**A Comprehensive Guide to Server-Side Rendering in React**

Improve your app performance with React server-side rendering.

**Article Contents**

[What Is Server Side Rendering?](https://www.bairesdev.com/blog/server-side-rendering-react/#what-is-server-side-rendering?)

[Implementing Server Side Rendering in React](https://www.bairesdev.com/blog/server-side-rendering-react/#implementing-server-side-rendering-in-react)

[Server-Side Rendering: SEO vs. Performance](https://www.bairesdev.com/blog/server-side-rendering-react/#server-side-rendering:-seo-vs.-performance)

[Final Thoughts](https://www.bairesdev.com/blog/server-side-rendering-react/#final-thoughts)

[Frequently Asked Questions (FAQs)](https://www.bairesdev.com/blog/server-side-rendering-react/#frequently-asked-questions-(faqs))

Server-side rendering in React is a powerful tool for creating dynamic web applications. It allows developers to create interactive and performant user interfaces without relying on client-side JavaScript execution. By utilizing server-side rendered components within our application codebase, we can leverage modern technologies like [Node.js](https://nodejs.org/) launched in [May 2009](https://en.wikipedia.org/wiki/Node.js#:~:text=Node.js%20was%20written%20initially,and%20later%20sponsored%20by%20Joyent.), which provide an optimized way of delivering content from the backend directly into our frontend codebase.

This approach speeds up loading times and offers unprecedented control over what is displayed on each page request while providing us with features like state management that come naturally when using a library such as [React](https://www.bairesdev.com/technologies/react/).

This article will explore the benefits of using server-side rendering in React, how it works under the hood, and how best to implement it into existing projects. We’ll also look at some of the challenges associated with server-side rendering, such as performance optimization and maintenance considerations. Finally, we’ll provide useful tips & tricks for getting started with server-side rendering in React today!

**What Is Server Side Rendering?**

Server Side Rendering (SSR) is used to render web pages on the server before sending them to the client. This allows for faster page loads, improved performance, and an SEO-friendly rendering solution for React applications. In addition, SSR can provide a better experience for users with slower internet connections or devices with limited memory and processing power by performing the initial rendering of components on the server.

SSR in React can improve page load times by eliminating unnecessary roundtrips between client and server. Server Side Rendering in React provides more control over how content appears within search engine results pages (SERPs). Since search engine crawlers rely heavily on JavaScript while indexing websites, websites built entirely with Client-Side Rendering may not appear correctly within SERPs due to their inability to parse JavaScript code.

Server Side Rendering, compared to Client-Side Rendering is that it helps ensure consistency across different browsers; since much of modern web [development](https://www.bairesdev.com/software-development-services/software-outsourcing/) relies heavily upon browser specific features like APIs or custom event handlers – these types of features may not always behave properly when rendered through Client-Side techniques alone but will function normally if prerendered via Server Side methods beforehand.

The main advantages and disadvantages of server-side rendering with React are as follows:

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| SSR provides faster initial page loads because all the necessary data is already rendered on the server before being sent to the client’s browser. | SSR requires additional infrastructure, time, and effort during development to support it properly. This isn’t necessary when building a standard website since all content can be rendered on the client side through HTML or JavaScript code. |
| When using traditional SPAs for web applications, there can often be issues related to SEO due to Google being unable to crawl through all your dynamic content during indexing. This is caused by only serving up JavaScript files that cannot easily be indexed by bots such as those used by Google or Bing etc. | SSR increases complexity within an application by introducing separate layers of code needed for both server-side rendering and browser-side scripting. |
| By having most parts of your application already rendered ahead of time you effectively eliminate any potential lag times while waiting for components or sections within an app/site. | React offers great performance benefits compared to others when using SSR correctly, it still cannot match native mobile apps in terms of speed and responsiveness due largely in part because those apps run entirely locally so they don’t need any sort of network connection at all times like webpages do. |

**Implementing Server Side Rendering in React**

Let’s now dive a bit deeper into how server-side rendering works in Next.js and Express.js by exploring a use case where it can be particularly useful.

**Use Case: An E-Commerce Website**

Let’s consider a use case where server-side rendering can be particularly useful.

An e-commerce website typically has many pages. Each page displays a product or product category. In addition, these pages are usually dynamic and frequently updated, so it’s important to make sure they are easily discoverable by search engines and accessible to all users.

To achieve this, you can build your e-commerce website with server-side rendering in Next.js or Express.js. This approach would allow you to generate an HTML markup for each page on the server. Thus, making it easy for search engines to crawl and index the content.

**Implementing Server Side Rendering using Next.js**

Let’s now look into how we can implement server-side rendering in Next.js for an e-commerce website.

**Step 1: Create a new Next.js project**

To get started, you will have to create a new Next.js project by running the following commands in your terminal:

npx create-next-app my-ecommerce-app

cd my-ecommerce-app

**Step 2: Add required dependencies**

Next, you’ll need to add the following dependencies to your project:

npm install react react-dom next

This is how the package.json looks right now.

{

"name": "my-ecommerce-app",

"version": "0.1.0",

"private": true,

"scripts": {

  "dev": "next dev",

  "build": "next build",

  "start": "next start",

  "lint": "next lint"

},

"dependencies": {

  "@next/font": "13.1.6",

  "eslint": "8.34.0",

  "eslint-config-next": "13.1.6",

  "next": "13.1.6",

  "react": "18.2.0",

  "react-dom": "18.2.0"

}

}

Please note the versions of the packages mentioned above, as they were used when this guide was created.

**Step 3: Set up the environment configuration**

We will now define an environment variable to be stored in a .env.local file, which stores configuration settings that can be used across different environments (e.g. development, staging, production).

To set an environment variable in your project. You will have to create a file called .env.local in the root of your project directory and add a line like the following:

API\_URL=http://localhost:3000

It is important to note that you should not check this file into source control, as it may contain sensitive information such as database credentials or API keys.

Instead, you can create a template file called .env.example that contains placeholder values for your environment variables and check that file into source control. Other [developers](https://www.bairesdev.com/hire-software-developers/) can then copy this file and fill in the values for the environment variables.

**Step 4: Create a new page**

Next.js uses a file-based routing system, which means that a file in the pages directory represents each page in your application. To create a new page, simply create a new file in the page’s directory with the desired URL path. For example, to create a page displaying a product list, you could create a file called pages/products/index.js.

In this file, you can define a React component that will be rendered when the user visits the /products URL path. Here’s an example component that fetches a list of products from an API and displays them in a list:

function ProductsPage() {

  const [products, setProducts] = useState([])

  useEffect(() => {

    async function fetchProducts() {

      const res = await fetch('/api/products')

      const products = await res.json()

      setProducts(products)

    }

    fetchProducts()

  }, [])

  return (

    <div>

      <h1>Products</h1>

      <ul>

        {products.map((product) => (

          <li key={product.id}>{product.name}</li>

        ))}

      </ul>

    </div>

  )

}

export default ProductsPage

**Step 5: Create an API endpoint**

To fetch the list of products, we’ve used an API endpoint at /api/products. Next.js provides a built-in API routing system that makes it easy to create serverless API endpoints.

To create an API endpoint, create a new file in the pages/api directory. For example, to create an API endpoint that returns a list of products, you could create a file called pages/api/products.js.

In this file, you can define a function that will be executed when the user requests the API endpoint. For the purpose of this guide, we will use an example function that fetches a list of products from a mock API:

const products = [  { id: 1, name: 'Product 1' },  { id: 2, name: 'Product 2' },  { id: 3, name: 'Product 3' },]

export default function handler(req, res) {

  res.status(200).json(products)

}

**Step 6: Update the page to use server-side rendering**

By default, Next.js uses client-side rendering (CSR) to render pages, which means that the JavaScript code is executed in the user’s browser. To switch to server-side rendering (SSR), you’ll need to update your page component to use a getServerSideProps function.

The getServerSideProps function is a special function that runs on the server before the page is rendered. It can be used to fetch data from an API or database, and return it as props to the page component.

Here’s an updated version of the pages/products/index.js file that uses getServerSideProps to fetch the list of products on the server:

import { useState } from 'react'

function ProductsPage({ products }) {

  const [loading, setLoading] = useState(false)

  return (

    <div>

      <h1>Products</h1>

      <ul>

        {products.map((product) => (

          <li key={product.id}>{product.name}</li>

        ))}

      </ul>

    </div>

  )

}

export async function getServerSideProps() {

  const res = await fetch(`${process.env.API\_URL}/api/products`)

  const products = await res.json()

  return { props: { products } }

}

export default ProductsPage

Note that we’ve moved the useState hook for the loading state outside of the getServerSideProps function, since it needs to be initialized on the client as well.

**Step 7: Start the development server**

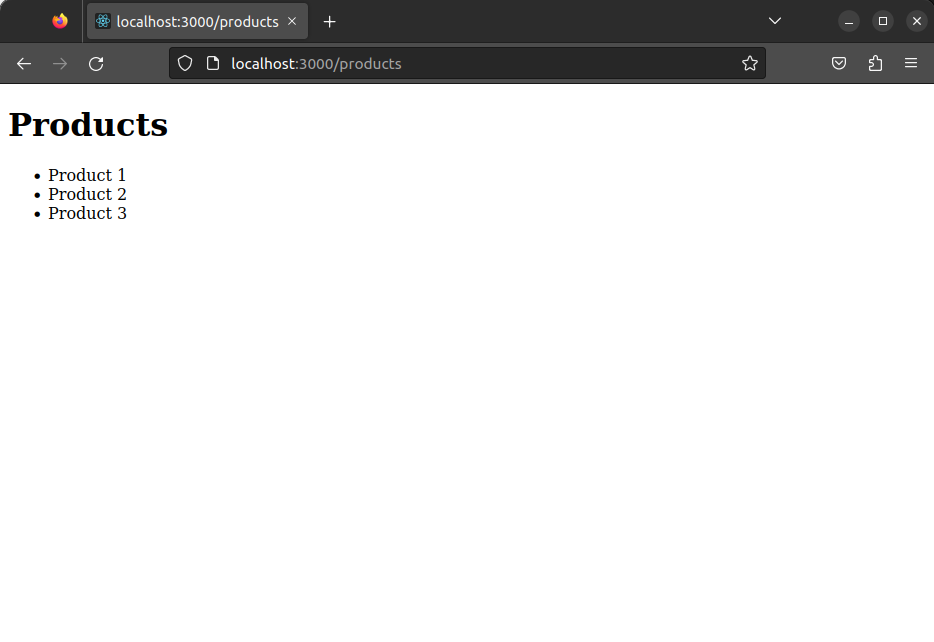
You can now start the development server by running the following command in your terminal:

npm run dev

This will start a local development server at http://localhost:3000.

**Step 8: Test the application**

You can now test the application by visiting the /products URL path in your web browser. You should see a list of products displayed on the page.

A screenshot of a computer

Description automatically generated

If you view the page source in your browser, you’ll also see that the list of products is included in the HTML markup, meaning the page was rendered on the server.

Congratulations, you now know how to implement server-side rendering in a Next.js application!

**Implementing Server Side Rendering using Express.js**

Let’s now look into how we can implement the same use case in an Express.js application:

**Step 1: Create a new Express.js application**

To get started, you will have to create a new directory for your project and run the following command in your terminal:

npm init

You can now see a package.json file in your project directory.

Next, install Express.js and the necessary dependencies by running the following command:

npm install express react react-dom next

This is how the package.json looks right now.

{

"name": "express-demo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

  "test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC",

"dependencies": {

  "express": "^4.18.2",

  "next": "^13.1.6",

  "react": "^18.2.0",

  "react-dom": "^18.2.0"

}

}

Please note the versions of the packages mentioned above, as they were used when this guide was created.

**Step 2: Set up the environment configuration**

We will now define an environment variable to be stored in a .env.local file, which stores configuration settings that can be used across different environments (e.g. development, staging, production).

To set an environment variable in your project. You will have to create a file called .env.local in the root of your project directory and add a line like the following:

API\_URL=http://localhost:3000

It is important to note that you should not check this file into source control, as it may contain sensitive information such as database credentials or API keys.

Instead, you can create a template file called .env.example that contains placeholder values for your environment variables and check that file into source control. Other developers can then copy this file and fill in the values for the environment variables.

**Step 3: Set up the server**

Now create a new file called server.js in the root of your project directory, and add the following code:

const express = require('express')

const next = require('next')

const dev = process.env.NODE\_ENV !== 'production'

const app = next({ dev })

const handle = app.getRequestHandler()

app.prepare().then(() => {

  const server = express()

  server.get('/', (req, res) => {

    return app.render(req, res, '/home')

  })

  server.get('/products', (req, res) => {

    return app.render(req, res, '/products')

  })

  server.all('\*', (req, res) => {

    return handle(req, res)

  })

  server.listen(3000, (err) => {

    if (err) throw err

    console.log('> Ready on http://localhost:3000')

  })

})

This code sets up an Express.js server that listens for incoming requests on port 3000. The app object is a Next.js application instance, which we use to render pages in response to requests.

In this example, we’ve set up two routes: / and /products. When a request comes in for either of these routes, the server calls app.render() to render the corresponding page. If the requested route doesn’t match either of these, the server falls back to the handle function, which serves the appropriate page using client-side rendering.

**Step 4: Create the home page**

Create a new file called pages/home.js in a directory called pages in the root of your project directory, and add the following code:

import Link from 'next/link'

function HomePage() {

  return (

    <div>

      <h1>Welcome to our e-commerce website!</h1>

      <Link href="/products">

        <a>View our products</a>

      </Link>

    </div>

  )

}

export default HomePage

This code defines a simple home page that displays a welcome message and a link to view the products page.

**Step 5: Create the products page**

Create a new file called pages/products.js in the pages directory, and add the following code:

import { useEffect, useState } from 'react';

function ProductsPage() {

const [products, setProducts] = useState([]);

useEffect(() => {

  async function fetchProducts() {

    const response = await fetch('/api/products');

    const data = await response.json();

    setProducts(data.products);

  }

  fetchProducts();

}, []);

return (

  <div>

    <h1>Products</h1>

    <ul>

      {products.map(product => (

        <li key={product.id}>{product.name}</li>

      ))}

    </ul>

  </div>

);

}

export default ProductsPage;

This code defines a products page that displays a list of products fetched from an API endpoint. The useEffect hook is used to manage state and fetch data from the server. When the component mounts, the useEffect hook calls the fetchProducts function to retrieve the products from the API.

**Step 6: Create the API endpoint**

You will now have to add an API endpoint for products in server.js file present in the root of your project directory, and add the following code:

  server.get('/api/products', (req, res) => {

    const products = [

      { id: 1, name: 'Product 1' },

      { id: 2, name: 'Product 2' },

      { id: 3, name: 'Product 3' },

    ];

    res.status(200).json({ products });

  });

This code defines an API endpoint that returns a list of products. This endpoint would fetch data from a database or other data source in a real-world scenario.

The updated server.js should look as follows:

const express = require('express')

const next = require('next')

const dev = process.env.NODE\_ENV !== 'production'

const app = next({ dev })

const handle = app.getRequestHandler()

app.prepare().then(() => {

  const server = express()

  server.get('/', (req, res) => {

    return app.render(req, res, '/home')

  })

  server.get('/products', (req, res) => {

    return app.render(req, res, '/products')

  })

  server.get('/api/products', (req, res) => {

    const products = [

      { id: 1, name: 'Product 1' },

      { id: 2, name: 'Product 2' },

      { id: 3, name: 'Product 3' },

    ];

    res.status(200).json({ products });

  });

  server.all('\*', (req, res) => {

    return handle(req, res)

  })

  server.listen(3000, (err) => {

    if (err) throw err

    console.log('> Ready on http://localhost:3000')

  })

})

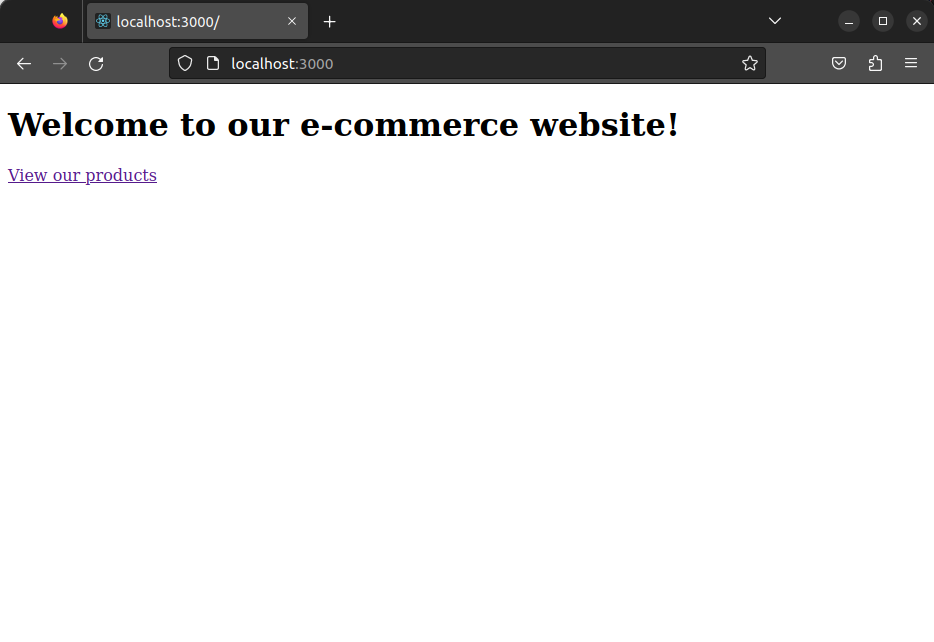
**Step 7: Start the server**

Start the Express.js server by running the following command in your terminal:

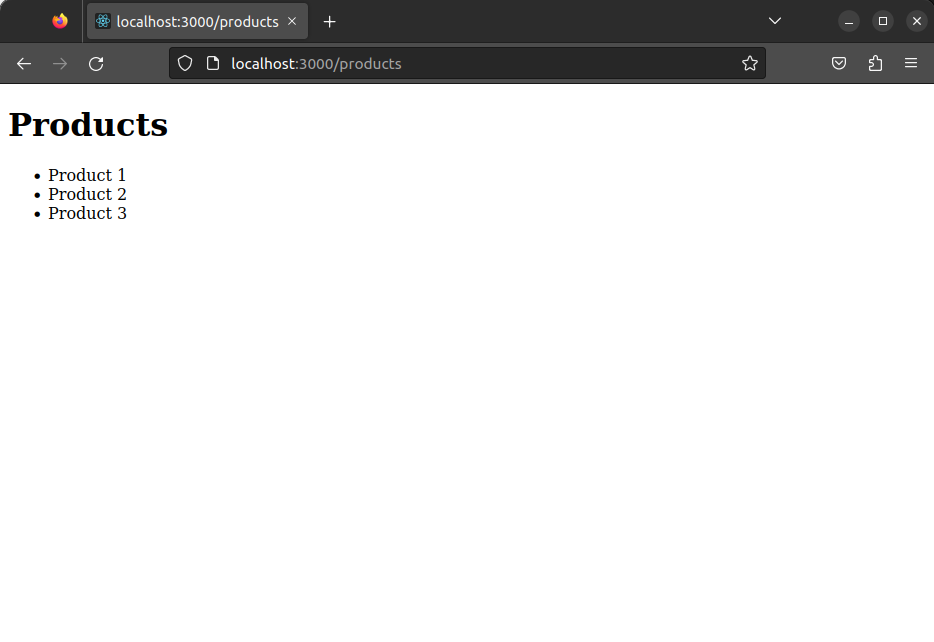
node server.js

This will start the server and make it available at http://localhost:3000.

When you navigate to http://localhost:3000/, you should see the home page with a link to the products page.



Clicking the link should take you to the products page, displaying a list of products fetched from the API endpoint.



Congratulations, you now know how to implement server-side rendering in a Next.js and an Express.js application!

**Server-Side Rendering: SEO vs. Performance**

Server-side rendering (SSR) can have both SEO and performance benefits, but there can be trade-offs.

**SEO Benefits**

* SSR improves SEO by making it easier for search engines to crawl and index content.
* SSR sends fully rendered HTML to the client, making it easier for search engines to understand the content.
* Search engines may rank SSR pages higher because they provide a better user experience.
* SSR can help ensure that all content on the page is visible to search engines, including content generated by JavaScript.

**Performance Benefits**

* CSR can offer faster initial page load times because the browser can start rendering the page as soon as it receives the initial HTML and JavaScript files.
* SSR can be slower for initial page loads because the server has to render the HTML, CSS, and JavaScript before sending it to the client.
* Once the page is loaded, subsequent navigation within site can be faster because the server has already done the rendering work.
* SSR can reduce client-side processing, which would benefit slower devices.
* SSR can help reduce the number of network requests required to load a page, improving performance.

**Trade-offs**

* The trade-off between SEO and performance may not be significant for some websites.
* For complex sites with a lot of dynamic content, the performance hit of SSR may outweigh the SEO benefits.
* SSR can be more difficult to implement and maintain than CSR, adding to development costs.

**Final Thoughts**

In conclusion, server-side rendering in React offers a powerful solution for creating dynamic and performant web applications. We can optimize page load speeds, provide a better experience for users with slower devices or limited processing power, and maintain consistency across various browsers by rendering web pages on the server before delivering them to the client.

While some challenges are associated with server-side rendering, such as increased complexity and maintenance considerations, modern technologies like Next.js and Express.js have made it easier to implement and optimize. With caching strategies and proper infrastructure support, server-side rendering can greatly enhance the overall user experience and provide a more SEO-friendly solution for complex websites like e-commerce platforms.

Overall, server-side rendering should be considered a valuable tool in the React developer’s toolkit, especially for projects with large amounts of dynamic content or heavy backend logic. It may provide considerable benefits in performance, user experience, and search engine optimization if proper design and implementation are taken care of.

As React continues to gain popularity, it’s crucial to have skilled and experienced developers to bring your projects to life. You might want to check this out to understand how to [hire a React Developer](https://www.bairesdev.com/technologies/react/hire-developers/) who can deliver high-quality solutions tailored to your needs.

**Frequently Asked Questions (FAQs)**

**Is React server-side rendering faster?**

React server-side rendering can be faster than client-side rendering in certain scenarios. For example, if your application has a large amount of content or data that needs to be loaded before rendering the page, SSR can provide a faster initial load time than CSR. However, in cases where most of the content is dynamically generated by JavaScript, CSR may be faster.

**Is SSR worth it?**

Whether or not SSR is worth it depends on your application’s specific needs and requirements. SSR can provide benefits such as improved initial load times, better SEO, and improved performance on low-end devices or slow network connections. However, implementing SSR can also add complexity to your application and may not be necessary for all use cases.

**Is Facebook a CSR or SSR?**

Facebook uses a combination of CSR and SSR in its applications. They use SSR for the initial load of some of their pages, such as the newsfeed, but rely heavily on CSR for dynamic updates and interactions.

**When should I use SSR?**

SSR can be useful in several scenarios, including when you have a large amount of data or content to load, to improve SEO, or to improve performance on slow network connections. SSR may also be useful for applications that require a high level of accessibility or for applications that need to render on low-end devices. However, SSR can add complexity and overhead to your application, so it’s important to carefully consider the tradeoffs before deciding to use it.