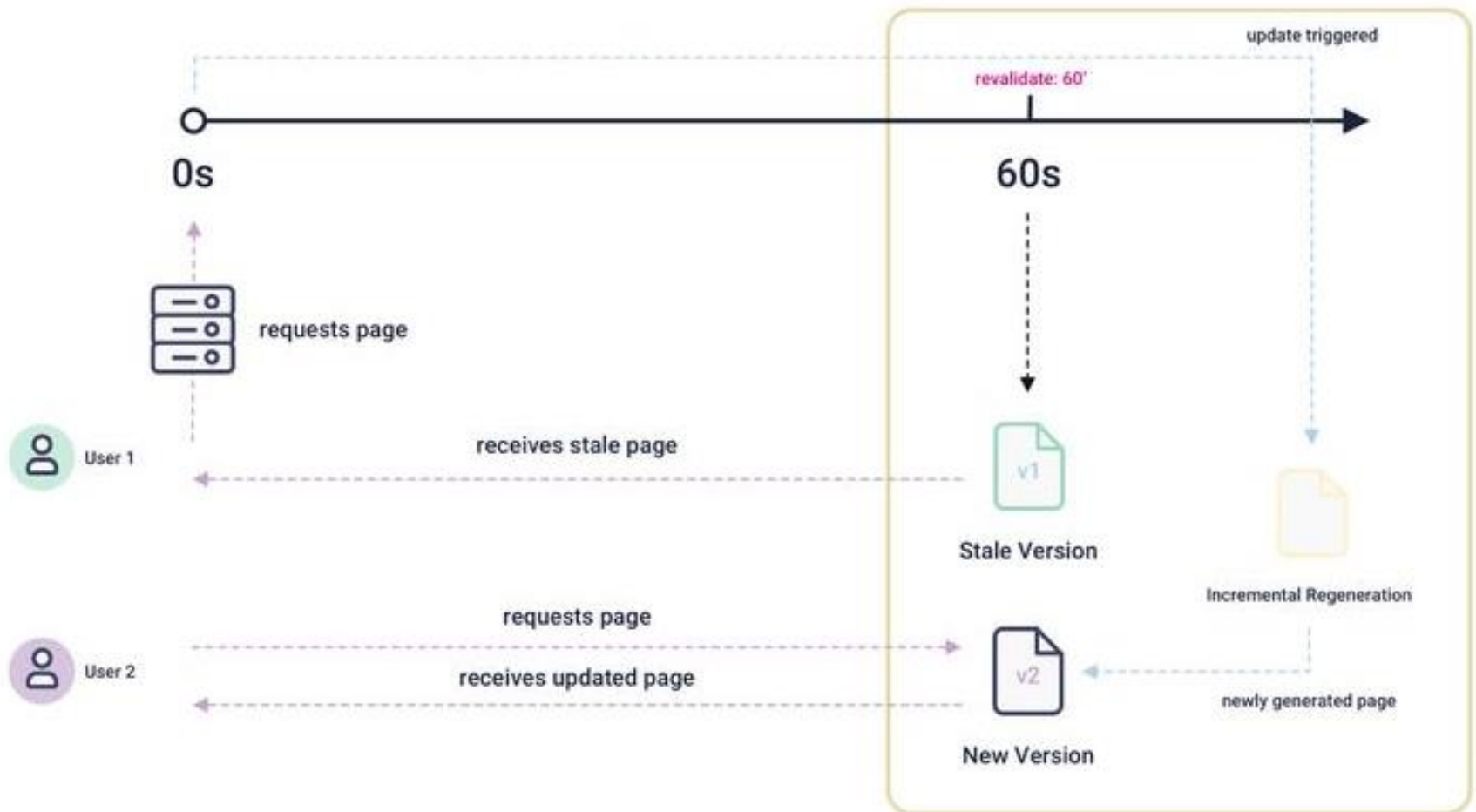
- "blocking" : new path will be waiting for HTML to be generated (via SSR)
 - there will be no loading state(no fallback page)
 - new path will be cached (later requests will result in cached page)

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
  }
}
```

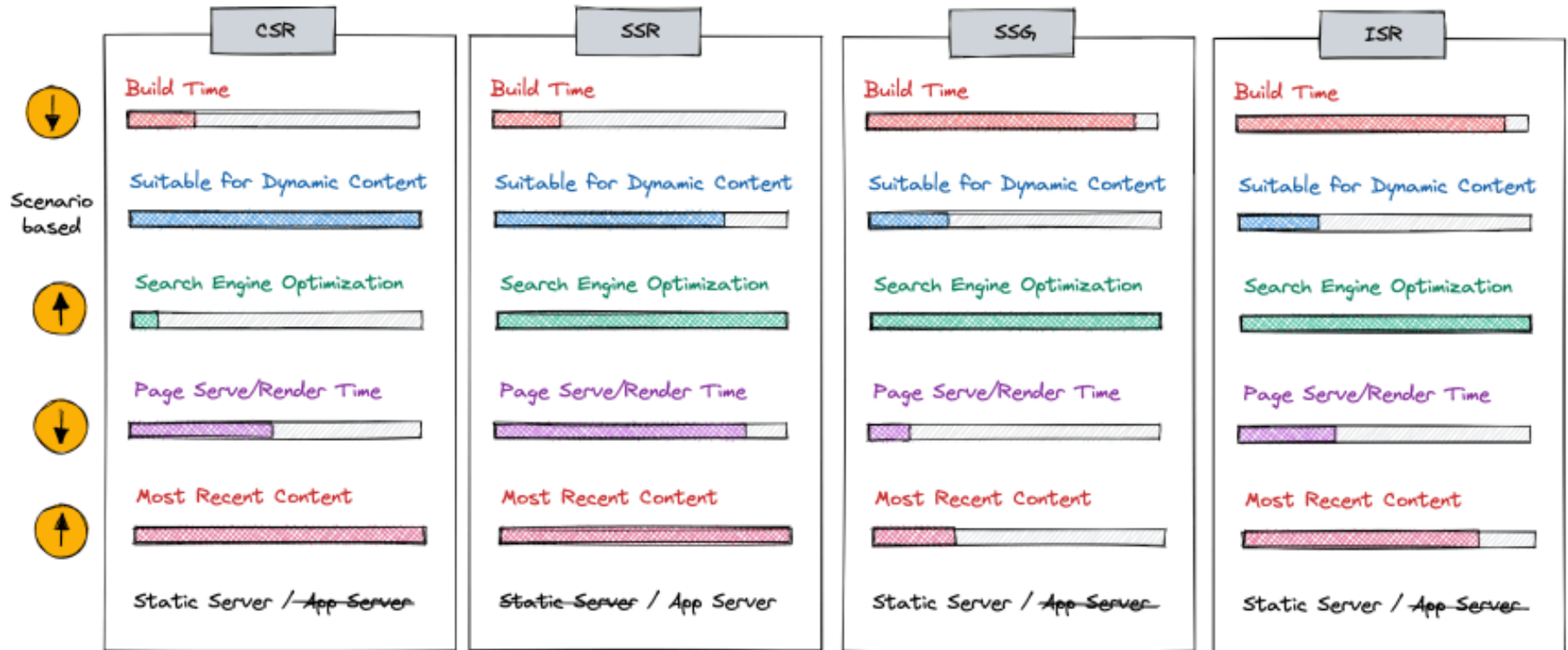
ISR

- 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



Static Server is good
due to less resources

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

Summary

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>