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

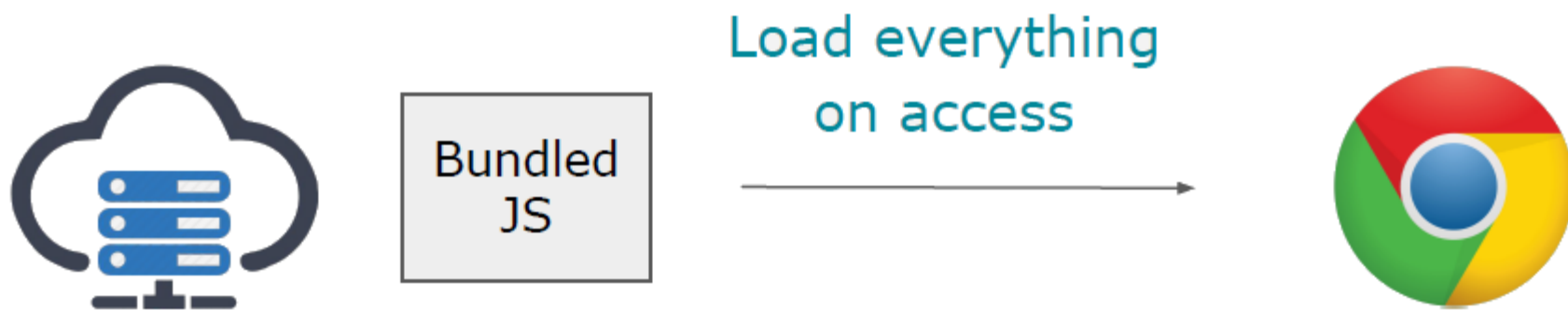7. CSS and Built-In Styling Methods

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 application

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:

  o Automatic code-splitting

  o File system-based routing systems

  o Route prefetching

  o API Routes

  o Automatic image optimization

  o Different rendering strategies: Server-side rendering, Static site generation, Incremental static generation

  o Support for internationalization

  o Fast refresh on the development environment

4

Code splitting

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

Load everything on access

Bundled JS

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

On access to index

On access to about

JS for index

JS for about

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 for routing, every JavaScript file inside it will be a page

    - the pages/ directory is a container for the app 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
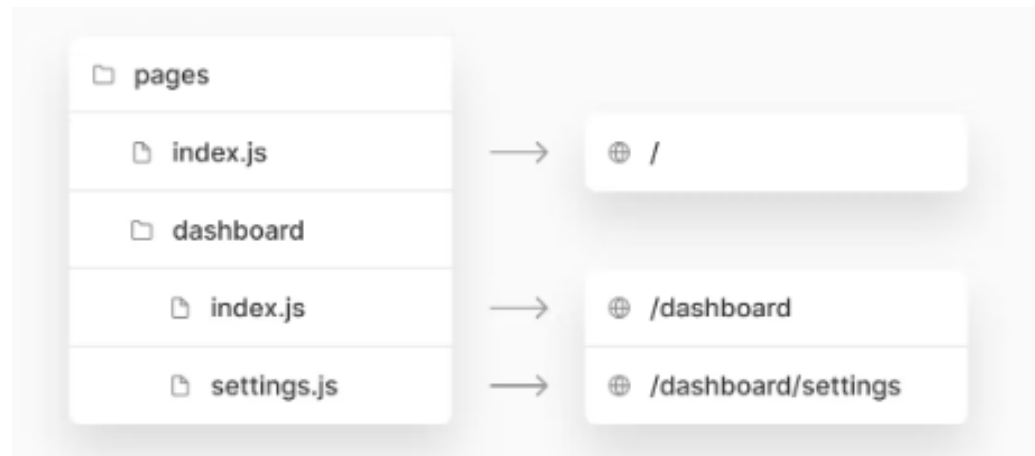
Routing

- Next.js has a **file-system based router**: when a file is added to the **pages** directory, it's automatically available as a route

- Index routes: the router automatically routes files named index to the root of the directory

```
`pages/index.js` → `/`
`pages/blogs/index.js` → `/blogs`
```

- Nested routes: nested folder structure is automatically mapped to the corresponding routes

```
`pages/blogs/first-post.js` → `/blogs/first-post`
`pages/dashboard/settings/profile.js` → `/dashboard/settings/profile`
```

Dynamic Routes

- In Next.js you can add brackets to the file name of a page to create a dynamic route

```
`pages/blogs/[id].js` → `/blogs/:id` (`/blogs/9`)
`pages/[username]/settings.js` → `/:username/settings` (`/foo/settings`)
```

- **catch-all dynamic routes**: allows a dynamic route to catch all paths by simply adding three dots (…) inside the brackets

e.g., The catch all page in **pages/blogs/[...params].js** will match any path underneath **/blogs** such as: /blogs/2022, /blogs/2022/3/10, and so on

- Matched parameters will be assigned as an array to the **router.query** property, so the path /blogs/2022/3/10 will have the following query object: { **"params"**: ["2022", "3", "10"] }

Optional catch all routes

- Catch all routes can be made optional by including the parameter in double brackets (**[[...params]]**).

    o route without the parameter is also matched

- For example, pages/posts/**[[...params]]** will match /blogs and any path underneath it such as: /blogs/2022, /blogs/2022/3/10, and so on

- The query objects are as follows:

```
{ } // GET `/blogs` (empty object)
// `GET /blogs/2022` (single-element array)
{ "params": ["2022"] }
// `GET /blogs/2022/3` (multi-element array)
{ "params": ["2022", "3"] }
```
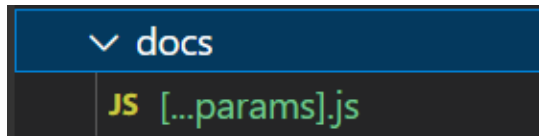
Catch-All Route Example

See posted example
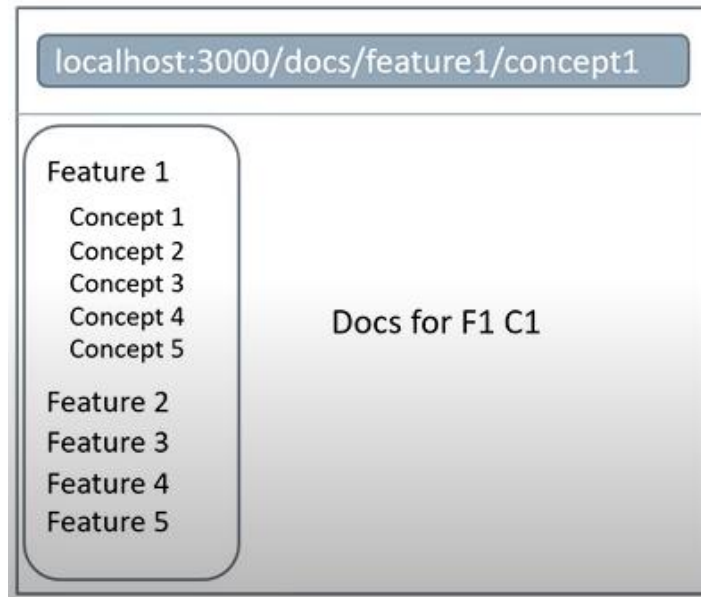

```
const router = useRouter()
const { params } = router.query
console.log(params)

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>
}
```

Linking between pages

- The Next.js router provides a React component called **Link** to do **client-side route** transitions between pages, similar to a single-page application

    o **href** specify the route associated with the link

    o Pages for any <Link /> in the viewport (visible to the user) will be prefetched by default (including the corresponding data) for pages using Static Generation. The corresponding data for server-rendered routes is not prefetched.

```
import Link from 'next/link'
export default function Home() {
return ( <ul>
    <li> <Link href="/"> <a>Home</a> </Link> </li>
    <li> <Link href="/about"> <a>About Us</a> </Link> </li>
  </ul>)
}
```

Linking to dynamic paths

- Links can be created for dynamic paths

E.g., creating links to access posts for a list which have been passed to the component as a prop

```
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blogs/${post.id}`}>
            <a>{post.title}</a>
          </Link>
        </li>
      ))}
    </ul>
  )
}
```

useRouter

- **useRouter** hook to access the router object inside any app component

- Router properties include:
  - ○ **query**: returns the query string parsed to an object, including dynamic route parameters
  - ○ **asPath**: returns the path as shown in the browser including the query params

```
import { useRouter } from 'next/router'
const Post = () => {
    const router = useRouter()
    const { pid } = router.query
    return <p>Post: {pid}
      Path: router.asPath </p>
} export default Post
```

For /posts/1
pid will be **1**
Router.asPath will return /posts/1

Router push method

- Router **push** method can be used for programmatic client-side routing

E.g., navigating to *pages/about.js*

```javascript
import { useRouter } from 'next/router'

export default function ReadMore() {
  const router = useRouter()

  return (
    <button onClick={() => router.push('/about')}>
      Click here to read more
    </button>
  )
}
```

API Routes

- Simply add a handler function to a js file under `pages/api`

- Every file inside api is treated as a Web API endpoint e.g., **hello.js** in **pages/api**

```
// req = HTTP incoming message, res = HTTP server response
export default function handler(req, res) {
  res.status(200).json({ text: 'Hello' });
}
```

- Visiting http://localhost:3000/api/hello will return a json document

```
{ "text": "Hello" }
```

Dynamic API Routes

- API routes support dynamic routes, and follow the same file naming rules used for pages.

- For example, the API route pages/api/posts/[pid].js has the following code:

```
export default function handler(req, res) {
  const { pid } = req.query
  res.end(`Post: ${pid}`)
}
```
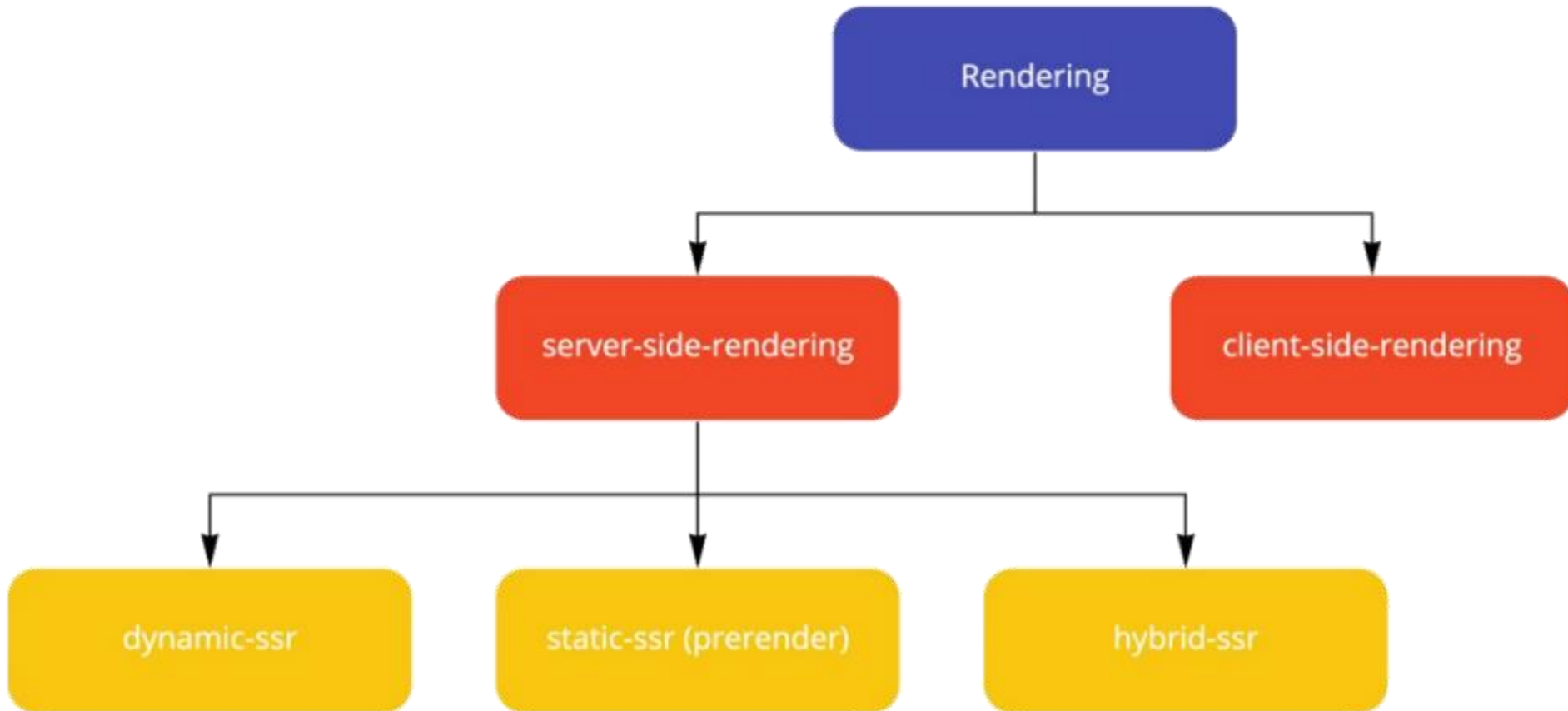
Handling HTTP Verbs

- By default, all HTTP request methods (GET, PUT, DELETE, and so on) will return the same response

- To handle different methods in an API route we can write the handler function with switch statement

```javascript
export default function handler(req, res) {
  const requestMethod = req.method;
  const body = JSON.parse(req.body);
  switch (requestMethod) {
    case 'POST':
      res.status(200).json({ message: `You submitted the following data: ${body}` })

    // handle other HTTP methods
    default:
      res.status(200).json({ message: 'Welcome to API Routes!'})
  }
}
```
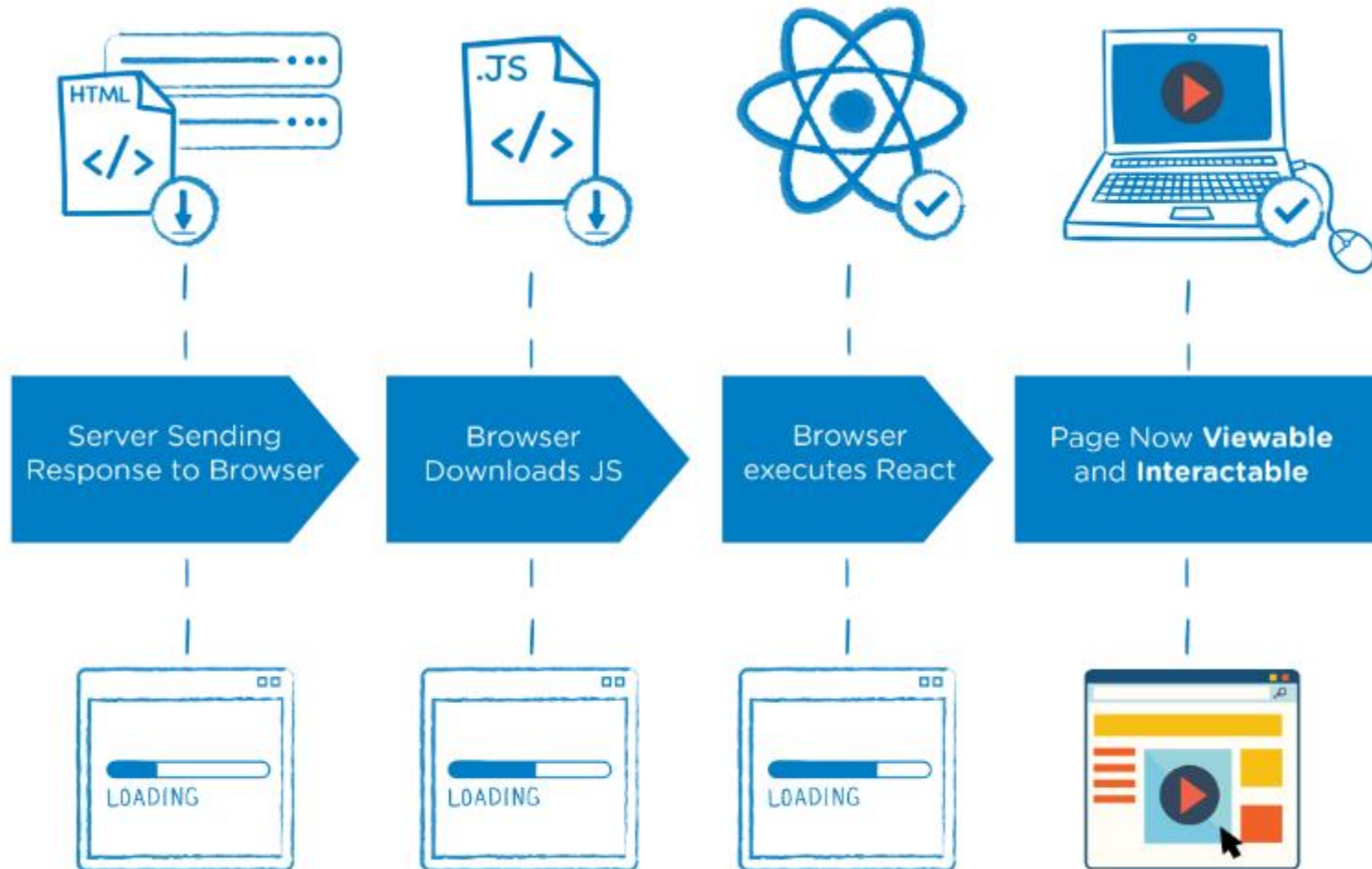
Rendering Strategies

- 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
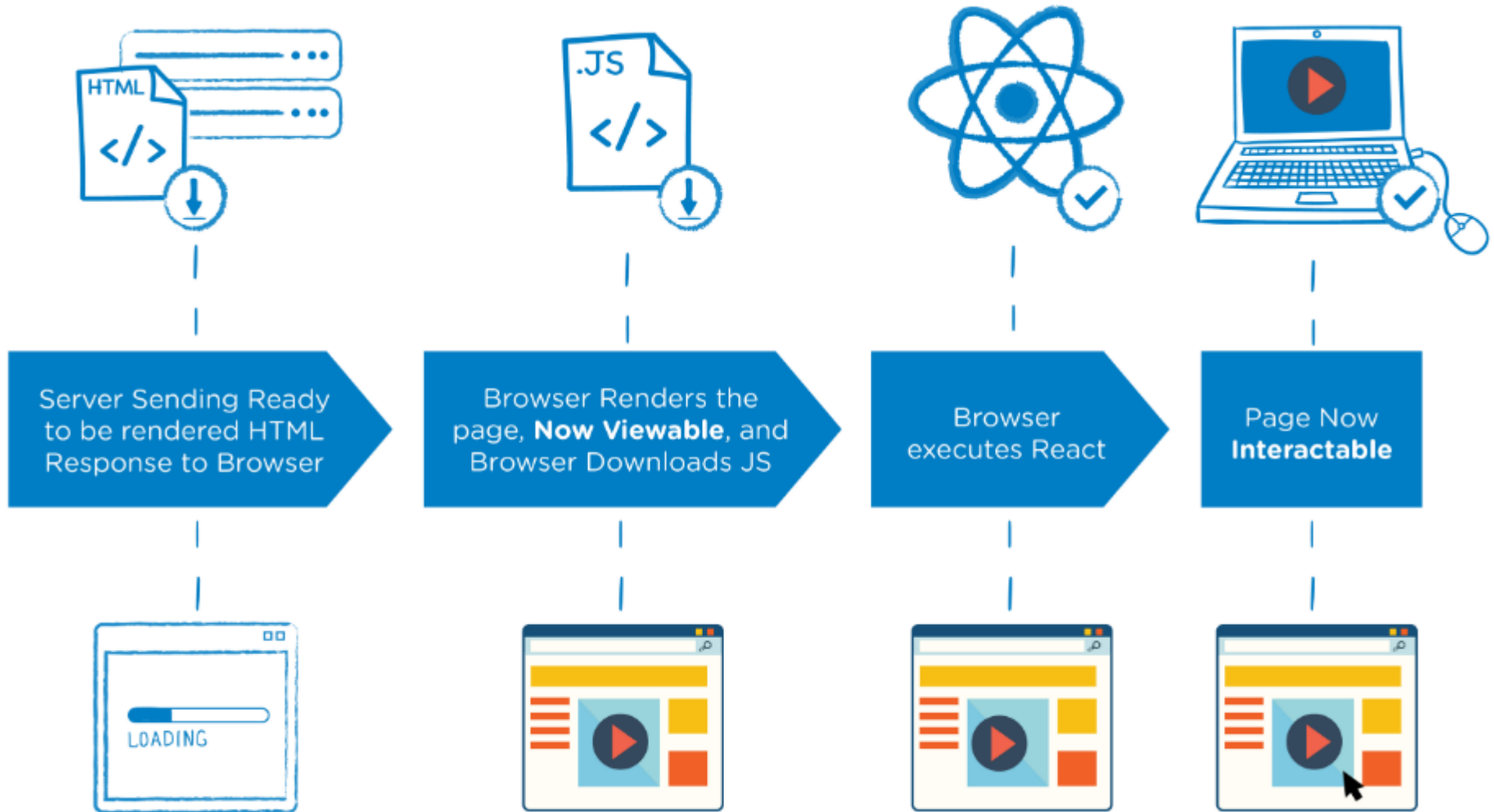
Rendering Strategies

CSR

Server Sending Response to Browser → Browser Downloads JS → Browser executes React → Page Now **Viewable** and **Interactable**

SSR

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**

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

```jsx
// /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;
```

SSG

BUILD TIME

Data / Content / Templates → Static Site Generator → Web Site

REQUEST TIME

Users → Hosting

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

27
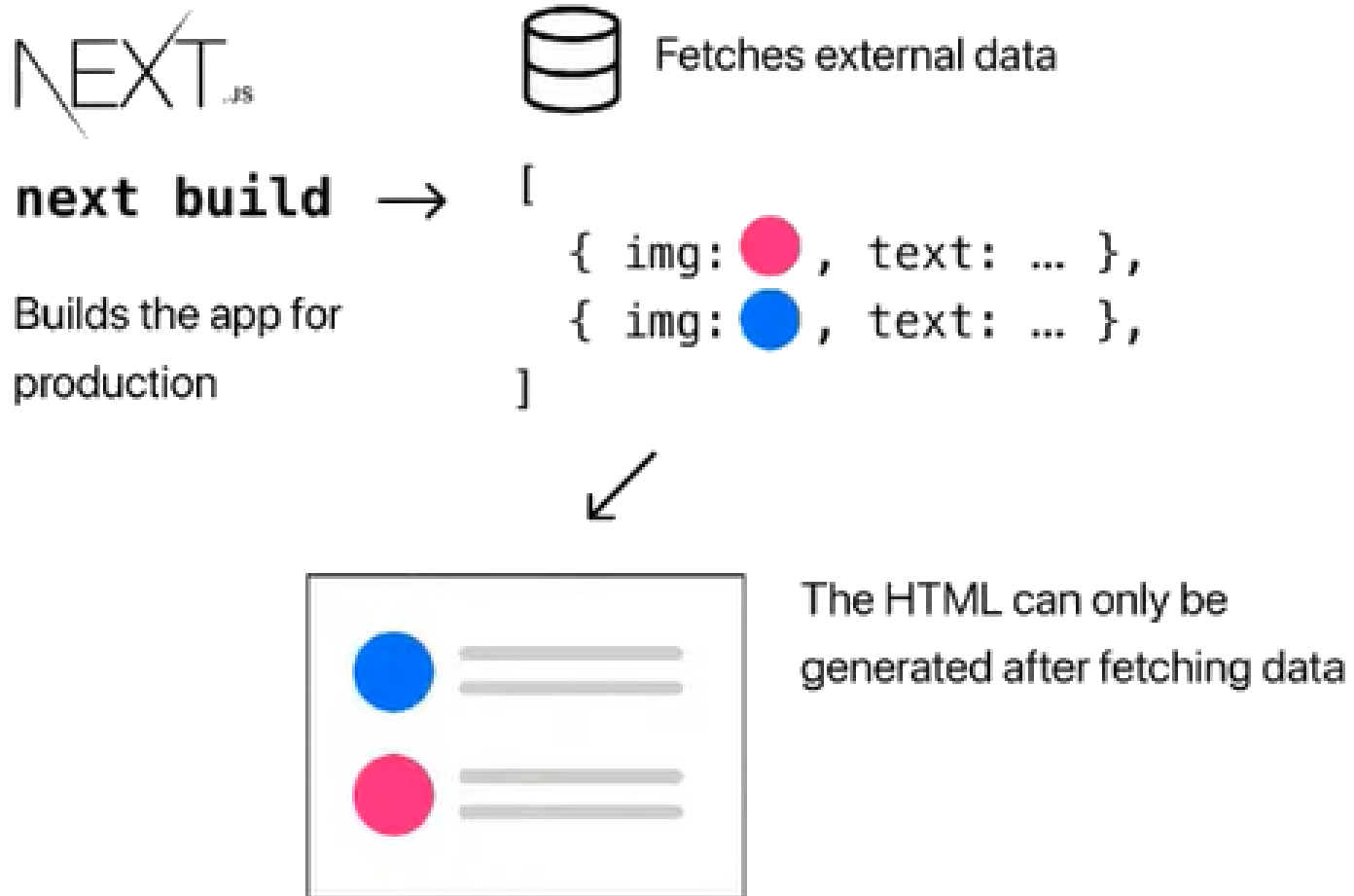
Static Generation with Data

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

NEXT.js

next build →

Builds the app for production

Fetches external data

```
[
  { img: ● , text: … },
  { img: ● , text: … },
]
```

The HTML can only be generated after fetching data

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

```javascript
// 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

  o Static content is fast, resilient to downtime, and immediately indexed by crawlers

  o For building a large-scale static site, it may take hours for your site to build.

  o 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

    o 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
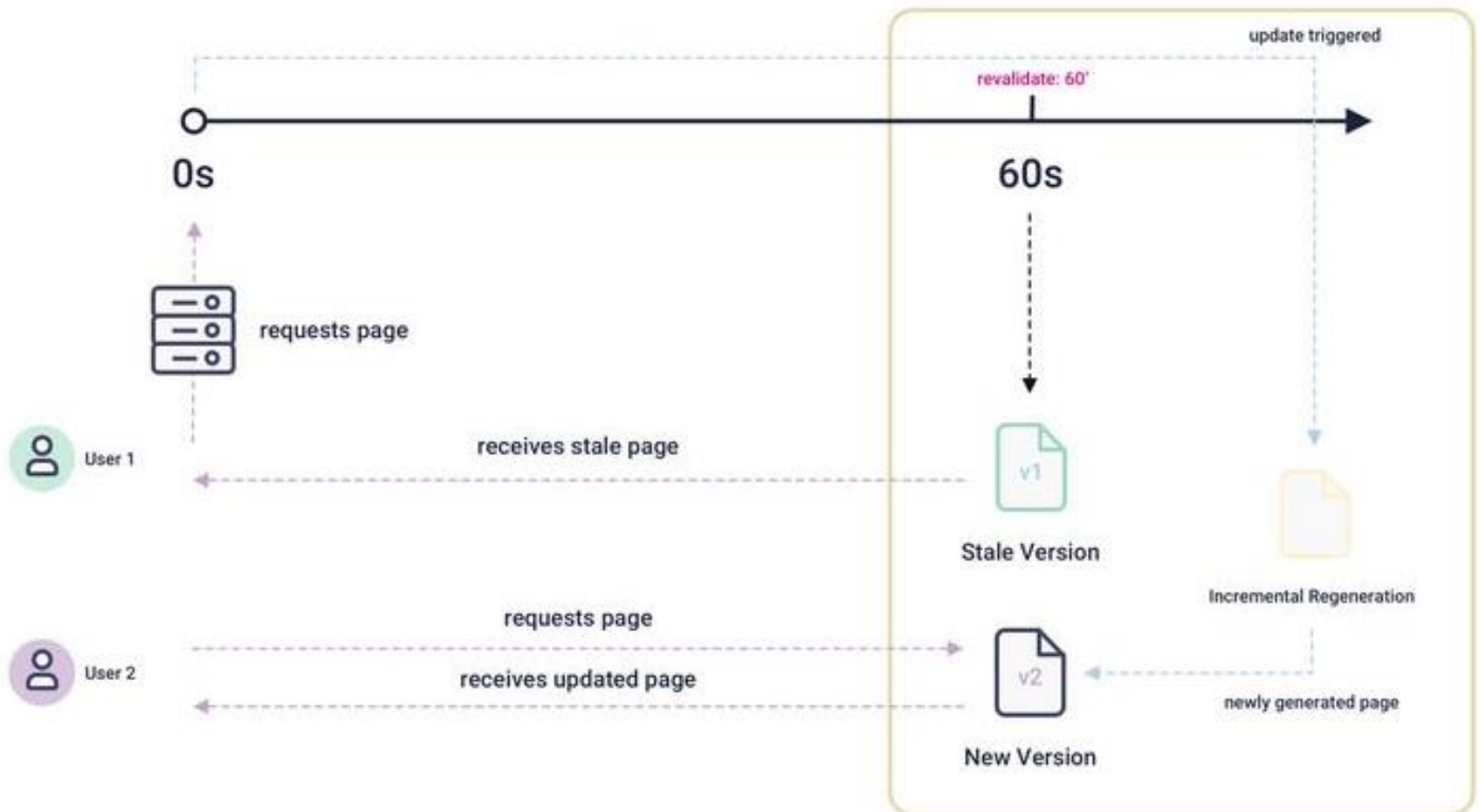
ISR Example

```
// 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
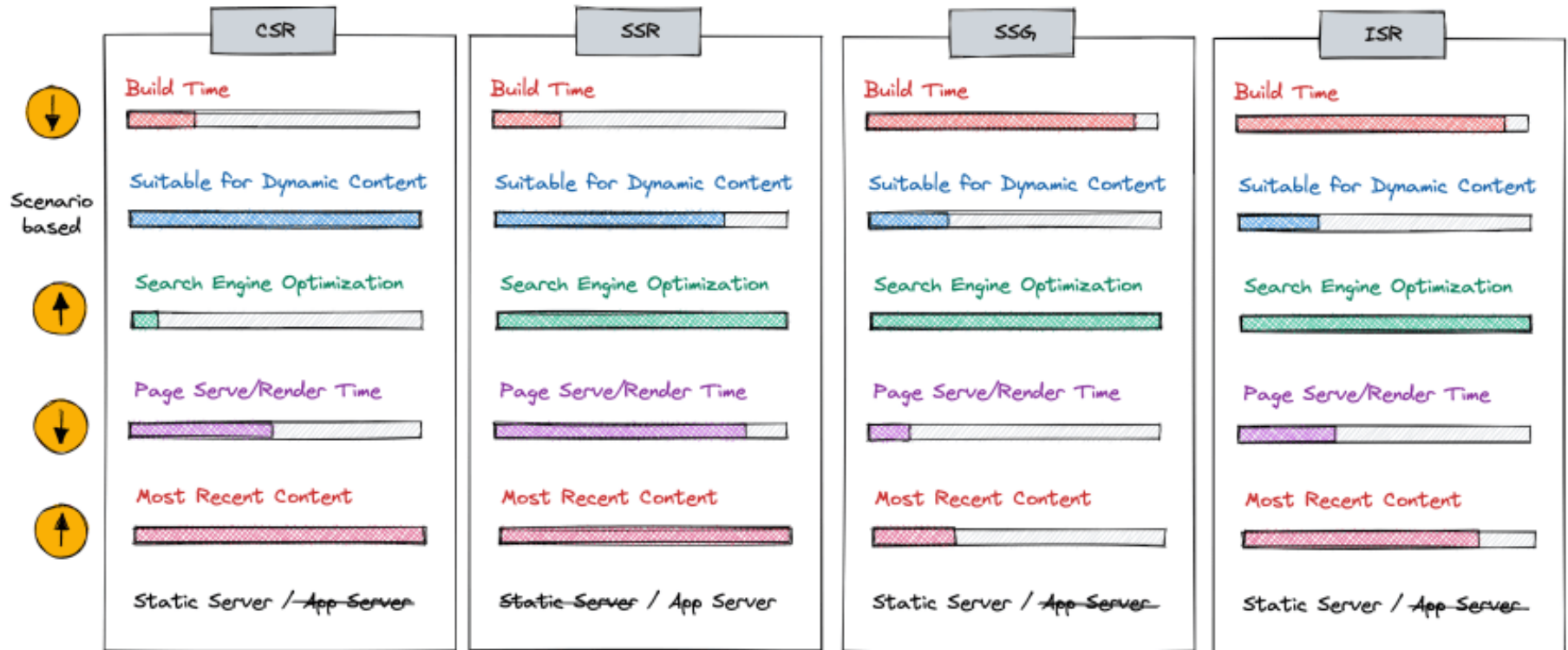
Summary

- 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