Aidar Mukhamadiev

Transitioning from server-side to client-side rendering of the web-based user interface: a performance perspective

Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

28 November 2018

Metropolia

Author	Aidar Mukhamadiev
Title	Transitioning from server-side to client-side rendering of the web-based user interface: a performance perspective
Number of Pages	41 pages + 14 appendices
Date	28 November 2018
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Professional Major	Web Development
Instructors	Marko Klemetti, Chief Technical Officer Ilkka Kylmäniemi, Senior Lecturer

The goal of the thesis is to implement a single page application using React and Redux as a replacement for the case company's existing solution of server-side rendered Ruby on Rails interface. Consequently, new implementation is compared to the old one from the perspective of performance only. Caused impact on both client and the server, although significantly restricted, is considered.

The investigation is limited to the desktop browsers only such as Google Chrome, Mozilla Firefox and Apple Safari, among which the former is selected as a major reference for the results and the following discussion.

Keywords	web development, performance, benchmarking, react

Metropolia

Contents

List of Abbreviations

API	Application Programming Interface
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
SPA	Single Page Application
REST	Representational State Transfer
DNS	Domain Name System
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
TLS	Transport Layer Security
HTTPS	Hypertext Transfer Protocol Secure
RTT	Round-Trip Time
CDN	Content Delivery Network
URI	Uniform Resource Identifier
UI	User Interface
DOM	Document Object Model
CSSOM	Cascading Style Sheets Object Model
XHR	XMLHttpRequest
AJAX	Asynchronous JavaScript and XML

JSON	JavaScript Object Notation
SEO	Search Engine Optimization
SSR	Server-side rendering
CSR	Client-side rendering

1 Introduction

As a response to the case company's growth and new business requirements, Trail's web interface is currently under an ongoing step-by-step transition to single page application (SPA), powered by React, a JavaScript view library developed by Facebook. In this thesis, the terms SPA and client-side rendered interface are used interchangeably to mean a web application HTML markup of which is constructed in the browser entirely. Similarly, terms client and browser are to convey the same meaning across the study.

Given the above, this paper attempts to investigate implementation and the caused effect of Trail being partially moved from server-rendered to client-rendered interface. Among all the potential changes introduced by the shift, the scope of the research for Trail is limited to the domain of performance, both from consumer and provider perspectives. The main targets of this thesis are to construct production-level SPA components and consequently provide quantitative answers to the following pair of questions:

- In the context of a user, how does client-side rendered interface perform in comparison to server-side one?
- How does client-side rendering (CSR) impact the server load and capacity?

It is important to point out that the thesis does not engage with mobile web benchmarking, but desktop only. In addition, to clarify, the variables such as an operating system, network bandwidth and computing power are abstracted out and considered to be constant. The primary focus of the study revolves around the amount of transmitted business data (e.g. table list of items) and the effect of cache in the context of multiple browsers.

Based on the scope, this paper has been divided into six parts. It begins with an introduction, the purpose of which is to provide a proper frame the further discussion belongs to. Particularly, state the questions the case company is attempting to address in the space of the technical changes it experiences.

The study goes on to theoretical framework, where the literature scope for the research is identified. For the most part, it includes the topics of networking followed by view rendering. The goal is to present how communication is performed, how browsers work and what architectures exist to render view data to the user.

The third section focusing on methods and materials presents the reader sufficient information on SPA implementation and benchmark design applied to the former. Frameworks, libraries, browsers, standalone applications and other items used in the research are listed.

Part four reports raw research results obtained using previously described methods. Findings are listed in structural manner without any interpretation.

The fifth part is dedicated to the discussion of benchmark design validity, and the final interpretation of results. Moreover, research limitations and further improvements are stated.

Finally, the conclusion provides a summary of the previous sections and gives answers to originally stated research questions.

2 Theoretical framework

2.1 Performance value

In the space of rising demand for a better experience, new browser features and resulting complexity, the performance variable is fundamental. It is becoming a key topic requiring particular attention, and the issues are common to the extent, that, as Grigorik (2014) defines it, web performance turned into an application's feature [1, p. 3]. Likewise, Wagner lists multiple reasons why well performing pages are important [2].

First of all, performance optimization improves user retention and conversion. Obviously, speed impacts the success of any online business, be it driven by sales or advertisement. To provide an example, Pinterest engineers Meder S et al. (2017) ran an experiment and found a 40% decrease in user wait time leading to 15% increase in search engine traffic and similar boost in conversion rate to account registration [3]. By contrast, DoubleClick (2016; Google Marketing Platform), in its study on the impact of mobile speed, reported that 53% of the visits are dropped out when the site takes more than 3 seconds to load, and that generally half of the users expect the load time to be less than 2 seconds [4].

Secondly, performance contributes to the quality of UX. For instance, Nielsen (2010) considers responsiveness as an essential user interface design law, guided by human limitations and goals. Indeed, waiting and feeling of fading of short-term memory information do not lead to user's productivity. On top of that, lack of quick service causes the provider to be perceived incapable and presumptuous, not only during the actual moment of experience, but also in the context of general brand values associations. [5.] In other words, poor performance can cause long-term and contagious emotions.

Finally, poor performance can pose tangible costs for people. Schwarz (2017) points out lack of general awareness of the ways and conditions under which people access the Web worldwide. He notes general unavailability of fast and inexpensive network bandwidth, and dominance of low-end, resource-limited mobile devices in the market. [6.] GSM Association (2018) reports that in 2017 globally among total cellular connections excluding Internet of Things devices 40% were done using 2G technology and expected to be of 4% only by 2025 [7]. To illustrate, why this could be problematic, as for September 15, 2018 median size of all uncompressed resources fetched by a page is 1270.3KB

and 1544.7KB for mobile and desktop, respectively [8]. By approximating its compressed size down to 300KB, 2G downloading of the page of that size could be expected to take up a significant amount of time. In addition to that, there is a time spent to parse that amount of data, which depends on computing power of the device, which, as noted by Osmani (2018), often happen to be low in computing performance [9].

Given all above, it is good practice to provide extensive functionality while also trying to fit the user's constraint space, consisting of network and client capabilities. Quick service access improves user satisfaction, which in turn is expected to have a positive impact on both retention and conversion. Without a doubt, the practice would be beneficial to any of online service providers, among which Trail Systems Oy is not an exception.

2.2 The case company overview

Trail Systems Oy is a Finnish software as a service provider. Established in 2010, its main product is Trail [10], an interactive application for a fixed asset management, which helps organization to keep track of the assets of various kinds. Trail consumers include both local and international players such as The National Theatre of London, YLE, Kone etc. Being originally developed for the domain of performing arts, Trail extended its functionality to fit other businesses, for instance, fields of construction, media and logistics.

On a technical side Trail is represented as a server accessed by a browser and mobile applications for Android and iOS operating systems. The server is built on a Ruby on Rails framework, providing server-side rendered views along with the representational state transfer (REST) endpoints.

2.3 Fundamental phases

Essentially, the procedure of displaying web content to the user can be organized into two phases: resource fetching and resource processing. To clarify, it is helpful to refer to the specification of Navigation Timing Level 2 interface, providing access to thorough, high-resolution data on timing for document navigation [11]. Particularly, it is the processing model, which sparks most of the interest as it provides a visual structure of the web performance from the perspective of the client. The model is depicted in Figure 1.

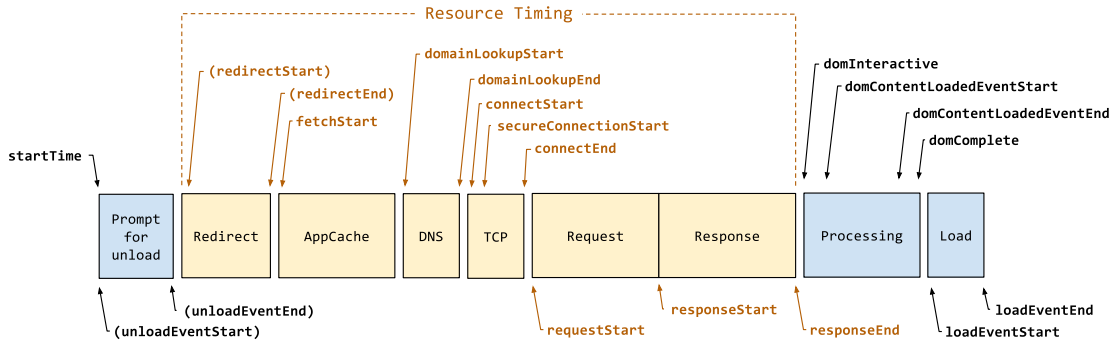

Figure 1. Navigation Timing interface processing model. [11]

Figure 1 illustrates the events accompanying the loading process of the root document and associated with it set of timing attributes. First, the previous document, if any, is unloaded. DNS query is then made, and by using the retrieved IP address, the TCP based HTTP request inquires the document data, which, as the response comes, is processed, loaded and finally displayed to the end user. During the procedure, the page is redirected if required, and the application cache is utilized, if available. [11.] To note, there is a clear separation between the concerns.

In the setting of current investigation, it would be essential to explore how the web resources are retrieved and rendered by a browser, and, based on that, what kind of rendering architectures are prevalent in the industry.

2.3.1 Resource fetching

HTTP protocol is a core instrument for a resource retrieval. RFC7230 defines it based on top of reliable transport/session channel, which is commonly represented as TCP or as Transport Layer Security (TLS) layer on top of the former [12, p. 7; 13 p. 76]. Injecting TLS in between extends the idea of HTTP to HTTP Secure (HTTPS) with additional properties of connection privacy and data integrity [14, p. 4].

The topics of particular interest are the TCP, TLS and HTTP, the last two of which are referred to in relation to specific versions across the thesis, namely 1.1 for HTTP and 1.2 for TLS, unless specified otherwise. First, however, it is essential to introduce two core elements defining the performance of a network transmission.

2.3.1.1 Bandwidth and latency

Bandwidth refers to a maximum amount of data that can be transmitted across the communication path at a unit of time [15, p. 35]. Latency is the time from the moment of the source sending the packet to the moment of the destination receiving it [1, p. 3]. Given the destination sends back an acknowledgment packet, the total time between sending and getting acknowledgment is then defined as a round-trip time (RTT) [15, p. 803].

There are multiple factors contributing to latency such as speed of light, packet header analysis etc. In contrast to bandwidth, there is limited space for an improvement [1, pp. 4-12.] Therefore, because of short-lived nature of HTTP requests, latency plays a more significant role compared to bandwidth, as it was demonstrated by Belshe (2010) and depicted in Figure 2 [16].

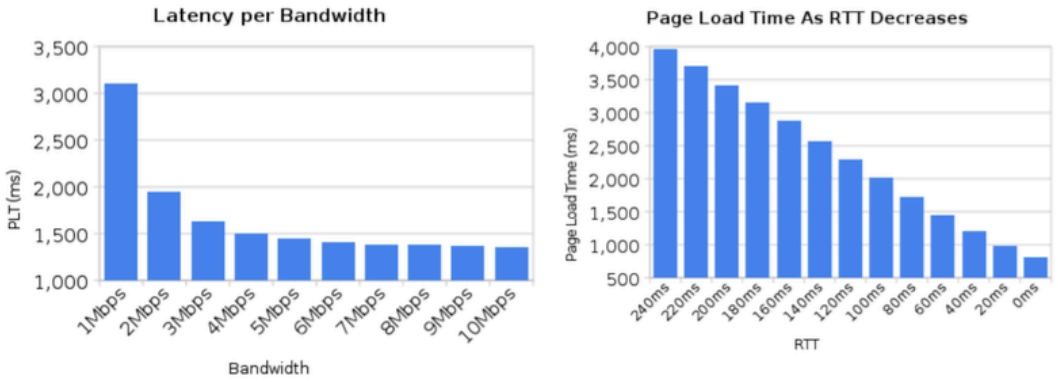

Figure 2. Comparison of the effects of bandwidth and round-trip time (RTT) on page load. [16]

Figure 2 on the left shows time required to load a page per bandwidth. As it increases, the load time also decreases, but quickly approaches the state of diminishing returns. RTT latency, in turn, indicates a continuous efficiency – decrease in RTT consistently decreases the page load time. It can be concluded, therefore, that the lower RTT or overall amount of round-trips, the better. In this context, the following review is to point out the impact technologies have on the overall accumulation of RTT.

2.3.1.2 TCP

Any browser-initiated resource request starts in the space of TCP - a reliable, connection-oriented protocol, allowing a pair of endpoints to establish a virtual connection and

bidirectionally exchange the data [15, p. 691]. If there are two devices, intending to exchange data over TCP, first, the connection must be established, and that procedure is referred to as the three-way handshake, illustrated in Figure 3.

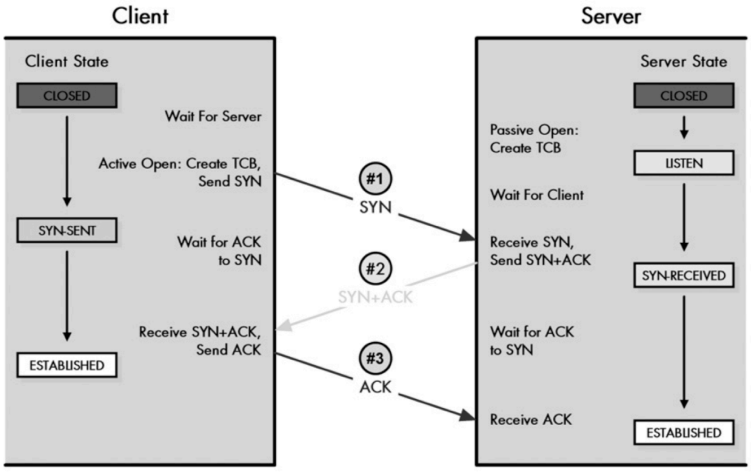

Figure 3. TCP three-way handshake procedure. [15, p. 755]

As displayed in Figure 3, during the handshake, the TCP client sends a synchronization control (SYN) message. After receiving it, the server sends back acknowledgment (ACK) for the client's SYN and own SYN, joined in a single message. Finally, the client responds with ACK for the SYN of the server. [15, p. 753.] The procedure is of three steps in total and represents a significant cost in terms of latency. Therefore, in the context of performance optimization the concept of TCP connection reuse is significant [1, p. 15].

When the handshake is completed, the data is ready to be passed, bringing sliding window acknowledgment system into action. It is based on the idea of a receiver's feedback, recognizing the fact of a successful segment transmission. Fundamentally, there is a finite number of pending unacknowledged segments are allowed. Such an amount is defined as a window. [15, pp. 732-740.] The size of it is dictated by a minimum of receive (*rwnd*) and congestion (*cwnd*) window values, which are determined by a receiver and network buffer capabilities, respectively [1, p. 19-20].

During the transmission, the path could be under a heavy load due an extreme amount of simultaneous connections, resulting into segment loss for each. Referred to as a congestion, such an overload can be escalated to the degree of a congestion collapse, turning the network unusable. To avoid the problem, there is a specific mechanism constructed, consisting of slow start and congestion avoidance algorithms. [15, pp. 816-817.]

The key idea behind slow start is to examine the network and identify usable capacity in an exponential way to avoid network congestion due to injection of an excessive amount of data at the start. The window size, at the range of which the network examination occurs, is defined as a slow start threshold (*ssthresh*). When the window exceeds the threshold, a congestion avoidance algorithm is executed, task of which is also to probe the network, but in a slower, linear manner. When the segment loss occurs, the value of the threshold is redefined, meaning the responsibility space of the algorithms is also altered. [17, pp. 4-6.] To clarify, Figure 4 presents the way the mechanism is conducted.

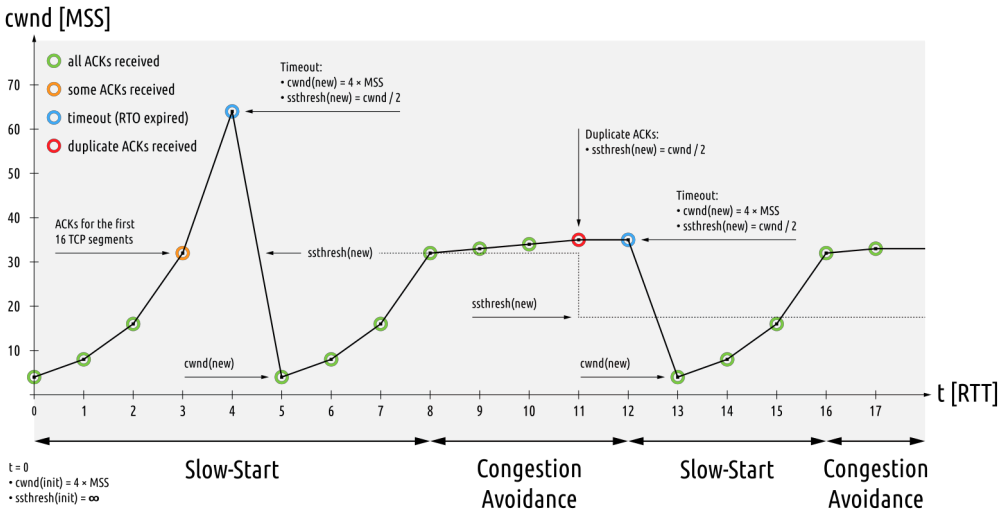

Figure 4. Slow start and congestion avoidance algorithms in action. [18]

As illustrated in Figure 4, that as soon the TCP connection is initiated, slow start is executed with an explicit initial congestion window [17, p. 5], and arbitrary, but high threshold value. At round-trip 4 the segment is lost due to a timeout, which results in a programmatic reduction of the threshold [17, p. 7] and new slow start execution. It completes at round-trip 8, where congestion window reaches the threshold value, passing the network control to congestion avoidance algorithm. Later, another threshold reduction occurs at round-trip 11 because of duplicate acknowledgments caused by out-of-order segment delivery which, however, does not trigger the slow start algorithm [17, p. 7-8], unlike the loss of the following segment at round-trip 12.

The congestion window behavior follows the entire communication process, until it is closed. It provides network reliability, but does it at the expense of performance, regardless of an actual algorithm implementation [1, p. 27]. The effect, in turn, depends on the nature of the communication: it is higher with respect to a short request compared to

large streaming download. The reason is that it is often possible to terminate the former before reaching the maximum window size. [1, p. 22.] Considering the prevalence of short-time requests, preceded with the three-way handshake phase, TCP alone introduces a certain amount of performance inefficiency for modern web applications. On top of that, considering the secure communication is the preference, there are additional costs included, which is presented by TLS.

2.3.1.3 TLS

TLS is a cryptographic, application independent protocol providing a secure, private communication between the two parties [14, p. 4-5]. If there are third-party observers of the channel, they could only deduce the endpoints, an encryption type, amount of data sent, but not read or mutate the substantial plaintext data [1, p. 47]. TLS allows the endpoints to authenticate each other and negotiate on protocol versions, encryption algorithms and the keys before actual data exchange happens. [14, p. 4.] Such a handshake procedure is illustrated in Figure 5.

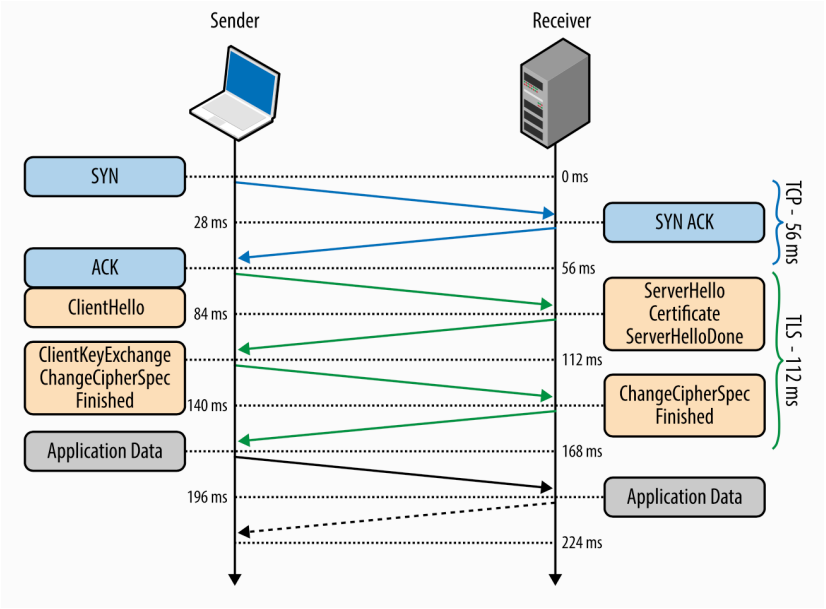

Figure 5. TCP with TLS Handshake Protocol in action. [1, p. 51]

As it is demonstrated in Figure 5, first, the TCP communication is established through the procedure of three-way handshake. After that the sender transmits various meta information such as TLS version, supported set of cipher suites etc. The receiver responds with the selected TLS version, a cipher suite, and the certificates. In addition, it could

request for more information if needed e.g. sender's certificate. As the second TLS round-trip starts, the sender generates a symmetric secret, encrypts it using the receiver's public key and dispatches it. The receiver, in turn, decrypts the delivered key, verifies MAC and responds with an encrypted message of "Finished" back to the sender. Finally, the sender decrypts the message by the symmetric key it previously generated and checks the MAC. If validation succeeds, a secure communication tunnel is set.

It can be seen that the TLS handshake does two round-trips, twice as much as required for a TCP handshake, imposing a significant performance penalty. [1, pp. 51-52]. The way to mitigate it is to reuse the results of the negotiation across multiple connections, for instance, by utilizing session identifiers or session tickets. Interestingly, in the situation of the multiple request to the same TLS server, most modern browsers purposely wait for the first one to complete to reuse the session. [1, pp. 55-57.]

As soon as TLS channel is ready, the client and the server are able to perform secure HTTP communication, the mechanism of which is described in the following chapter.

2.3.1.4 HTTP

HTTP is a stateless protocol for distributed hypertext information systems. It is based on the request-response transactional model, and operating on top of a reliable channel, namely TCP along with TLS for a secure communication. [12, p. 7.] The communication happens in the context of resources, targeted by HTTP using uniform resource identifiers (URI) [12, p. 16]. When client makes request to URI, it defines the purpose, described by method tokens. Along with the token, request is allowed to supply header fields, representing various meta information. [19, p. 33]. The corresponding response, along with headers and payload, if any, delivers the request status code. [19, pp. 47].

Request and responses operate with messages, which contain entities. Entities consist of headers and body. The latter is the transferred content, if any, while the former is the content's metadata. Content could be of different types, and be encoded or compressed, thereby improving performance [13, pp. 342-354] Unlike the content, the headers are not compressible, thus introducing an overhead to the degree, that they often reserve more space than substantial data [1, p. 200-201]. The manner in which the content transferred is encodable as well, specifically it is deliverable in chunks, rather than in a single

transaction. That is helpful when the size of the response is not identifiable quickly due to its size or dynamicity, or, for example, when the HTML is served in parts in the situation with progressive rendering. [13, pp. 356-357; 20.]

HTTP uses a persistent kind of connections by default, allowing multiple requests to be performed in the life-time of a single TCP connection. In addition, the requests could be pipelined and sent over without waiting for the previous transaction response to arrive [12, p. 51-54] However, the feature has not been widely adopted, and most of the browser disabled it due to various network crashing bugs and complications. [1, p. 195; 21] Alternatively, to escape blocking, HTTP requests are sent in parallel over multiple connections, but the amount of those is recommended to be restricted because of various performance issues. [12, p. 55; 1, p. 197.] According to Browserscope, browsers Chrome 50 and Firefox 46 limit the amount down to 6 connections per host [22].

From the performance viewpoint, a key property of HTTP is cacheability. An HTTP cache is a local repository, designed to store the response messages for identical request to improve performance. Cached response is regarded as fresh if it is retrievable without validation, meaning communicating the server and examining the response for its relevance. [23, p. 4.] The moment for the response between being fresh or not is defined by an expiration time, set either by the server or the client [23, p. 11]. Servers specify the time using the response headers of Expires or Cache-Control [13, p. 176]. If the headers are not supplied, the client tries to heuristically estimate the expiration time e.g. by an LM-Factor algorithm [13, p. 184]. After the expiration time is reached, next request contains specific conditional headers, compared against the headers of the response. If the condition matches, the new resource is sent. Otherwise, it is only the headers delivered, possibly with the new freshness expiration date. [13, pp. 177-180.]

2.3.1.5 Optimizations to consider

In the context of costly TCP and TLS handshakes, followed by congestion control mechanism, the priority is to minimize the amount and the payload of HTTP requests. This implies content compression, with minification in case of scripts, usage of CSS sprites, establishing persistent connections and appropriate cache utilization [24, p. 11]. On top of that, to reduce latency, it is beneficial to minimize the distance of communication, which is achievable with the Content Delivery Network (CDN), efficiently serving static

assets across the globe [24, pp. 19-20]. Finally, communication should happen within minimum number of domains to reduce the amount of potential DNS lookups [24 p. 68].

2.3.2 Resource processing

As soon as the required content is transported, browser loads it into the rendering pipeline and initiates the processing stage. Current chapter explores the details of that procedure and how it fits in the space of browser architecture.

2.3.2.1 Browser architecture

A browser is a multiplatform software application, key function of which is to retrieve the web resource of various media and display it to the user [25]. Among numerous browser developers, the majority of the market is hold by Google's Chrome, resulting in an inordinate 61.51% of the share in October 2018. It is followed by Apple's Safari (15.16%), Mozilla's Firefox (5.02%). [26.]

Browser's high-level architecture consists of several components such as UI, browser engine, rendering engine, networking, UI backend, JavaScript interpreter and, finally, data persistence unit [25]. The UI represent interactive elements such as address bar, bookmarks etc. Data persistence layer is accountable for local information storage e.g. cookies. The browser engine is a mediator between the previous two and the rendering engine, which, given the requested resource is supplied by the networking component, sequentially orchestrates generation of the view out of it. JavaScript interpreter parses and runs JavaScript code, accompanying the view construction. The drawing of the browser's UI and the view, in turn, is performed by UI backend layer. [25.]

Browsers use different rendering engines. For instance, Firefox uses Gecko, but Chrome uses Safari's WebKit fork called Blink [25]. Similarly, JavaScript interpreters also vary: Chrome runs V8, Firefox works on top of SpiderMonkey, and Safari uses JavaScriptCore [27]. Despite the divergence, modern browsers tend to follow and consistently implement the official HTML, CSS and ECMAScript specifications, thereby generally operating in analogous way. In the past, for instance, browsers were following their own set of specifications along the way, causing compatibility related problems for web developers. [25.]

The scope of the topic of resource processing is limited mainly to the layer of the rendering, as it gives a sufficient outlook on the mechanism behind preparations for the page to be displayed to the user. Given the context of the previous review on resource fetching, it is considered that the resources at this point are retrieved and prepared to be served for rendering from the networking component.

2.3.2.2 Rendering pipeline

The rendering pipeline graph is illustrated in Figure 6.

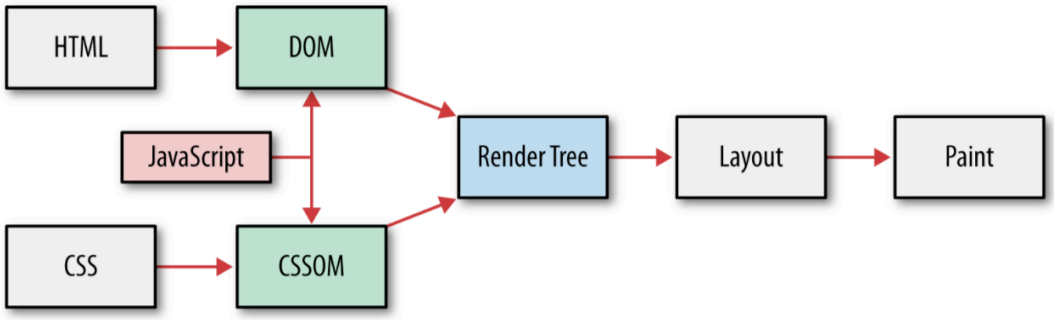

Figure 6. Rendering pipeline. [1, p. 168]

Specifically, Figure 6 demonstrates the sequence of phases for HTML, CSS and JavaScript assets to be handled. The steps include constructions of object models, render tree assembling, followed by layout process and painting at the end. [28.]

To begin with, given there is an HTML byte code supplied, the parsing of it starts. Bytes are converted to the stream of characters based on the encoding specified e.g. UTF-8. Characters, in turn, are classified into specific tokens such as <*div*>, <*span*> etc. The tokens are then converted into the objects with additional specific properties. Finally, by defining relationships between the derived objects, the DOM tree is constructed. This tree defines features and connections of the document markup but does not provide any information on the visual style, responsibility for which is encoded in CSSOM. [28.]

As soon as the stylesheets are reached during the HTML parsing process, they are processed in an asynchronous manner. They could be retrieved directly or require an HTTP request. Similar to HTML, CSS bytes are then converted into a tree structure referred to as CSSOM. [28.]

Another type of asset embeddable to HTML document is JavaScript code in <*script*> tag. Same as with CSS, the code could be inline or linked to an individual file, thereby requiring an explicit retrieval. What makes it different, however, is its ability to block parsing because the script could inject tag tokens into the stream through *document.write()* API [29]. In addition, JavaScript cannot be executed until the CSSOM is constructed, because the script could contain CSS related logic and thereby introduce a race condition. In the situation when the script includes DOM associated code, referencing unparsed or non-existing node at the time, results in an error. Given the constraints, it is helpful to apply <*script*> attributes of *async* and *defer*. They both command to asynchronously fetch the script file, but the difference is that *async* implies immediate execution on file availability, when *defer* on document parsing completion. [28.] In other words, the former is unpredictable in timing, while the latter is not. Specifically, *defer* script execution always performed before DOMContentLoaded event is fired. In contrast, *async* kind of scripts do not block DOMContentLoaded, but they are executed before the final window load event is triggered. [29.]

As soon as both the DOM and CSSOM are assembled, they are combined into a single union structure. DOM tree is traversed, and the invisible tags are omitted, such as <*meta*>, <*script*>, or the ones with corresponding styles e.g. *none* for *display*. Visible nodes are then mapped to associated styles in CSSOM. By joining these two set of properties together, a new tree is produced, referred to as render tree. At this point it is important to note, that linked CSS is by default a rendering blocker, meaning the render tree cannot be constructed without complete CSSOM tree. CSS <*link*> tags, however, can specifically set the media context using *media* attribute, and if it matches the current state, the file is to be parsed entirely and potentially block the rendering, otherwise render tree is available to be constructed without it. [28.]

When the render tree is prepared, the process moves to the next phase defined as layout or reflow, where the nodes' absolute positions and dimensions within the viewport are computed. Layout recursively iterates elements, starting from the root, which is <*html*> [25]. The output of the process is presented as a box model. At this point, spatial properties of the visible nodes are known along with the associated styling characteristics. Such an information is sufficient to be passed as an in input to the painting or rasterizing process, which converts the data provided to graphical artifacts on the user's screen. [28.] Supplied box model consists of layered stacks, painting order of which proceeds from the back elements to the ones on the top [25].

As long as there are dynamic changes introduced to the painted page, browser attempts to cope with them in an efficient manner. For instance, element color change results only in painting of the element. Change in the position triggers both layouting and paintings of the element and its children. Insertion of a new DOM node induces layout and paint execution on the node. Global changes, however, such as changing the font size of the *<html>* tag, affect the entire tree, thereby being computationally expensive. [25.]

2.3.2.3 Optimizations to consider

In the context of render tree construction, supplying HTML along with CSS in front is critical for the first paint timing. CSS files a recommended to be situated at the top of the markup file, because it ensures the stylesheet is quickly retrieved, and object model trees are assembled in parallel. Besides, scripts should not block the pipeline wastefully – they are to be placed at the bottom of the markup, and minified to reduce time required for execution, which should proceed in asynchronous manner, if possible [24, pp. 45, 69].

2.4 Rendering architectures

Considering previously reviewed stages of resource fetching and resource processing, existing rendering architectures represent an attempt to manipulate these variables to achieve specific goals, for instance, richer user experience or better performance. To avoid confusion, it is important to clarify that the term rendering, when referenced from architectural perspective, is to mean responsibility for HTML markup assembling rather than browser related rendering, consisting of HTML parsing and painting.

2.4.1 Server-side rendering

In contrast to contemporary web experience, pioneering web pages were uncomplicated, containing HTML with the content of plain text and images, limited to functionality of sharing of scientific research documentation. Pages were connected through links and each transition required an entire page reload. Later, server-side programming languages such as PHP (1995) introduced the concept of dynamic HTML, rendered by the server and responded with to client, subsequently making it possible for pages to react to the requests of a more complex kind. Nevertheless, an entire page refresh was

required on every action. [30, p. 3; 31, pp. 6-8.] Defined as server-side rendering (SSR) architecture, its mechanism is presented in Figure 7.

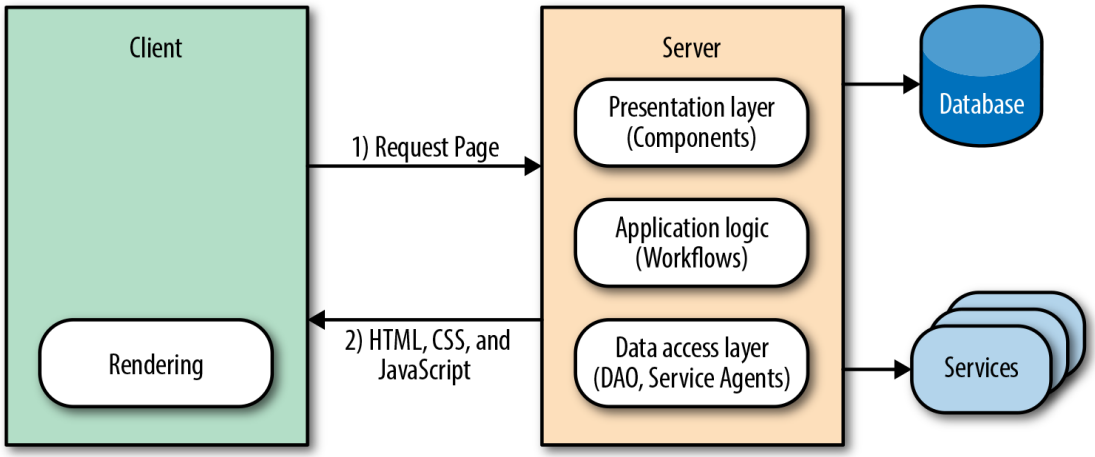

Figure 7. Server-side rendering. [32, p. 6]

As it can be seen in Figure 7, that the only responsibility of the browser is to render server constructed markup, while the server handles the view, application logic and persistence domains. The communication protocol is elementary and consists of a page request, following delivery of various assets on initial load.

Later, the idea of interactivity was augmented by a new client-side scripting language of JavaScript (1995), later accompanied with DOM (1998). Following the development and adoption of XMLHttpRequest (XHR) browser API a paradigm shift for JavaScript was highlighted by Garrett (2005), who used the term of Asynchronous JavaScript and XML (AJAX) to refer to an industry-proven set of techniques for making client-to-server interaction asynchronous. Namely, Google was one of the first companies heavily using the approach, what was reflected in its products such as Gmail and Google Maps. Those kinds of applications were different from a classical request-refresh model in having an additional JavaScript layer responsible for both asynchronous server data access and successive client-side HTML rendering. As a result, continuous interaction requires less page reloads for the same set of use cases, thus bringing a richer user experience. [31.]

The changes AJAX introduces to the scheme are shown in Figure 8.

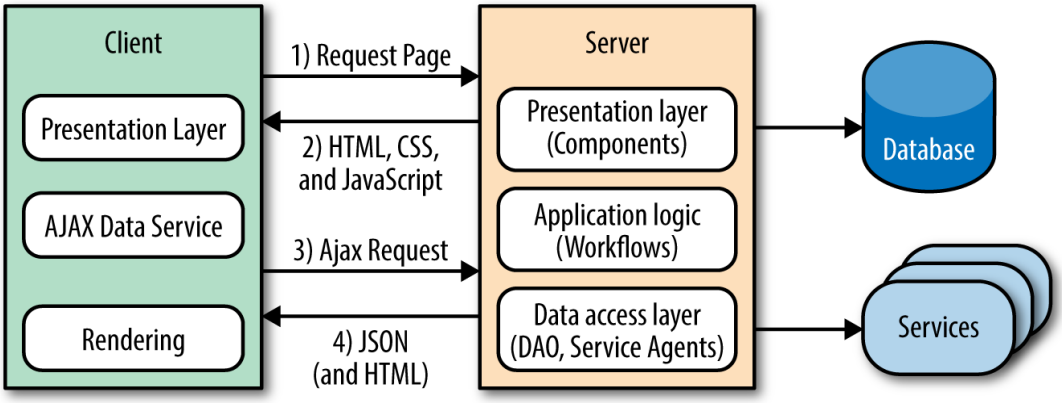

Figure 8. AJAX-augmented SSR architecture. [32, p. 8]

Figure 8 demonstrates the client extending the space of its responsibility. By using AJAX, the client is able to manage asynchronous data retrieval affecting the current presentational layer. Communication intricacy grows, as the overall request count increases. To point out, most of the functionality is conducted on the side of server.

2.4.2 Client-side rendering

Major problem with AJAX approach is introduction of additional application complexity. It causes duplication in views, models and assets required for the client, complicating the process of maintaining and keeping track of the codebase as new features are implemented. To avoid such a problem, the AJAX locality can be scaled to the entire domain, meaning management of the HTML markup construction becomes responsibility of the client only. [31 p. 8-9.] The resulting application is referred to as SPA. It consists of a single document communicating to the server for JSON data only, and brings no full refreshes on interaction, resembling a cross-platform, native user experience [30 p. 11].

SPA architecture is depicted in Figure 9.

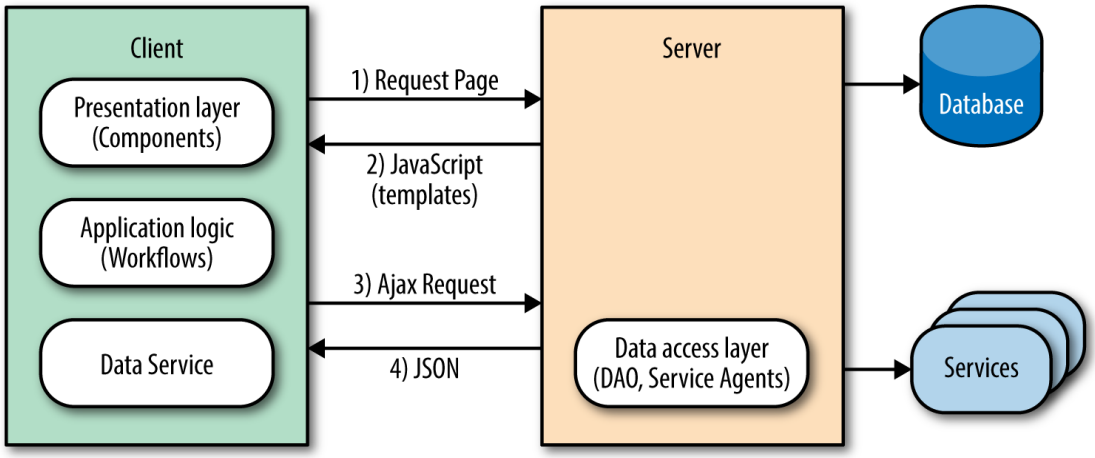

Figure 9. SPA architecture [32 p. 9]

Figure 9 shows the degree to which the responsibility of the server narrowed and transferred to the client, which governs both views and application logic to the whole extent. Impact on the server is reduced significantly, and server's purpose is to provide a data access medium to underlying database and services. Ideally, there is a precise separation of concerns. However, as Airbnb engineering team points out, some logic ends up being duplicated eventually, for example, date formatting and form validation [34].

SPA architecture suffers from several major drawbacks. First of all, the time required for an initial load can be high, because an entire application needs to be fetched first before it becomes interactive [32, p. 9]. Twitter, for instance, after implementing SPA in 2010, moved rendering back to the server in 2012, achieving one fifth of the previous page loading time [35]. Secondly, SPA is unreachable for web crawlers by default, thereby unsearchable [32, p. 10]. On top of that, SPA has been subjected to considerable criticism by Navis (2018), who points out an unjustified price of SPA and lack of business reasons for such an architecture in comparison to SSR. Specifically, he calls attention to frontend statefulness and the variety of issues it causes. [36.] However, Petrina (2018) has challenged some of Navis's conclusions, arguing that the major advantage SPA offers is ability to construct a component-based system on the client, which is easier to build, maintain, and therefore are less expensive. In contrast, he notes, writing proper reusable code with server templating systems is a non-trivial task. [37.]

2.4.3 Isomorphic rendering

In recent years, there has been an increasing interest in a new type of applications referred to as isomorphic or universal. Fundamentally, it represents a conceptual merge of both SSR and SPA, leveraging the finest features of theirs. Based on the ability to run same piece of code on JavaScript environments of both client and server, isomorphic application initially loads a server-rendered view containing the scripts, required to bootstrap the SPA. There are multiple benefits this approach offers. To begin with, first meaningful paint is rendered by browser quickly, thereby improving user's perceived performance despite the fact that eventual page load time is comparable to pure SPA. As a consequence, page is indexable by web crawlers. Maintenance of the application is considered to be consistent because regardless of where the code is run, it needs to be written once. Drawbacks, however, consist of handling browser and server differences, resulting in challenging debugging, testing and global variable management. It is also important to note, that although the first paint is quick, the interface is not interactive until the scripts are finished executing [38, pp. 4-18.]

There are many companies in the industry, who implemented isomorphic principles on production level successfully. To name a few, for instance, Netflix Website UI Engineering team achieved 70% reduction in page load time after replacing SPA stack with universal one [39]. Walmart had a positive experience as well and reported an increase in users' engagement thanks to early rendering on Electrode platform [40].

2.5 Summary

Web application performance is important and affects multiple dimensions of any online business. For a user, the timing between the page being requested and the page being displayed in the browser is contributed by numerous intermediary details in the domains of networking and rendering, which need to be considered and coped with. While there are several factors out of control, one can mitigate an impact if needed, and choose suitable architecture based on available resources and business requirements.

3 Methods and materials

The goal of this chapter is to describe the systematic steps required for achieving the objectives stated in introduction. The chapter is divided into two parts: implementation and benchmark design. The former lays out the key features of constructed SPA, paying particular attention to the component under the test. Design of the test is presented in the latter part and defines the benchmark environment along with the metrics of choice with the goal of answering the question of how the interface transformation affects the performance from the perspective of the client and the server. In both sections, issues regarding validity and reliability of the methods are pointed out.

3.1 Implementation

3.1.1 General overview

The proposed application's architecture was based on Facebook's JavaScript view library of React, accompanied with the functionality of Redux container for the client's state management, Semantic UI library for ready-made components along with the styles, and Webpack bundler for transforming and packaging the resources. As a complete reference, Appendix 1 contains the list of each of the installed packages along with the versions scope.

Constructed by using the mentioned tools, the resulting SPA consisted of numerous views meeting different business requirements, among which one was selected to be the target of performance testing, and therefore is the one the implementation inspection was focused on. Namely, the page of choice displays a dynamic list of table entries, which in the domain of Trail are referred to as consumable models. There are multiple reasons behind the decision of aiming attention at that particular view. First of all, it is functionally and visually close to a previous server-side implementation. Secondly, the dynamical nature of the list allows to represent the interface of various sizes and thereby manipulate the load applied to both the client and the server. That is helpful for the purpose of benchmarking as it increases the space for an experimentation. Finally, from the architectural perspective the new view uses set of generic patterns, therefore providing a sufficient high-level information on how the rest of the SPA operates. Given all that, to provide a better context, Image 1 and Image 2 illustrate visual differences between the

new client-rendered and old server-rendered consumable models list interfaces, respectively.

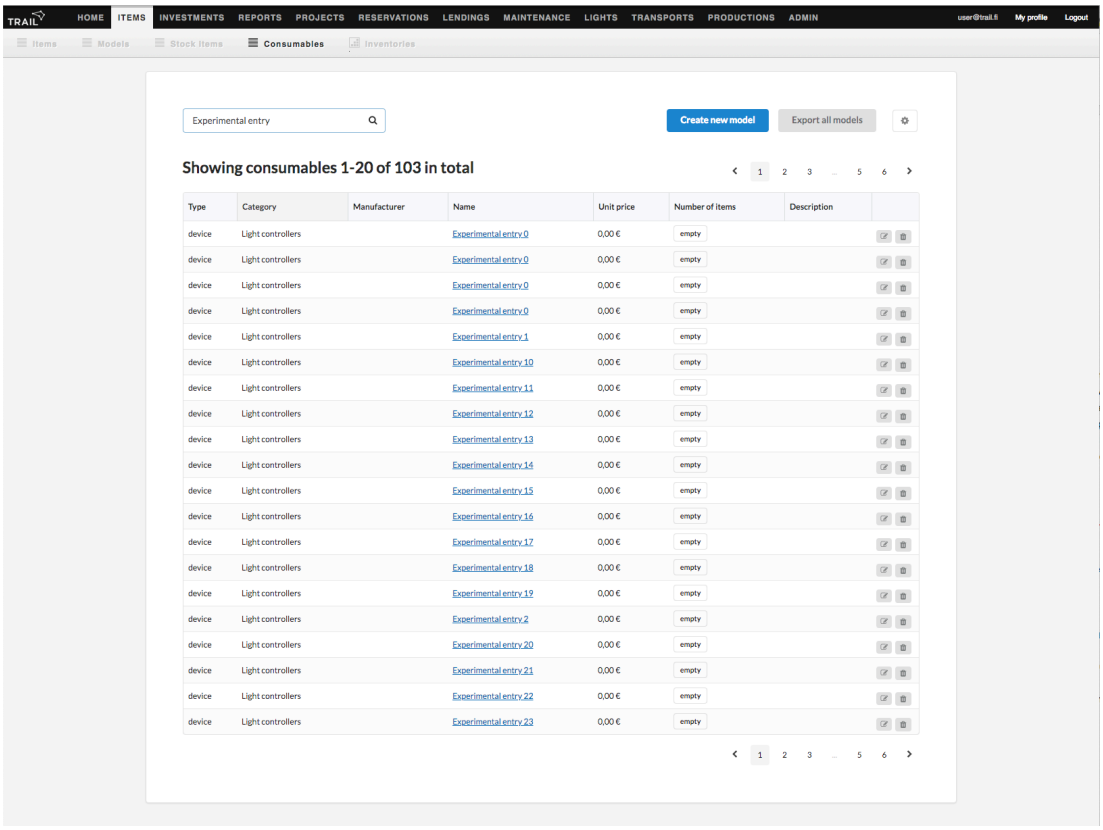

Image 1. New consumable list view generated by the client

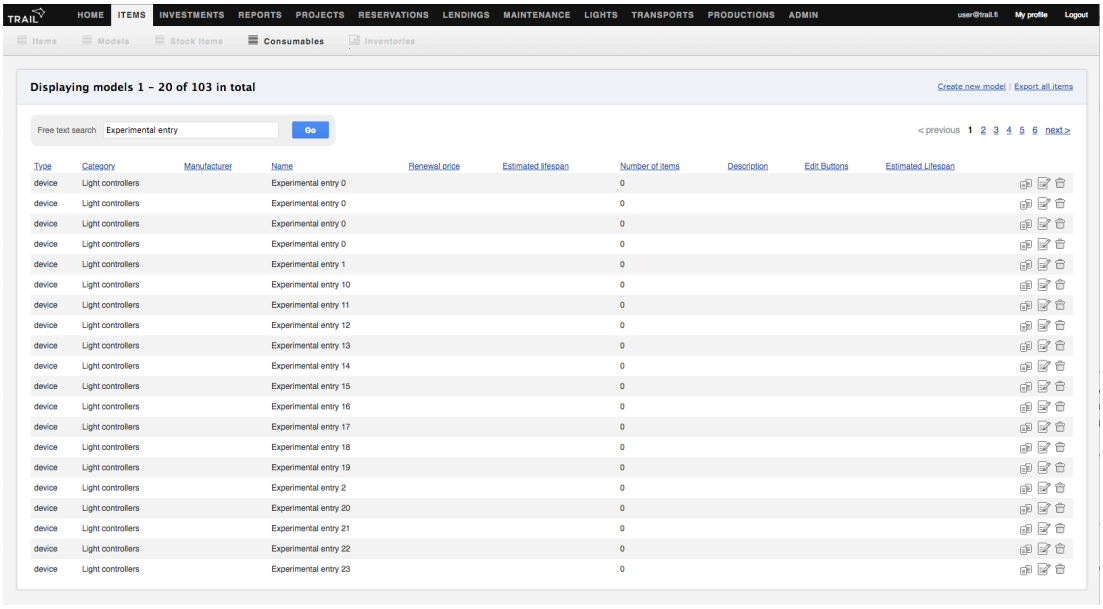

Image 2. Preceding consumable list view generated by the server

As it is seen in Images 1 and 2, both views component-wise consist of the navigation bar, search input and paginated table. The difference in the number of table columns is dictated by the altered design requirements and considered to be insignificant on the setting of the current investigation. Importantly, it can be seen in Image 1 that the constructed application was not replacing the server-side rendered views to the whole extent but was integrated into it. In fact, the SPA was responsible for the core functionality only and altering the body of the page but keeping the original navigation layout. Such an approach allowed the interface transition to be experienced by users in a continuous manner. That strategy is recommended to be followed until the SPA is capable of providing functionally identical set of views as the server and recreate the navigation segment.

It is essential to point out, that the new interface implementation is considered to correspond to proper production level standards, thereby it should not introduce an error during the process of benchmarking. In order to support the statement, the following sections provide answers the questions of how the core components of the view operate and how the SPA is integrated into the base of Trail application.

3.1.2 View internals

To begin with, it is important to review the SPA's entry point, presented in Appendix 2. At the top of the hierarchy the application, encapsulated in the App component, was augmented by other components, providing it the access to the content translations and global Redux store, which holds the global state. The resulting structure was abstracted out and rendered under the DOM element with the id of app. The App, the code for which is listed in Appendix 3, is responsible for loading of critical data on the startup and wrapping the Router along with MessageBucket, displaying error, success and warning messages to the end users. The Router represents a map between the current location path and the corresponding component to be rendered at the placement of the Router. Among all the entries, the one of particular interest is the following:

```
<Route exact path={'/models/consumables'} component={ConsumablesListPage} />
```

Listing 1. Route mapping between the location and component

In Listing 1 the path corresponds to component previously selected to be put under the test. ConsumableListPage (Appendix 4) serves as the table segment, visualized in Image 1. It can be seen in the code, that by using *connect* function ConsumableListPage

was attached to the state of a considerable size and several server API methods such as loadConsumables, for instance, which loads table data immediately as soon as the component is to mount or in the response to various user inputs:

```
componentWillMount() {
  this.fetchConsumablesByUrlHash();
}
```

Listing 2. Loading of table entries before the component mounts

The complete API method list with the corresponding actions for the domain of consumable models is located in a separate file (Appendix 5). The methods operate in a similar way, namely they mutate the application's state tree or reducer (Appendix 6) by dispatching actions with an optional payload on initialization, success or failure of the request sent. These, for instance, would trigger the page's loading indicator, load the list of the requested models, or call for an error message to be displayed, as it, to give an example, was done for actions of CONSUMABLES_LOAD_REQUEST shown in Listing 3, CONSUMABLES_LOAD_SUCCESS and CONSUMABLES_LOAD_FAILURE.

```
const loadConsumablesRequest = () => ({
  type: types.CONSUMABLES_LOAD_REQUEST,
});

case types.CONSUMABLES_LOAD_REQUEST: {
  return { ...state, loading: true, messages: [] };
}
```

Listing 3. Action with the corresponding state mutation

Server API methods rely on the API middleware (Appendix 7). It is the class of the methods of *get*, *post*, *put* and *delete*, encapsulating GET, POST, PUT and DELETE HTTP request methods across the application. They, in turn, are based on the modern Fetch API for making the AJAX requests. Reponses propagate back to actions, and custom HttpError and ApiError implementations are utilized on error, if necessary. Important to note, that because by default Fetch does not manage the cookies, it needs to be explicitly set to send the user credentials:

```
credentials: 'include'
```

Listing 4. Fetch option to send the credentials

In addition, it is important to supply the X-CSRF-Token header along with the requests, and in the context of the SPA being a standalone injection, the token is retrieved by

programmatically accessing the DOM of the Rails generated HTML tag, as it can be seen in Listing 5.

```
const extractCSRFToken = () => {
  const el = document.getElementsByName('csrf-token')[0];
  const CSRFToken = el ? el.content : '';

  return CSRFToken;
};
```

Listing 5. CSRF token extraction

The code in Listing 5 implies existing of the HTML meta tag with the content of the CSRF token in the current document. Consecutively, it is supplied within the header of each request.

Given above, and assuming the loadConsumable's GET request completed, the list of the consumable models is stored in the reducer (Appendix 6) and are retrieved by the selector of getConsumables in ConsumablesListPage (Appendix 4). The list page includes the custom-made component of ListTable (Appendix 8), the responsibility of which is to generate the table given there are consumable models supplied. To explain briefly, it is based on the Semantic UI library's Table component, and dynamically builds the columns and rows from the JSON configuration and business data provided. The component is assumed to be a highly expensive part of the view in terms of HTML parsing and painting. In fact, it is not only the text the table can display but the content of any complexity.

While exploring the code, it becomes obvious there are other components involved in a view construction. However, the current explanation is considered to be sufficient in the setting of the performance testing scope, explained in the later section. Thereby the rest of the elements are assumed to be insignificant. At this point, it is important to be aware of the operational flow of how the page loads: first, the navigation request made, then basic skeleton view is constructed, followed by an immediate request for a models list and displayed loading indicator, and, finally, when the response arrives, its data is mapped to the table and rendered on the screen.

3.1.3 Resource packaging and integration

In order to package the developed scripts and stylesheets into single files of the corresponding format, for them to be used in production-level context, the tool of Webpack was utilized. Appendix 9 lists the configurations applied. It can be seen that although the implementation in written in modern JavaScript notation, at the end of the pipeline it is compiled by Babel to a widely compatible code, which was, additionally, minified. Along with regular language, Fetch API is recommended to be converted into a function consumable by old web browsers. For these cases babel-polyfill and whatwg-fetch packages were integrated.

The Webpack configuration results in generation of three JavaScript output files based on the entry points defined and the shared bundle behind them. Similarly, the stylesheets are also packaged into a single CSS file. On top of that, client related translation files are copied to the scope of Rails application. The complete list of the SPA assets along with their sizes in original and compressed states are shown in Table 1.

File	**Format**	**Original size, KB**	**Compressed size, KB**
vendor.js	JavaScript	1020	295
main.js	JavaScript	946	227
navigation-search.js	JavaScript	11	3
main.css	CSS	824	301
common.json (en)	JSON	28	-
common.json (fi)	JSON	28	-

Table 1. SPA production files and their sizes

In Table 1 it is the compressed file size which is of high importance as it indicates the load to be transferred through the network. The biggest asset in the list is main CSS file of 301KB, mostly contributed by Semantic UI style definitions. The smallest, in turn, is the navigation search JavaScript file of 3KB. In the context of consumables list page, it does not play a role, because it represents a standalone functionality out of its scope. common.json files store English and Finnish textual content and they are fetched automatically by the i18next framework on the application mount. In contrast to the rest of the assets, translation files are not compressed.

The way generated JavaScript and CSS files are integrated into the Ruby on Rails application is shown in Listing 6.

```
- content_for :content do
  = include_stylesheets :react_main
  %div#app

  = include_javascripts :react_main
```

Listing 6. A Haml-based React template for a Ruby on Rails view

Listing 6 presents a Haml template including SPA related stylesheets at the top and the scripts at the bottom. Naming of react_name defines grouping of the files in the Rails's asset configuration setting. Important to note that the application stores the assets in the public folder of its own, meaning the resource request and the server share an identical domain. Given the template, Rails can render it in the response to a specified URL path. When its rendered and the assets are loaded, the SPA is bootstrapped and div element with the id of app becomes a central point of React environment.

3.2 Benchmark design

3.2.1 Metrics

To restate the first research question, the understanding is needed of how introduction of the SPA affects the performance of the application from the perspective of the user. In order to capture the characteristics defining the performance in the context of the page navigation, the following metrics were chosen: "DOM loading", "DOM complete", "usable", "first useful" and "second useful".

"DOM loading" indicates the moment the HTML is loaded and is ready to be passed to a rendering engine for processing. When it finishes, the "DOM complete" event is marked. Next timing point to be waited for are "usable" and "first useful". The former defines the moment, when the interface has a property of being interacted in any sensible kind, and the latter represents the moment, when the integral data is presented, which is the list of consumable models for the current case. In the context of SSR both "usable" and "first useful" are mapped to loadEventEnd event. They are identical as the server-rendered document does not have an intermediary state – when it is usable it is immediately useful. The situation for the SPA is completely different, however, because it is of a stateful

nature. It is possible for it to be usable first and useful after a certain amount of time, because it is able to render a usable interface while asynchronously doing complicated requests, the arrival of which along with the following rendering indicate the moment of it being useful. To provide an example, Image 3 shows the SPA at the state of being usable:

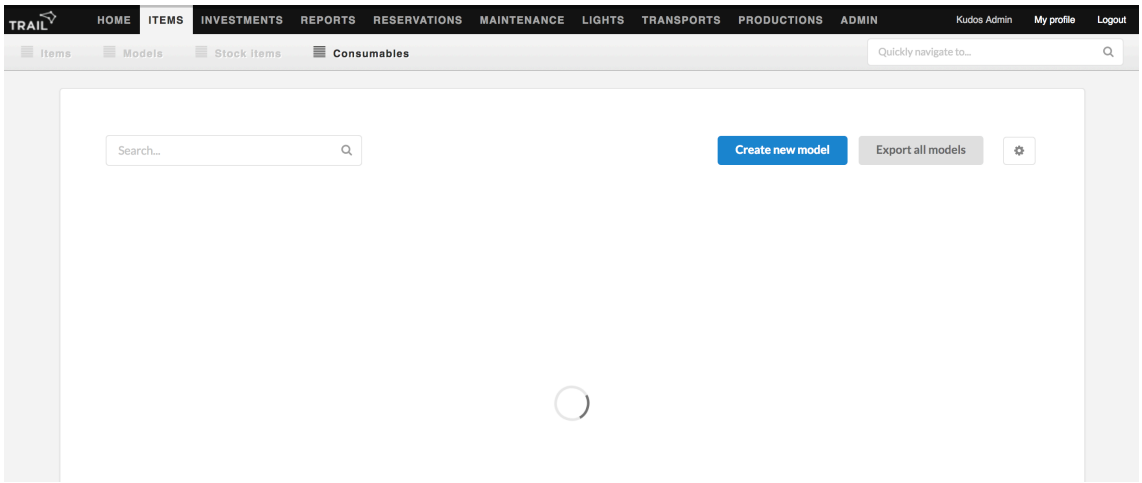

Image 3. The usable state of the SPA

It can be seen in Image 3 that while the table is loading, which is indicated by an animated spinner, the interface is usable and interactive. For instance, the search input could be altered, or the button clicked. When the table is loaded, the view becomes useful because it fully satisfies the requirements it imposes as it is shown in Image 1.

The last and key metric to be considered is “second useful” because it leverages the advantages of the client-rendered interface. It is to mark the timing required for a consecutive reloading of table data immediately after “first useful” was identified. That would, for example, represent a use case when a user loads the page for first time and does not locate the required model in the table loaded. Then they would modify the request using the search functionality and reload the table. To point out, in such situation the server-side rendered architecture would require a complete page refresh, while the SPA to trigger a single AJAX request. While the “first useful” is assumed to be time consuming for the SPA, the “second useful” is expected to be more performant because the application is considered to be already loaded in the browser.

3.2.2 Data collection

3.2.2.1 Client data

The script in Appendix 10 was injected to the HTML page and shows how the metric related data was collected. For the purpose PerformanceTiming API was used. It is seen that when the event of completeObservation is triggered, the data is allocated to the browser's session storage and page is automatically reloaded thus denoting an observation to be completed. Number of observations is defined by observationsLeft variable, which is defined in the stage benchmark setup through a console in the following way:

```
sessionStorage.setItem('data', JSON.stringify([]));
sessionStorage.setItem('observationsLeft', 10);
location.reload();
```

Listing 7. Benchmark setup

In Listing 7 the benchmark setup is shown. The data is defined to be an empty array, and the number of observations is set to 10. When the code is executed in the browser console, the page is automatically reloaded and the script (Appendix 10) is to have appropriate conditions to start repeatedly doing the observations.

It is shown in Appendix 10 that while for both test cases "DOM loading" and "DOM complete" are mapped to the domLoading and domComplete performance entries, the rest of the metrics differ. In the context of server-rendered page, "usable" and "first useful" are set to be equal to loadEventStart performance entry. "Second useful" is defined to be double of loadEventStart for the sake of simplicity under the assumption that the second load would require the same amount of time as the first one.

In the situation of SPA testing the performance entries need to be marked and accessed manually by the API's mark() and getEntriesByName() methods, respectively. To assess the measurements correctly there are two React components (Appendices 11 and 12) were added. Usable.jsx (Appendix 11) loads when the ConsumablesListPage.jsx (Appendix 4) mounts and sets the callback before the browser's repaint using window.requestAnimationFrame() method. The repaint process is expected to happen when the loading indicator (Image 3) starts to be painted. At that point, consequently, the "usable" metric is assessed. Useful.jsx (Appendix 12) operates in a similar manner. Specifically, it mounts when loadConsumables methods returns the response, and immediately sets

the callback on repaint, which is called when the table painting is initiated, thus marking the timing of “first useful”. Immediately after that the table is explicitly loaded anew by the passed this.props.loadListAnew() function. Then, similarly, it waits for the repaint to start and marks the “second useful” timing when the table painting process starts again. At this stage, all the required metrics are measured, and the observation is ready to be completed by dispatching an event for execution of the data collecting script (Appendix 10) as it is shown in Listing 8.

```
performance.mark('secondUseful');

const completeObservation = new Event('completeObservation');
window.dispatchEvent(completeObservation);
```

Listing 8. Marking of “second useful” and completing the observation by event dispatching

In turn, the moment when the observation is to be completed for the server-side rendered interface is defined by the window’s load event:

```
window.addEventListener('load', () => {
  const completeObservation = new Event('completeObservation');
  window.dispatchEvent(completeObservation);
})
```

Listing 9. Dispatching an observation triggering event on window load

Listing 9 represents dispatching of completeObservation event when the window is loaded in the setting of server-side rendered view. The code is included to the HTML page and sets up an event listener before the load is completed.

3.2.2.2 Server data

Due to the lack of resources server data collection in the current research is significantly limited to the cloud monitoring tool generated graphs only. Namely, the ones describing CPU utilization and system load. Imposed restrictions result in absence of details and quality data, but, nevertheless, are considered to partly able to provide a high-level perspective on the differences between the impact caused by investigated rendering architectures, thus providing a basic answer to the question raised earlier. To clarify, the monitoring tool used for the server under the test was IDERA by CopperEgg.

3.2.3 Testing cases and environment

To recall, size of the table is dynamic. User may choose the maximum number of entries per page among the options of 20, 50 and 100. The goal is to measure performance of the page with all available options to the user along with artificial settings of 0 and 300 items per page. Those are the independent variables, while the dependent one is timing in milliseconds. With each number of page entries, the views were tested with the HTTP cache turned on and off by using the browsers' developer tools. Important to note that, unlike the assets, the request for the list of consumable models was never cached in order to simulate the uniqueness of the data with each page load. It was done with the response header of Cache-Control set to no-store later augmented by a random number parameter (Listing 10) with each request, because Safari did not operate properly with the cache control header. Specifically, in the context of SPA, it always cached the first request, but not the second one.

```
const resource = `/models/consumables.json?nocache=${Math.random()}`;
```

Listing 10. Avoiding cache by attaching a random parameter value with each request

To gather the perspective on the server performance, similar set of number of entries per page was chosen with the corner case of 1000, however. Besides, it is only the uncached environment which was tested on the server.

Number of observations for each case is 100, given which the means and standard deviations were calculated.

The designed benchmark was executed on the laptop of MacBook Pro 2016 running on macOS Mojave 10.14.1 with the CPU of 2,9 GHz Intel Core i5, RAM of 16 GB 2133 MHz LPDDR3 and graphics card of Intel Iris Graphics 550 1536 MB.

The browsers under the test were Google Chrome 70.0.3538.102, Mozilla Firefox Quantum 63.0.3 and Apple Safari 12.0.1. Such a selection was dictated by their market share and the fact that each of them operate on different rendering and JavaScript engines. Additionally, it would be beneficial to examine the Microsoft's Edge browser, because it too runs on unique pair of rendering and scripting engines, EdgeHTML and Chakra, respectively. However, due to the lack of resources, such a case was not concern of the current research but could be considered as a part of the further one.

The browsers communicated to the Ruby on Rails 3.2 application on a remote Apache HTTP server 2.2.22 with Phusion Passenger 4.0.53 integration, set up on Ubuntu 12.04.4 LTS Linux distribution. To point out, the data was specifically prepared to be of production quality.

To manage the bandwidth in a consistent manner and simulate a more realistic testing environment the throttling was introduced by using Charles 4.2.7, a web debugging proxy tool. The 8 Mbps ADSL2 preset was selected with the following properties: download bandwidth of 8192Kb/sec with minimal round-trip latency of 40ms. These settings fit the underlying assumption of the average quality of the connection the customers of the case company use. To note, the focus of the investigation is a desktop application, thereby it is the corresponding kind of network is expected rather than a mobile one. It is important to clarify that the primary reason for using a standalone throttling tool is the lack of such functionality in Safari browser. In comparison, for instance, it is possible to set throttling settings directly in the developer tools of both Chrome and Firefox browsers.

On top of that, importantly, the SPA integration was tailored for the purpose of yielding a performance data without an overhead and imitating the product of complete and standalone kind. Specifically, the original navigation bar along with corresponding stylesheets and scripts were ignored. In that setting, it was assumed that the SPA generated assets are self-contained, and that absence of visually rendered navigation bar was insignificant.

4 Results

4.1 User perspective

4.1.1 Browser comparison

Figures 10 and 11, based on data from Appendices 13 and 14, show the comparison of benchmark results for all the browsers with and without resource cache, respectively.

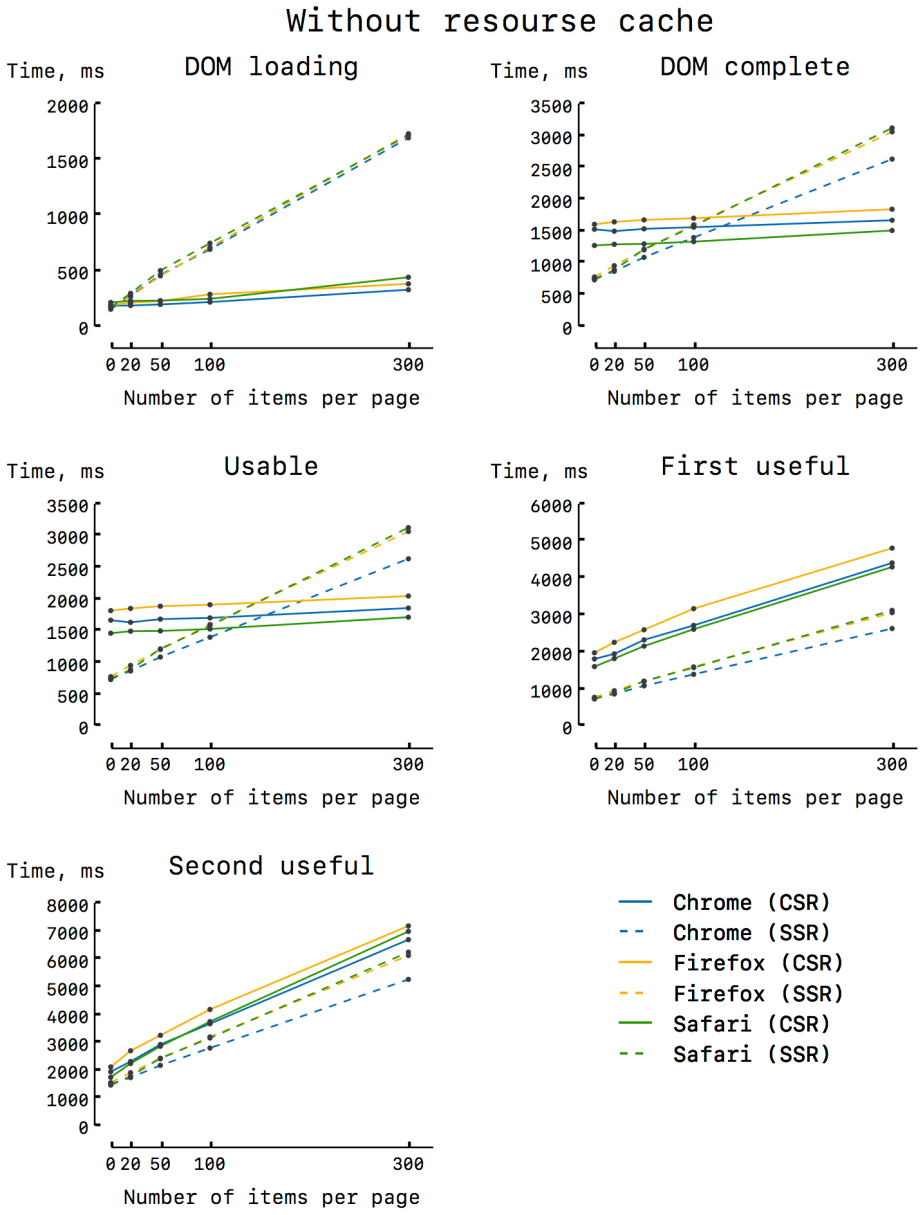

Figure 10. Browsers load comparison without resource cache

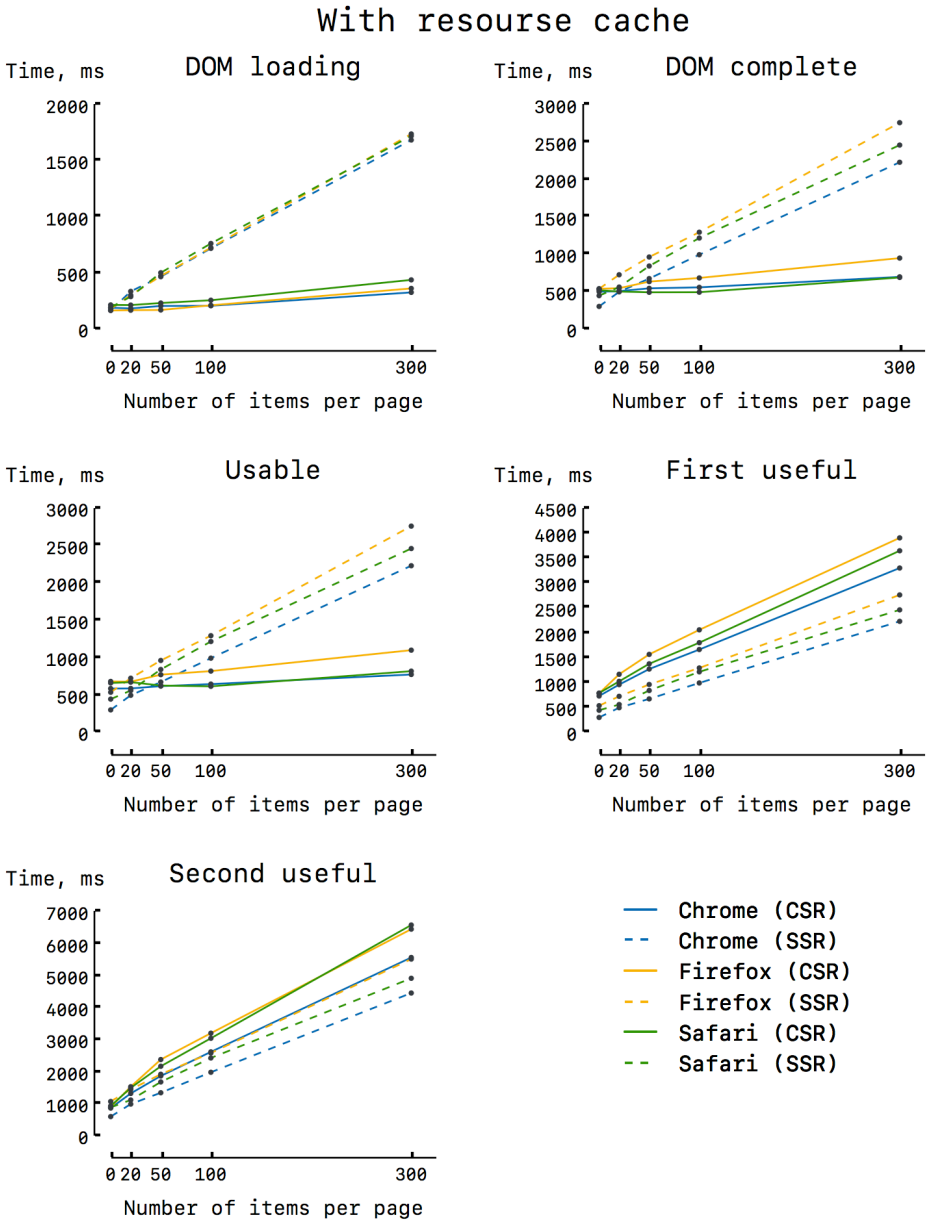

Figure 11. Browsers load comparison with resource cache

By investigating Figures 10 and 11, without going into the details, it can be seen that all the browsers manifest similar, close to linear, trends in the context of CSR and SSR comparison. It is essential to point out, that the characteristics of results presentation are interchangeable, thereby it is totally sufficient to focus on a single browser data to support the upcoming discussion. Because it is difficult to reason which browser is more performant that requires a deeper approach deserving to be the topic of further development. In fact, there is significant amount of deviation introduced at different stages of measurement, especially in the case of Mozilla Firefox. The differences are acceptable, however,

as the environment under the test is of high complexity and is expected to impose a certain amount of error. There are multiple vectors of intricacy such as engine internals differences, garbage collection pauses, multi-threading etc. What is more important for current investigation is that the data provides a basis for a solid assumption, that in the setting of a single browser CSR and SSR behave relatively similar as in the context of other browsers.

Given the above, Google Chrome was chosen for further results exploration, because it clearly demonstrated the most stable and consistent results across all levels of benchmarking.

4.1.2 Google Chrome

Figures 12 and 13, display the benchmark results for Google Chrome with and without resource cache, respectively. There are two graphs on each of the charts. The blue graph shows the performance of CSR architecture, while the green one deals with the performance of SSR.

“DOM loading” measurements for both SSR and CSR are similar in comparison to cached and uncached environments. While SSR is more performant when there are 0 items on the page, the time required to receive the DOM increases significantly as the number of items per page grows. Meanwhile, the CSR values remain relatively stable. “DOM complete” results demonstrate similar to “DOM loading” trends. What is different is the point of intersection at which CSR starts showing better results compared to SSR. Without resource cache the point is at around 130 items per page with approximated time of 1600ms. Introduction of cache alters the point down to 20 items with the timing close to 500ms.

There is practically none of the differences between the measurements of “DOM complete” and “usable” for SSR as the latter immediately follows the former one. CSR, in turn, shows a slight and consistent gap between those, of around 150ms without cache and 90ms with. Thereby the results of “usable” are close to the results of “DOM Complete” and generally shows the same tendency.

“First useful” results show no sign of intersection. In fact, the gap between CSR and SSR in uncached setting is around 1000ms at 0 items per page and it almost doubles at 300

items per page. By turning cache on the tendency stays the same, while the gap decreases approximately by 60%. The results are comparable to the data for “Second useful”, presenting a smaller gap, which is, however, is not eliminated even with the resource cache.

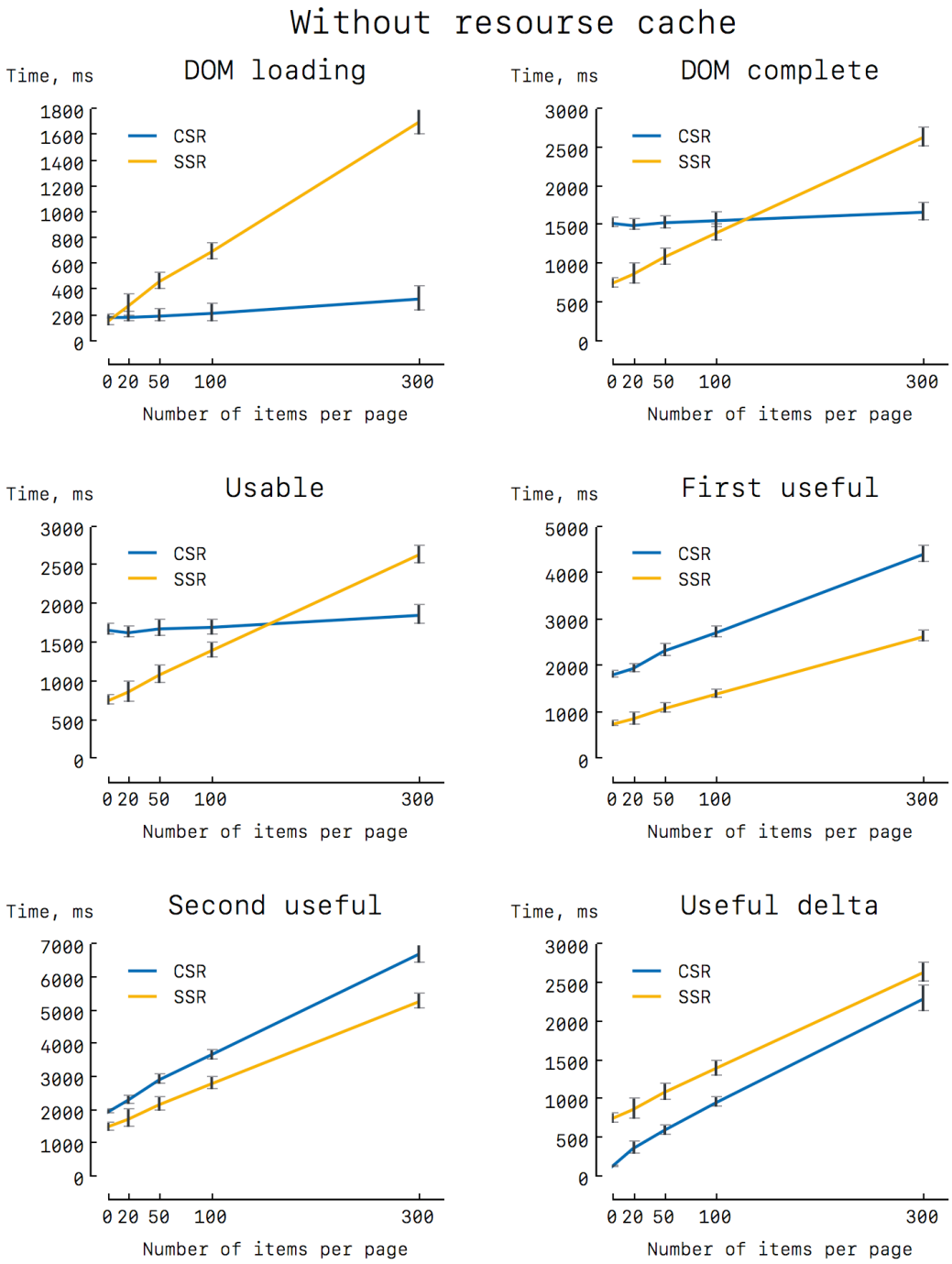

Figure 12. Google Chrome load without resource cache

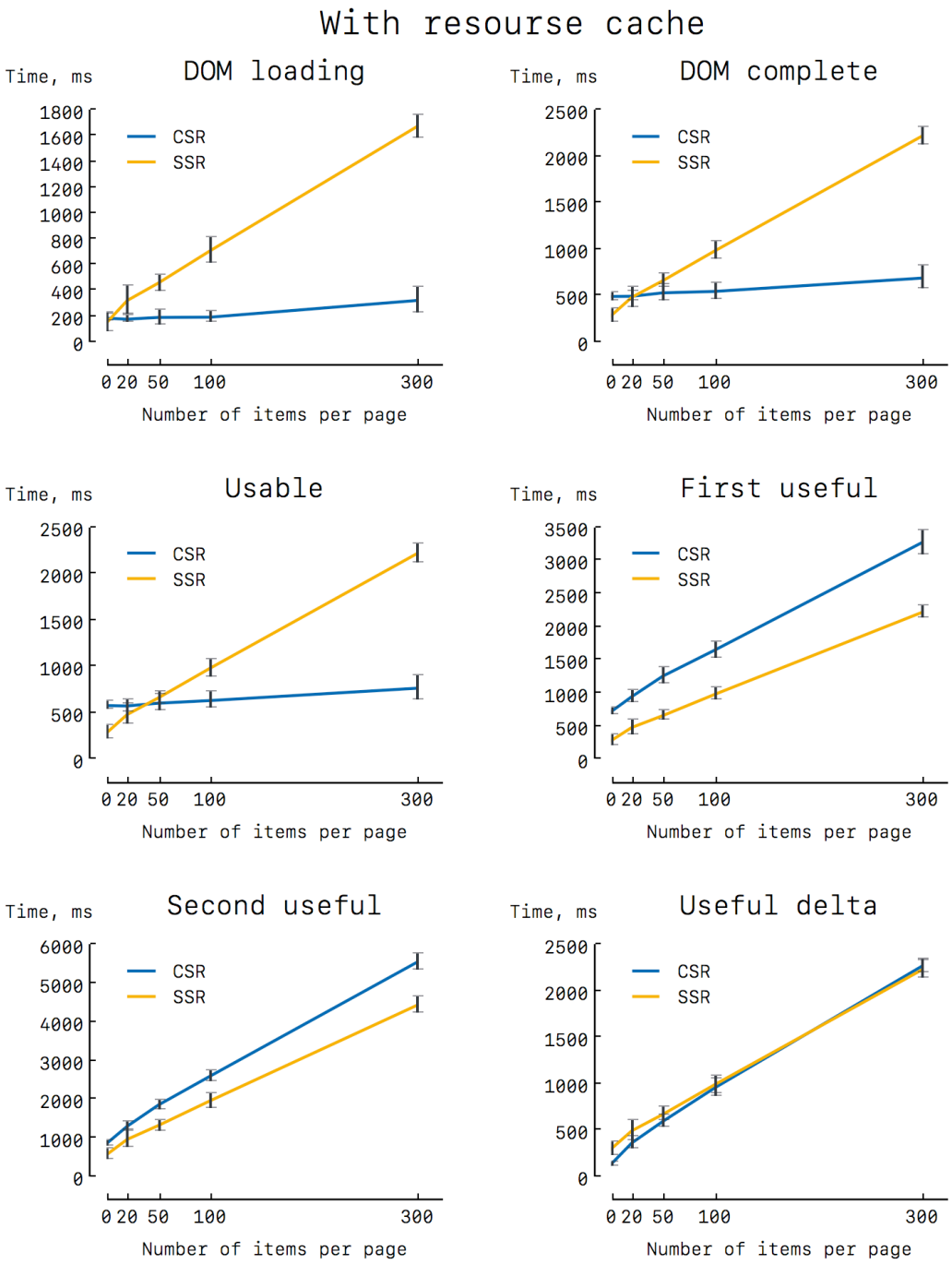

Figure 13. Google Chrome load with resource cache

Useful delta is difference between second and first useful. It shifts the point of reference and shows the results from the perspective of the SPA already being loaded in the browser and being allocated in continuous “usable” state. It can be seen that in such a setting the CSR and SSR graphs swap the places when there is no cache but show near values in cached context with the CSR being slightly faster with lower page entries.

4.2 Server perspective

Figures 14 and 15 display CPU utilization and system load on the server under the numerous successive requests initiated by SSR. Correspondingly, Figures 16 and 17 are related to CSR. The numbers 0, 20, 50, 100 etc. written on top of the chart sections indicate the number of items per page included in the page loaded.

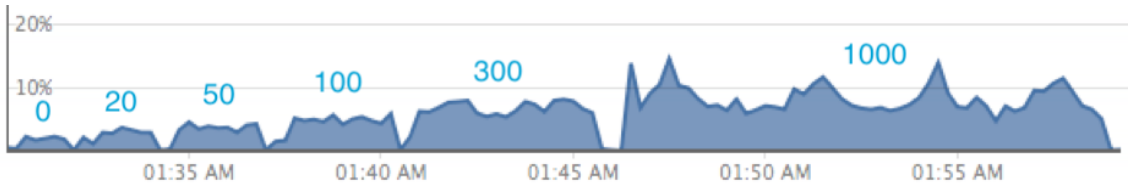

Figure 14. CPU utilization by SSR

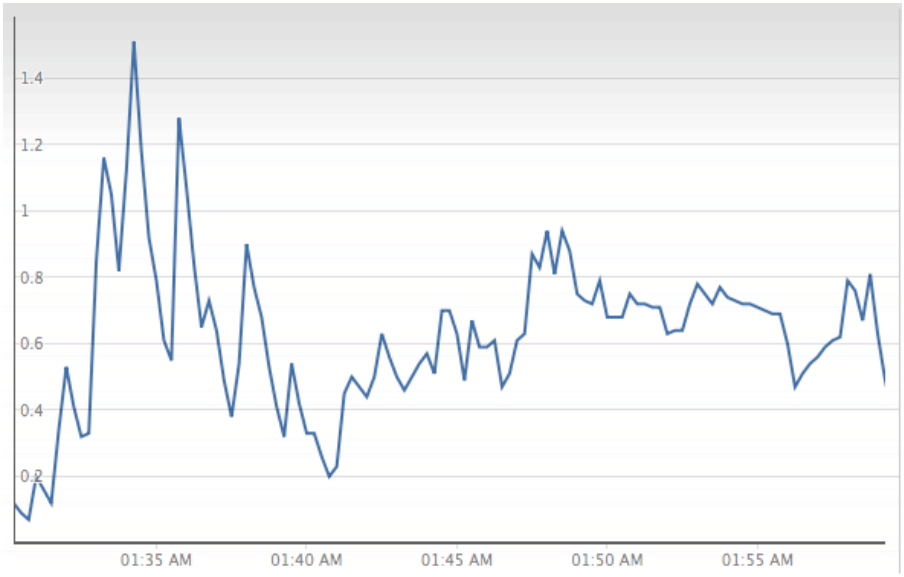

Figure 15. System load by SSR

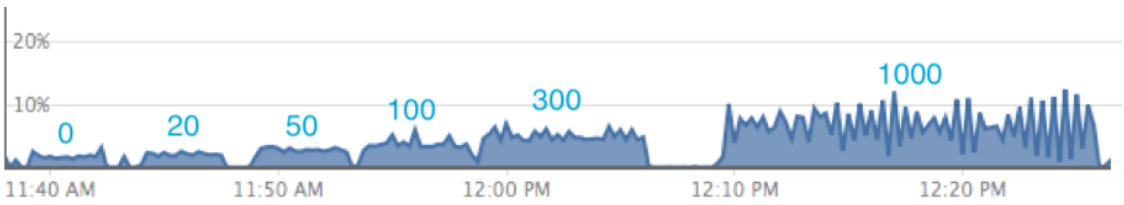

Figure 16. CPU utilization by CSR

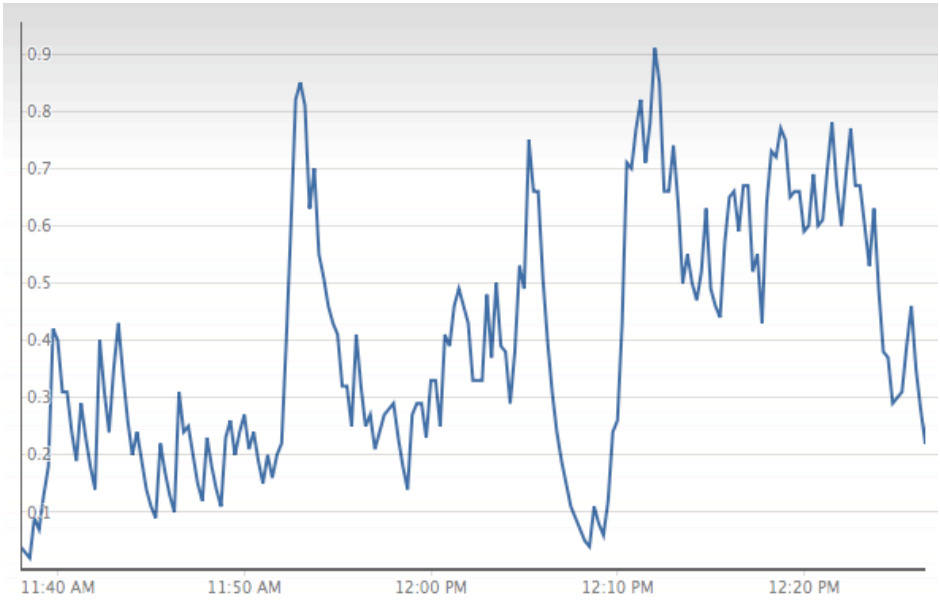

Figure 17. System load by CSR

By comparing the CPU utilization Figures 14 and 16 it is considered that the utilization is 1-2% lower in the case of CSR, which can be clearly seen for all the page entries with an exception for 1000, where it is problematic to deduce the difference. System load data represents an issue as well due its inconsistency towards the request pattern. What could be stated, however, is that CSR did not cause load higher than 0.9, while SSR spiked the load at maximum of 1.5.

5 Discussion

Given the results it could be seen that when the web page was loaded for the first time, the CSR architecture was not able to perform better than the SSR in terms of being useful e.g. displaying the table of consumable models. Also, it could not demonstrate impressive results in its quickness of being usable or interactive. Specifically, if the assets were cached, the CSR could surpass SSR at around 40 items per page. Considering 85% of Trail users prefer to list 50 items and less, and the fact that at 50 items the gap between CSR and SSR “usable” reached only 50ms, CSR performance advantage is practically unnoticeable. Given all that it can be stated that the CSR is comparatively not capable of achieving a quick startup time, which is of high importance for user retention and conversion in online retail business, for instance [3].

To recall, half of the users expect the page to be loaded in 2 seconds or less [4]. Both SSR and CSR architectures demonstrated ability to fit such a constraint in a real use case scenario if assets were cached. If not, CSR was able to serve the page in under 3 seconds, which is the edge when half of the users are ready to drop off the page [4]. Arguably, it could be considered that most of the time Trail users utilize browser’s cache for the assets. It is due to comparatively low frequency of releases, which means users do not need to reload newly generated packages frequently. At this point it is also important to be aware of the limitation of the current research, namely the SPA implementation was far from being complete and examined as an ideal benchmark candidate. In other words, the size of the assets is expected to increase as long as the new features are added, thus in the cached context the loading time would grow as well.

The CSR performance situation drastically changed when the perspective was shifted, and it was assumed the SPA was already loaded in the browser and was interacted through a frame of native desktop experience. At this stage the application was constantly usable, and was expected to be quicker than SSR, because any data retrieval became the question of an asynchronous request rather than a complete page reload. With the 30-160ms performance advantage over 100 items per page range, the CSR potentially manifested a faster user experience for 100% of the Trail users. What was surprising during the investigation is CSR’s performance outrun, which happened to be lower than expected. Thereby it was assumed that React, as a complex front-end framework, brings a performance overhead, the detailed investigation of which could be a goal of the further research. If it is true, important to point out that it can be considered as a

cost for advantages the framework provides such as code maintainability, project development efficiency, advanced user experience etc. [41].

Caused server impact is considered to be smaller for CSR. Despite the lack of data precision, it is possible to point out approximately a 1% decrease in CPU load. Although the change does not look significant alone, in the context of thousands of users initiating simultaneous requests and calling for multiple CPU cores, it represents a huge value for the business. Indeed, the process of page assembling was delegated to the client's machine, and the major responsibility of the server becomes processing of API requests. It could be even more efficient if the assets were delivered not by the server, but by CDN, which additionally would map the client to the closest server available. The lower the impact, the higher the server capacity which consequently, results in a safer and cheaper application.

6 Conclusion

The SPA was implemented by using React and Redux libraries along with the best practices in the context of available resources. One SPA view was selected and compared to the corresponding previous implementation from the perspective of performance. Google Chrome was selected as a major reference for benchmark results. Consequently, it was concluded that depending on the context of usage, new interface implementation yields various timings but operates within the acceptable boundaries. In uncached and cached settings for 100 items per page the view is useful under 3 and 2 seconds respectively. Nevertheless, it is higher compared to SSR, for which the corresponding timing values are 1.5 and 1 seconds. Given the SPA is loaded in the browser, it is able to outperform the SSR, but with a small margin of 30ms. Considering the perspective of the server impact it is approximated that CSR improves the CPU's capacity by 1% and generally forces a lower system load.

References

1 Grigorik I. High-Performance Browser Networking. Sebastopol, CA: O'Reilly Media; 2013.

2 Wagner J. Why Performance Matters [online]. URL: https://developers.google.com/web/fundamentals/performance/why-performance-matters/. Accessed on 5 October 2018.

3 Meder S, Antonov V, Chang J. Driving user growth with performance improvements [online]. URL: https://medium.com/@Pinterest_Engineering/driving-user-growth-with-performance-improvements-cfc50dafadd7. Accessed 5 October 2018.

4 Google. The need for mobile speed: How mobile latency impacts publisher revenue [online]. URL: https://www.thinkwithgoogle.com/intl/en-154/insights-inspiration/research-data/need-mobile-speed-how-mobile-latency-impacts-publisher-revenue/. Accessed 5 October 2018.

5 Nielsen J. Website Response Times [online]. URL: https://www.nngroup.com/articles/website-response-times/. Accessed 5 October 2018.

6 Ben Schwarz. Beyond the Bubble: Real world performance [online]. URL: https://building.calibreapp.com/beyond-the-bubble-real-world-performance-9c991dcd5342. Accessed 5 October 2018.

7 GSM Association. The Mobile Economy 2018. 2018

8 HTTP Archive. State of the Web. URL: https://beta.httparchive.org/reports/state-of-the-web. Accessed on 5 October 2018.

9 Addy Osmani. The Cost of JavaScript in 2018 [online]. URL: https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4. Accessed on 13 October 2018.

10 Fixed asset management software in cloud - Trail Systems [online]. URL: https://www.trail.fi. Accessed on 5 October 2018.

11 W3C. Navigation Timing Level 2 [online]. URL: https://www.w3.org/TR/navigation-timing-2. Accessed on 18 October 2018.

12 Fielding R, Reschke J. RFC7230. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing [online]. URL: https://tools.ietf.org/html/rfc7230. Accessed on 13 October 2018.

13 Gourley D, Totty B. HTTP: The Definitive Guide. Sebastopol, CA: O'Reilly Media; 2002.

14 Dierks T, Rescorla E. RFC5246. The Transport Layer Security (TLS) Protocol. Version 1.2 [online]. URL: https://tools.ietf.org/html/rfc5246. Accessed on 13 October 2018.

15 Kozierok CM. The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference. San Francisco, CA: No Starch Press, Inc.; 2005.

16 Belshe M. More Bandwidth Doesn't Matter (much) [online]. URL: https://docs.google.com/a/chromium.org/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2I4YzI3NzE2. Accessed on 16 October 2018.

17 Allman M, Paxson V. RFC5681. TCP Congestion Control [online]. URL: https://tools.ietf.org/html/rfc5681. Accessed on 20 October 2018.

18 Wikimedia Commons. File: TCP Slow-Start and Congestion Avoidance.svg [online]. URL: https://commons.wikimedia.org/wiki/File:TCP_Slow-Start_and_Congestion_Avoidance.svg. Accessed on 20 October 2018.

19 Fielding R, Reschke J. RFC7231. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content [online]. URL: https://tools.ietf.org/html/rfc7231. Accessed on 20 October 2018.

20 Steele-Idem P. Async Fragments: Rediscovering Progressive HTML Rendering with Marko [online]. URL: https://www.ebayinc.com/stories/blogs/tech/async-fragments-rediscovering-progressive-html-rendering-with-marko. Accessed on 27 October 2018.

21 The Chromium Projects. HTTP Pipelining [online]. URL: https://www.chromium.org/developers/design-documents/network-stack/http-pipelining. Accessed on 27 October 2018.

22 Browserscope. Network [online]. URL: http://www.browserscope.org/?category=network&v=1. Accessed on 27 October 2018.

23 Fielding R, Nottingham M, Reschke J. RFC7234. Hypertext Transfer Protocol (HTTP/1.1): Caching [online]. URL: https://tools.ietf.org/html/rfc7234. Accessed on 27 October 2018.

24 Souders S. High Performance Web Sites. Sebastopol, CA: O'Reilly Media; 2007.

25 Garsiel T. How browsers work [online]. URL: https://taligarsiel.com/Projects/howbrowserswork1.htm. Accessed on 27 October 2018.

26 StatCounter. Browser Market Share Worldwide – October 2018 [online]. URL: http://gs.statcounter.com/. Accessed on 1 November 2018.

27 Bynens M, Meurer B. JavaScript engine fundamentals: Shapes and Inline Caches [online]. URL: https://mathiasbynens.be/notes/shapes-ics. Accessed on 1 November 2018.

28 Grigorik I. Critical Rendering Path [online]. URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/. Accessed on 27 October 2018.

29 W3. HTML5.2: The HTML Syntax [online]. URL: https://www.w3.org/TR/html5/syntax.html#the-end. Accessed on 1 November 2018.

30 Fink G, Flatow I. Pro Single Page Application Development: Using Backbone.js and ASP.NET. New York, NY: Apress Media; 2014.

31 Asleson R, Schutta NT. Foundations of Ajax. New York, NY: Apress Media; 2006.

32 Strimpel J, Najim M. Building Isomorphic JavaScript Apps. Sebastopol, CA: O'Reilly Media; 2016.

33 Garrett JJ. Ajax: A New Approach to Web Applications [online]. URL: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. Accessed on 1 October 2018.

34 AirbnbEng. Isomorphic JavaScript: The Future of Web Apps [online]. URL: https://medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc. Accessed on 4 November 2018.

35 Twitter. Improving performance on twitter.com [online]. URL: https://blog.twitter.com/engineering/en_us/a/2012/improving-performance-on-twittercom.html. Accessed on 4 November 2018.

36 Navis G. The Architecture No One Needs [online]. URL: https://www.gregnavis.com/articles/the-architecture-no-one-needs.html. Accessed on 4 November 2018.

37 Petrina T. Architecture no one needs is server side templating [online]. URL: https://itnext.io/architecture-no-one-needs-is-server-side-templating-78331391274. Accessed on 4 November 2018.

38 Gordon EK. Isomorphic Web Applications. Shelter Island, NY: Manning Publications; 2018.

39 Baxter K. Making Netflix.com Faster [online]. URL: https://medium.com/netflix-techblog/making-netflix-com-faster-f95d15f2e972. Accessed on 4 November 2018.

40 Grigoryan A. The Benefits of Server Side Rendering Over Client Side Rendering [online]. URL: https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8. Accessed on 4 November 2018.

41 Sethi A. The Baseline Costs of JavaScript Frameworks [online]. URL: https://blog.uncommon.is/the-baseline-costs-of-javascript-frameworks-f768e2865d4a. Accessed on 28 November 2018.

Complete list of installed packages (package.json)


```
{
 "dependencies": {
    "animated": "^0.2.1",
    "babel-polyfill": "^6.26.0",
    "d3-array": "^1.2.1",
    "d3-axis": "^1.0.8",
    "d3-color": "^1.0.3",
    "d3-dsv": "^1.0.7",
    "d3-format": "^1.2.0",
    "d3-interpolate": "^1.1.6",
    "d3-scale": "^1.0.6",
    "d3-selection": "^1.1.0",
    "d3-tip": "^0.7.1",
    "history": "^4.5.1",
    "i18n-iso-countries": "^3.7.3",
    "i18next": "^10.0.1",
    "i18next-xhr-backend": "^1.4.3",
    "lodash": "^4.17.4",
    "moment": "^2.22.1",
    "numeral": "^2.0.6",
    "qs": "^6.5.1",
    "react": "^16.1.1",
    "react-addons-shallow-compare": "^15.6.0",
    "react-dates": "^15.1.0",
    "react-dom": "^16.1.1",
    "react-dropzone": "^4.2.7",
    "react-i18next": "^6.0.6",
    "react-redux": "^5.0.6",
    "react-router-dom": "^4.2.2",
    "react-transition-group": "^2.2.1",
    "redux": "^3.7.2",
    "redux-actions": "2.3.0",
    "redux-form": "^7.1.2",
    "redux-thunk": "^2.2.0",
    "semantic-ui-css": "2.2.12",
    "semantic-ui-react": "0.76.0",
    "shortid": "^2.2.8",
    "whatwg-fetch": "^2.0.4"
  },
  "devDependencies": {
    "babel-core": "^6.26.2",
    "babel-eslint": "^7.1.1",
    "babel-jest": "^23.4.2",
    "babel-loader": "^7.1.4",
    "babel-minify-webpack-plugin": "^0.3.1",
    "babel-plugin-transform-builtin-extend": "^1.1.2",
    "babel-preset-env": "^1.6.1",
    "babel-preset-react": "^6.24.1",
    "babel-preset-stage-2": "^6.24.1",
    "copy-webpack-plugin": "^4.1.1",
    "css-loader": "^0.28.4",
    "eslint": "^4.6.1",
    "eslint-config-airbnb": "^15.1.0",
    "eslint-plugin-import": "^2.7.0",
    "eslint-plugin-jest": "^21.21.0",
    "eslint-plugin-jsx-a11y": "^5.1.1",
    "eslint-plugin-react": "^7.3.0",
    "extract-text-webpack-plugin": "^3.0.2",
    "file-loader": "^0.11.1",
    "html-webpack-plugin": "^2.28.0",
```

```
    "jest": "^23.5.0",
    "less": "^2.7.2",
    "less-loader": "^4.0.3",
    "react-test-renderer": "^16.3.2",
    "style-loader": "^0.18.2",
    "transfer-webpack-plugin": "^0.1.4",
    "url-loader": "^0.5.8",
    "webpack": "^3.11.0",
    "webpack-dev-server": "^2.3.0",
    "webpack-merge": "^4.1.0"
  }
}
```

Entry point of the SPA (client.jsx)

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { I18nextProvider } from 'react-i18next';
import 'react-dates/initialize';

import 'semantic-ui-css/semantic.min.css';

import configureStore from './store/configureStore';
import i18n from './i18n';

import App from './containers/App';

import './assets/stylesheets/main.less';


const store = configureStore();

const router = (
  <I18nextProvider i18n={i18n}>
    <Provider store={store}>
      <App />
    </Provider>
  </I18nextProvider>
);

const app = document.getElementById('app');
if (app !== null) {
  ReactDOM.render(router, app);
}
```

Main application component (App.jsx)

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { BrowserRouter } from 'react-router-dom';
import isEmpty from 'lodash/isEmpty';
import moment from 'moment';
import { translate } from 'react-i18next';

import { loadCurrentUser } from 'actions/currentUser';
import { getCurrentUser } from 'reducers/currentUser';
import { loadOrganization } from 'actions/organization';
import { dismissMessage } from 'actions/globalMessages';
import MessageBucket from 'components/messages/MessageBucket';
import i18n from 'i18n';
import Router from './Router';


class App extends Component {
  componentWillMount() {
    const { user } = this.props;

    if (!isEmpty(user)) {
      i18n.changeLanguage(user.locale);
      moment.locale(user.locale);
      moment.weekdays(true);
    }

    this.props.loadCurrentUser();
    this.props.loadOrganization();
  }

  render() {
    const { messages, dismiss, t } = this.props;

    return (
      <div>
        <BrowserRouter>
          <div>
            <MessageBucket
              messages={messages}
              dismiss={dismiss}
              t={t}
            />
            <Router />
          </div>
        </BrowserRouter>
      </div>
    );
  }
}

const mapStateToProps = state => ({
  user: getCurrentUser(state),
  messages: state.globalMessages.entries,
});

const mapDispatchToProps = {
  loadCurrentUser,
  loadOrganization,
  dismiss: dismissMessage,
};
```

```
export default translate('', { wait: true })(
  connect(mapStateToProps, mapDispatchToProps)(App)
);
```

Page displaying the list of consumable models (ConsumablesListPage.jsx)

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { Segment, Grid, Button } from 'semantic-ui-react';
import isEmpty from 'lodash/isEmpty';
import { translate } from 'react-i18next';

import * as currentUserSelectors from 'reducers/currentUser';
import * as consumablesSelectors from 'reducers/consumables';
import * as consumablesActions from 'actions/consumables';

import { updateUrlHash, extractUrlHash } from 'utils/urlHelpers';

import * as modelsListsColumns from 'componentsConfigs/modelsListsColumns';
import DataTableHeader from 'components/tables/DataTableHeader';
import ListTable from 'components/tables/ListTable';
import PaginationButtons from 'components/tables/PaginationButtons';
import MessageBucket from 'components/messages/MessageBucket';
import SearchBox from 'components/utils/SearchBox';

import TableSettingsPopup from 'components/tables/TableSettingsPopup';

class ConsumablesListPage extends Component {
  componentWillMount() {
    this.fetchConsumablesByUrlHash();
  }

  getDisplayedConsumableColumns() {
    const { t, displayedColumnNames } = this.props;

    const columns = modelsListsColumns.getColumnsConfigs(t);
    const displayedColumns = [];

    displayedColumnNames.forEach((key) => {
      const column = columns[key];
      if (column) { displayedColumns.push(column); }
    });

    return displayedColumns;
  }

  fetchConsumablesByUrlHash() {
    const { location, updatePage, updateSearchInput, updateOrder } =
this.props;
    const { page, search, orderBy, orderDirection } = extractUrlHash(loca-
tion);

    updatePage(page);
    updateSearchInput(search);
    updateOrder(orderBy, orderDirection);

    this.fetchConsumables();
  }

  async fetchConsumables() {
    await this.props.loadConsumables();
    this.updateHash();
  }

  handleSearchDebounce = () => {
    const firstPage = 1;
```

```
    this.props.updatePage(firstPage);
    this.fetchConsumables();
  }

  handlePageNumberChange = (newPage) => {
    this.props.updatePage(newPage);
    this.fetchConsumables();
  }

  handleSort = (orderBy, orderDirection) => {
    this.props.updateOrder(orderBy, orderDirection);
    this.fetchConsumables();
  }

  updateHash() {
    const { history, location, page, search, orderBy, orderDirection } =
this.props;
    const newHash = { page, search, orderBy, orderDirection };

    updateUrlHash(history, location, newHash);
  }

  render() {
    const {
      search, loading, consumables, page, totalPages, perPage, dismissMessage,
      totalEntries, orderBy, orderDirection, t, messages, updateSearchInput,
columns,
    } = this.props;

    const SearchInput = (
      <SearchBox
        value={search}
        onDebounce={this.handleSearchDebounce}
        onChange={updateSearchInput}
        t={t}
      />
    );

    const TableHeader = (
      (!(isEmpty(consumables) && loading)) &&
      <DataTableHeader
        page={page}
        totalPages={totalPages}
        totalEntries={totalEntries}
        perPage={perPage}
        entryLocale="consumable"
        t={t}
      />
    );

    const PageButtons = (
      <PaginationButtons
        currentPage={page}
        totalPages={totalPages}
        onPageChange={this.handlePageNumberChange}
      />
    );

    const Messages = (
      <MessageBucket
        t={t}
        messages={messages}
        dismiss={dismissMessage}
      />
```

```
    );

    const consumableColumns = this.getDisplayedConsumableColumns();
    const ConsumablesListTable = (
      <ListTable
        columns={consumableColumns}
        entries={consumables}
        sortColumnName={orderBy}
        sortDirection={orderDirection}
        onSort={this.handleSort}
        tableProps={{
          sortable: true,
          selectable: true,
          striped: true,
          compact: 'very',
          size: 'small',
        }}
      />
    );

    const ColumnsSelectPopup = (
      <span className={'table-settings'}>
        <TableSettingsPopup columns={columns} />
      </span>
    );

    const buttons = [
      {
        color: 'blue',
        href: `${window.location.origin}/models/new`,
        content: t('actions.create_new_model'),
      },
      {
        href: `${window.location.origin}/models/export`,
        content: t('actions.export.all_models'),
      },
    ];

    const Buttons = buttons && buttons.map((button, i) => (
      <Button {...button} key={i.toString()} style={{ marginLeft: i !== 0 ?
'10px' : 0 }} />
    ));

    return (
      <div>
        <Segment.Group className="segment-group">
          <Segment basic>
            <Grid stackable divided="vertically" verticalAlign="middle">
              <Grid.Row columns={2}>
                <Grid.Column>{SearchInput}</Grid.Column>
                <Grid.Column textAlign="right">
                  {Buttons}
                  {ColumnsSelectPopup}
                </Grid.Column>
              </Grid.Row>
            </Grid>
          </Segment>
          <Segment basic loading={loading} className="loading-segment">
            <Grid stackable divided="vertically">
              <Grid.Row columns={2}>
                <Grid.Column>{TableHeader}</Grid.Column>
                <Grid.Column>{PageButtons}</Grid.Column>
              </Grid.Row>
            </Grid>
            {Messages}
```

```
            {ConsumablesListTable}
            {PageButtons}
          </Segment>
        </Segment.Group>
      </div>
    );
  }
}

const mapStateToProps = state => ({
  currentUser: currentUserSelectors.getCurrentUser(state),
  consumables: consumablesSelectors.getConsumables(state),
  page: consumablesSelectors.getConsumablesMetadataPage(state),
  totalPages: consumablesSelectors.getConsumablesMetadataTotalPages(state),
  perPage: consumablesSelectors.getConsumablesMetadataPerPage(state),
  totalEntries: consumablesSelectors.getConsumablesMetadataTotal-
Entries(state),
  orderBy: consumablesSelectors.getConsumablesMetadataOrderBy(state),
  orderDirection: consumablesSelectors.getConsumablesMetadataOrderDirec-
tion(state),
  loading: consumablesSelectors.getConsumablesLoading(state),
  messages: consumablesSelectors.getConsumablesMessages(state),
  search: consumablesSelectors.getConsumablesSearchInputValue(state),
  columns: currentUserSelectors.getConsumableModelColumns(state),
  displayedColumnNames: currentUserSelectors.getDisplayedConsumable-
ModelColumnNames(state),
});

const mapDispatchToProps = {
  loadConsumables: consumablesActions.loadConsumables,
  dismissMessage: consumablesActions.dismissConsumablesMessageById,
  updateSearchInput: consumablesActions.updateConsumablesSearchInput,
  updateOrder: consumablesActions.updateConsumablesOrder,
  updatePage: consumablesActions.updateConsumablesPage,
};

export default translate('', { wait: true })(
  connect(mapStateToProps, mapDispatchToProps)(ConsumablesListPage)
);
```

Consumables actions (actions/consumables.js)

```
import * as types from 'actions/actionTypes';
import { constructSuccessMessage } from 'utils/messages';
import { handleError } from 'utils/utils';
import * as consumablesSelectors from 'reducers/consumables';

const loadConsumablesRequest = () => ({
  type: types.CONSUMABLES_LOAD_REQUEST,
});

const loadConsumablesSuccess = payload => ({
  type: types.CONSUMABLES_LOAD_SUCCESS,
  payload,
});

const loadConsumablesFailure = message => ({
  type: types.CONSUMABLES_LOAD_FAILURE,
  payload: { message },
});

const addConsumablesSuccess = (model, message) => ({
  type: types.CONSUMABLES_ADD_SUCCESS,
  payload: { model, message },
});

const addConsumablesFailure = message => ({
  type: types.CONSUMABLES_ADD_FAILURE,
  payload: { message },
});

const destroyConsumableSuccess = message => ({
  type: types.CONSUMABLE_DESTROY_SUCCESS,
  payload: { message },
});

const destroyConsumableFailure = message => ({
  type: types.CONSUMABLE_DESTROY_FAILURE,
  payload: { message },
});

const dismissConsumablesMessage = messageId => ({
  type: types.CONSUMABLES_MESSAGE_DISMISS,
  payload: { messageId },
});

const changeConsumablesSearchInputValue = value => ({
  type: types.CONSUMABLES_SEARCH_INPUT_VALUE_CHANGE,
  payload: { searchInputValue: value },
});

const changeConsumablesOrder = (orderBy, orderDirection) => ({
  type: types.CONSUMABLES_ORDER_CHANGE,
  payload: { orderBy, orderDirection },
});

const changeConsumablesPage = page => ({
  type: types.CONSUMABLES_PAGE_CHANGE,
  payload: { page },
});
```

```
const loadConsumables = () =>
  async (dispatch, getState, api) => {
    const resource = '/models/consumables.json';

    dispatch(loadConsumablesRequest());

    try {
      const response = await api.get(resource, {
        'page': consumablesSelectors.getConsumablesPage(getState()),
        'search[free]': consumablesSelectors.getConsumablesSearchInput-
Value(getState()),
        'search[order_by]': consumablesSelectors.getConsuma-
blesOrderBy(getState()),
        'search[order_direction]': consumablesSelectors.getConsuma-
blesOrderDirection(getState()),
      });

      dispatch(loadConsumablesSuccess(response));
    } catch (error) {
      const messageContent = { text: 'consumables.messages.consuma-
bles_list_failure' };

      handleError(error, messageContent, message => dispatch(loadConsuma-
blesFailure(message)));
    }
  };

const destroyConsumable = ({ modelId, modelName }) =>
  async (dispatch, getState, api) => {
    const resource = `/models/${modelId}.json`;

    dispatch(loadConsumablesRequest());

    try {
      await api.delete(resource);

      const message = constructSuccessMessage({
        header: modelName, text: 'models.messages.model_delete_success',
      });
      dispatch(destroyConsumableSuccess(message));
    } catch (error) {
      const messageContent = { header: modelName, text: 'models.mes-
sages.model_delete_failure' };

      handleError(error, messageContent, message => dispatch(destroyConsuma-
bleFailure(message)));
    }
  };

const addConsumables = (
  { modelId, modelName },
  { quantityDiff, price, locationId, departmentId }) =>
  async (dispatch, getState, api) => {
    const resource = `/models/${modelId}/add_consumables.json`;

    try {
      const response = await api.post(resource, {
        quantity_diff: quantityDiff,
        price: price,
        location_id: locationId,
        department_id: departmentId,
      });

      const message = constructSuccessMessage({
```

```
        header: modelName, text: 'stock_balances.messages.balance_update_suc-
cess',
      });
      dispatch(addConsumablesSuccess(response.model, message));
    } catch (error) {
      const messageContent = {
        header: modelName,
        text: 'stock_balances.messages.balance_update_failure',
      };

      handleError(error, messageContent, message => dispatch(addConsuma-
blesFailure(message)));
    }
  };

const dismissConsumablesMessageById = id =>
  async dispatch => dispatch(dismissConsumablesMessage(id));

const updateConsumablesSearchInput = value => async dispatch => dis-
patch(changeConsumablesSearchInputValue(value));

const updateConsumablesOrder = (orderBy, orderDirection) => async dispatch =>
  dispatch(changeConsumablesOrder(orderBy, orderDirection));

const updateConsumablesPage = page => async dispatch =>
  dispatch(changeConsumablesPage(page));

export {
  loadConsumables,
  destroyConsumable,
  addConsumables,
  dismissConsumablesMessageById,
  updateConsumablesSearchInput,
  updateConsumablesOrder,
  updateConsumablesPage,
};
```

Consumables reducer and selectors (reducers/consumables.js)

```
import * as types from 'actions/actionTypes';
import { appendMessage, removeMessageById } from 'utils/messages';
import { replaceArrayEntry } from 'utils/utils';

const stopLoading = () => ({
  loading: false,
});

const defaultState = {
  entries: [],
  messages: [],
  loading: false,
  loaded: false,
  page: 1,
  searchInputValue: '',
  orderBy: '',
  orderDirection: 'asc',
  metadata: {
    page: 1,
    totalPages: 1,
    perPage: 20,
    totalEntries: 0,
    orderBy: '',
    orderDirection: 'asc',
  },
};

const consumables = (state = defaultState, action) => {
  const { type, payload } = action;

  switch (type) {
    case types.CONSUMABLES_LOAD_REQUEST: {
      return { ...state, loading: true, messages: [] };
    }
    case types.CONSUMABLES_LOAD_SUCCESS: {
      const md = payload.metadata;
      return {
        ...state,
        ...stopLoading(),
        loaded: true,
        entries: payload.data,
        metadata: {
          page: md.page,
          totalPages: md.total_pages,
          perPage: md.per_page,
          totalEntries: md.total_entries,
          orderBy: md.order_by,
          orderDirection: md.order_direction,
        },
      };
    }
    case types.CONSUMABLES_LOAD_FAILURE: {
      return { ...state, ...stopLoading(), messages: appendMessage(pay-
load.message, state.messages) };
    }
    case types.CONSUMABLES_ADD_SUCCESS: {
      return { ...state, entries: replaceArrayEntry(payload.model, state.en-
tries) };
    }
    case types.CONSUMABLES_ADD_FAILURE: {
```

```
      return state;
    }
    case types.CONSUMABLE_DESTROY_SUCCESS: {
      return { ...state, ...stopLoading() };
    }
    case types.CONSUMABLE_DESTROY_FAILURE: {
      return { ...state, ...stopLoading() };
    }
    case types.CURRENT_USER_LIST_COLUMNS_UPDATE_FAILURE: {
      const messages = [...state.messages, action.payload.message];

      return { ...state, ...stopLoading(), messages };
    }
    case types.CONSUMABLES_MESSAGE_DISMISS: {
      return { ...state, messages: removeMessageById(payload.messageId,
state.messages) };
    }
    case types.CONSUMABLES_SEARCH_INPUT_VALUE_CHANGE: {
      return { ...state, searchInputValue: payload.searchInputValue };
    }
    case types.CONSUMABLES_ORDER_CHANGE: {
      const { orderBy, orderDirection } = payload;

      return { ...state, orderBy, orderDirection };
    }
    case types.CONSUMABLES_PAGE_CHANGE: {
      return { ...state, page: payload.page };
    }
    default: {
      return state;
    }
  }
};

const getConsumables = state => state.consumables.entries;
const getConsumablesLoading = state => state.consumables.loading;
const getConsumablesLoaded = state => state.consumables.loaded;
const getConsumablesMessages = state => state.consumables.messages;
const getConsumablesSearchInputValue = state => state.consumables.searchInput-
Value;
const getConsumablesPage = state => state.consumables.page;
const getConsumablesOrderBy = state => state.consumables.orderBy;
const getConsumablesOrderDirection = state => state.consumables.orderDirec-
tion;
const getConsumablesMetadata = state => state.consumables.metadata;
const getConsumablesMetadataPage = state => getConsuma-
blesMetadata(state).page;
const getConsumablesMetadataTotalPages = state => getConsuma-
blesMetadata(state).totalPages;
const getConsumablesMetadataPerPage = state => getConsuma-
blesMetadata(state).perPage;
const getConsumablesMetadataTotalEntries = state => getConsuma-
blesMetadata(state).totalEntries;
const getConsumablesMetadataOrderBy = state => getConsuma-
blesMetadata(state).orderBy;
const getConsumablesMetadataOrderDirection = state => getConsuma-
blesMetadata(state).orderDirection;


export default consumables;
export {
  getConsumables,
  getConsumablesLoading,
  getConsumablesLoaded,
  getConsumablesMessages,
```

```
  getConsumablesSearchInputValue,
  getConsumablesPage,
  getConsumablesOrderBy,
  getConsumablesOrderDirection,
  getConsumablesMetadata,
  getConsumablesMetadataPage,
  getConsumablesMetadataTotalPages,
  getConsumablesMetadataPerPage,
  getConsumablesMetadataTotalEntries,
  getConsumablesMetadataOrderBy,
  getConsumablesMetadataOrderDirection,
};
```

Server API middleware (api/Api.js)

```
import qs from 'qs';

import { ApiError, HttpError } from 'actions/actionErrors';


const extractCSRFToken = () => {
  const el = document.getElementsByName('csrf-token')[0];
  const CSRFToken = el ? el.content : '';

  return CSRFToken;
};

const getHeaders = () => {
  const headers = {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  };

  const CSRFToken = extractCSRFToken();
  if (CSRFToken) {
    headers['X-CSRF-Token'] = CSRFToken;
  }

  return headers;
};

class Api {
  config = {};
  dispatch = null;

  constructor() {
    const origin = window.location.origin;
    let endpoint = '';
    if (window.location.hostname.includes('localhost')) {
      if (window.location.port === '9292') {
        // Robot environment
        endpoint = origin;
      } else {
        endpoint = process.env.ENDPOINT;
      }
    } else {
      // E.g. ORGNAME.trail.fi
      endpoint = origin;
    }

    this.config.baseURL = endpoint;
  }

  throwErrorsIfExist = async (response) => {
    if (response.status === 400) {
      const responseJson = await response.json();
      const { errors } = responseJson;

      if (errors) {
        throw new ApiError(errors, response.status);
      } else {
        throw new HttpError(response.statusText, response.status);
      }
    } else if (response.status < 200 || response.status >= 300) {
      throw new HttpError(response.statusText, response.status);
```

```
    }
  }

  async get(uri, params = {}) {
    const query = qs.stringify(params);
    const init = {
      headers: getHeaders(),
      credentials: 'include',
    };
    const separator = uri.includes('?') ? '&' : '?';

    const response = await fetch(`${this.config.baseURL}${uri}${separa-
tor}${query}`, init);

    await this.throwErrorsIfExist(response);

    return response.json();
  }

  async post(uri, body = {}, isFile = false) {
    const headers = getHeaders();

    if (isFile) {
      delete headers['Content-Type'];
    }

    const response = await fetch(`${this.config.baseURL}${uri}`, {
      method: 'POST',
      headers: headers,
      body: isFile ? body : JSON.stringify(body),
      credentials: 'include',
    });

    await this.throwErrorsIfExist(response);

    if (response.status === 204) {
      return {};
    }

    return response.json();
  }

  async put(uri, params = {}) {
    const response = await fetch(`${this.config.baseURL}${uri}`, {
      method: 'PUT',
      headers: getHeaders(),
      credentials: 'include',
      body: JSON.stringify(params),
      credentials: 'include',
    });

    await this.throwErrorsIfExist(response);

    return response.json();
  }

  async delete(uri) {
    const response = await fetch(`${this.config.baseURL}${uri}`, {
      method: 'DELETE',
      headers: getHeaders(),
      credentials: 'include',
    });

    await this.throwErrorsIfExist(response);
```

```
    if (response.status === 204) {
      return {};
    }

    return response.json();
  }
}

export default Api;
```

Table generating component (components/tables/ListTable.js)

```
import React from 'react';
import { Table } from 'semantic-ui-react';
import isEmpty from 'lodash/isEmpty';

import { getOppositeDirection, getDirectionFullName } from 'utils/utils';

const ListTable = (props) => {
  const { entries, columns, sortDirection, sortColumnName, onSort, tableProps
} = props;

  if (isEmpty(entries)) {
    return null;
  }

  const handleSort = (columnName) => {
    const direction = (sortColumnName !== columnName) ? 'asc' : getOppositeDi-
rection(sortDirection);

    onSort(columnName, direction);
  };

  const HeaderCells = columns.map((column) => {
    const headerCellProps = column.headerCellProps || {};
    if (column.sortable) {
      const direction = getDirectionFullName(sortDirection);

      headerCellProps.sorted = (sortColumnName === column.name) ? direction :
null;

      if (onSort) {
        headerCellProps.onClick = () => handleSort(column.name);
      }
    }

    return (
      <Table.HeaderCell key={column.name} {...headerCellProps}>
        {column.label || ''}
      </Table.HeaderCell>
    );
  });

  const HeaderRow = <Table.Row>{HeaderCells}</Table.Row>;

  const BodyRows = entries.map((entry) => {
    const BodyCells = columns.map((column) => {
      const tableCellProps = column.tableCellProps || {};
      const TableCellContent = column.component || entry[column.name];

      return (
        <Table.Cell key={column.name} className={`column_${column.name}`}
{...tableCellProps}>
          {column.component ? <TableCellContent entry={entry} /> : TableCell-
Content }
        </Table.Cell>
      );
    });

    return <Table.Row key={entry.id}>{BodyCells}</Table.Row>;
  });
```

```
  return (
    <div>
      <Table {...tableProps}>
        <Table.Header>{HeaderRow}</Table.Header>
        <Table.Body>{BodyRows}</Table.Body>
      </Table>
    </div>
  );
};

export default ListTable;
```

Webpack configuration files (config/webpack/base|production.js)

```
// Base configuration

const path = require('path');
const webpack = require('webpack');
const ExtractTextPlugin = require('extract-text-webpack-plugin');


module.exports = {
  entry: {
    main: ['babel-polyfill', 'whatwg-fetch', path.join(__dirname,
'../../src/client.jsx')],
    'navigation-search': path.join(__dirname, '../../src/containers/naviga-
tionSearch/bootstrap.js'),
  },
  output: {
    filename: 'javascripts/react/[name].js',
    path: path.resolve(__dirname, '../../../public'),
    publicPath: '/',
  },
  resolve: {
    extensions: ['.js', '.jsx', '.json', '.less'],
    modules: [
      path.resolve('./src'),
      path.join(__dirname, '../'),
      'node_modules',
    ],
  },
  plugins: [
    new ExtractTextPlugin('stylesheets/react/[name].css'),
    new webpack.optimize.CommonsChunkPlugin({
      name: 'vendor',
    }),
    new webpack.ContextReplacementPlugin(/moment[/\\]locale$/, /en|fi/),
    new webpack.ContextReplacementPlugin(/i18n-iso-countries[/\\]langs$/,
/en|fi/),
  ],
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        include: path.resolve(__dirname, '../../src'),
        loader: 'babel-loader',
        query: {
          plugins: [
            ['babel-plugin-transform-builtin-extend', { globals: ['Error'],
approximate: true }],
          ],
          presets: ['env', 'stage-2', 'react'],
        },
      },
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract({ fallback: 'style-loader', use:
'css-loader' }),
      },
      {
        test: /\.less$/,
        loader: ExtractTextPlugin.extract({ fallback: 'style-loader', use:
['css-loader', 'less-loader'] }),
      },
```

```
      {
        test: /\.json$/,
        loader: 'json-loader',
      },
      {
        test: /\.(woff|woff2|ttf|eot|svg|png)(\?v=[a-z0-9]\.[a-z0-9]\.[a-z0-
9])?$/,
        loader: 'url-loader?limit=100000',
      },
    ],
  },
};



// Production configuration

const merge = require('webpack-merge');
const webpack = require('webpack');
const CopyWebpackPlugin = require('copy-webpack-plugin');

const MinifyPlugin = require('babel-minify-webpack-plugin');
const config = require('./webpack.config.base');

const GLOBALS = {
  'process.env.NODE_ENV': JSON.stringify('production'),
  'process.env.ENDPOINT': JSON.stringify(process.env.ENDPOINT || 'http://lo-
calhost:3000'),
};

module.exports = merge(config, {
  stats: 'errors-only',
  plugins: [
    new webpack.NoEmitOnErrorsPlugin(),
    new webpack.DefinePlugin(GLOBALS),
    new MinifyPlugin({}, { sourceMap: null }),
    new CopyWebpackPlugin([
      {
        from: 'src/public/locales',
        to: 'locales',
      },
    ]),
  ],
});
```

Observational script

```
window.addEventListener('completeObservation', () => {
        const observationsLeft = parseInt(sessionStorage.getItem('observa-
tionsLeft'));
        if (!observationsLeft || observationsLeft < 1) return;

        const timing = performance.timing;
        const loadEventStart = timing.loadEventStart - timing.navigationStart;
        const usableEntries = performance.getEntriesByName('usable');
        const firstUsefulEntries = performance.getEntriesBy-
Name('firstUseful');
        const secondUsefulEntries = performance.getEntriesBy-
Name('secondUseful');

        const domLoading = timing.domLoading - timing.navigationStart;
        const domComplete = timing.domComplete - timing.navigationStart;
        const usable = usableEntries.length ? Math.ceil(usableEn-
tries[0].startTime) : loadEventStart;
        const firstUseful = firstUsefulEntries.length ?
Math.ceil(firstUsefulEntries[0].startTime) : loadEventStart;
        const secondUseful = secondUsefulEntries.length ?
Math.ceil(secondUsefulEntries[0].startTime) : 2 * loadEventStart;

        const observation = { domLoading, domComplete, usable, firstUseful,
secondUseful }
        const data = JSON.parse(sessionStorage.getItem('data'));
        const newData = [...data, observation];

        sessionStorage.setItem('data', JSON.stringify(newData));
        sessionStorage.setItem('observationsLeft', observationsLeft - 1);

        location.reload();
      });
```

Usable.jsx

```
import { Component } from 'react';


class Usable extends Component {
  componentDidMount() {
    window.requestAnimationFrame(() => performance.mark('usable'));
  }

  render() {
    return '';
  }
}

export default Usable;
```

Useful.jsx

```
import { Component } from 'react';


class Useful extends Component {
  componentDidMount() {
    window.requestAnimationFrame(this.handleFirstAnimationFrameRequest);

    window.addEventListener('loadListAnew', async () => {
      await this.props.loadListAnew();

      window.requestAnimationFrame(this.handleSecondAnimationFrameRequest);
    });
  }

  handleFirstAnimationFrameRequest = () => {
    performance.mark('firstUseful');

    const loadListAnew = new Event('loadListAnew');
    window.dispatchEvent(loadListAnew);
  }

  handleSecondAnimationFrameRequest = () => {
    performance.mark('secondUseful');

    const completeObservation = new Event('completeObservation');
    window.dispatchEvent(completeObservation);
  }


  render() {
    return '';
  }
}


export default Useful;
```

Benchmark results (without resource cache)

	0	20	50	100	300
DOM loading	155 (34)	275 (81)	466 (64)	694 (66)	1699 (99)
DOM complete	754 (61)	869 (131)	1090 (108)	1402 (97)	2638 (116)
Usable	754 (61)	870 (131)	1090 (108)	1402 (97)	2638 (116)
First useful	754 (61)	870 (131)	1090 (108)	1402 (97)	2638 (116)
Second useful	1508 (123)	1740 (263)	2180 (216)	2804 (194)	5277 (233)

Table 1. Google Chrome, server-side rendering results. Means with standard deviations

	0	20	50	100	300
DOM loading	188 (17)	188 (40)	198 (45)	220 (67)	331 (98)
DOM complete	1532 (66)	1502 (70)	1539 (75)	1565 (93)	1674 (118)
Usable	1670 (66)	1637 (73)	1687 (100)	1707 (96)	1862 (122)
First useful	1817 (66)	1953 (88)	2331 (129)	2723 (122)	4412 (165)
Second useful	1951 (67)	2318 (115)	2930 (134)	3681 (139)	6709 (246)

Table 2. Google Chrome, client-side rendering results. Means with standard deviations

	0	20	50	100	300
DOM loading	177 (50)	270 (62)	456 (66)	712 (88)	1736 (220)
DOM complete	780 (92)	958 (163)	1208 (196)	1601 (230)	3069 (428)
Usable	780 (92)	958 (163)	1208 (195)	1601 (230)	3069 (428)
First useful	780 (92)	958 (163)	1208 (195)	1601 (230)	3069 (428)
Second useful	1560 (185)	1916 (327)	2417 (391)	3203 (460)	6138 (857)

Table 3. Mozilla Firefox, server-side rendering results. Means with standard deviations

	0	20	50	100	300
DOM loading	191 (56)	214 (105)	229 (146)	290 (81)	384 (150)
DOM complete	1610 (396)	1648 (454)	1680 (437)	1706 (174)	1848 (223)
Usable	1821 (396)	1856 (454)	1891 (441)	1914 (175)	2051 (232)
First useful	1984 (437)	2266 (488)	2606 (490)	3175 (246)	4811 (311)
Second useful	2123 (437)	2707 (520)	3262 (485)	4194 (245)	7204 (497)

Table 4. Mozilla Firefox, client-side rendering results. Means with standard deviations

	0	**20**	**50**	**100**	**300**
DOM loading	170 (33)	298 (85)	503 (95)	750 (100)	1724 (94)
DOM complete	734 (74)	892 (115)	1218 (148)	1583 (126)	3129 (120)
Usable	734 (74)	892 (115)	1218 (148)	1583 (126)	3129 (120)
First useful	734 (74)	892 (115)	1218 (148)	1583 (126)	3129 (120)
Second useful	1469 (148)	1785 (230)	2437 (297)	3167 (252)	6259 (240)

Table 5. Apple Safari, server-side rendering results. Means with standard deviations

	0	**20**	**50**	**100**	**300**
DOM loading	216 (45)	231 (36)	232 (35)	249 (44)	443 (134)
DOM complete	1277 (154)	1295 (71)	1302 (71)	1337 (75)	1513 (156)
Usable	1465 (157)	1497 (81)	1501 (75)	1531 (86)	1718 (153)
First useful	1610 (173)	1827 (85)	2164 (99)	2620 (141)	4302 (262)
Second useful	1752 (176)	2247 (213)	2869 (108)	3757 (195)	7006 (386)

Table 6. Apple Safari, client-side rendering results. Means with standard deviations

Benchmark results (with resource cache)

	0	20	50	100	300
DOM loading	166 (70)	336 (108)	467 (63)	720 (96)	1687 (88)
DOM complete	303 (73)	498 (109)	675 (69)	995 (94)	2232 (96)
Usable	303 (73)	498 (109)	675 (69)	995 (94)	2232 (96)
First useful	303 (73)	498 (109)	675 (69)	995 (94)	2232 (96)
Second useful	606 (147)	997 (219)	1351 (139)	1990 (189)	4465 (192)

Table 7. Google Chrome, server-side rendering results. Means with standard deviations

	0	20	50	100	300
DOM loading	191 (27)	183 (22)	206 (55)	207 (46)	327 (96)
DOM complete	500 (44)	507 (56)	545 (87)	558 (82)	699 (124)
Usable	589 (50)	590 (60)	621 (89)	648 (88)	777 (124)
First useful	733 (53)	962 (93)	1274 (117)	1666 (119)	3303 (186)
Second useful	872 (57)	1329 (122)	1874 (124)	2627 (136)	5575 (212)

Table 8. Google Chrome, client-side rendering results. Means with standard deviations

	0	20	50	100	300
DOM loading	180 (55)	302 (66)	473 (73)	726 (74)	1740 (116)
DOM complete	538 (87)	727 (159)	964 (207)	1296 (155)	2762 (358)
Usable	538 (87)	727 (159)	964 (207)	1296 (155)	2762 (358)
First useful	538 (87)	727 (159)	964 (207)	1296 (155)	2762 (358)
Second useful	1077 (175)	1454 (318)	1929 (415)	2592 (311)	5525 (717)

Table 9. Mozilla Firefox, server-side rendering results. Means with standard deviations

	0	20	50	100	300
DOM loading	166 (63)	168 (75)	170 (68)	212 (126)	362 (149)
DOM complete	539 (120)	547 (201)	635 (432)	684 (487)	950 (870)
Usable	683 (138)	680 (201)	774 (434)	822 (487)	1102 (873)
First useful	785 (135)	1169 (258)	1569 (505)	2061 (575)	3908 (1204)
Second useful	902 (143)	1539 (271)	2387 (529)	3211 (641)	6460 (1416)

Table 10. Mozilla Firefox, client-side rendering results. Means with standard deviations

	0	**20**	**50**	**100**	**300**
DOM loading	193 (57)	289 (73)	502 (104)	763 (117)	1723 (102)
DOM complete	447 (73)	562 (88)	843 (144)	1218 (174)	2462 (328)
Usable	447 (72)	562 (88)	844 (144)	1218 (174)	2462 (328)
First useful	447 (72)	562 (88)	844 (144)	1218 (174)	2462 (328)
Second useful	894 (145)	1124 (177)	1688 (289)	2437 (349)	4925 (657)

Table 11. Apple Safari, server-side rendering results. Means with standard deviations

	0	**20**	**50**	**100**	**300**
DOM loading	215 (72)	214 (55)	232 (180)	257 (114)	439 (178)
DOM complete	520 (170)	502 (194)	492 (211)	493 (138)	691 (199)
Usable	663 (190)	674 (348)	628 (221)	618 (152)	821 (202)
First useful	792 (201)	1027 (394)	1380 (293)	1804 (249)	3649 (528)
Second useful	938 (223)	1503 (463)	2178 (347)	3052 (274)	6588 (707)

Table 12. Apple Safari, client-side rendering results. Means with standard deviations