React.js

Complete Guide To Server-Side Rendering

By Gerard van der Put

React.js

Complete Guide To Server-Side Rendering

Gerard van der Put, May 2020

Introduction

Many React developers find the term **server-side rendering** a scary one. It sounds like a concept from another universe. "Rendering a front-end React application on a server? But what happened to rendering it in the browser? Like I always do?"

That must be something for wizards only.

And if you finally dare to take the step to start searching the internet for information about this topic you will be pushed away even further: new terminology, difficult tutorials and as always countless opinions that could not differ more from each other.

Throughout many years I have collected all the small bits and pieces that are actually useful and compressed them into one single guide that will not only provide you with solutions; it will also explain them thoroughly so you will get a **solid foundation of knowledge** about this topic. Knowledge that you can bring with you when you start developing and maintaining your own server-side rendered React applications.

We will keep it minimalistic and as simple as possible.

Before we start to write code let's answer an important question first.

What Is Server-Side Rendering?

In a regular React application (without server-side rendering) we write our JavaScript and bundle this together in one or multiple files. Most commonly we then serve our users an HTML file with an empty body which - once loaded - will execute the JavaScript from our bundle. This React code will build and manipulate the DOM while the user is interacting with our application. We describe this scenario as **client-side rendering** since the rendering happens in the users browser. It happens on their machine.

Notice how we mentioned that "we serve the users an HTML file with an

empty body". This is the crucial part. When we talk about an application with server-side rendering we will not serve an empty HTML file. Instead, we will let our server serve an HTML file **with a pre-rendered DOM**. After that has happened everything will continue as normal. The JavaScript bundle will be loaded and executed. And React "will take over" and make our application interactive.

Benefits

Faster initial server response

When our users visit a traditional React application with client-side rendering they have to wait for the following events to complete *before they see the rendered DOM elements on the screen* in their browser, in chronological order:

1. Browser request to get the index file for the application
2. Server response with initial index HTML file (with empty body)
3. Browser request to get the JavaScript bundle
4. Server response with JavaScript bundle
5. Browser executes JavaScript code
6. JavaScript code (React) updates the DOM

Let's be fair: in most cases this happens all rather quick. But each of the six theoretical steps (*in a real-world example there would be more steps - think about loading additional assets such as CSS, images and fonts*) take a bit of time and it all adds up. You've probably experienced long initial loading times yourself when opening a website and I bet it frustrated you.

> *When initial loading times are too long users might "bounce". They don't bother to wait and leave your application before they've seen it.*

If you have a large (enterprise-) React application with a large JavaScript bundle the risk for this happening increases significantly. If your users have a rather slow internet connection on top of that you might struggle with

"grabbing their attention" in the first place.

By implementing server-side rendering your users will see your application already after step 2 ("Server response with initial index HTML file"). Sure, they still have to wait for step 6 to be finished before they can fully interact with the application (more about this later on) but at least we already have their attention.

Big players in the tech world take site speed very serious as well. Google wrote already in 2010 that site speed is one of the factors in determining page ranks in their search results[(1)] and Facebook announced something similar in 2017 mentioning that they prioritize fast-loading websites in their news feed[(2)].

(1) https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html
(2) https://about.fb.com/news/2017/08/news-feed-fyi-showing-you-stories-that-link-to-faster-loading-webpages/

SEO (Search Engine Optimization)

Another benefit is that we can optimize our application for search engine crawlers when we use server-side rendering.

Imagine a crawler visiting our React application. If it's a traditional React SPA (Single Page Application) it has to execute the JavaScript before it can parse and process the information that's available in the DOM. How would it navigate? Can it discover all HTML elements to which you've bound click handlers? And even if it would manage that, it has to work hard to gather a summary of the "page" that it's looking at.

When you want to have full control over what is shown by search engines when they list our website pages in their search results we have to manually add meta tags to each important page of a website that we want to be listed (or: indexed). We have to provide keywords, a description of the page and perhaps some other additional meta data.

Server-side rendering gives us the opportunity to do exactly that, in a straight-forward manner.

Disadvantages

In my opinion there is one mayor drawback worth mentioning when it comes to implementing server-side rendering:

It makes the code and hosting of your application more complex.

And it's a big one. You and your team have to strongly consider whether or not it's worth it. All team members have to be aware of the fact that the application has server-side rendering and need to know about how it works. Their local development environment becomes more complex. More things can go wrong.

But we will not dive to deep into this topic. Let's move on.

Proof of Concept Application (POC)

Before we start writing the code for our Proof of Concept application (I will refer to it as "POC" from now on) I want to plant an idea in your head. A thought which might help with processing everything that follows:

When creating an application with server-side rendering you are basically creating and maintaining two versions of your application: a client- and a server-side version.

Luckily they share the same source code. But it's something that we have to keep in mind at all times. There will be two versions of our application, which has consequences. Sometimes we have to make sure that part of our code is conditionally executed: only on the client side and not on the server side, or visa versa. But we will get to that.

Time to write some code!

Chapter I: Setup

Creating a basic React application with Webpack

Note up front: I assume you have basic to intermediate knowledge about developing React applications. Which means I will not write down prerequisites such as "make sure you have yarn and node installed locally".

Create a new working directory and initialize a new yarn project:

```
$ mkdir /var/www/poc
$ cd /var/www/poc
$ yarn init
```

You can enter all the default answers when you run "yarn init".
A new file "package.json" will be created.

Install the following packages:

```
$ yarn add react react-dom
$ yarn add webpack webpack-cli webpack-dev-server
$ yarn add babel-loader @babel/core @babel/preset-env @babel/preset-react
$ yarn add html-webpack-plugin
```

We use **Webpack** as our module- and assets bundler. **Babel** and the related packages are used for transcompiling our code into backwards-compatible plain JavaScript. This way we can use all modern JavaScript features such as spread operators and destructuring assignments without having to worry about compatibility and browser support. And it will allow us to write JSX-syntax; it will be compiled into plain JavaScript as well.

The **html-webpack-plugin** we will get back to soon when we are going to configure webpack.

Before we configure webpack let's create our first two JavaScript files:

src/components/App.jsx

```
import React from 'react';

export default () => {
```

```
  return (
    <div>Hello world!</div>
  )
}
```

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

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

Our **App** component is a minimal functional React component (rendered as a div with some text).

The **index** file contains standard React bootstrap code. The render function from react-dom renders the App component into the provided container (an HTML element with id "root").

Let's create the index file that contains this root element (a div in this case):

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

This is an interesting and relevant file! When we talked about client-side rendering earlier I mentioned that *the server responded with an initial index HTML file (with an empty body)*. That is what we're seeing here.

Let's get our facts straight: the body is not completely empty. But other than our root div there is nothing to see.

Soon when we fire up our POC application and a user requests it in their browser, this file is what the server will send to them. With nothing in the body; JavaScript will fill it once the bundle is loaded and executed.

My apologies for perhaps being a bit too repetitive. But having good understanding of this will become more important later on. Bare with me.

Configure Webpack

Next, let's create the webpack configuration file. We start with the lines:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const rules = [
  {
    test: /\.jsx?$/,
    exclude: /node_modules/,
    use: {
      loader: 'babel-loader'
    }
  }
];
```

We create one rule for now which instructs webpack to compile .js and .jsx files (as long as they're not in the node_modules directory) with the **babel-loader**. We need to instruct babel which presets it should use though.

Create a new file for this:

.babelrc

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

preset-env allows us to use the latest JavaScript syntax and **preset-react** allows us to use JSX-syntax. There are more presets but this will be sufficient for our POC.

Back to our webpack config file; we're not done yet with it. Append the following configuration to it:

webpack.config.js

```
...

module.exports = {
  watchOptions: {
    ignored: /node_modules/
  },
```

```
  entry: path.join(__dirname, 'src', 'index.js'),
  resolve: {
    extensions: ['.jsx', '.js']
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, './build'),
    publicPath: '/'
  },
  module: {
    rules
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html'
    })
  ]
};
```

We don't have to look at every line, but notice that:

- src/index.js is used as the entry point
- our bundle will be saved to build/bundle.js
- public/index.html is used as our template

The last thing we need to do is adding a script called **start** to package.json which will start a development server:

package.json

```
...

  "scripts": {
    "start": "webpack-dev-server --watch --mode development --open --hot"
  }
}
```

Run it from the command line:

```
$ yarn run start
```

It will open the POC in a new tab in your browser, with the address **http://localhost:8080**. Try to change the text "Hello world!" in App.jsx and notice how the application is automatically reloaded in the browser after you save your changes.

Source code
You can find the code from this chapter at gitlab.com in the branch part-1.

Chapter II: Styling, assets and production builds

One might think that it is not necessary to add a chapter about styling and assets to this book. And to some extend that's true. However, when we're creating an application with server-side rendering we run into some situations where we have to account for having styles and assets. We have to deal with this in different ways on the client- and server side. Which is why we have to spent a bit of time working with this and adding it to our POC, so we can illustrate these situations and learn how to deal with them later on.

CSS/SCSS

First things first: we add some new dependencies to our project.

```
$ yarn add css-loader node-sass sass-loader
$ yarn add mini-css-extract-plugin
```

The loaders will be used to instruct webpack and tell it how to process (s)css files. The **mini-css-extract-plugin** will extract CSS into separate files after it has been processed by our loaders.

Now we extend our webpack configuration. Add a new rule to the **rules** constant:

webpack.config.js

```
...

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const rules = [
  ...
  {
    test: /\.s?css$/,
    exclude: /node_modules/,
    use: [
      {
        loader: MiniCssExtractPlugin.loader,
        options: {
          hmr: process.env.NODE_ENV === 'development'
        }
      },
      "css-loader",
      "sass-loader"
    ]
```

```
  }
];

...
```

This will instruct webpack to compile .scss and .css files with the sass-loader, css-loader and the loader from the MiniCssExtractPlugin.

We also have to register the plugin by adding it to the plugins section.

webpack.config.js

```
...

  plugins: [
    ...
    new MiniCssExtractPlugin()
  ]

...
```

Next we create a file containing some basic styling.

src/index.scss

```
@import url('https://fonts.googleapis.com/css2?
family=Source+Sans+Pro:wght@300;400;700&display=swap');

$fontDefault: 'Source Sans Pro';

body {
  font-family: $fontDefault;
  font-size: 1em;
  color: #444;
}
```

And we import it into our index.js file.

src/index.js

```
import './index.scss';
...
```

Since we've made changes to our webpack configuration we have to restart our **start script**. Kill the running process from before (CTRL + C) and re-run it.

```
$ yarn run start
```

Our styling is now applied when we view our POC in the browser.

Static assets

Before we move on to creating a production build of our POC we will add an image to our page. I copied an image of a cat (we all love cats, don't we?) to **public/static/kitten.jpg**. You can find it in the gitlab.com branch which is linked at the end of this chapter. Or simply use another image.

Render the image inside App.jsx by changing its contents into:

src/components/App.jsx

```
import React from 'react';

export default () => {
  return (
    <>
      <div>Hello world!</div>
      <img src="/static/kitten.jpg" alt="kitten" />
    </>
  )
}
```

Comment: notice how we use a React fragment (short syntax): <>

And finally make sure that webpack copies our static assets to our build folder. We will use the dependency copy-webpack-plugin for this. Add it to our project:

```
$ yarn add copy-webpack-plugin
```

And adjust our webpack config:

webpack.config.js

```
...

  plugins: [
    ...
    new CopyWebpackPlugin([
      {
        from: path.resolve(__dirname, './public/static/'),
        to: path.resolve(__dirname, './build/static/') }
    ])
  ]

...
```

Restart our **start script** once more and you can admire our little kitten in the POC.

```
$ yarn run start
```

Production build

In the next chapter we start with the actual server-side rendering. But before we do so I want to make sure that we're able to create a production build of our application. Again, like we argued in the beginning of this chapter when we started talking about styling and assets, this will be important later on when we illustrate certain aspects of server-side rendering.

Add a new script called "build" to package.json which will compile and collect all files required to run our POC in a production environment.

package.json

```
...

  "scripts": {
    ...
    "build": "webpack --mode production"
  }
}
```

> If you are using git make sure to never commit directories that are generated by scripts. When we run our newly created **build** script it will create a directory "build" which contains all files of our production build. It would be bad practice to commit them. Add them to your **.gitignore** file.

Our new build script will use the default (and only...) webpack configuration file in our project. This means both our development server (started with "yarn run start") and production build will share the same webpack config file. In a mature project you would probably want to use separate configuration files for each process. But for the purpose of this POC sharing the same file will be sufficient.

Run our new build script:

```
$ yarn run build
```

The "build" directory in our root working directory will now contain all files for our production build. If you would like to see if it works you could use the popular serve package from npm to serve the contents locally:

```
$ yarn global add serve
$ cd /var/www/poc/build
$ serve
```

Source code
You can find the code from this chapter at gitlab.com in the branch part-2.

Okay!

We now have a very minimalistic React application with standard client-side rendering. We can use styles, assets and we are able to create a production build. So far so good. Grab something warm to drink and let's add some server-side rendering...

Chapter III: Server-Side Rendering

Make sure our development server is running:

```
$ yarn run start
```

Open our POC in the browser (http://localhost:8080). Let's have a look at the **page source** by right clicking and selecting "view page source". You are now viewing the source that was returned by the server when we initially requested the page. As we have mentioned several times before already, you can see that the body of our HTML page is empty in this source. It only contains a div element with an id-attribute, value "root". Close the page source again.

Now **inspect** the page instead by right clicking and selecting "Inspect" (Chrome) or "Inspect element" (Firefox). This will show us the current state of the DOM. Have a look inside our root div. There is our rendered App component. It was added to the DOM by React when our JavaScript bundle was loaded and executed!

What we will do next is making sure that our **page source** will already contain our rendered App component, inside the root div. Or, in other words, that our server will return our rendered application instead of an index HTML file with an empty body. This is server-side rendering in its essence.

> *(...) our server will return our rendered application instead of an index HTML file with an empty body. This is server-side rendering in its essence.*

Sounds simple enough. But once your application becomes more mature (and we will make sure our POC becomes that, later on) when we add for example state management, there are quite some things to consider. And as with most matters the initial setup is the hardest part.

We will move forward step-by-step.

Add the following new dependencies to our project:

```
$ yarn add ignore-loader nodemon express webpack-node-externals
```

Soon we will create a separate webpack configuration for rendering our application server-side. We then need the **ignore-loader** to ignore certain modules (files) when we compile our source code.

Express is the most popular web application (server-) framework for Node.js[3]. We will set up our server with it. **Nodemon** is a tool that automatically restarts a node server when changes in files are detected, which will help us during development. It would be very cumbersome to have to restart our server manually every time we've made changes to its source code. And finally we will be using **webpack-node-externals** in order to prevent that our dependencies are bundled by webpack when we build our server files.

(3) May 2020: Express has more than 48k stars on github.com

Don't worry if none of that made sense. It will all become clear soon.

Create a new file called server.js inside our src directory. Start with importing and requiring some dependencies:

src/server.js

```
import App from "./components/App";
import React from 'react';

const fs = require('fs');
const path = require('path');
const http = require('http');
const express = require('express');
const reactDOMServer = require('react-dom/server');
const { renderToString } = reactDOMServer;
```

App and React don't require explanation I assume. **fs** will help us with working with files (e.g. reading them). **path** helps us with resolving filename paths and **http** allows us to use the HTTP server so we can receive requests and send responses; it works tightly together with express. And finally we will use the **renderToString** function from react-dom/server to render our App component into a string.

After our imports we append the following to our server.js file in order to initiate a small express server:

src/server.js

```
...

let app = express();
app.server = http.createServer(app);

app.use('/static', express.static(path.resolve('build/static')));
app.use('/main.css', express.static(path.resolve('build/main.css')));
app.use('/bundle.js', express.static(path.resolve('build/bundle.js')));

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

  const filePath = path.resolve('build', 'index.html');
  fs.readFile(filePath, 'utf-8', (err, data) => {

    // todo
  });
});

const port = process.env.PORT || 8080;
app.server.listen(port);
console.log(`Listening on port ${port}`);
```

First we initialize our server instance.

The lines where we call **app.use()** define that we want our server to serve some specific static content:

- the build/static directory (containing our cute kitten)
- build/main.css
- build/bundle.js

For all other requests we will serve the build/index.html file[4].
See the line where we call **app.get()**.

Once the file is read into memory a callback is executed. The callback is provided as the third parameter in the readFile call. We will have a look at what we will do inside it next.

Finally we define on which port our server should run (fallback value 8080), we define that our express server should listen on that port, and we print a message to indicate that the server has started and is listening.

(4) Hey! That's our index HTML file again with the empty body! But not for long...

We replace the "todo" comment from the previous code fragment with the following:

src/server.js

```
...

    if(err) {
      console.error(err);
      res.status(404).send('Error: 404');
    }

    const reactHtml = renderToString(<App />);
    const html = data.replace('{{HTML}}', reactHtml);
    res.status(200).send(html);

...
```

Look at this code for a brief moment and some puzzle pieces should fall on their place now.

The error handling is straight forward and self-explanatory. It will kick in when we - for some reason - are not able to read the build/index.html file (remember that we're inside the callback handler of the readFile call).

But the last three lines are very interesting. We render our App component into a string... Then we replace a placeholder-string {{HTML}} in the contents of our index.html file with that string... And we send the results back as a response.

At this point you should be thinking of two things, being that - when we run this server - we:

1. ...don't have an empty body in our index HTML file anymore!
2. ...don't have a {{HTML}} string in our index file, do we?

1) Correct! 2) Correct as well. Let's fix that. Our complete index.html file should look like this:

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
</head>
<body>
  <div id="root">{{HTML}}</div>
</body>
</html>
```

Now those were probably quite some new concepts and quite a bit of new code. Let's see if we can run all this. In order to do that we need a new webpack configuration, specific for this express server: **webpack.server.config.js**. Again, we start with the first part:

webpack.server.config.js

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

const rules = [
  {
    test: /\.jsx?$/,
    exclude: /node_modules/,
    use: {
      loader: 'babel-loader'
    }
  },

  {
    test: /\.s?css$/,
    exclude: /node_modules/,
    use: ["ignore-loader"]
  },
  {
    test: /\.(png|jpe?g|gif|svg)$/,
    use: ["ignore-loader"]
  }
];
```

The important part is that we *ignore* (s)css files and image files with the **ignore-loader**, which we mentioned earlier. Why? Because we don't have use for them in our express server. These static assets are already copied to the correct directory (build/) by our client-side application build script.

Let's create the final part of the file as well, by appending the following:

webpack.server.config.js

```
...

module.exports = {
  watchOptions: {
    ignored: /node_modules/
```

```
  },
  target: "node",
  entry: path.join(__dirname, 'src', 'server.js'),
  externals: [nodeExternals()],
  resolve: {
    extensions: ['.jsx', '.js']
  },
  output: {
    filename: 'server.js',
    path: path.resolve(__dirname, './build-server'),
    publicPath: '/'
  },
  module: {
    rules
  },
  plugins: [
    new webpack.DefinePlugin({
      __isClientSide__: "false"
    })
  ]
};
```

The differences between this code and our original webpack.config.js file are:

1. target "node" will make sure our code is compiled for usage in a Node environment
2. different entry point (src/server.js)
3. different output path and filename (build-server/server.js)
4. No plugins, except one new one (see next paragraph)

The new plugin that was added (**webpack.DefinePlugin**) allows us to define a variable that will be available inside our application code. Before we explain what it does let's also add it to our client-side webpack configuration file:

webpack.config.js

```
...

  plugins: [
    ...
    new webpack.DefinePlugin({
      __isClientSide__: "true"
    })
  ]
};
```

Now we can use the variable **__isClientSide__** inside our application code to determine if the code is currently executed client-side/in the browser (value: "true") or server-side on our node server (value: "false"). We will use it soon.

A brief moment of reflection; the way we've configured our express server, and our webpack configuration files, is of course discussable. Things can be handled in multiple ways. I'm sure I will receive feedback and comments. But the scope of this book is much broader than configuring express and webpack. My main focus is creating a React application with server-side rendering and explaining you, the reader, the concepts and theory behind it in an easy to follow manner.

Scripts

Adjust the scripts section in package.json so it looks like this:

package.json

```
...

    "scripts": {
      "start": "webpack-dev-server --watch --mode development --open --hot",
      "build": "webpack --watch --mode production",
      "server:build": "webpack --watch --config webpack.server.config.js --mode
production",
      "server:start": "export PORT=8081 && nodemon build-server/server.js"
    }

  ...
```

We added the **--watch** flag to the already-existing **build** script. This will make sure that our client-side production bundle is rebuild when we make changes in our application code.

We added two new scripts: **server:build** and **server:start**. The former will compile our server.js file into a production build (it will be written to build-server/server.js) and it will rebuild when we make changes. The latter will serve this production build (it will start our server) by starting a node process which is monitored by nodemon. As mentioned earlier, by using nodemon we will make sure that the server process restarts when we make changes to our server code, which is convenient.

Now kill all processes that you might have running still, and run the following three scripts:

```
// terminal 1
  $ yarn run build

  // terminal 2
  $ yarn run server:build
```

```
  // terminal 3
  $ yarn run server:start
```

Once the three processes are running, open our POC in the browser by visiting **http://localhost:8081**. You will see our application. But how disappointing, everything looks the same as before? Actually, *quite some things have changed*.

First of all, look at the **page source** again. The initial server response didn't return an empty body with just our root div; it actually returned our rendered App component inside it:

view-source:localhost:8081

```
...

<div id="root">
  <div>Hello world!</div>
  <img src="/static/kitten.jpg" alt="kitten"/>
</div>

...
```

Server-side rendering!

Once our browser displays this data our JavaScript bundle will be loaded, the code is executed and React takes control. Because everything happens so fast, especially while we're developing on our local machine, it's hard to see what's going on. But we can visualize it with some small tricks.

Change App.jsx into:

src/components/App.jsx

```
import React from 'react';

export default () => {

  const env = __isClientSide__ ? 'client' : 'server';

  return (
    <>
      <div>Hello from {env}!</div>
      <img src="/static/kitten.jpg" alt="kitten" />
    </>
  )
}
```

Notice how we use the __isClientSide__ variable that's declared in the webpack configuration files. If you refresh our POC in the browser you will see the text "Hello from client!". We don't see the HTML that's generated by the server because everything happens so fast. We can simulate that it takes a while to execute the code in our JavaScript bundle by wrapping the "react-dom.render()" call inside index.js with a time-out:

src/index.js

```
...

setTimeout(() => {
  ReactDOM.render(
    <App/>,
    document.getElementById('root')
  );
}, 1000);
```

Now refresh our POC again in the browser. The first second you will see the HTML generated by the server ("Hello from server!"), and after our time-out has passed by (1 second) we will see that our "react-dom.render()" from index.js gets executed and the text "Hello from client!" is rendered on screen. React has taken over.

This very nicely visualizes the server-side rendering concept. We leave the time-out inside index.js for now.

Source code
You can find the code from this chapter at gitlab.com in the branch part-3.

Chapter IV: State management

Almost every front-end application needs some form of state management. While React hooks can be extremely useful and easy to use we go for a more solid approach here. I don't want to start the heated discussion "Hooks vs Redux" right now. Let's just say that **Redux** is more fitted for the challenges that lay ahead of us in the next part.

Server-side rendering introduces some challenges when it comes to state management. We will have a look at them. I assume you have good basic knowledge about Redux. If not, please read and learn a bit about it before continuing.

Set up Redux with Redux-Toolkit

On March 24th 2020 Redux-Toolkit v1.3.0 was released. Using the toolkit will reduce the amount of boilerplate code that we need to set up state management in our application. The toolkit bundles Redux with some handy tools such as **redux-thunk**, **reselect** and **Immer** (more about Immer later on). In general it will simplify things for us.

Add the following packages to our project:

```
$ yarn add @reduxjs/toolkit react-redux
```

Create a new file which we use for defining a simple reducer and some actions:

src/reducers/person.js

```
import { createSlice } from "@reduxjs/toolkit";

const initialState = {
  name: 'John Doe',
  age: 35
};

export const slice = createSlice({
  name: "person",
  initialState,
  reducers: {
    ageIncrement: state => { state.age += 1 },
```

```
    ageDecrement: state => { state.age -= 1 }
  }
});

export const { ageIncrement, ageDecrement } = slice.actions;

export default slice.reducer;
```

Our simple state contains two variables: name and age. We create two actions which we can use to increment and decrement the age. We export the reducer that's created by *createSlice*(5), and the two actions.

Note how we manipulate the state object directly inside the ageIncrement function, for example. Normally this is a big no-no. Because our state should be considered to be immutable and we should return a new state object with the changes applied. But this is one of the advantages of using Redux-toolkit. It comes bundled with a library called **Immer** which allows us to manipulate objects and arrays in the traditional way. And it will make sure that a new (state-) object is automatically generated. Also see that we *don't return* the state object. We simply manipulate it inside the function body. That's enough. Read more about Immer on their website, they won several prices in 2019.

(5) **createSlice** is (quote) "a function that accepts an initial state, an object full of reducer functions, and a slice name, and automatically generates action creators and action types that correspond to the reducers and state". From redux-toolkit.js.org

Create a new **store** in which we keep track of our person state:

src/store.js

```
import person from './reducers/person';
import { configureStore } from "@reduxjs/toolkit";

export default () => configureStore({
  reducer: {
    person
  }
});
```

And make the store available in our application code, inside index.js:

src/index.js

```
...

import { Provider } from 'react-redux';
import createStore from './store';
```

```
setTimeout(() => {

  const store = createStore();

  ReactDOM.hydrate(
    <Provider store={store}>
      <App/>
    </Provider>,
    document.getElementById('root')
  );

}, 2000);
```

We made a couple of changes:

I. Create a new instance of our store, by calling createStore
II. Wrap the App component in a Provider, which holds a reference to the store
III. Replaced ReactDOM.render with ReactDOM.hydrate
IV. Increased our time-out a bit, to two seconds

A) The new store instance will contain the **initial person state** (name: John Doe, age: 35). **B)** By wrapping our root App component in a Redux Provider we make the store **available** to all components in our application. **C)** We replaced the react-dom.render() call with **react-dom.hydrate()**. This optimizes our render process because we can leverage the fact that our server has already rendered our application. Client-side we don't have to do that again. We can simply assume that our server-side renderer did a good job(6). **D)** A bigger delay before executing react-dom.hydrate() so we have a bit more time to see what's going on in the browser.

(6) Source: https://reactjs.org/docs/react-dom.html#hydrate

Now we display some data from our store in our App component:

src/components/App.jsx

```
import React from 'react';
import { connect } from 'react-redux';

const App = ({ name, age }) => {

  const env = __isClientSide__ ? 'client' : 'server';

  return (
    <>
      <p>Hello {name}, from {env}!</p>
      <p>Your age is: {age}</p>
```

```
      <img src="/static/kitten.jpg" alt="kitten" />
    </>
  )
};

const mapStateToProps = state => ({
  name: state.person.name,
  age: state.person.age
});

export default connect(
  mapStateToProps,
  null
)(App);
```

This is the standard way of using data from a Redux store: we *connect* the functional component with the store and define in the constant **mapStateToProps** which data from the store we want to send to our component props. If the data changes, our component will rerender automatically.

Try to refresh our POC in the browser. We get an error (you can see the details in the terminals where we run our scripts). The reason is straight forward: when we use the connect() function, Redux assumes that our component has access to a store. We wrapped our App component in the high-order Provider component which does exactly that. However, we didn't do this on the server-side yet! Let's fix that inside server.js:

src/server.js

```
...

import { Provider } from 'react-redux';
import createStore from './store';

...

    const store = createStore();

    const reactHtml = renderToString(
      <Provider store={store}>
        <App />
      </Provider>
    );

...
```

Refresh the POC in the browser again and all is good now. Some of you have noticed that we now have *two instances* of our store: one on the client-side, and one on the server-side. That's asking for trouble. We will look into

synchronizing their data soon, when it becomes relevant.

Dispatch actions client side

Before we move on we will add a new **setAge** action to our *slice* (src/reducers/person.js) and create three buttons in the App component which will dispatch our actions, so we can manipulate the data in our store dynamically:

src/reducers/person.js

```
...

export const slice = createSlice({
  ...
  reducers: {
    ...
    setAge: (state, action) => { state.age = action.payload }
  }
});

export const { ageIncrement, ageDecrement, setAge } = slice.actions;
...
```

src/components/App.jsx

```
...
import { connect } from 'react-redux';
import { ageIncrement, ageDecrement, setAge } from "../reducers/person";

const App = ({ name, age, ageIncrement, ageDecrement, setAge }) => {

  ...

  return (
    <>
      <p>Hello {name}, {env}!</p>
      <p>Your age is: {age}</p>
      <p>
        <button onClick={ageDecrement}>younger</button>
        <button onClick={ageIncrement}>older</button>
        <button onClick={() => setAge(50)}>age = 50</button>
      </p>
      ...
    </>
  )
};

...

const mapDispatchToProps = {
  ageIncrement,
  ageDecrement,
  setAge
};
```

```
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

Refresh the POC in the browser and see how we can manipulate the age value by clicking the buttons now.

> **Hint**
> Try clicking the buttons while you see the text "Hello John Doe, from **server**!". You will notice that nothing happens. That's because at that moment we only have static HTML in our DOM (which was rendered server-side) and our React code is not executed yet, on the client side. So no event handlers (e.g. click) are bound to our DOM elements yet.

Dispatch actions server side

What if we want to dispatch actions on the server side? Maybe we want to fetch some remote data, save it in our store, render our application and then send the server response (index HTML file with rendered application) to the client. Let's start a bit simpler, with calling **setAge** inside server.js.

src/server.js

```
...

import { setAge } from "./reducers/person";

...

    const store = createStore();
    store.dispatch(setAge(75));

    const reactHtml = renderToString(
      <Provider store={store}>
        <App />
      </Provider>
    );

...
```

Check in the browser what happens now when we refresh our POC. When the server-side rendered HTML is displayed we see the text "Your age is: 75" on screen. As expected. But after two seconds react-dom.hydrate() is called and it jumps back to "You age is: 35".

That's because we never dispatched the action on our *client side store*. As

mentioned before, we have two store instances. And they are not synchronized. As soon as the client side code is executed a new store is initialized and used, and it will contain an **initial state** (name: John doe, age: 35) again.

But there is a solution.

Sharing state between server and client

We can send the state from the store on the server side, to the client. And use that state as our **initial client side state**, in order to make sure that our states contain the same data.

First, add a new package:

```
$ yarn add serialize-javascript
```

This allows us to serialize the data object from our store into a string.
We make some changes in server.js and index.html:

src/server.js

```
...
import serialize from 'serialize-javascript';

...

    const html = data
      .replace('{{HTML}}', reactHtml)
      .replace('{{INITIAL_STATE}}', serialize(store.getState(), { isJson: true }));

...
```

public/index.html

```
<body>
  <div id="root">{{HTML}}</div>
  <script>
    window.__INITIAL_STATE__ = {{INITIAL_STATE}}
  </script>
</body>
```

See how we serialize the data from our store and pass it to our index file where it is stored as a global variable with the name **__INITIAL_STATE__**.

We can now use this variable to initialize our store on the client side. First we make sure our createStore functionality inside store.js is able to receive an

initialState variable:

src/store.js

```
export default initialState => configureStore({
  ...
  preloadedState: initialState
});
```

Then we can pass our newly created variable to it:

src/index.js

```
...
  const initialState = window.__INITIAL_STATE__ || {};
  const store = createStore(initialState);
...
```

Have a look at the POC in the browser again. The age is 75 to begin with, and it stays like that!

A small recapitulation: we're initializing a new store server-side, make changes to its data by dispatching an action (setAge), we then send the state serialized to the client where it is picked up when we initialize our client-side store.

Source code
You can find the code from this chapter at gitlab.com in the branch part-4.

Chapter V: Asynchronous state manipulation

More often than not we need to fetch remote data in our applications by calling an API. Let's have a look at how we can make that work in our POC with server-side rendering. Since we will be fetching data from both client-side (browser) and server-side (our node express server) we will use a fetch function that can be used by both. Install the new package:

```
$ yarn add isomorphic-fetch
```

Fun-fact: as of May 2020 it is downloaded more than 4.5 million times per week!

Before we begin we need an API that we can call. We create a dummy API end-point in server.js which returns an array with the names of five of the friends of John Doe:

src/server.js

```
...

app.get('/api/friends', (req, res) => {
  setTimeout(() => {
    res.json({
      "friends": [
        'James', 'Eric', 'Olivia', 'Emma', 'Charlotte'
      ]
    });
  }, 1000);
});

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

As we have done several times before we add a bit of artificial network delay; one second. Check if our API works by opening http://localhost:8081/api/friends. After one second the json should be returned.

Client side implementation

First make some changes to our reducer, person.js:

src/reducers/person.js

```
...

const initialState = {
  name: 'John Doe',
  age: 35,
  friends: {
    isLoading: false,
    data: []
  }
};

export const slice = createSlice({
  ...
  reducers: {
    ...
    fetchFriendsStart: state => { state.friends.isLoading = true },
    fetchFriendsSuccess: (state, action) => {
      state.friends.isLoading = false;
      state.friends.data = action.payload;
    }
  }
});
```

We add a new property to our state called **friends**, which holds information about fetching John Doe's friends. Then we added two new actions which we will dispatch when we start fetching the data (**fetchFriendsStart**) and when we receive the response (**fetchFriendsSuccess**). Normally you would also add a third action which can handle the situation where something goes wrong (fetchFriendsError) but let's focus on what's important for us at this moment instead.

You notice that we have not created the actual functionality that fetches the data. That's what we do now. Since this will be asynchronous logic (it takes a bit of time before we get our response back from the API) we have to create a so-called *thunk*(7) for this:

(7) Read more about redux-thunk middleware on their github page.

src/reducers/person.js

```
import fetch from 'isomorphic-fetch';

...

export const fetchFriends = () => dispatch => {

  const { fetchFriendsStart, fetchFriendsSuccess } = slice.actions;

  dispatch(fetchFriendsStart());

  return fetch("http://localhost:8081/api/friends")
    .then(response => response.json())
```

```
    .then(json => json.friends)
    .then(friends => dispatch(fetchFriendsSuccess(friends)))
};
...
```

It's rather straight-forward what's going on here. In the function body the first thing we do it dispatching our action that notifies our store about the fact that we start fetching the data. And finally we return a Promise (isomorphic-fetch's "fetch" function returns a Promise) which will dispatch our **fetchFriendsSuccess** action once we've received and extracted the data from the response.

Now we can dispatch our new **fetchFriends** asynchronous action inside our application code. Since our App component is a functional component (and not a class - in which case we could've used the componentDidMount lifecycle method) we can use the useEffect hook that was introduced in React 16.8 in order to fetch the friends when the component is mounted to the DOM:

src/components.App.jsx

```
...
import {ageIncrement, ageDecrement, setAge, fetchFriends} from "../reducers/person";

const App = ({ name, age, ageIncrement, ageDecrement, setAge, fetchFriends, friends,
isLoading }) => {

  useEffect(() => {
    fetchFriends();
  }, []);

  const env = __isClientSide__ ? 'from client' : 'from server';

  return (
    <>
      <p>Hello {name}, {env}!</p>
      <p>Your age is: {age}</p>
      <p>Is fetching: {(isLoading ? 'yes, please wait...' : 'no')}</p>
      <p>Friends: {friends.join(', ')}</p>
      ...
    </>
  )
};

const mapStateToProps = state => ({
  ....
  friends: state.person.friends.data,
  isLoading: state.person.friends.isLoading,
});

const mapDispatchToProps = {
  ...
```

```
  fetchFriends
};

...
```

We also display information about the friends when we render our component so we can see their names, and whether or not the request is pending.

Open our POC in the browser again. This is what happens now, in chronological order:

1. We see the HTML that's returned by the server
2. After two seconds, React takes over
3. At *that moment* we start fetching the friends
4. After one second, the friend-names are shown on screen

...which is great. But why are the friends not fetched in the first place, when our App is rendered server-side? Why are their names not rendered in the index HTML file that is returned by the server?

The answer is simple: React does not execute hooks when it's rendering server-side.

It requires a manual effort.

Server side implementation

We already dispatch one action server side: setAge (synchronous logic). We now want to dispatch an additional action: fetchFriends (asynchronous logic). We alter our server.js file as following:

src/server.js

```
...
import { setAge, fetchFriends } from "./reducers/person";
...

    const store = createStore();

    const promises = [
      store.dispatch(setAge(75)),
      store.dispatch(fetchFriends())
    ];
```

```
    Promise
      .all(promises)
      .then(() => {

        ...
        res.status(200).send(html);
      })

...
```

We treat both dispatch calls as Promises. The first one isn't, but that's okay: "*The Promise.all() method returns a single Promise that fulfills when all of the promises passed as an iterable have been fulfilled or when the iterable contains no promises or when the iterable contains promises that have been fulfilled and non-promises that have been returned*"(8).

(8) Source: https://developer.mozilla.org/.../Global_Objects/Promise/all

In plain English: when both **setAge** and **fetchFriends** are done, we send our HTML. That has the drawback that our respond time is increased by the amount of time it takes to fetch our friends. We added a delay of one second to our /api/friends end-point which means that when we open our POC now in the browser, it takes approximately one second before we get the initial response from the server.

Comment
In real-life such important initial API calls should not take one second. Such results should be cached server-side, and should be available immediately for our server. Popular frameworks such as Gatsby do exactly that.

The good news is that the friends are now fetched server-side. They are rendered server-side and are added to the body inside the index HTML. They are also transferred to the client-side store state, in the same way the **age** is transferred (in the __INITIAL_STATE__ global variable).

But! If you look closely to our POC at this moment you will notice some unwanted behavior. Once our client-side JavaScript takes over (in other words, when the text "Hello John Doe, from client!" appears in the browser) it starts to fetch the friends again. This is redundant. We already have that data in our store. We can prevent it by altering our useEffect call inside the App component slightly:

src/components/App.jsx

```
...

  useEffect(() => {
    if(!friends.length) {
      fetchFriends();
    }
  }, [friends]);

...
```

I won't say this is a proper solution. But it illustrates our point.

Finally let's remove our artificial delay in the /api/friends API end-point:

src/server.js

```
...

app.get('/api/friends', (req, res) => {
  res.json({
    "friends": [
      'James', 'Eric', 'Olivia', 'Emma', 'Charlotte'
    ]
  });
});

...
```

Refresh the POC one more time in the browser and see how smooth it behaves now!

In the final part we will add routing to our application. Once that's done (and we have removed all other artificial delays that we've added along the way) we can proudly conclude that we have a full-blown React single-page application, with server-side rendering and global state management.

Source code
You can find the code from this chapter at gitlab.com in the branch part-5.

Chapter VI: react-router

Routing

We only have one single route in our POC application. However, more often than not real-life applications have multiple routes. We will use several packages to create a dynamic and configurable routing setup. It will also provide us with the possibility for having sub-routes. Needless to say we will make sure everything works both client- and server-side.

Add the following packages to our project:

```
$ yarn add react-router react-router-dom react-router-config
```

React-router contains core functionality and **react-router-dom** provides components that we can use inside our application components for setting up the routing logic. The **react-router-config** package helps us configuring our routes in a dynamic, normalized and abstract way so that the routes (and what they will render) are easy to maintain.

Let's start off with creating two brand-new functional components, **Home** and **Cat**:

src/components/Home.jsx

```
import React from 'react';

export default () => {
  return (
    <p>Welcome to the home page</p>
  )
}
```

src/components/Cat.jsx

```
import React from 'react';

export default () => {
  return (
    <p>
      <img src="/static/kitten.jpg" alt="kitten" />
    </p>
  )
}
```

Since we now render the cat image in our Cat component, we **remove it** from the App component.

Next, create a new file called **routes.js** in which we will configure two routes:

src/routes.js

```
import Home from "./components/Home";
import Cat from "./components/Cat";

export default [
  {
    name: "Home",
    path: "/",
    exact: true,
    component: Home
  },
  {
    name: "Cat",
    path: "/cat",
    component: Cat
  }
];
```

For each route we have configured several properties:

- An arbitrary **name**
- The **path** for the route
- Which **component** to render for the route

For the Home route we also specify that it should only be rendered when there is an *exact location match* (as opposed to the Cat route which is more "liberal"; it will be rendered when the path is "/cat", "/cat/name", etc.).

Just like we did when we wrapped our App component in a Provider component in order for our component-tree to be able to access the Redux store, we will wrap it in another high-order component called **BrowserRouter**.

This way all our components can have access to routing-related information:

src/index.js

```
...
import {BrowserRouter} from "react-router-dom";
...
  ReactDOM.hydrate(
```

```
    <Provider store={store}>
      <BrowserRouter>
        <App/>
      </BrowserRouter>
    </Provider>,
    document.getElementById('root')
  );
...
```

The next step is to use the function **renderRoutes** from react-router-config at the place where we want to render our "route dependent" components (inside App.jsx):

src/components/App.jsx

```
...
import { renderRoutes } from "react-router-config";
import routes from '../routes';
...

  return (
    <>
      ...
      <p>
        <button onClick={ageDecrement}>younger</button>
        <button onClick={ageIncrement}>older</button>
        <button onClick={() => setAge(50)}>age = 50</button>
      </p>
      {renderRoutes(routes)}
    </>
  )
...
```

We pass our routes configuration to this function as a parameter and it will return the component that should be rendered for the current route. If we would visit the location *http://localhost:8081* the Home component is returned, and if we visit *http://localhost:8081/cat* our Cat component is returned.

As usual we also have to make changes inside server.js. When we render our App component there we also need to wrap it in a Router component:

src/server.js

```
...
import {StaticRouter} from "react-router-dom";
...
        const reactHtml = renderToString(
          <Provider store={store}>
            <StaticRouter location={req.url}>
              <App />
            </StaticRouter>
          </Provider>
```

```
        );
```

Server-side we don't have a browser context so we have to be more explicit about what's going on. We use the **StaticRouter** component instead and tell it which location we are currently rendering. This information is available in the variable "req" (which stands for request) which was passed to our callback function that's used when we're processing an incoming request.

Before we check the results in the browser (spoiler: it works) we will add a simple menu to our application which allows the user to navigate. By using the Link component from react-router-dom we will get some powerful functionality.

Create a new component. We will use it to render a single button in our upcoming main menu:

src/components/MenuLink.jsx

```
import React from 'react';
import { Link, matchPath, withRouter } from 'react-router-dom';

const MenuLink = ({ route, location }) => {

  const isActive = matchPath(location.pathname, route) !== null;

  return (
    <div className={"menuLink" + (isActive ? ' active' : '')}>
      <Link to={route.path}>{route.name}</Link>
    </div>
  );
};

export default withRouter(MenuLink);
```

Our functional component has two properties. **route** is one of the objects from our routes configuration file (routes.js). We will pass it when we start using our new component. The second property **location** is passed to our component because at the end of the file, where we export our function, we wrap it in a function call to **withRouter**. withRouter passes a location object as a property to the given component. It contains information about the current route. It also passes several other properties(9).

(9) More information: https://reacttraining.com/.../api/withRouter

Now we can use the location property together with a helper function called

matchPath to determine whether or not the link should be rendered as the *currently active link*. Before we create our main menu add some simple styling to our scss file:

src/index.scss

```
...

div.menuLink {

  display: inline-block;
  padding: 5px 10px;
  margin-right: 1px;
  background-color: #ddd;

  & > a {
    color: blue;
  }

  &.active {
    background-color: #bbb;
  }
}
```

Next we create the **main menu** inside App.jsx. We iterate over the routes config array and render a MenuLink component for each route:

src/components/App.jsx

```
import MenuLink from "./MenuLink";
...
  return (
    <>
      ...
      <div>
        {routes.map(route => (
          <MenuLink route={route} />
        ))}
      </div>
      {renderRoutes(routes)}
    </>
  );
```

Now open our POC in the browser and refresh the page. We have a functional main menu and we can navigate between Home and Cat. It's important to note and interesting to learn that react-router is only manipulating the DOM when we navigate. Traditionally when we click a link (an HTML a-element with a href attribute) a new request is made to the server and the entire DOM is replaced by the response. But by using react-router we have a real single-page application: our DOM is never completely replaced. This results in a fast and responsive user experience.

But what if we open a new tab in our browser and navigate directly to **/cat**? Go ahead and try it. Everything works as expected. Our server is aware of routing since we're using StaticRouter. So it's able to render the correct components and return them, just like we expect.

Page source
Try to navigate directly to http://localhost:8081/cat in a new tab in your browser. Right click the page and view its source. Our server returned the rendered components that we expected to be returned, including Cat.jsx.

Sub-routing

As a last exercise we will have a quick look at sub-routing and query parameters. They are common needs. Perhaps we want to have two (sub-) pages *within our Cat route*? I will not explain too much about the new code because the principles are the same as before, when we created our main routing logic.

First create two new components:

src/components/CatName.jsx

```
import React from 'react';

export default () => {
  return (
    <p>
      I am a kitten and my name is Felix.
    </p>
  )
}
```

src/components/CatColor.jsx

```
import React from 'react';
import { useParams } from 'react-router';

export default () => {

  const { color } = useParams();

  return (
    <p>
      ...and I am <strong>{color}</strong>!
    </p>
  )
}
```

Notice how we use a new function called **useParams** from react-router in order to be able to receive the values of provided query parameters.

Now update our routes configuration file and add two *sub routes*. We do so by adding an array of regular route objects to the route property called **routes**:

src/routes.js

```
...
import CatName from "./components/CatName";
import CatColor from "./components/CatColor";

export default [
  ...
  {
    name: "Cat",
    path: "/cat",
    component: Cat,
    routes: [
      {
        name: "Name",
        path: "/cat",
        exact: true,
        component: CatName
      },
      {
        name: "Color",
        path: "/cat/color/:color",
        exact: true,
        component: CatColor,
        linkPath: "/cat/color/orange"
      }
    ]
  }
];
```

There are two interesting things:

- One of the paths contains a query parameter placeholder (**:color**). You've already seen how we will obtain the value, inside CatColor.jsx
- The Color route got an additional property called **linkPath**. It will be used as the "to" value for the button that we will create in our upcoming sub menu.

Update the MenuLink component:

src/components/MenuLink.jsx

```
...
    <Link to={route.linkPath || route.path}>{route.name}</Link>
```

```
...
```

Next update our Cat component. We want to render a sub menu and, for lack of a better word, our "sub components":

src/components/Cat.jsx

```
import React from 'react';
import { renderRoutes } from "react-router-config";
import MenuLink from "./MenuLink";

export default ({ route }) => {
  return (
    <>
      <p>
        <img src="/static/kitten.jpg" alt="kitten" />
      </p>
      <div>
        {route.routes.map(route => (
          <MenuLink route={route} />
        ))}
      </div>
      {renderRoutes(route.routes)}
    </>
  )
}
```

Last but not least **remove the setTimeout function** from "src/index.js" so we get rid of the artificial delay.

Now refresh our POC in the browser and experience our completed application, **with server-side rendering**! Try to turn off JavaScript as well, as a little exercise. You will notice that our application is still rendering. You can even navigate between routes by using our menus, without JavaScript.

Source code
You can find the code from this chapter at gitlab.com in the branch part-6.