JOONAS STENGÅRD

SSR IN MODERN REACT DEVELOPMENT

Server-side rendering in React applications from fundamentals to advanced techniques

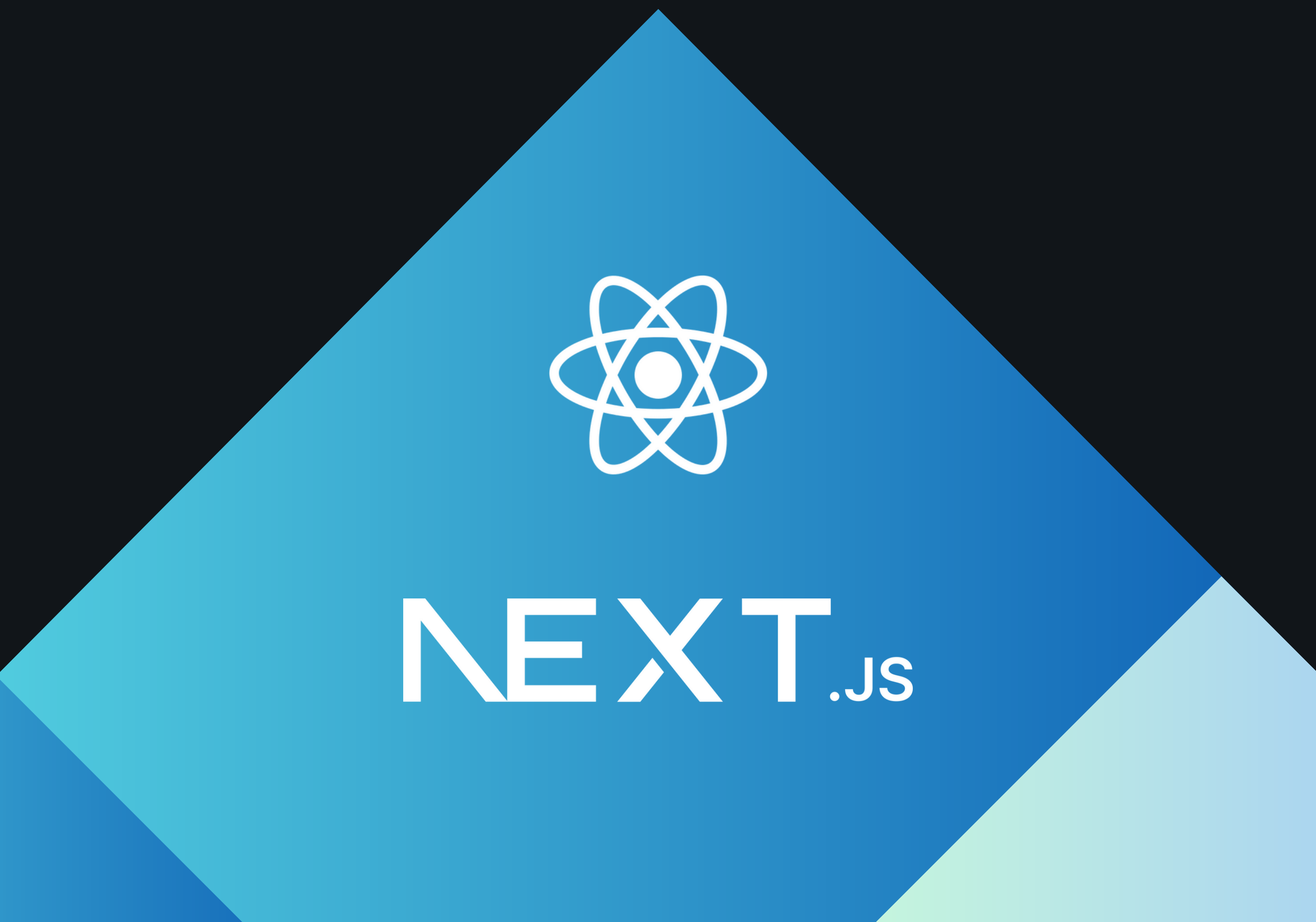


TABLE OF CONTENT

1: Introduction to server-side rendering in web development

SSR
Server-Side Rendering

CSR
Client-Side Rendering

The two web page rendering methods of the industry

There are two main approaches to rendering web pages in web development. While the topic of this book is server-side rendering, first we must understand the main differences between the methods to gain a better understanding on the benefits of each approach. This chapter of the book is programming language agnostic as dealing with the very introductory fundamentals is core to all web development. In later chapters of the book we will venture further into React specific topics.

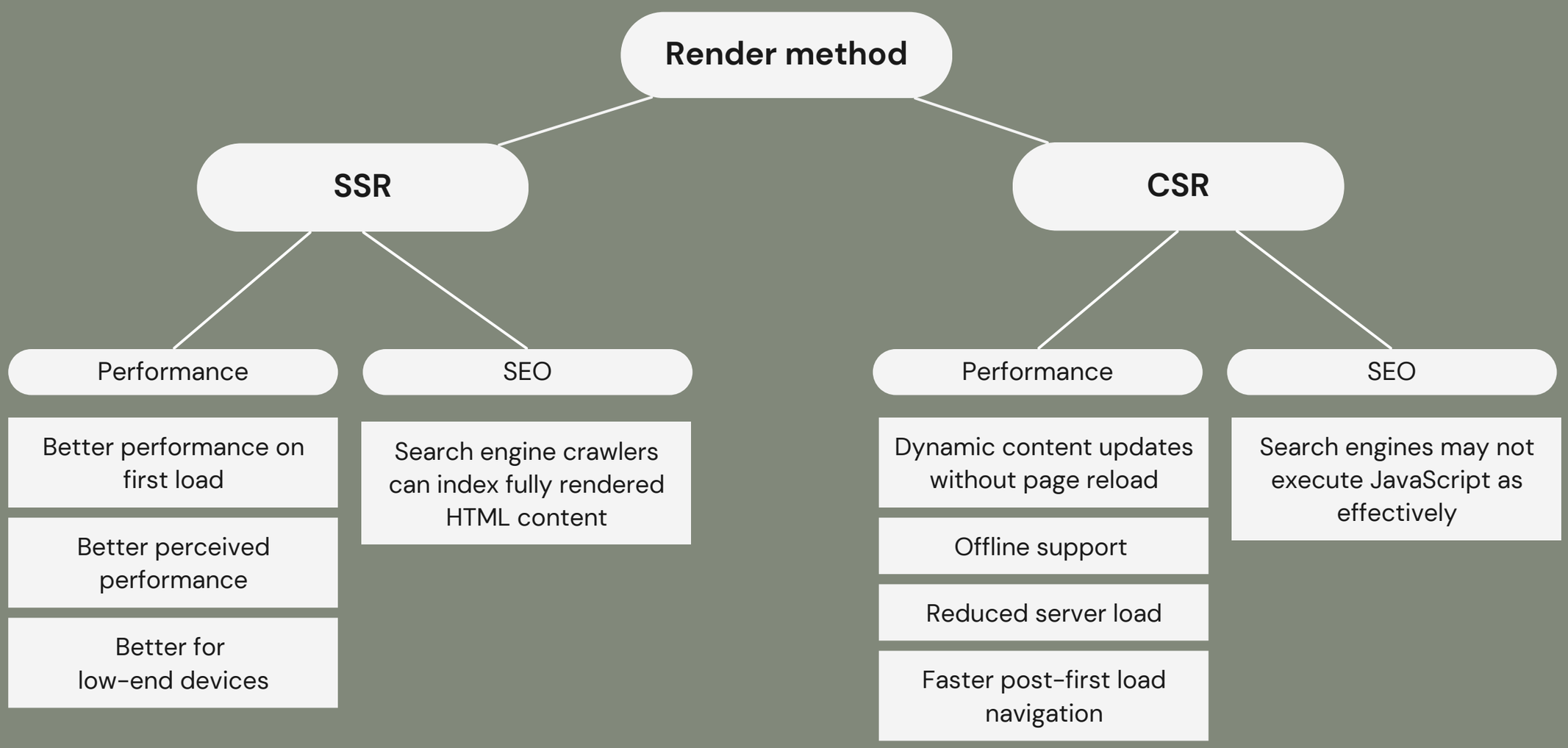
Server-Side Rendering (SSR) is a technique used in web development to improve the performance and search engine optimization of web applications, including those built with React. SSR involves rendering web pages on the server side (i.e., on the web server) before sending them to the client's web browser. This is in contrast to traditional client-side rendering (CSR), where web pages are primarily rendered in the browser using JavaScript. (Nextjs, 2023)

Benefits of SSR and CSR

In a CSR approach, the browser initially receives a minimal HTML file and a JavaScript bundle. The JavaScript code is responsible for fetching data from a server and rendering the content on the client-side. This approach is used in single-page applications (SPAs) and can result in slower initial page loads because the browser must wait for the JavaScript to download and execute before rendering the content. (Gawai, C. 2021)

In SSR, the server processes the request, fetches the required data, generates the HTML for the page, and sends a fully populated HTML page to the client. The client's browser can render the page immediately, even before JavaScript files are downloaded and executed. This leads to faster perceived page load times and better SEO because search engines can index the content directly from the HTML. (Nextjs, 2023)

SSR and CSR performance and SEO comparison



While techniques like SSR (using frameworks like Next.js) can be used with CSR to improve SEO, pure CSR has clear SEO challenges, as search engines may not execute JavaScript as effectively. SSR is more SEO-friendly out of the box because search engine crawlers can easily index the fully rendered HTML content. This leads to better search engine rankings. (Nextjs, 2023)

TTFP and TTI performance

Time to First Paint (TTFP) and Time to Interactivity (TTI) are performance metrics used to assess the user experience of web applications. TTFP measures the time it takes for the browser to render the first content on the screen after a user requests a web page. (Pavithra, P. 2023)

TTI measures the time it takes for a web page to become fully interactive for the user. In other words, it gauges how long it takes for the user to be able to interact with elements like buttons, forms, and links on the page. TTI is a more comprehensive metric than TTFP because it takes into account not only the initial rendering but also the time required for JavaScript to become available and responsive. (Pavithra, P. 2023)

In SSR TTFP is typically shorter compared to CSR. This is because, in SSR, the server pre-renders the initial HTML content on the server and sends it to the browser. So, the browser can start painting the page as soon as it receives this pre-rendered HTML. (Nextjs, 2023)

In CSR TTFP can be longer because the browser needs to first download the JavaScript bundle, execute it, and then request and render the page content. This process can introduce delays, especially on slower devices or networks. (Nextjs, 2023)

For SSR the TTI is typically shorter compared to CSR because most of the JavaScript code required for interactivity is included in the initial server-rendered HTML. Users can start interacting with the page sooner. This is included in the comparison chart on page 4 as better perceived performance (for the user) as well as better performance on first load. (Chetan, G. 2021, Nextjs, 2023)

TTI can also be affected by the complexity of the client-side code and the performance of the device running the application. This is especially evident in CSR as the browser must first download and execute the JavaScript code, which can be a significant delay. As users see the content slower, it can create a perception of a slower-loading page leading to a worse perceived performance. (Okafor, N. 2023)

Hybrid approach

Some applications use a third method - the hybrid approach, combining SSR and CSR to strike a balance between performance and interactivity. This is known as "Hydration," where the server sends a pre-rendered page, and then the client-side JavaScript takes over to enhance the interactivity. The Hydration method is covered through its usage in SSR later in the book, however before delving deeper into the intricacies of complex SSR topics it’s beneficial to get a brief understanding of the method.

The Hydration method combines qualities covered in the chapter between SSR and CSR and can be used to leverage the benefits of both approaches. The method can be used to improve resource efficiency on both the server and client sides by using SSR for static content and CSR for dynamic content. Hybrid rendering can also be used for code splitting techniques to minimize initial load times to more in-line with the faster SSR approach as we learned in the previous sub-chapter. (Gawai, C. 2021)

As you have learned by now, SSR is the superior method for SEO optimization. As such it can be used through the hybrid method for improved SEO with CSR features included in the application.

2: Basic React SSR

Did you know?

SSR (Server-Side Rendering) has been around since the early days of web development and predates React by a wide margin.

Understanding the React rendering process on the server

After reading chapter 1 of the book you should already have a solid grasp on what is server-side rendering (SSR) and how it compares to client-side rendering (CSR) in the concept of web development as a whole. The contents of this chapter will no longer be language agnostic, as we are now diving into the basics of using SSR as a rendering method in React development.

There are two pre-requisites to enabling server-side rendering in a React application: a server environment that can execute JavaScript and a server framework that can handle server-side rendering. Technically an SSR framework is not a hard requirement, however in modern web development there is little point in not using a framework such as Next.js for server-side rendering. (Szczeciński, B. 2023)

In the following pages the post-requisite steps of the server-side rendering process are covered.

Request handling

When a user requests a page, the server handles the request.
The request determines which React components are required for the route.

Data fetching

If the page requires data from an API or a database, the server fetches this data. Data fetching can be asynchronous.
It's common to use libraries like axios or fetch for this purpose.

Component rendering

With the required data in hand, the server renders the relevant React components.
The server executes the component code and generates HTML markup for each component.

HTML generation

The generated HTML from rendering React components is assembled into a complete HTML page.
This HTML page includes the initial state of the React application.

Sending HTML to the client

The server sends the complete HTML page as a response to the user's request.

Client-side hydration

When the browser receives the HTML page, it starts rendering it.
React, on the client side, takes over and hydrates the page. The concept of the hydration method was covered in page 5 of the book.
React compares the initial server-rendered HTML with the client-side components and updates the DOM as needed.

Client-side routing

Once hydration is complete, client-side routing takes over.
Subsequent page transitions are handled on the client side without full-page reloads, providing a smoother user experience.

SSR in Next.js

Next.js is a popular React framework that offers Server-Side Rendering (SSR) capabilities out of the box. It is designed to simplify and streamline the process of building React applications with SSR. (Nextjs, 2023)

The framework provides automatic SSR for React components. When creating a page in a Next.js application, it will be rendered on the server by default, and the resulting HTML is sent to the client. This helps with SEO and initial page load performance as demonstrated in detail in chapter 1 of the book.

A data fetching mechanism is provided in Next, allowing the fetching of data during the SSR process. Functions like *getServerSideProps* or *getStaticProps* can be used to fetch data on the server and pass it as props to a React component. The traditional client-side routing method of React is modified in Next with its built-in routing system which has support for SSR pages. (Nextjs, 2023)

Next.js optimizes initial load time through automatic code splitting. This essentially means that only the JavaScript that is required for the initial render is sent to the client, instead of a full CSR approach. Next utilizes hydration (covered in chapter 1) of pages on the client side. After the initial HTML is loaded, Next reattaches event handlers and makes the page interactive using React. (Szczeciński, B. 2023, Nextjs, 2023)

While Next.js provides a built-in server for SSR, it can also be extended and customized for advanced use cases, which is covered in detail later in the book. Next also supports Static Site Generation (SSG) where pages can be pre-rendered at build time, allowing for even faster performance and reduced server load. (Nextjs, 2023)

Main take-away

Next.js is widely used in the React community for building applications that benefit from server-side rendering.

Next.js sample app using SSR data fetching

NEXT.JS pages/index.js

```
import axios from 'axios';

// This function runs on the server-side (SSR) and fetches data
export async function getServerSideProps() {
  // Fetches data from any API
  const response = await axios.get('URL');
  const posts = response.data;

  // Passes the data to the page component as props
  return {
    props: {
      posts,
    },
  };
}

function Home({ posts }) {
  return (
    <div>
      <h1>Our Next.js App with SSR fetched data</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
}

export default Home;
```

3: Advanced React SSR Techniques

Code-splitting (or code splitting)
The process of splitting code into smaller chunks to improve applications load time.

Code-splitting in web development

Code-splitting is a technique used in web development to improve the performance and load times of web applications. It involves breaking down a large JavaScript bundle into smaller, more manageable chunks that can be loaded on-demand. It allows for reduced initial load time of web applications by only loading the code that is needed for the current page or user interaction. (Next.js 2023)

Large JavaScript bundles can take a long time to download and parse, especially on slower connections. Code-splitting reduces the initial bundle size, resulting in quicker page loads. Code-splitting allows for loading the bare minimum JavaScript required for rendering the page that the user is currently on. (Bece, A. 2022)

In addition to faster initial load, code-splitting also provides benefits in improved UX, resource usage optimizing and caching. Users can start interacting with an application that uses code-splitting faster because they don’t have to wait for the entire codebase to load, leading to better UX. Resource utilization is improved by loading less code overall when rendering and smaller bundles can be cached more effectively by browsers and CDNs, reducing subsequent load times for returning users. (Bece, A. 2022)

Code-splitting was less useful in the past and early days of web development. However, with modern web applications the codebases are larger and dependencies more complex, leading to more significant benefits from utilizing code-splitting. (Shipton, E. 2023)

Code-splitting in React

React itself doesn't provide a built-in code-splitting solution. Code-splitting can be achieved by using exterior React methods and libraries. One popular approach is to use dynamic imports with React's Suspense and lazy features, which are available in React 16.6 and later. The following example uses React.lazy function to import a sample Chart component to code-split and it’s wrapped in a Suspence component to handle loading. (React, 2023)

REACT.JS APP.JS

```
import React, { lazy, Suspense } from 'react';

// Dynamically import the sample Chart component
const Chart = lazy(() => import('./Chart'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <Chart />
      </Suspense>
    </div>
  );
}

export default App;
```

When building the sample app of previous page with npm run build, the code-splitting will be automatically handled, creating separate bundles for each dynamically imported component. This reduces the initial load size for the application.

Code-splitting with Next.js

Next.js provides built-in support for code-splitting through automatic page-based code-splitting and dynamic imports. With Next applications, using dynamic imports allows us to load components only when needed. The following sample is the same app as the previous pages' React application, with the difference of using the built in Next.js method for code-splitting. (Nextjs, 2023)

NEXT.JS PAGES/INDEX.JS

```
import dynamic from 'next/dynamic';

// Dynamically import the sample Chart component
const DynamicChart = dynamic(() => import('../components/Chart'));

function HomePage() {
 return (
  <div>
   <DynamicChart />
  </div>
 );
}

export default HomePage;
```

As demonstrated, the built-in Next.js method of the previous page uses comparatively less code and is included in the framework itself. While the manual route is present in vanilla React (page 12), in modern web development using Next.js is a more robust method to achieve efficient code-splitting when rendering web applications.

Caching

Scott Gary argues that Caching plays a crucial role in SSR React development and is a tool for making SSR React applications faster and more efficient. Caching is the practice of storing and reusing data or rendered content to improve the performance of web applications. In the context of SSR React applications, caching can be applied at various levels to reduce server load, improve response times, and enhance user experience. (Gary, S. 2023)

To reduce the load on the server and speed up SSR, server-side caching mechanisms can be implemented to any React application. This involves storing pre-rendered HTML for specific routes or components and serving them to subsequent requests instead of re-rendering the content. Common caching solutions for SSR include in-memory caches such as Redis, and disk-based caching. (Szczeciński, B. 2023)

Once the SSR-rendered content reaches the client, it can be cached on the client-side to avoid unnecessary re-renders and improve navigation speed. This can be achieved using browser cache mechanisms, Service Workers, or client-side libraries such as Redux or Mobx to store and manage cached data. Content Delivery Networks can be leveraged to cache static assets such as JavaScript bundles, CSS files, images and serve them from geographically distributed edge servers. CDNs can significantly reduce the latency and load on your application server by delivering cached assets to users from the nearest edge server. (Gary, S. 2023)

Content delivery network / Content distribution network (CDN)

Network of proxy servers and their data centers, with the goal of providing high availability and performance by distributing the service relative to end users. (Dilley, J. 4)

In addition to caching the entire HTML page, specific content fragments or API responses can be cached as well in a React application. For example, the results of frequently requested API calls on the server or client-side caching libraries like SWR or react-query for caching API responses on the client. (React, 2023)

Main take-away

Storing data in a temporary storage to reduce the need for redundant requests and unnecessary data fetching in the form of caching can lead to faster load times and better performance in React applications utilizing SSR.

Other optimization strategies

Other optimization strategies that can be used to improve performance, efficiency and user experience of SSR React applications include tree shaking, image optimization, lazy loading and separating critical CSS. These topics are covered briefly in the book on a surface level.

Tree shaking is a technique which is not limited to React. It's used in a variety of modern JavaScript build processes. In SSR React applications, tree shaking relies on the ES6 module system of React, where functions, classes, and variables can be exported and imported from one module to another. The tree shaking system provides a clear structure for static analysis and optimization. (Egwuenu, G. 2019)

The technique identifies and eliminates code that is not used in your application. It does this by analyzing the import and export statements in your modules. If a module exports something that is never imported elsewhere, that code is considered dead by the method of tree shaking and can be safely removed. (Egwuenu, G. 2019)

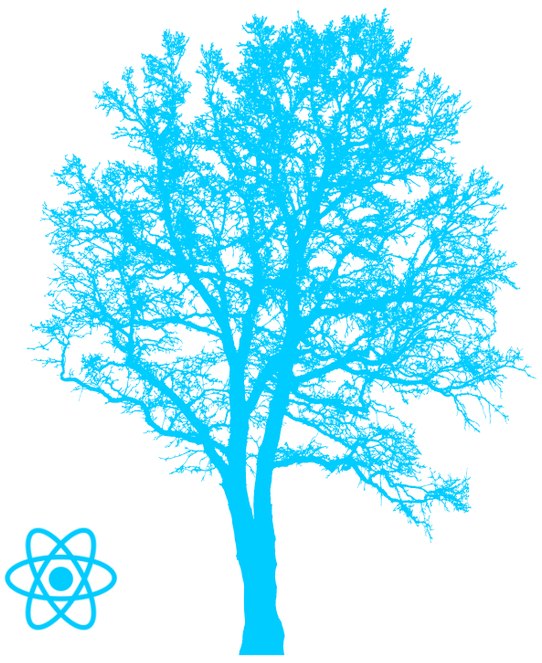


Image optimization can be utilized in SSR by pre-rendering the images along with other HTHML content on the server. WebP is a performance format for further optimization and size of the image can be optimized based on the client's device and viewport, srcset attribute can be used to deliver images in a responsive format. (Shipton, E. 2023)

Lazy loading is an optimization method that can be used to delay the loading of non-essential resources, such as images below the fold, until they are needed. React's Suspense can be used to utilize Lazy loading in React applications, through React.lazy function. Lazy loading is particularly useful in large applications where loading all components at once might lead to slower initial load times. Below is a sample of using the React Suspense and React.lazy function to achieve this.

REACT.JS APP.JS

```
import React, { Suspense } from 'react';
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
 return (
  <div>
   <Suspense fallback={<div>Loading...</div>}>
    <LazyComponent />
   </Suspense>
  </div>
 );
}
```

Critical CSS is the last optimization method that we will cover in this chapter. It is a technique used in web development in general as well as React to optimize the loading and rendering of web pages. The primary goal of critical CSS is to improve the perceived page load performance by delivering the minimal CSS required for rendering the above-the-fold content of a web page as quickly as possible. (Sethi, P. 2022)

Critical CSS refers to the subset of an application's CSS that is required to render the part of the page that's immediately visible to the user (above-the-fold content). It doesn't include styles for components or elements that are below the fold or outside the user's initial viewport. After identifying React applications critical CSS, it can be separated and load directly into the HTML response using inline CSS. Non-critical CSS can then be written on an external CSS file which can be loaded asynchronously to improve performance on loading the critical CSS. (Sethi, P. 2022, Mihajlija, M. 2019)

Below is a sample of loading non-critical CSS asynchronously in a React application from an exterior CSS file.

APP.JS

REACT.JS

```
<link rel="stylesheet" href="non-critical.css" media="print"
onload="this.media='all'">
<noscript><link rel="stylesheet" href="non-critical.css"></noscript>
```

Handling authentication and authorization on the server

Handling authentication and authorization in SSR React applications involves a combination of server-side and client-side techniques. SSR ensures that initial authentication and authorization checks are performed on the server before rendering the initial page, and then the client takes over for subsequent interactions. This is where Next.js comes in again, as it can be used for setting up the server that handles SSR. (Nextjs, 2023)

In a Next.js app using server for authentication and authorization users submit their credentials, which are verified on the server. Next.js allows for the handling of submissions and server-side logic in API routes. It allows for usage of server-side session management, such as express-session or custom middleware, for creating and managing user sessions. (Nextjs, 2023)

Middleware is then used in Next.js to protect routes and handle authentication logic. Specific pages or API routes can be wrapped with middleware to ensure that only authenticated users can access them. Then, once a user is authenticated, their user data can be loaded on the server side and passed as props to React components that need it. The data can be fetched during the server-side rendering using *getServerSideProps* function, or pre-rendering it at build time with *getStaticProps* function. (Nextjs, 2023)

4: Data fetching in SSR

Data fetching process

Now we will dive a bit deeper into the data fetching process in SSR React development. Data fetching is by definition a fundamental aspect of building any types of dynamic web applications. The process in SSR applications involves retrieving data from various sources, such as APIs or databases, and rendering it on the server before sending the fully populated HTML to the client.

Next.js simplifies not only SSR integration in React, but also data fetching. *getServerSideProps* is a Next.js function that allows for the fetching of data on the server-side and passing it as props to any page component. It runs on every request, making it suitable for dynamic data. getStaticProps is a function used for pages with static data or data that changes infrequently. Next.js generates static HTML at build time, and this data is then used to serve subsequent requests. (Next.js, 2023, Okafor, N. 2023)

getInitialProps is not recommended for new projects, as getServerSideProps and getStaticProps are more powerful, but getInitialProps can still be used to fetch data. It's used in class-based components and older Next.js projects. (Okafor, N. 2023)

On the following pages you can find sample apps of *getServerSideProps* function and *getStaticProps* function in use in a Next.js applications.

Next.js sample app using getServerSideProps function to fetch data

NEXT.JS pages/fetchSample.js

```
function FetchSample({ data }) {
  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.description}</p>
    </div>
  );
}

export async function getServerSideProps(context) {
  //Here we fetch the required data from any API
  const res = await fetch('API URL);
  const data = await res.json();

  //Here we pass the fetched data as props to the component
  return {
    props: {
      data,
    },
  };
}

export default FetchSample;
```

Next.js sample app using getStaticProps function

NEXT.JS pages/index.js

```
function FetchSample({ data }) {
  return (
    <div>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export async function getStaticProps() {
  //Here we fetch the required data from any API
  const response = await fetch('API URL');
  const data = await response.json();

  return {
    props: {
      data,
    },
  };
}

export default FetchSample;
```

Data hydration on the client side

Data hydration on the client-side in the context of React apps is the process of preloading or initializing data on the client-side before rendering the user interface. As you have learned by know, in React applications utilizing SSR, the initial HTML of a web page is generated on the server, including the data needed for the page. When a user first requests a page, they receive a fully populated HTML document with the content and data, not just an empty shell. (Szczecinski, B. 2023)

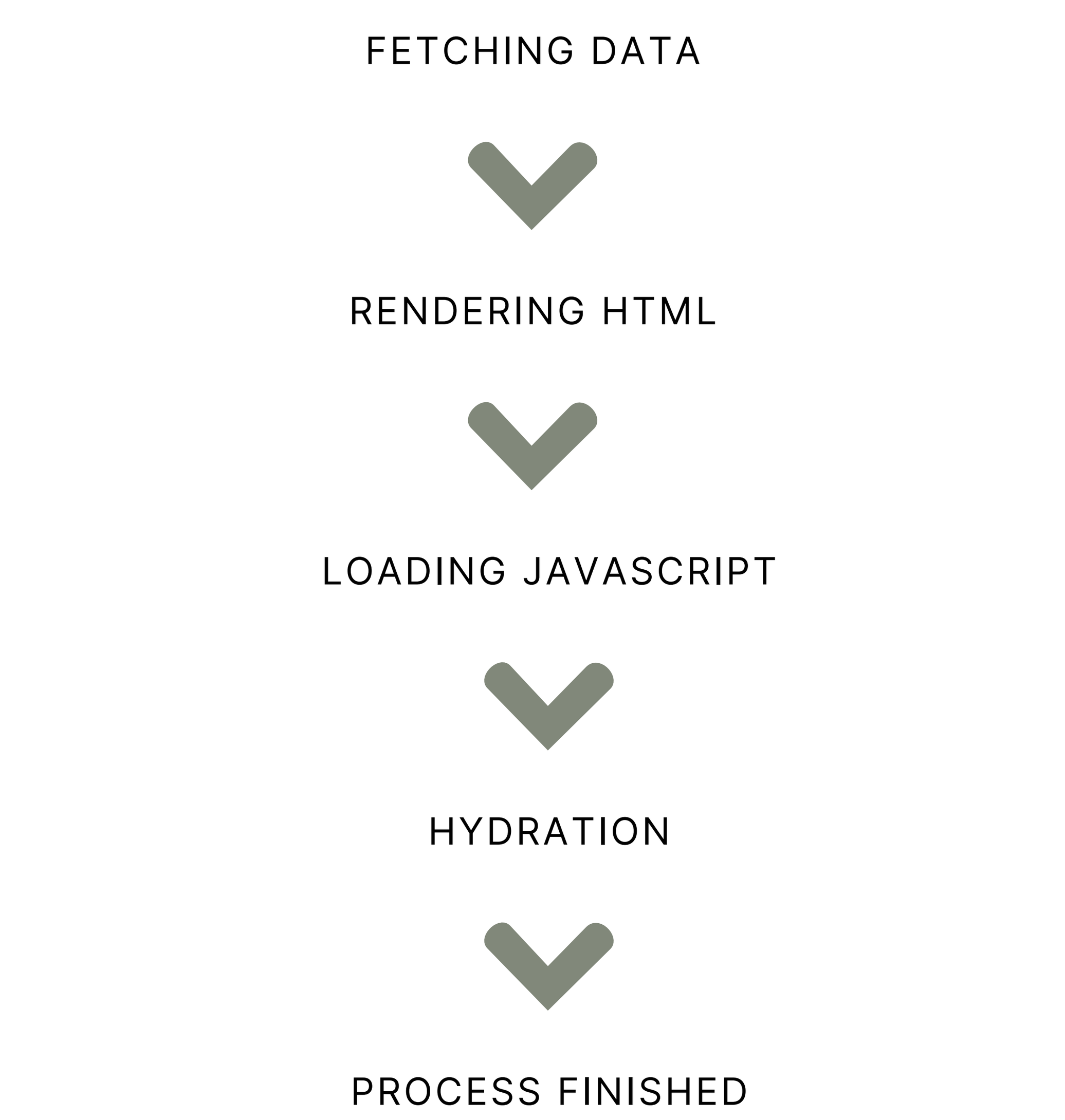
The first step before client rehydration takes effect in the data hydration process is generating the initial HTML of a web page on the server, as per the SSR process you are familiar by now. When a user first requests a page, they receive a fully populated HTML document with the content and data, not just an empty shell. React components are often rendered on the server as well, so the HTML also contains the initial component markup.

Once the initial HTML is loaded in the browser, React on the client side takes over. It recognizes the server-rendered HTML and the initial data and rehydrates the components. Rehydration means that React attaches event handlers, sets up interactivity, and connects to the initial data used to render the components. (Szczecinski, B. 2023)

Then as the last step of the process after rehydration, React components may continue to fetch additional data from APIs or other sources to update the page with dynamic content or respond to user interactions. This asynchronous data fetching ensures that the page stays up-to-date with the latest data.

The overall benefit of data hydration is that it provides a fast initial load experience because users see content immediately without waiting for JavaScript to fetch and render everything, as per the SSR process. It also aids in SEO because search engine crawlers can index the fully populated HTML content. Once the JavaScript loads, the app becomes interactive and can fetch additional data as needed. (Gawai, C. 2021)

Below you can see the server rendering process imagined with Hydration being the last step as covered in this chapter.



This marks the end of the theoretical section of the book. By now you should have a solid grasp on the fundamentals of the full server-side rendering process in React applications as well as on advanced SSR techniques for improved performance and how to use Next.js to facilitate the process. In the following chapters we will move on to more practical topics and learn about using the theory learned so far in practice.

5: Deployment and scaling

Deployment options for SSR React applications

In this chapter, we will cover the most relevant options available for deploying SSR React applications. Before getting in detail with each of these approaches, the most important thing to consider when choosing the deployment option is your applications infrastructure and other requirements. All of the following methods have their use cases in modern React development depending on the needs and requirements of the project.

DEPLOYMENT OPTION 1

Traditional Server Deployment

In the traditional server deployment approach, the application is deployed on a traditional web server. Examples of these include options such as Apache, Nginx and Express.js. With this option, the server handles incoming requests and renders React components on the server before sending the HTML to the client. (Apache, 2023)

Traditional server deployment is a straightforward approach suitable for applications with moderate amounts of traffic.

DEPLOYMENT OPTION 2

Containerization

Containerization technologies such as Docker and orchestration tools such as Kubernetes can be used to deploy server-side React applications. Docker containers encapsulate the application and its dependencies, making it easier to manage and scale. Kubernetes provides scalability and load balancing capabilities, making it suitable for high-traffic scenarios.

DEPLOYMENT OPTION 3

Platform as a Service (PaaS)

There are Platform as a Service (PaaS) providers available in the market for deploying SSR and other React applications. These providers include commonly used ones such as Heroku, Vercel and Netlify. Using a PaaS provider can simplify the deployment process as they offer easy integration with CI/CD pipelines, automatic scaling, and managed infrastructure. (Edwards, R. 2023)

PaaS providers can be an ideal solution for small to medium-sized SSR projects or projects with dynamic scaling requirements.

DEPLOYMENT OPTION 4

Serverless SSR with AWS Lambda

AWS Lambda is a deployment method for running serverless SSR React applications. Through this method, Lambda functions can be triggered by API Gateway or other AWS services. This option can be a cost-effective choice, especially for non-enterprise React applications with low-to-moderate amounts of traffic. (Beswick, J. 2021)

Scaling SSR for high traffic

Load balancing

The process of distributing incoming network traffic across multiple servers to ensure that no single server becomes a bottleneck.

Load balancing can be set up to distribute incoming requests to different instances of the applications SSR server. This helps in achieving better resource utilization and improved response times.

Common approaches include using hardware load balancers, software load balancers such as Nginx or HAProxy, or cloud-based load balancers such as AWS Elastic Load Balancing. Load balancers can be configured for various algorithms, such as round-robin, least connections, or IP hash, however detailing these configurations is outside of the scope of this book. (Cunha, L. 2021)

Caching mechanisms can be implemented to reduce the load on the applications SSR server. Caching involves storing frequently accessed data or rendered HTML content so that it can be served quickly without re-generating it each time a user requests it. It can be applied at multiple levels, including client-side caching, server-side caching and content caching. In a hybrid rendering method, all of these caching options can be used in scaling. Caching as a concept in React applications is explained in detail in chapter 3 of the book.

Horizontal scaling is a process that involves adding more servers or instances to an application to handle increased traffic. In the context of SSR React applications, this means deploying multiple instances of an SSR server behind the load balancer. Each instance should be identical, allowing for horizontal scaling as needed. Containerization and orchestration tools like Docker and Kubernetes (which are covered in previous page) can simplify the process of managing multiple instances and automatically scaling based on traffic load. (Yu, J. 2022)

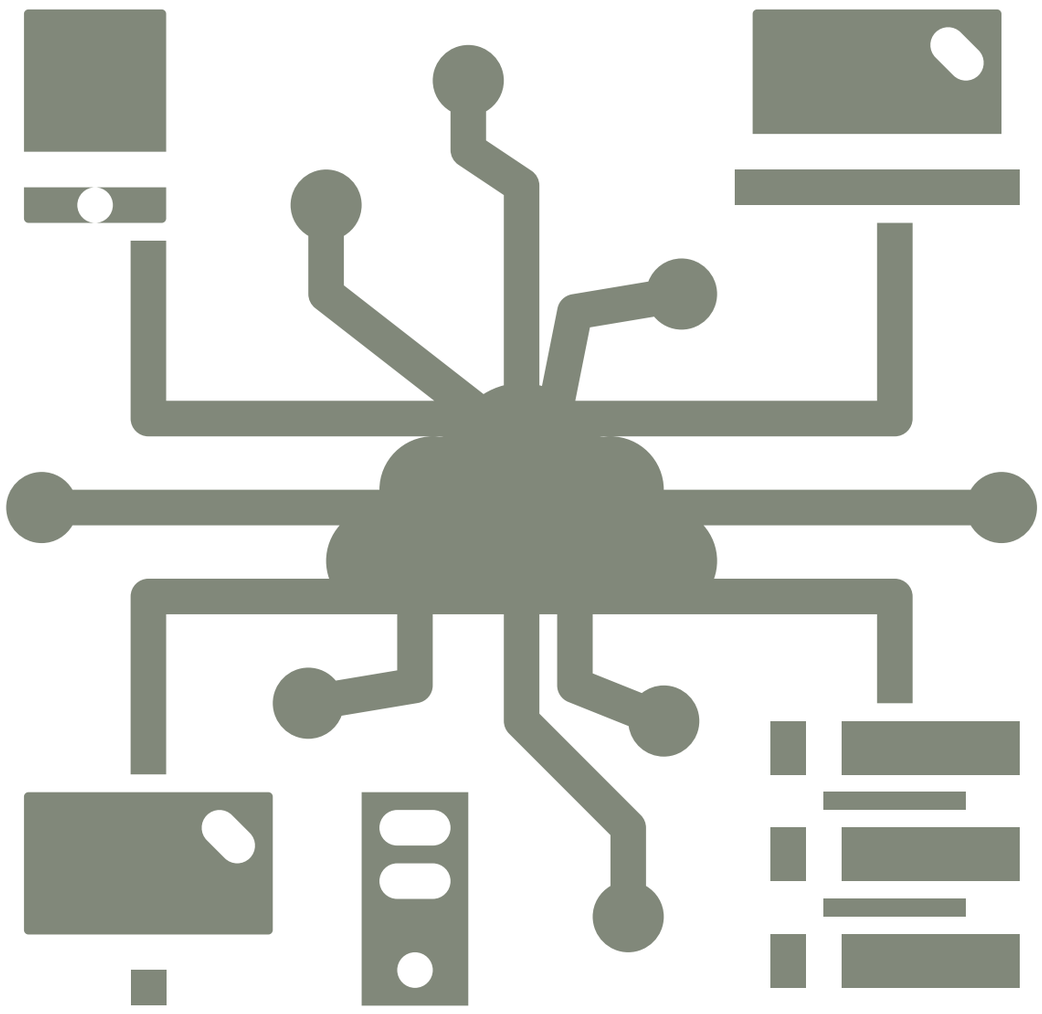
Database optimization can be a vital tool for scaling SSR applications as database queries can be a common performance bottleneck. Without going too much in detail to the optimization methods themselves, they include query optimization, database query result caching, sharding and read replicas.

Content Delivery Networks (CDNs) are the last method that we are covering and also revisiting in this book, this time from the perspective of scaling SSR React applications. CDNs are a network of distributed servers that store cached copies of an applications static assets, including images & CSS and JavaScript files. When a user requests these assets, the CDN serves them from the nearest server, reducing latency and offloading to the applications deployment server.

In the process of scaling SSR React applications, CDNs can be utilized to reduce latency by serving content from servers geographically closer to the user, reducing the time it takes for content to reach the user's device. This results in faster page load times and a better user experience. (Reactjs, 2023)

CDNs can handle high levels of traffic and distribute it across their network of servers. This can help mitigate traffic spikes and prevent the applications origin server from becoming overloaded during traffic surges. As they distribute content across multiple servers, they can also reduce the risk of server failures or network issues affecting content delivery. If one server goes down, requests are automatically routed to a healthy server. (Singhal, G. 2020)

CDNs are a network of servers used when scaling web applications.



Handling serverless SSR with AWS Lambda

AWS Lambda

Event-driven compute service for running serverless applications.

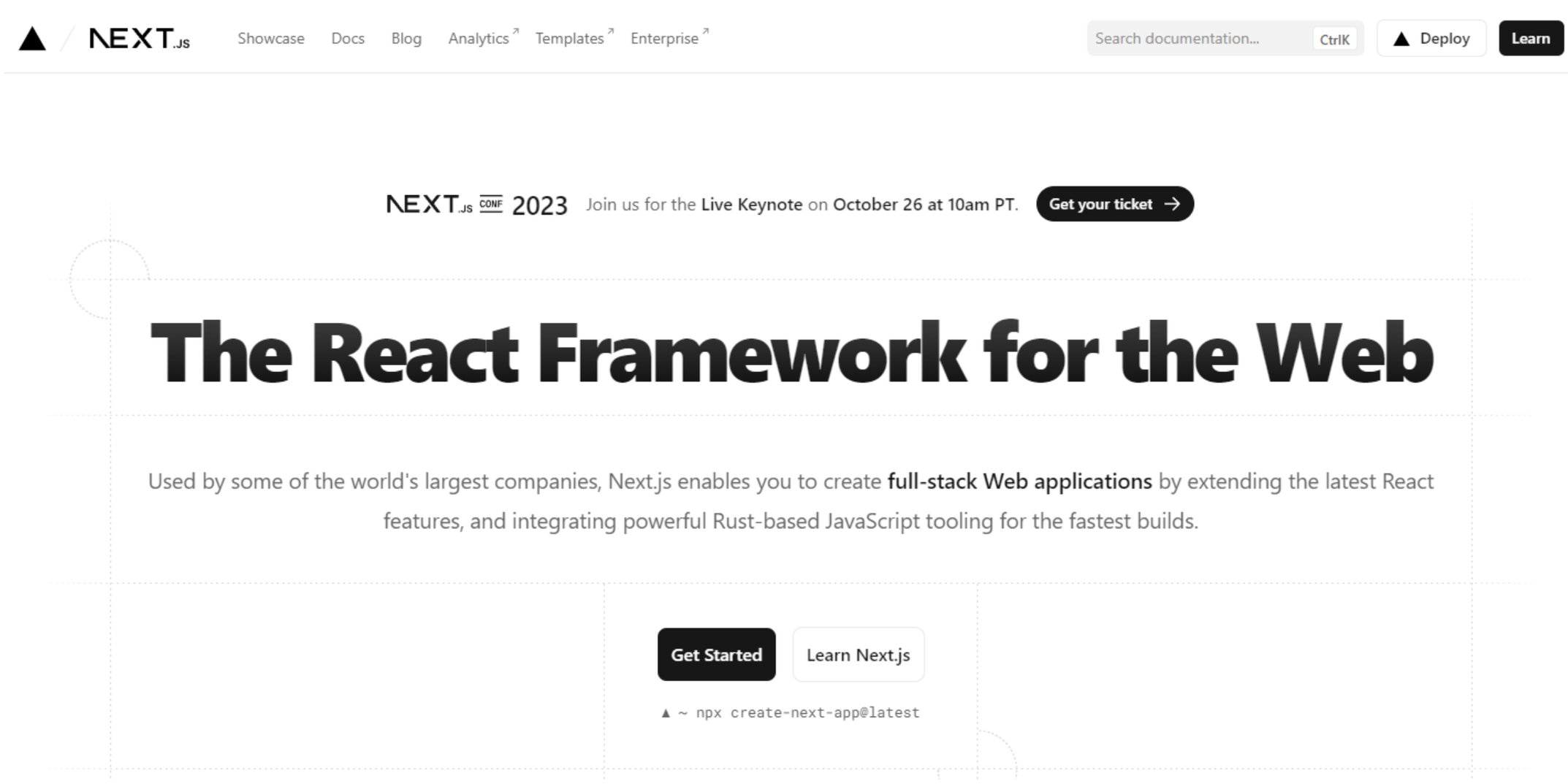
AWS Lambda is a serverless compute service that allows for the running of code in response to events without provisioning or managing servers. Lambda can be used to execute SSR code for React applications. AWS Lambda function can handle SSR requests and it can be setup using frameworks such as Serverless Framework or AWS SAM. (Amazon, 2023)

After packaging the application as a Lambda function with dependencies, the next step is configuring API Gateway for triggering the Lambda function on HTTP requests. A CDN, the concept of which is explained in previous chapter of the book, can be used for global content distribution for faster access to the SSR React application. (Amazon, 2023)

The last remaining steps in the process are setting up environment variables for configurations, including database connetions, and ensuring a proper cold start mitigation strategy during Lambda function initialization. The specific implementation details and configurations vary based on the application's requirements and AWS setup. AWS documentation can be consulted for more detailed instructions on each step of the process. (Amazon, 2023)

6: Examples and case studies

Next.js



www.next.js

The official website of Next.js is built using Next.js, using the server-side rendering methods covered in the first five chapters of this book. The site uses the Next.js namesake React framework for including the standard functionality provided by the framework for improved SEO and faster initial page load experience.

The site uses Vercel as a PaaS (Platform as a Service) method for deployment, which you should be familiar with if you studied the contents of chapter 5 of the book.

Facebook and Instagram

Facebook is the company responsible for creating the React.js framework. The main pages, Facebook Ads manager and the mobile apps are built with React. Facebook.com and Instagram.com websites use a mixture of client-side rendering and server-side rendering techniques, known as the Hybrid approach as covered in chapter 1 of the book. (Reactjs, 2023, Kruglyak, O. 2020)

The hybrid approach is used in these websites to combine the faster initial load times of server-side rendering with the faster post-first page loading of client-side rendering.

Hulu

Streaming service Hulu.com uses Next.js for their website as well as their greenfield applications. In a Next.js case study, a representative of Hulu cites server-side rendering as a “critical requirement” for their web application. Hulu uses Next.js to handle and simplify the client hydration and data fetching logic & processes. (Nextjs, 2023)

Sources

1. Bece, Adrian. Improving JavaScript Bundle Performance With Code-Splitting. 2022. https://smashingmagazine.com/
2. Beswick, James. Building server-side rendering for React in AWS Lambda-2021. https://aws.amazon.com/
3. Case studies - Hulu. 2023 & Documentation. https://nextjs.org/
4. Cunha, Leonard. Deploying a React app and a Node.js server on a single machine with PM2 and Nginx. 2023. https://medium.com/
5. Dilley, John. Maggs, Bruce. Parikh, Jay. Prokop, Harald. Sitaraman, Ramesh. Weihl, Bill. Globally Distributed Content Delivery. 2022. https://people.cs.umass.edu/~ramesh/Site/PUBLICATIONS_files/DMPPSW02.pdf
6. Egwuenu, Gift. Tree-Shaking Basics for React Applications. 2019. https://telerik.com/
7. Gary, Scott. Caching in React – How to Use the useMemo and useCallback Hooks. 2023. https://freecodecamp.org/
8. Gawai, Chetan. Understanding Hydration in React applications (SSR). 2021. https://blog.saeloun.com/
9. Mihajlija, Milica. Extract critical CSS. 2019. https://web.dev/
10. Okafor, Noble. How to Handle Data Fetching in Next.js. 2023. https://makeuseof.com/
11. Parthiban, Pavithra. From Loading to Interaction: A Guide to Time to Interactive Improvement. 2023. https://atatus.com/
12. Reference documentation. 2023. https://react.dev/
13. ReactDOMServer. 2023. https://legacy.reactjs.org/
14. Sethi, Punit. My Experiences with Optimizing CSS for React Frontends. 2022. https://punits.dev/
15. Shipton, Elizabeth. Abstract. 2023. https://abstractapi.com/
16. Singhal, Gaurav. Using React Router with CDN Links. 2020. https://pluralsight.com/
17. Szczeciński, Bartosz. Understanding ReactJS — data hydration / initialization. 2023. https://medium.com/
18. The Easy Way to Run Your React Apps. 2023. https://divio.com/
19. Yu, Jeffrey. 5 Good practices to scale your React projects easily. 2022. https://dev.to/