Bachelor's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Server-side rendering of React applications in enterprise portals

Václav Jančařík

Supervisor: Ing. Martin Ledvinka
Field of study: Software Engineering and Technology
May 2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Jančařík** Jméno: **Václav** Osobní číslo: **466301**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vykreslování React aplikací na straně serveru v enterprise portálech

Název bakalářské práce anglicky:

Server-side rendering of React applications in enterprise portals

Pokyny pro vypracování:

1. Analyze the current state of the art in the field of server-side rendering of React applications and running React applications in portal solutions.
2. Design fundamental principles of integration of server-side rendering of React applications in the context of portal environments.
3. Based on your design, implement a server-side rendering solution for React applications embedded in enterprise portals.
4. Demonstrate the correctness of your solution by comparing client-side and server-side rendering output of an example application.
5. Compare the performance of your server-side rendering solution with standard client-side rendering.

Seznam doporučené literatury:

[1] K. Konshin, Next.js Quick Start Guide: Server-side rendering done right, Packt Publishing, 2018
[2] R. Sezov, Liferay in Action: The Official Guide to Liferay Portal Development, Manning Publications, 2011
[3] R. Wieruch, The Road to learn React: Your journey to master plain yet pragmatic React.js, 2018

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **22.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **19.02.2021**

Ing. Martin Ledvinka podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would first like to thank Ing. Martin Ledvinka for great communication, support, and valuable feedback as the thesis supervisor. I am also grateful to Mgr. Jiří Kadlec and Ing. Tomáš Konrády for developing and maintaining the React Union project. I would also like to thank my friends and family for their continuous endurance and encouragement.

Declaration

I hereby declare that I have completed this thesis independently and that I have mentioned all the used information sources in accordance with the Guideline for compliance with ethical principles in the course of writing final theses.

Prague, 21 May 2019

. . . .

Václav Jančařík

Abstract

Single-page applications utilizing modern JavaScript libraries have many advantages regarding development and user experience. Integrating them into enterprise portals combines their respective benefits and is highly requested in large corporates. However, the content of single-page applications is usually generated in the browser, measurably affecting load speed and search engine optimization. Although these applications can be dynamically rendered on the server, no solution for enterprise portals was available.

One possible approach to this issue is piping the HTML output of the enterprise portal to an HTTP server with a JavaScript interpreter, which will handle the rendering. This thesis includes a functioning prototype, which is further extended with other useful features for load speed optimization.

Keywords: JavaScript, React, Node.js, Liferay, WordPress, enterprise portal, content management system, server-side rendering, search engine optimization

Supervisor: Ing. Martin Ledvinka
Prague, Resslova 9, E-116

Abstrakt

Single-page aplikace využívající moderní JavaScript knihovny mají spoustu výhod, co se vývoje a uživatelského prožitku týče. Jejich integrace do enterprise portálů kombinuje výhody obou technologií a je velmi žádaná ve velkých společnostech. Obsah single-page aplikací je však běžně generován až v prohlížeči, což má měřitelný dopad na rychlost načítání a optimalizaci pro vyhledávače. Ačkoliv mohou být tyto aplikace dynamicky vykreslovány na serveru, pro enterprise portály nebylo dostupné žádné řešení.

Jeden z možných způsobů, jak k tomuto problému přistoupit, je posílat HTML výstup enterprise portálu na HTTP server s JavaScript interpretem, který se o vykreslování postará. Součástí této práce je funkční prototyp, který je navíc rozšířen o další užitečné funkce pro optimalizaci rychlosti načítání.

Klíčová slova: JavaScript, React, Node.js, Liferay, WordPress, enterprise portál, systém pro správu obsahu, vykreslování na straně serveru, optimalizace pro vyhledávače

Překlad názvu: Vykreslování React aplikací na straně serveru v enterprise portálech

Contents

1 Rendering of web pages 1
1.1 Client-side rendering drawbacks . 1
1.1.1 Load speed 1
1.1.2 Search engine optimization . . . 2
1.2 Server-side rendering 2
1.2.1 Possible approaches 3
1.2.2 Environment restrictions 3
1.2.3 Optimization opportunities . . . 4
1.3 Static site generators 5
1.4 Objective of the thesis 6

2 React and enterprise portals 7
2.1 Widget-based design 7
2.2 Introduction to React 8
2.3 React Union project 9
2.4 Other means of integration 10

3 Architecture 11
3.1 Node.js solution analysis 11
3.2 High-level overview 12
3.3 Rendering process 12

4 Implementation 15
4.1 Node.js rendering service 15
4.2 Refactoring of existing packages 17
4.2.1 `react-union` 17
4.2.2 `react-union-scripts` 18
4.2.3 `babel-preset-react-union` 18
4.3 Example applications 18

5 Load speed improvements 21
5.1 Measurement setup 21
5.2 JavaScript bundle size 22
5.3 Back-end latency 22
5.4 HTML payload size 23
5.5 Real-world impact 23

6 Conclusion 25

A Guide to local deployment 27

B Contents of the enclosed CD 29

C References 31

D Abbreviations 35

Figures

3.1 Example SSR architecture 13
3.2 Rendering process UML diagram 14

Listings

2.1 React application example 9
2.2 Sample enterprise portal output 10
4.1 Rendering service signatures . . . 16
4.2 Rendering process pseudocode . . 16

Tables

5.1 JavaScript bundle sizes 22
5.2 Medians of back-end latencies . . 22
5.3 HTML payload sizes 23
5.4 Mobile network speeds 23
5.5 Load speed improvements with SSR enabled 24

Chapter 1

Rendering of web pages

Historically, most dynamic web pages have had their content generated on the server, utilizing scripting languages like PHP. With JavaScript (JS), the possibilities of interactivity can be nearly endless, but complex applications quickly become unmaintainable and cumbersome to build upon – that is, unless a single-page application (SPA) framework or library is utilized. When these libraries (e.g. React) came along, the complexity of the applications could be much higher. However, SPAs usually come with a major drawback: their content is generated in the browser, not on the server.

1.1 Client-side rendering drawbacks

Generally, SPAs work in the following way:

1. User opens a web page.
2. User agent[1] downloads an empty page and some JS source code.
3. JS source code is executed and the application is rendered.

It is apparent that there is a delay between opening the web page and rendering it. This delayed loading of content poses challenges in various areas of web development.

1.1.1 Load speed

Because the browser has to download the JS source code before anything can be rendered, the impact on First Contentful Paint (and other related metrics) can be quite extreme, especially with a slow internet connection.

> First Contentful Paint (FCP) measures the time from navigation to the time when the browser renders the first bit of content from the DOM [1].

[1] Software acting on behalf of the user, e.g. a web browser.

Although there is room for improvement in the form of caching and code splitting[2], the load speed never comes close to that of a purely server-rendered application, as some JS is still necessary for the initial render.

1.1.2 Search engine optimization

While some web crawlers, such as Google's Googlebot [2], are able to crawl and index client-rendered applications as well, it requires way more resources and is therefore deferred. Google uses two waves of indexing to process these kinds of applications:

1. In the first wave, HTML and CSS will be crawled and indexed, almost instantly.
2. In the second wave, Google will come back to render and index JavaScript-generated data, which can take from a few hours to over a week [3].

Other web crawlers cannot handle client-rendered applications at all. In contrast, server-rendered applications can be indexed by Google immediately and can be easily processed by other search engines as well. To solve this problem, Google recommends using dynamic rendering [4].

Dynamic rendering (or prerendering) is the principle of sending client-rendered content to users and server-rendered content to search engines and other crawlers [2, 4]. Instead of serving an empty web page to the crawler, the content is prerendered by a headless browser[3] and cached.

Because the web page output is simulated in a full-featured browser, dynamic rendering is applicable to all types of SPAs, regardless of the underlying framework or system. As a result, it is possible to delegate dynamic rendering to hosting providers – this is directly supported e.g. by Netlify [5].

1.2 Server-side rendering

Server-side rendering (SSR) is quite similar to dynamic rendering, but slightly harder to implement. Instead of server-rendering the web page for crawlers only, it is rendered on the server per each request. Client-side JS can then hook into the server-rendered application, attaching event listeners to the existing markup – "hydrate" it. SSR is more expensive than dynamic rendering, but ultimately results in noticeable load speed improvements, especially when using a slower network connection.

It is important to mention that the concept of rendering applications on the server is nothing new, as libraries like React or Angular were developed with SSR in mind. Today, there are many platforms which support SSR, such as Next.js or Electrode [6, 7, 8]. However, because these platforms are highly opinionated in terms of structure and how they serve content, they are unsuitable for integration into other systems.

[2]Splitting the JS bundle into smaller chunks, which can be loaded on demand.
[3]A web browser without a graphical user interface.

1.2.1 Possible approaches

There are essentially two feasible approaches to rendering SPAs on the server:

- Rendering the application using a JS runtime or a virtual machine (e.g. Node.js or GraalVM). For example, React comes with a `renderToString` function, making it possible to render the entire application as a string and inject it into the web page which would otherwise be sent empty.
- Using a headless browser (e.g. Google Chrome or Mozilla Firefox), it is possible to open the requested web page on the server, wait for the content to load, and then send the final HTML to the client.

Both have their advantages and disadvantages, but the JS runtime approach is more customizable in terms of load speed optimization and should technically be more performant as well[4]. The headless browser approach is thus often used as a universal solution for dynamic rendering, but not so much for reducing load times.

1.2.2 Environment restrictions

When developing a server-rendered application, especially one which will not be rendered in a headless browser, there are some technical challenges and restrictions to keep in mind.

Browser API

Because the JS source code is not be evaluated in a standard browser environment, global variables such as `window` or `document` are not available. There are multiple ways of handling this:

- In many applications, it is not necessary to use these variables at all, meaning that this issue does not have to be dealt with.
- Any use of these variables can be wrapped in a conditional statement to prevent runtime errors.
- Defining the global variables as NOOP stubs (e.g. `global.window = {}`).
- Emulating a subset of the browser environment. Although this is possible with the use of some libraries, there are no clear reasons for doing so. Accessing these variables is usually necessary because of event listeners and/or very specific DOM manipulation (such as scrolling behaviour), none of which need to be addressed during SSR.

Conditional statements and NOOP stubs are clearly the way to go. The decision can be left to the developer of the server-rendered application, because these approaches require no additional tooling setup.

[4] Headless browsers come with overhead in the form of unnecessary browser features.

Singleton pattern

The singleton pattern is a software design pattern that restricts the possibility of having more instances of specific objects while providing a global point of access to them [9]. In the JS ecosystem, it is often used to avoid dependency injection of objects that are only present once throughout the application (likely because dependency injection is not natively supported). However, the use of this pattern is highly discouraged, mainly because of the following reasons:

- It introduces global state to the application.
- Testability. Although this issue can be solved by various mocking libraries, the use of this pattern is often not necessary and thus the tests become unnecessarily complex.
- Server-side rendering. Because there is only one instance of the object throughout the entire application, this instance would be reused for each request, resulting in data leakage across sessions and possibly serious security vulnerabilities.

In this case, we are concerned mainly with SSR. Although it is possible to achieve total isolation by creating a new execution environment (i.e. virtual machine) per each request, this often comes with a performance cost, so it is better to avoid the singleton pattern altogether.

1.2.3 Optimization opportunities

Executing JS code on the server comes with interesting opportunities for further load speed improvements.

Server-side prefetching of data

Because the application is often rendered in the same physical location as its data source, it is possible to further optimize the delivery of content by enabling server-side prefetching of data. This means that instead of rendering e.g. an empty list and then sequentially fetching the necessary data, the list can be filled on the server.

Furthermore, server-side rendering is often executed synchronously, as is the case with React's `renderToString` function. In contrast, AJAX requests are inherently asynchronous (hence *Asynchronous* JavaScript and XML). Because of this, the fetching must occur before rendering the application[5].

Wave reduction

This concept may not be as obvious and simple to grasp, but can nevertheless improve web page load speed significantly. Most large JS applications utilize

[5] Asynchronous calls are deferred, not affecting the synchronously rendered application.

some form of code splitting, which needs to be handled both when bundling the JS modules (e.g. by Webpack or Parcel.js) and when executing the resulting source code.

- When bundling the modules, whenever a dynamic `import()` call is encountered, a new chunk (a separate JS file) may be created[6].
- When executing the source code, whenever a dynamic `import()` call is encountered, the appropriate chunk will be fetched (if not already available). Note that `import()` returns a `Promise`, meaning that any dependent code will be run asynchronously.

The necessary chunks are thus fetched when they are actually needed. The asynchronous fetching of chunks is handled by a runtime injected into the entry chunk[7] by the bundler. For clarity, let us assume a scenario of three JS files/chunks:

- `entry.js` containing `await import("./foo.js")`
- `foo.js` containing `await import("./bar.js")`
- `bar.js` containing `console.log("Hello from bar!")`

With appropriate bundler configuration, using `entry.js` as a bundle entry results in three separate chunks: `app.js`, `app~foo.js`, and `app~bar.js`. If each chunk takes one second to fetch (including the application entry), "Hello from bar!" will be logged after a total of three seconds.

This is not ideal, because the chunks are fetched "in waves". Technically, it should be possible to serve all the necessary chunks immediately, because the application has already been rendered on the server. This can be done by "marking" the chunks when they are used on the server, i.e. during the rendering. Afterwards, the marked chunks can be included in the HTML that will be sent to the client. This techinque is called "wave reduction", term coined by Anton Korzunov [10].

Because it is necessary to somehow mark the used server-side chunks and map them to the client-side ones, wave reduction requires a very advanced tooling setup to pull off.

1.3 Static site generators

Using a JS-based static site generator, such as GatsbyJS, is a viable alternative to SSR [11]. Instead of rendering the web page on the server per each request, static site generators render the entire website once – at compile time. This website can then be hosted without the need of a dedicated Node.js server, tackling the same challenges as SSR, but more efficiently.

[6] This depends on the bundler configuration. Serving too many chunks may actually result in a performance drop due to too many requests and runtime overhead.

[7] The runtime can also be located in a separate file for long-term caching.

However, static site generators only make sense for serving data that does not depend on user context. Any user-specific data cannot possibly be rendered at compile time, making SSR the only way to provide load speed improvements for JS applications which serve data based on the currently authenticated user.

1.4 Objective of the thesis

The objective of this thesis is to design and implement a prototype for SSR of React applications embedded in enterprise portals and content management systems (CMSs). This prototype shall serve as a basis for measuring and evaluating the real-world benefits of SSR.

Although it is possible to render traditional SPAs on the server, the existing tools cannot be applied in the context of the aforementioned CMS-like systems, as explained in section 1.2. Moreover, SSR can be implemented using different SPA libraries and frameworks, each of them coming with their own set of challenges and supported features. In this thesis, React is selected as the main focus, primarily because of its popularity and simplicity.

While being specific to React, the solution should not be coupled to any enterprise portal or CMS platform. Instead, it should be possible to swap the underlying CMS-like system without any major refactoring. The following chapter will explore the possibilities of integrating React applications into enterprise portals in more detail.

Chapter 2

React and enterprise portals

Enterprise portals and other CMS-like systems are attractive to large companies for various reasons. The stakeholders are able to be in control of what is shown on the website more directly, and these systems usually contain useful built-in features (such as user management), making them beneficial for cutting down costs as well. In contrast, React applications (and other SPAs) have benefits from the opposite side of the spectrum, such as the increased interactivity and the ability to cater to almost any business requirement related to user experience.

2.1 Widget-based design

What are the possibilities of combining React and CMS-like systems? The first option is to use a so-called headless CMS, which would serve as a back-end only content database[1]. This content can usually be accessed in the React application via a REST (or GraphQL) API. With this approach, however, we would give up the ability to manage the layout of the web pages through the CMS UI, which is often one of the main reasons for adopting a CMS or an enterprise portal.

Instead, in order to take advantage of as many features as we can, the React application could be split into separate widgets. The content manager can then create a web page, decide on the layout, and then drag and drop the widgets to where they should actually be rendered. The widget-based design needs to support all of the following features to be comparable to e.g. Liferay's portlets[2] or widgets in WordPress.

Dynamic combinations of widgets. The layout of the application should not be in the hands of developers. In other words, we do not know which widgets are actually going to be rendered, or in what combination. This means that the widgets should be isolated and should not depend on one another. Furthermore, the browser should not download any unnecessary JS code, because the entire application may consist of tens of widgets, with

[1] In essence, it means that the data layer is decoupled from the actual website, allowing multiple front-end applications to share the same back-end service (content database).

[2] In Liferay, a portlet is a web application that runs in a portion of the web page [12, 13].

only a small portion of them being rendered for a particular web page (code splitting *must* be implemented).

Sharing data between widgets. Although the widgets should keep their data isolated, they should be able to communicate with one another (if necessary). Assume two widgets: a map of branches and an onboarding form. When these two widgets are present on the same web page, it should be possible to preselect a branch in the onboarding form when the branch is selected in the map.

Custom configuration of widgets. Content managers should be able to configure the behaviour and appearance of widgets through the enterprise portal UI. Depending on the business requirements, it should be possible to change the localization, default form values, or any other variable, just like in widgets native to the platform.

Multiple instances of the same widget. When using the Redux library for application state management, it is common practice to only use a single store for the entire application [14]. When there are multiple instances of the same widget in a single web page, e.g. of an image gallery, the data location within the store cannot be defined statically. Instead, each widget should have a unique namespace and should store data accordingly. In practice, this means using `state.galleries[namespace]` instead of `state.galleries.food`, because the content manager can decide to have two food galleries in a single page (e.g. `food-top` and `food-bottom`). Ignoring this would result in the two galleries being "linked", causing unexpected behaviour.

2.2 Introduction to React

React is a JavaScript library for building component-based UIs using a declarative API. React is often used with JSX, an XML-like syntax extension to JS. In this minimal example (listing 2.1), the name Taylor is being passed as a prop[3] to the `HelloMessage` component. When `ReactDOM.render` is called, the entire application is rendered into an HTML element, displaying "Hello Taylor". This example would be very easy to replicate without the need of any libraries, but as the application grows in terms of complexity, React quickly becomes indispensable [15, 16].

Should the `name` prop change, React will *automatically* update the DOM[4] with the correct content. In this example, the prop is defined statically (and thus will not change), but the components can have their own encapsulated state and pass props down to other components, meaning that React needs to keep track of which components to update.

React does so by keeping a virtual representation of the UI in memory [17], synchronizing it with the real DOM when necessary – this process is called

[3] Short for property. It is essentially a function parameter to the component.
[4] Document Object Model, an API allowing programmatic access to HTML documents.

Listing 2.1: React application example

```
const HelloMessage = props => (
  <div>Hello {props.name}</div>
)

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  document.getElementById("root")
)
```

reconciliation [18]. Because accessing the DOM is slow, React is able to calculate the necessary updates to the component tree much faster, resulting in huge performance benefits. Another benefit of this approach is that React can be used even in non-browser environments, so a React application can be rendered as a native mobile application or even as a string.

Despite the fact that the DOM is a tree structure, React provides a useful feature: portals. They allow applications to render content outside their DOM scope; their intended use is described in the React documentation.

> A typical use case for portals is when a parent component has an `overflow: hidden` or `z-index` style, but you need the child to visually "break" out of its container. For example, dialogs, hovercards, and tooltips [19].

Although portals were designed to solve a completely different use case, we can utilize this behaviour to render the widgets anywhere in the document while keeping them in a single virtual DOM. This allows us to treat the widgets as a single React application, massively simplifying development.

2.3 React Union project

The React Union project is an open-source collection of libraries and tools for integrating React applications into various back-end systems [20]. The core mechanism is simple: instead of rendering the React widgets directly, the CMS-like system outputs HTML elements called *widget descriptors*, describing the widgets to render. The React application then scans the entire document and renders the widgets to their respective containers using React portals.

Listing 2.2 contains a sample HTML document generated by an enterprise portal. When the `app.js` script is run, the library[5] will scan the document, resulting in a single widget descriptor being found. This descriptor says that a navigation widget shall be rendered into a DOM element with an ID of `root`. Widget descriptors may optionally contain a JSON object that can be used to provide configuration data to the widget.

[5]To be more precise, the scanning will be initiated by the `Union` React component.

Listing 2.2: Sample enterprise portal output

```
<!doctype html>
<title>Enterprise portal</title>
<script src="app.js" defer></script>

<div id="root"></div>

<script data-union-widget="nav" data-union-container="root">
</script>
```

Because of a clearly defined, platform-agnostic interface, React Union is not tied to any CMS or enterprise portal platform, making it easy to use with WordPress, Liferay, Drupal, and many other solutions.

When developing the React application, there is no need to have the underlying enterprise portal running, as it is possible to use a static HTML file as the source of widget descriptors. Moreover, when it is necessary to debug the React application in the context of an enterprise portal, it is possible to use a proxy server that serves the React application *on top* of the enterprise portal. This allows developers to utilize hot replacement of JS modules for rapid development.

React Union comes with all of these features preconfigured in the form of React Union Scripts, a software development kit similar to Create React App's `react-scripts` package [21]. Furthermore, the React Union repository includes multiple example projects that are completely set up with all the necessary tooling and are ready to be extended with custom functionality [20].

2.4 Other means of integration

Besides React Union, no other platform-agnostic solution for integrating React applications into CMS-like systems has been publicly available. This might be because it is possible to use a headless CMS instead of a standard one, but the drawbacks of this approach were mentioned at the beginning of chapter 2.

Although some CMS-like systems claim that they support React applications in their documentation, most of them allow only the classic, "non-widget" approach, meaning that some of the desired features (such as having dynamic combinations of widgets) cannot be properly implemented.

Liferay is an exception to this with its `liferay-npm-bundler`, which allows us to have a JS application in a portlet [12, 22]. While this bundler does support code splitting, keep in mind that due to the nature of the JS ecosystem, "regular" bundlers like Webpack[6] are always going to be more advanced in terms of features and optimization. Unfortunately, Liferay's solution is not transferrable to other platforms and does not support SSR at all.

[6] Webpack is utilized extensively in the React Union project.

Chapter 3

Architecture

Given the circumstances explained in section 2.4, extending React Union is presumably the most logical way of implementing SSR of React applications in the context of enterprise portals. As mentioned in subsection 1.2.1, we want to use a JS runtime or a virtual machine instead of a headless browser, mainly because of the added flexibility and performance.

There are quite a few options to choose from, but not all of them are suitable for us. To keep the core React Union source code JS only, JVM-based software[1], such as Nashorn or GraalVM, is not ideal when compared to alternatives like Node.js.

3.1 Node.js solution analysis

Node.js is a JavaScript runtime built on Chrome's V8 JS engine [23]. We can use it to create an HTTP server which can execute JS source code, i.e. render a React application. This means that the rendering server would essentially be an "SSR as a service" Node.js microservice.

This is very similar to Airbnb's Hypernova [24], but due to the nature of React Union (scanning the document for widget descriptors), it is not possible to use Hypernova for this purpose. Furthermore, Hypernova does not support wave reduction and implementing it would likely prove to be difficult.

Because any two widgets should technically be able to communicate with one another, all the widgets must be rendered in a single phase (using a single call to `ReactDOMServer.renderToString`). In order to allow seamless integration with any CMS or enterprise portal, it is best to pipe all the HTML output of the CMS-like system to the rendering service, which will respond with the HTML to send to the client. On most platforms, this is easier to implement[2] than aggregating the rendered widget descriptors.

At first glance, piping all the HTML to a Node.js server might not look like a great idea performance-wise. There are two points where a bottleneck might occur: I/O and scanning the HTML for widget descriptors.

[1] Software which can execute JS source code using the Java virtual machine.
[2] Possible e.g. via output buffering or servlet filters (WordPress and Liferay, respectively).

- As measured by Airbnb, I/O load is negligible compared to the actual rendering of the application [25].
- There are multiple ways to scan an HTML document for widget descriptors. While implementing a subset of the DOM might turn out to be a bottleneck, using simple string manipulation would suffice.

This means that the most demanding part of the rendering process should be the rendering itself. However, because there is no way to avoid rendering the application altogether, piping all the HTML to a Node.js rendering service should be a suitable solution.

3.2 High-level overview

To summarize what we know so far in terms of requirements and necessary features, here are the main points:

- A Node.js HTTP server acting as a rendering microservice shall be implemented.
- In order to offload as much implementation logic to this tool as possible, the Node.js server shall accept the entire HTML output of an enterprise portal as its input.
- The new HTML (containing the rendered React application and used JS chunks) shall be sent back to the enterprise portal, replacing the original HTML as the client-facing output.

A simplified version of a possible underlying infrastructure is shown in figure 3.1, with server-side prefetching of data taking place during the rendering process (communication with the REST API).

Because this is a method of progressive enhancement, if anything related to SSR goes wrong, the application must still function properly. The enterprise portal should find out if the rendering service is running via a health check API endpoint. If the service returns a bad status code, the enterprise portal should fall back to relying on client-side rendering (CSR).

3.3 Rendering process

A couple of points have been metioned so far in terms of rendering:

- The HTML document shall be scanned for widget descriptors *before* rendering the React application.
- After the document is scanned, server-side prefetching of data shall occur, based on the widgets that are going to be rendered.

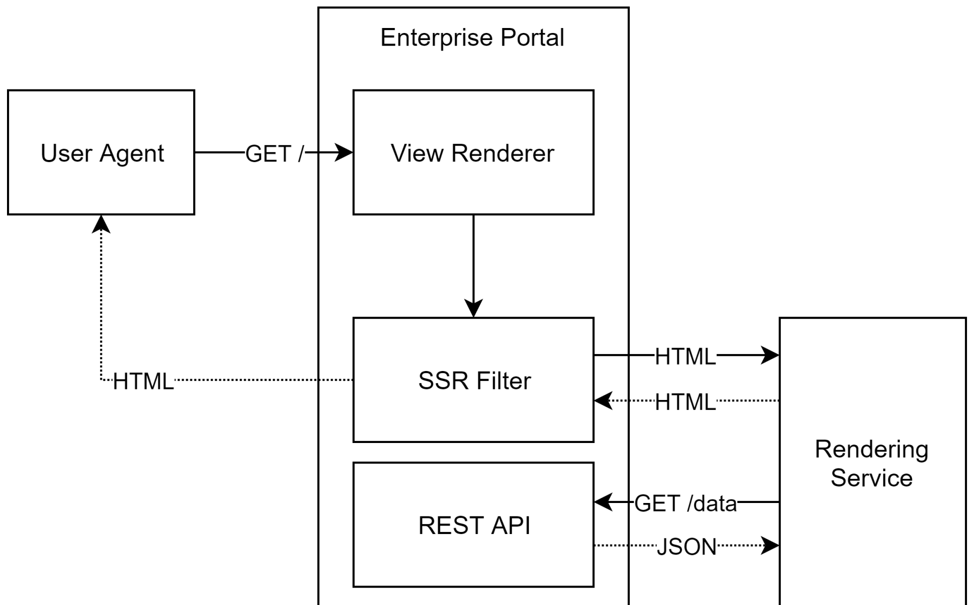


Figure 3.1: Example SSR architecture

- Because of React portals, we must pay close attention to where the widgets should actually be rendered.
- The used chunks shall be appended to the HTML as script elements to achieve wave reduction.

The rendering flow is thus depicted as a UML diagram in figure 3.2, taking the aforementioned points into consideration.

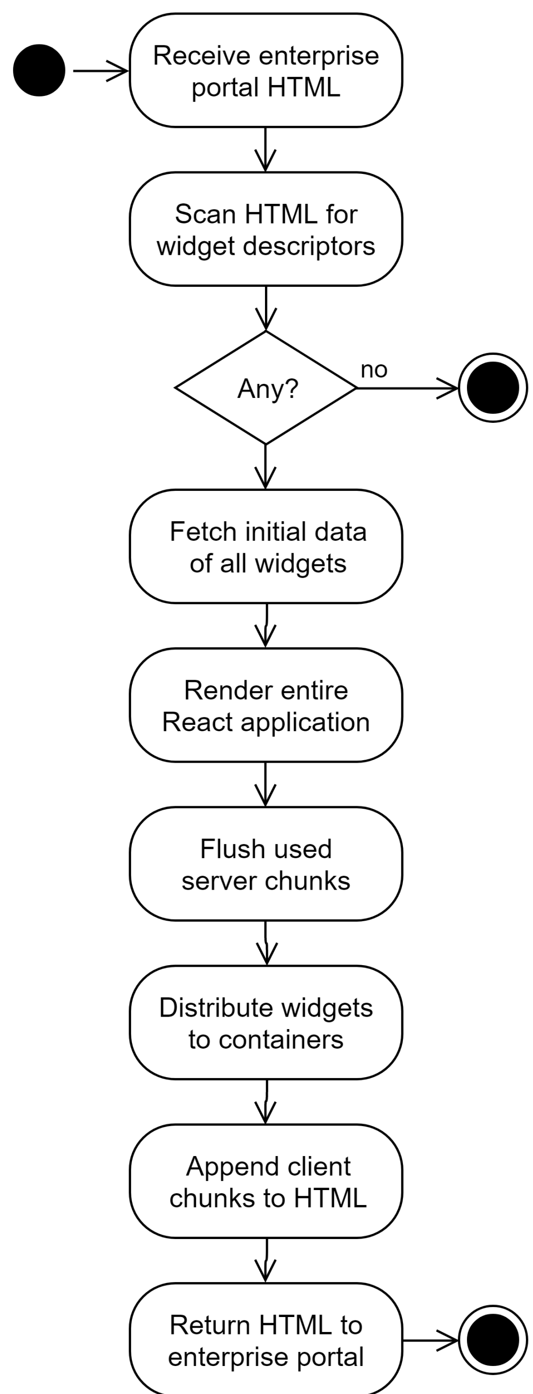


Figure 3.2: Rendering process UML diagram

Chapter 4
Implementation

The implementation of the proposed architecture consists of three parts:

1. The Node.js rendering service.
2. Refactoring of the existing React Union packages.
3. Example applications with out-of-the-box SSR support.

The source code of all parts is available in the `lundegaard/react-union` repository on GitHub [20]. A collection of JS libraries by James Gillmore has been extensively used in order to achieve simultaneous code splitting, SSR, and wave reduction [26, 27, 28, 29].

4.1 Node.js rendering service

The Node.js rendering service was implemented as a standalone JS package[1], exposing a single function: `startRenderingService`. The function signature is described using Flow syntax in listing 4.1, which should be quite simple to grasp with basic knowledge of JavaScript [30].

This function accepts a handler to be invoked for each request. In this handler, the `render` function must be called at least once, rendering the entire React application into the HTML context of the request. This HTML can also be accessed and manipulated using the `head` and `body` Cheerio wrappers[2]. This design allows developers to have full control over the rendering process, with two-phase rendering[3] and other customization being possible as well. To better show how each request is actually handled, a pseudocode sample is included in listing 4.2.

The `startRenderingService` function also accepts an `options` object, which can be used for enabling/disabling some features, as well as other configuration (e.g. setting the TCP port of the server). It is also possible to

[1] Available in the `npm` repository as `react-union-rendering-service`.

[2] Cheerio is a server-side implementation of jQuery [31].

[3] Rendering the application once, walking the component tree, and then rendering it again. This is necessary for libraries such as React Apollo [32].

Listing 4.1: Rendering service signatures

```
type RenderResult = {
  widgetConfigs: Object[],
  chunkNames: string[],
  initialProps: { [namespace: string]: Object }
}

type Render =
  (rootElement: React.Element, routes: Array) => RenderResult

type HandleRequest = ({
  render: Render,
  head: CheerioWrapper,
  body: CheerioWrapper,
  req: http.ServerRequest,
  res: http.ServerResponse
}) => RenderResult

type StartRenderingService =
  (handleRequest: HandleRequest, options?: Object) => void
```

Listing 4.2: Rendering process pseudocode

```
function renderingMiddleware(handleRequest, req, res) {
  originalHTML = cheerio.parse(req.body)
  head = originalHTML.head
  body = originalHTML.body

  function render(reactElement, routes) {
    widgets = scan(originalHTML, routes)
    data = widgets.map(prefetchData)
    reactHTML = renderToString(withData(data, reactElement))
    html = distributeWidgets(originalHTML, reactHTML)
    return html
  }

  context = { render, head, body, req, res }
  html = await handleRequest(context)
  assert(render).calledTimes(1)
  chunks = flushChunks()
  html.append(chunks)
  return html
}
```

configure the rendering service via a `union.config.js` file in the application root directory.

To improve developer experience as much as possible, the rendering service is polymorphic: it can also run as a middleware. When developing React applications, it is common to use the `webpack-dev-middleware` library [33], which intercepts requests to compiled JS files in the file system and instead serves them from memory, all the while watching for any source code changes. Additionally, `webpack-hot-server-middleware` can be used to hook a server-rendering middleware to these changes [34]. This complicated setup allows React Union applications to be rendered on the server even during development, without the need to manually restart the Node.js development server.

The nature of microservices allows the React Union rendering service to be easily scaled horizontally, utilizing off-the-shelf solutions such as Docker, Kubernetes, PM2, Nginx, and HAProxy. The service performance (i.e. latency) can be monitored using traditional distributed tracing tools.

4.2 Refactoring of existing packages

There were a total of three packages which needed refactoring: `react-union`, `react-union-scripts`, and `babel-preset-react-union`.

4.2.1 `react-union`

Previously, the `Union` component was responsible for the scanning. Because it is necessary to scan the widget descriptors before rendering on the server, the component now also accepts external scan results. Furthermore, in order to support scanning a string instead of a DOM element, support for Cheerio wrappers was added.

Unfortunately, React does not support SSR of portals at all. This issue had to be solved on both sides of the rendering process.

On the server. Instead of using portals directly, we can render all the widgets into a single parent container, wrapping each widget in an HTML element with a data attribute specifying where the widget should have been rendered. After the entire application is rendered, we can parse the HTML output and emulate the behaviour of React portals, although with a slight performance drop.

On the client. Unfortunately, the hydration of portals is not as straightforward as the rendering. Currently, instead of hooking into the existing markup, React will duplicate it. This can be solved by clearing the respective HTML containers prior to rendering the widgets and disabling any stateful features of the application, such as input fields, to prevent unexpected behaviour when the application is hydrated.

4.2.2 `react-union-scripts`

The Webpack configuration had to be altered to allow building server bundles [35]. Furthermore, in order to enable wave reduction, the server bundle and the client bundle must be compiled at the same time[4]. As a result, the build process was completely revamped, taking advantage of Webpack being able to reuse modules and chunks during compilation.

React Union Scripts support customization via a `union.config.js` file. This file now also accepts several options related to the Node.js rendering service under the `renderingService` property.

4.2.3 `babel-preset-react-union`

React Union Scripts use Babel to transpile modern JS syntax for old browsers. Babel is configurable via presets and this is the one used in the React Union project. Because the selected collection of libraries contains a Babel plugin for passing contextual metadata to imported components, this plugin was added to the preset for convenience [29].

4.3 Example applications

Two SSR-ready projects were added to the React Union repository: `ssr-basic` and `liferay-ssr`. They are heavily based on the existing boilerplate projects[5]. Let us take a closer look at their structure.

A standard React Union project consists of multiple widgets and one or more applications, allowing developers to target multiple platforms and share source code across applications within a project. Every application should include the following files:

- `routes.js` defining the mapping of widgets (React components) and widget descriptors.
- `index.js` being the client-side entry point of the entire application (akin to the `main` method in Java). Somewhere in this file, `ReactDOM.render` should be called.
- `index.ssr.js` optionally being the server-side entry point of the application. It must export the return value of calling `startRenderingService`.

Each example project contains a single application and two simple widgets, rendering only some basic text content or an image.

In the `liferay-ssr` project, an SSR servlet filter was implemented for integration with the Node.js rendering service. This filter pipes the Liferay HTML output to the rendering service, replacing the client-side output with

[4]Wave reduction depends on marking used server chunks and then serving the appropriate client chunks, meaning that simultaneous access to both is needed at compile time.

[5]Preconfigured applications serving as a source code base for custom functionality.

the service response. If the rendering service is not available, Liferay falls back to CSR, periodically attempting to re-enable SSR by calling the health check API of the service. The default interval has been set at 5 minutes.

The new boilerplates are completely set up with tooling and are ready to be extended with custom functionality. Appendix A contains a guide on how to build these projects and deploy them in a local environment.

Chapter 5

Load speed improvements

As explained in section 1.2, the main reason for adopting SSR should be improvements in load speed. This chapter attempts to measure the real-world performance impact, utilizing the implemented prototype. The following application parameters must be taken into account:

- JavaScript bundle size.
- Back-end latency.
- HTML payload size.

Because the React application needs to be rendered on the server before being sent to the client, the back-end latency is expected to slightly increase. The size of the initial HTML payload is expected to increase as well, but the amount depends on the tested application. The size of the JavaScript bundle is not affected by the rendering method.

5.1 Measurement setup

To measure the impact on the individual parameters as accurately as possible, the analysis was performed on the following existing projects:

- Project A, a small project containing 1 application and 2 simple widgets.
- Project B, a medium-sized project containing approximately 25 000 lines of functional JS code, 2 applications, 27 internal libraries, and 19 widgets of varying complexity, mostly forms and tables. To simulate a real application configuration, two combinations of widgets have been used, containing 2 widgets and 5 widgets, respectively.

The applications were compiled using `react-union-scripts` and processed by a Node.js script[1] ensuring that the Liferay portal can load them on demand. The resulting bundles were deployed to an instance of Liferay Portal 7.0 CE GA7 [36].

[1] AMD loader tools are available in all Liferay boilerplates in the React Union repository [20].

The back-end latency was measured using Apache JMeter™ [37, 38], the size of the HTML payload was inspected using Chrome DevTools [39].

5.2 JavaScript bundle size

The JS bundle size is essentially the main cause of the delayed rendering of web pages. The data in table 5.1 is the result of a quick inspection using an appropriate file explorer.

	Project A 2 widgets	Project B 2 widgets	Project B 5 widgets
Application entry	10 kB	101 kB	101 kB
Webpack runtime	2 kB	3 kB	3 kB
Libraries	236 kB	700 kB	700 kB
Widget chunks	3 kB	181 kB	229 kB
Sum	**251 kB**	**985 kB**	**1 033 kB**

Table 5.1: JavaScript bundle sizes

5.3 Back-end latency

The latency was measured by sending 500 sequential GET requests to the Liferay portal, which in turn calls the Node.js rendering service if SSR is enabled. Because the rendering service itself is single-threaded and the `renderToString` function is synchronous, a real-world setup would consist of running multiple services behind a load balancer. However, with the requests being sent sequentially, running a single instance of the rendering service does not affect the latency.

As shown in table 5.2, enabling SSR does indeed have a slight impact on back-end latency when compared to CSR. On average, the latency medians have increased by approximately 30 ms.

	Project A 2 widgets	Project B 2 widgets	Project B 5 widgets
CSR latency	28 ms	18 ms	17 ms
SSR latency	54 ms	48 ms	56 ms
Difference	**26 ms**	**30 ms**	**39 ms**

Table 5.2: Medians of back-end latencies

5.4 HTML payload size

To measure the difference in HTML payload sizes, there is no need to have the React applications embedded in Liferay. Instead, it is faster and simpler to use the development server of `react-union-scripts`. As seen in table 5.3, the increase in payload size is negligible, possibly due to the nature of the selected projects. To put the values into perspective, the initial HTML payload of Wikipedia's home page was measured to be approximately 20 kB.

	Project A 2 widgets	Project B 2 widgets	Project B 5 widgets
CSR size	1.2 kB	3.5 kB	4.6 kB
SSR size	4.2 kB	7.7 kB	19.2 kB
Difference	**3.0 kB**	**4.2 kB**	**14.6 kB**

Table 5.3: HTML payload sizes

5.5 Real-world impact

The collected data allows us to estimate the impact of SSR on different kinds of networks. As a guildeline, table 5.4 lists the typical real-world mobile network speeds based on their type [40].

Network type	Typical real-world download speed [40]
3G	375 kBps
3G HSPA+	750 kBps
4G LTE	2 500 kBps
4G LTE-advanced	5 250 kBps

Table 5.4: Mobile network speeds

Table 5.5 lists the improvements in First Contentful Paint (FCP) when the projects use SSR [1]. The following equation was used to calculate the values:

$$\Delta FCP = \frac{JS - \Delta HTML}{download\ speed} - (\Delta latency \times 1000),$$

where ΔFCP is the reduced First Contentful Paint in seconds, JS is the JS bundle size in kilobytes, $\Delta HTML$ is the increased HTML payload in kilobytes, *download speed* is the download speed in kilobytes per second, and $\Delta latency$ is the increased back-end latency in milliseconds.

It is clear that SSR has a great impact on slower networks, improving load speed by up to 2.68 seconds in the worst case scenario. On the same network, the total FCP of the respective Liferay page with SSR disabled is approximately 11 seconds. In this case, SSR theoretically reduces FCP by

about a quarter. It should be noted that the bundle size of Project B has been optimized and reduced as much as the available tools allow, including manual inspection of transitive dependencies. This means that in larger, less optimized projects, the impact will be even more apparent.

	Project A 2 widgets	**Project B 2 widgets**	**Project B 5 widgets**
3G	0.64 s	2.59 s	2.68 s
3G HSPA+	0.30 s	1.28 s	1.32 s
4G LTE	0.07 s	0.36 s	0.37 s
4G LTE-advanced	0.02 s	0.16 s	0.15 s

Table 5.5: Load speed improvements with SSR enabled

On faster networks, the JS bundle size becomes less significant, as does the difference in HTML payload size, making the increased back-end latency progressively more important. However, as long as the latency increase stays in the range of tens of milliseconds, this should not become a problem.

The downside of SSR is the increased Time to Interactive (TTI), i.e. the delayed attaching of JS event handlers [41]. The following equation can be used to calculate the theoretical increase in TTI:

$$\Delta TTI = \frac{\Delta HTML}{download\, speed} + (\Delta latency \times 1000),$$

where ΔTTI is the increased Time to Interactive in seconds, $\Delta HTML$ is the increased HTML payload in kilobytes, $download\, speed$ is the download speed in kilobytes per second, and $\Delta latency$ is the increased back-end latency in milliseconds. On all the selected networks, ΔTTI is negligible.

Chapter 6

Conclusion

Server-side rendering is currently one of the most troublesome aspects of web development, mainly because there is no easy unopinionated solution. Nevertheless, there are many great libraries which aid in this area, making SSR possible to implement even in more complex systems. This is further demonstrated by the included prototype, proving that SSR does indeed have a positive impact on performance, as measured in chapter 5. The objective of this thesis was thus accomplished.

The load speed implications of SSR can be noticeable even on fast mobile networks. In practice, however, the difference depends on the type of the application, making it difficult to decide if SSR is actually worth the hassle. That being said, the React Union repository now contains a preconfigured, SSR-enabled boilerplate project with Liferay integration, which can serve as a basis for custom functionality [20].

Appendix A

Guide to local deployment

To run the applications locally, the following software must be installed and/or running on your device.

- Node.js, version 10.
- Yarn, version 1.
- Liferay CE, version 7.0 GA7 (only necessary for a production build).
- Gradle, version 5 (only necessary for a production build).

Installation instructions for all the abovementioned software are available online [23, 42, 36, 43].

Development

This section describes how to run the applications for development, without the need of deployment to a Liferay portal instance.

1. Navigate to the React Union root directory via the command line.
2. Run `yarn` to install the dependencies. You may skip this step if all the dependencies are already installed.
3. Navigate to the appropriate project directory. For example, by running `cd boilerplates/react-union-boilerplate-ssr-basic`.
4. Run `yarn start`. A browser tab will open shortly.

Production build

This section describes how to build the applications for production and deploy them to a local Liferay portal instance. This guide only applies to boilerplate projects with Liferay integration.

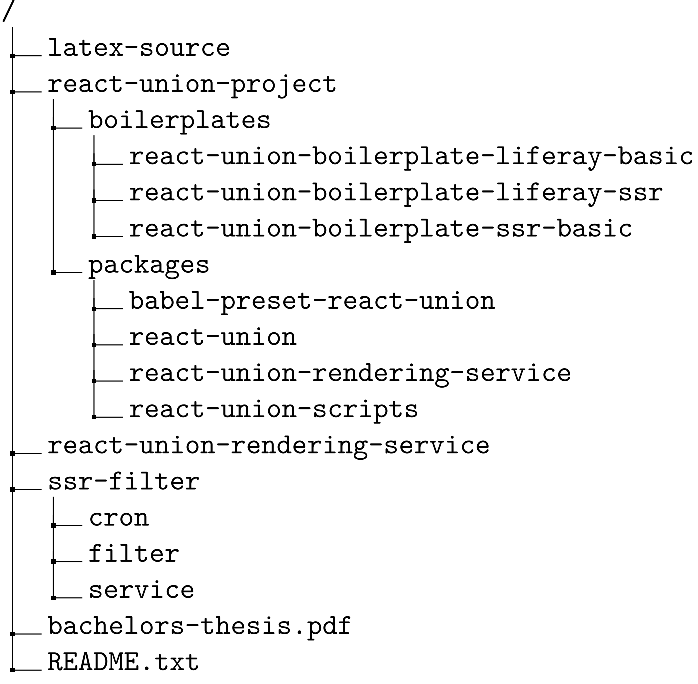
1. Navigate to the React Union root directory via the command line.

2. Run `yarn` to install the dependencies. You may skip this step if all the dependencies are already installed.
3. Navigate to the appropriate project directory. For example, by running `cd boilerplates/react-union-boilerplate-liferay-ssr`.
4. Run `gradle build` to build the application for production.
5. Run `node build/app-demo/server.js` to start the rendering service. This step should be skipped for applications without SSR.
6. Deploy the built `.jar` files by moving them into the Liferay `deploy` directory. These files can be found in `liferay/<module>/build/libs`.
7. React Union portlets should now be available in the Liferay portal.

If the Liferay servlet filter detects that the Node.js rendering service is not available, SSR will be immediately disabled. To manually attempt to re-enable SSR, the filter bundle should be restarted. This can be done via the Felix Gogo Shell [44].

Appendix B
Contents of the enclosed CD

Below is the directory structure of the enclosed CD. For brevity, only the notable inner directories are listed.



The source code of the implemented modules and example applications can be found in the `lundegaard/react-union` GitHub repository as well as on the CD [20]. The `react-union-project` directory contains a complete clone of the repository. It should be noted that the cloned repository might be different to the current `master` branch on GitHub, as the project is continuously being updated. Furthermore, the `react-union-rendering-service` and `ssr-filter` modules have been copied to the root directory of the CD for more convenient source code inspection.

Appendix C

References

[1] Google LLC. (2018). First Contentful Paint, Google Developers, [Online]. Available: https://developers.google.com/web/tools/lighthouse/audits/first-contentful-paint (visited on Dec. 24, 2018).

[2] R. Costello. (2018). A search marketer's guide to Google I/O 2018, DeepCrawl, [Online]. Available: https://www.deepcrawl.com/blog/events/a-search-marketers-guide-to-google-io-2018 (visited on Dec. 23, 2018).

[3] ——, (2018). Webinar recap: The chaotic landscape of JavaScript with Bartosz Goralewicz & Jon Myers, DeepCrawl, [Online]. Available: https://www.deepcrawl.com/blog/events/webinar-recap-javascript-bartosz-goralewicz/ (visited on Dec. 23, 2018).

[4] Google LLC. (2018). Get started with dynamic rendering, Google Developers, [Online]. Available: https://developers.google.com/search/docs/guides/dynamic-rendering (visited on Dec. 25, 2018).

[5] Netlify, Inc. (2019). Prerendering, Netlify, [Online]. Available: https://www.netlify.com/docs/prerendering/ (visited on Apr. 28, 2019).

[6] ZEIT, Inc. (2018). Next.js, [Online]. Available: https://nextjs.org/ (visited on Dec. 24, 2018).

[7] K. Konshin, *Next.js Quick Start Guide: Server-Side Rendering Done Right*. Packt Publishing, 2018, ISBN: 9781788993661. [Online]. Available: https://www.amazon.com/Next-js-Quick-Start-Guide-Server-side/dp/1788993667.

[8] Walmart Labs. (2018). Electrode, Universal React and Node.js application platform, [Online]. Available: https://www.electrode.io/site/web.html (visited on Dec. 24, 2018).

[9] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612. [Online]. Available: `http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612`.

[10] A. Korzunov. (2018). react-imported-component, GitHub, [Online]. Available: `https://github.com/theKashey/react-imported-component` (visited on Dec. 24, 2018).

[11] Gatsby contributors. (2019). GatsbyJS, [Online]. Available: `https://www.gatsbyjs.org/` (visited on May 12, 2019).

[12] Liferay, Inc. (2019). Portlets, Liferay Developer Network, [Online]. Available: `https://dev.liferay.com/en/develop/tutorials/-/knowledge_base/7-0/portlets` (visited on Jan. 7, 2019).

[13] R. Sezov, *Liferay in Action: The Official Guide to Liferay Portal Development*, 1st ed. Manning Publications, 2011, ISBN: 193518282X. [Online]. Available: `https://www.amazon.com/Liferay-Action-Official-Portal-Development/dp/193518282X`.

[14] Dan Abramov and the Redux documentation authors. (2019). Store setup, Redux, [Online]. Available: `https://redux.js.org/faq/store-setup#can-or-should-i-create-multiple-stores-can-i-import-my-store-directly-and-use-it-in-components-myself` (visited on Apr. 28, 2019).

[15] Facebook, Inc. (2018). React, A JavaScript library for building user interfaces, [Online]. Available: `https://reactjs.org/` (visited on Dec. 23, 2018).

[16] R. Wieruch, *The Road to Learn React: Your Journey to Master Plain yet Pragmatic React.js*. Independently published, 2018, ISBN: 9781720043997. [Online]. Available: `https://www.amazon.com/Road-learn-React-pragmatic-React-js/dp/172004399X`.

[17] Facebook, Inc. (2018). Virtual DOM and internals, React, [Online]. Available: `https://reactjs.org/docs/faq-internals.html` (visited on Dec. 24, 2018).

[18] ——, (2018). Reconciliation, React, [Online]. Available: `https://reactjs.org/docs/reconciliation.html` (visited on Dec. 23, 2018).

[19] ——, (2018). Portals, React, [Online]. Available: `https://reactjs.org/docs/portals.html` (visited on Dec. 24, 2018).

[20] Lundegaard a.s. (2018). React Union, GitHub, [Online]. Available: `https://github.com/lundegaard/react-union` (visited on Jan. 7, 2019).

[21] Facebook, Inc. (2019). Create React App, Set up a modern web app by running one command, [Online]. Available: `https://facebook.github.io/create-react-app/` (visited on Jan. 7, 2019).

[22] Liferay, Inc. (2019). liferay-npm-bundler, Liferay Developer Network, [Online]. Available: `https://dev.liferay.com/en/develop/tutorials/-/knowledge_base/7-0/liferay-npm-bundler` (visited on Jan. 7, 2019).

[23] Node.js Foundation. (2019). Node.js, [Online]. Available: `https://nodejs.org/en/` (visited on Jan. 7, 2019).

[24] Airbnb, Inc. (2019). Hypernova, A service for server-side rendering your JavaScript views, [Online]. Available: `https://github.com/airbnb/hypernova` (visited on Jan. 7, 2019).

[25] B. Hughes. (2018). Operationalizing Node.js for server side rendering, Medium, Airbnb, [Online]. Available: `https://medium.com/airbnb-engineering/operationalizing-node-js-for-server-side-rendering-c5ba718acfc9` (visited on Jan. 2, 2019).

[26] J. Gillmore. (2019). React Universal Component, GitHub, [Online]. Available: `https://github.com/faceyspacey/react-universal-component` (visited on May 17, 2019).

[27] ——, (2019). Webpack Flush Chunks, GitHub, [Online]. Available: `https://github.com/faceyspacey/webpack-flush-chunks` (visited on May 17, 2019).

[28] ——, (2019). extract-css-chunks-webpack-plugin, GitHub, [Online]. Available: `https://github.com/faceyspacey/extract-css-chunks-webpack-plugin` (visited on May 17, 2019).

[29] ——, (2019). babel-plugin-universal-import, GitHub, [Online]. Available: `https://github.com/faceyspacey/babel-plugin-universal-import` (visited on May 17, 2019).

[30] Facebook, Inc. (2014). Flow, A static type checker for JavaScript, [Online]. Available: `https://flow.org/` (visited on Jan. 7, 2019).

[31] M. Mueller. (2016). Fast, flexible, and lean implementation of core jQuery designed specifically for the server. GitHub, [Online]. Available: `https://github.com/cheeriojs/cheerio` (visited on Jan. 2, 2019).

[32] L. Chung. (2017). Server side rendering with GraphQL, Apollo GraphQL, [Online]. Available: `https://blog.apollographql.com/how-server-side-rendering-works-with-react-apollo-20f31b0c7348` (visited on Jan. 4, 2019).

[33] JS Foundation and other contributors. (2019). webpack-dev-middleware, A development middleware for Webpack, [Online]. Available: https://github.com/webpack/webpack-dev-middleware (visited on Apr. 28, 2019).

[34] 60frames. (2019). Webpack hot server middleware, Hot reload Webpack bundles on the server, [Online]. Available: https://github.com/60frames/webpack-hot-server-middleware (visited on Apr. 28, 2019).

[35] J. Gillmore. (2019). Universal Demo, GitHub, [Online]. Available: https://github.com/faceyspacey/universal-demo (visited on May 17, 2019).

[36] Liferay, Inc. (2019). Download Liferay Portal CE, Liferay, [Online]. Available: https://www.liferay.com/downloads-community (visited on May 12, 2019).

[37] The Apache Software Foundation. (2019). Apache JMeter, [Online]. Available: https://jmeter.apache.org/ (visited on May 11, 2019).

[38] Guru99. (2019). How to use JMeter for performance & load testing, [Online]. Available: https://www.guru99.com/jmeter-performance-testing.html (visited on May 11, 2019).

[39] Google LLC. (2019). Chrome DevTools, Google Developers, [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/ (visited on May 12, 2019).

[40] 4G.co.uk Limited. (2014). How fast is 4G?, 4G speeds and UK network performance, [Online]. Available: https://www.4g.co.uk/how-fast-is-4g/ (visited on May 12, 2019).

[41] Google LLC. (2019). Time to Interactive, Google Developers, [Online]. Available: https://developers.google.com/web/tools/lighthouse/audits/time-to-interactive (visited on May 12, 2019).

[42] Yarn contributors. (2016). Installation, Yarn, [Online]. Available: https://yarnpkg.com/en/docs/install (visited on Jan. 7, 2019).

[43] Gradle Inc. (2019). Gradle Build Tool, [Online]. Available: https://gradle.org/ (visited on May 14, 2019).

[44] Liferay, Inc. (2019). Felix Gogo Shell, Liferay Developer Network, [Online]. Available: https://dev.liferay.com/en/develop/reference/-/knowledge_base/7-0/using-the-felix-gogo-shell (visited on May 14, 2019).

Appendix D
Abbreviations

AJAX Asynchronous JavaScript and XML. 4

AMD Asynchronous Module Definition. 21

API application programming interface. 7, 8, 12, 19

CMS content management system. 6, 7, 9–11

CSR client-side rendering. 12, 19, 22, 23

CSS Cascading Style Sheets. 2

DOM Document Object Model. 1, 3, 8, 9, 12, 17

FCP First Contentful Paint. 1, 23

HTML Hypertext Markup Language. vi, ix, 2, 3, 5, 8–13, 15, 17, 21–24

HTTP Hypertext Transfer Protocol. vi, 11, 12

I/O input/output. 11, 12

ID identifier. 9

JS JavaScript. vi, ix, 1–8, 10–12, 15, 17, 18, 21–24

JSON JavaScript Object Notation. 9

JSX JavaScript XML. 8

JVM Java Virtual Machine. 11

NOOP no operation. 3

PHP PHP: Hypertext Preprocessor. 1

REST Representational State Transfer. 7, 12

SEO search engine optimization. vi

SPA single-page application. vi, 1–3, 6, 7

SSR server-side rendering. viii, ix, 2–6, 10–13, 15, 17–19, 21–25, 28

TCP Transmission Control Protocol. 15

TTI Time to Interactive. 24

UI user interface. 2, 7, 8

UML Unified Modeling Language. viii, 13, 14

XML Extensible Markup Language. 4, 8