An Introduction to React Server-Side Rendering

Server-side rendering (SSR) is a popular technique for rendering a normally client-side only single page app (SPA) on the server and then sending a fully rendered page to the client. The client's JavaScript bundle can then take over and the SPA can operate as normal. One major benefit of using SSR is in having an app that can be crawled for its content even for crawlers that don't execute JavaScript code. This can help with SEO and with providing meta data to social media channels.

SSR can also often help with performance because a fully loaded app is sent down from the server on the first request. For non-trivial apps though, your mileage may vary because SSR requires a setup that can get a bit complicated and it creates a bigger load on the server. In the end, whether to use server-side rendering for your React app really depends on your specific needs and on which trade-offs make the most sense for your use case.

We've covered the basics of server-side rendering for Vue.js so now let's do the same for React. We'll initialize a simple React app using Create React App and then modify things around to enable server-side rendering without having to eject.

Creating the React App

We'll go ahead and use npx to start up a new React app using the latest version of **Create React App**. Let's call our app `react-ssr-demo`:

```
$ npx create-react-app my-ssr-app
```

And then we can go ahead and `cd` into the new directory and start the our new client-side app:

```
$ cd my-ssr-app
$ yarn start
```

Let's create a simplistic `Home` component:

↓ src/Home.js

```
import React from 'react';

export default props => {
  return <h1>Hello {props.name}!</h1>;
};
```

And let's render the **Home** in the `App` component:

↓ src/App.js

```
import React from 'react';
import Home from './Home';

export default () => {
  return <Home name="Alligator" />;
};
```

Hydrate instead of render

In our app's `index.js` file, we'll use ReactDOM's hydrate method instead of `render` to indicate to the DOM renderer that we're rehydrating the app after a server-side render:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

```
import App from './App';

ReactDOM.hydrate(<App />, document.getElementById('root'));
```

Simple Express Server

Now that we have our simple app in place, let's setup a simple server that will send along a rendered version. We'll use Express for our server so let's go ahead and add it to the project:

```
$ yarn add express

# or, using npm
$ npm install express
```

Next we can create a `server` directory next to the apps `src` directory and an `index.js` file that will contain our Express server code:

```
$ mkdir server && touch server/index.js
```

Here's the content for our simple server:

↓ server/index.js

```
import path from 'path';
import fs from 'fs';

import React from 'react';
import express from 'express';
import ReactDOMServer from 'react-dom/server';

import App from '../src/App';

const PORT = process.env.PORT || 3006;
const app = express();

app.use(express.static('./build'));

app.get('/*', (req, res) => {
```

```
  const app = ReactDOMServer.renderToString(<App />);

  const indexFile = path.resolve('./build/index.html');
  fs.readFile(indexFile, 'utf8', (err, data) => {
    if (err) {
      console.error('Something went wrong:', err);
      return res.status(500).send('Oops, better luck next time!');
    }

    return res.send(
      data.replace('<div id="root"></div>', `<div id="root">${app}</div>`)
    );
  });
});

app.listen(PORT, () => {
  console.log(`😎 Server is listening on port ${PORT}`);
});
```

As you can see, we can import our app component from the cleint app directly from the server. Other than a basic Express app setup, 3 important things are taking place here:

- We tell Express to serve contents from the `build` directory as static files.
- We use a method from `ReactDOMServer`, `renderToString`, to render our app to a static HTML string.
- We then read the static `index.html` file from the built client app, inject our app's static content in the div with a `root` id and send that as the response to the request.

Configuring webpack & Babel

For our server code to work, we'll need to bundle and transpile it, using webpack and Babel. To accomplish this, let's add a bunch of dev dependencies to the project:

```
$ yarn add webpack webpack-cli babel-core babel-loader babel-preset-env babel-pres

# or, using npm:
$ npm install webpack webpack-cli babel-core babel-loader babel-preset-env babel-p
```

> We'll make use of `nodemon` and `npm-run-all` later in our npm scripts.

We can now create a Babel configuration file with the `env` and `react-app` presets:

↓ .babelrc

```
{
  "presets": ["env", "react-app"]
}
```

And we can add a simple webpack config for the server that uses Babel Loader to transpile the code. With the following configuration, our transpiled server bundle will be found in the `server-build` folder in a file called `index.js`:

↓ webpack.server.js

```
const path = require('path');
const nodeExternals = require('webpack-node-externals');

module.exports = {
  entry: './server/index.js',

  target: 'node',

  externals: [nodeExternals()],

  output: {
    path: path.resolve('server-build'),
    filename: 'index.js'
  },

  module: {
    rules: [
      {
        test: /\.js$/,
        use: 'babel-loader'
      }
    ]
  }
};
```

It's a pretty standard webpack config. Note the use of `target: 'node'` and `externals: [nodeExternals()]` form `webpack-node-externals`, which will omit the files from `node_modules` in the bundle; the server can access these files directly.

npm scripts

Let's add a few scripts to help run our SSR setup:

↓ package.json

```
"scripts": {
  "dev:build-server":
    "NODE_ENV=development webpack --config webpack.server.js --mode=development -w
  "dev:start": "nodemon ./server-build/index.js",
  "dev": "npm-run-all --parallel build dev:*",
  ...
},
```

We make use of `nodemon` to restart the server when we make changes to it and `npm-run-all` to run multiple commands in parallel. With this in place, you can run the following to build the client-side app, bundle/transpile the server code and start up the server on port 3006:

```
$ yarn run dev

# or, using npm:
$ npm run dev
```

Our server webpack config will watch for changes and our server will restart on changes. For the client app however we currently still need to build it each time we make changes. There's an open issue for that here.

And now... drumrolls 🥁... you can go to http://localhost:3006/ and you should see our basic server-side rendered app!

🤓 With this post we just scratched the surface at what's possible. Things tend to get a bit more complicated once routing, data fetching and/or Redux also need to be part of a server-side rendered app. We'll explore these topics in subsequent posts.

more react

Creating Complex Animations in React Using react-spring

react-notifications-component, a Powerful React Notifications Library

Intro to Animations in React Using React Spring

A Guide for Refs in React

all react posts

follow us @alligatorio

site sponsors

stay in the loop with our monthly newsletter:

you@awesome.com

Subscribe!

Published: May 27, 2018

Catch up with the latest in front-end web dev with our monthly newsletter:

you@unicorn.com

Subscribe