OULU UNIVERSITY OF APPLIED SCIENCES

Damien Morizur

ENHANCING MAGENTO FRONTEND PERFORMANCE WITH REACTJS AND COMPARING IT TO KNOCKOUT

ENHANCING MAGENTO FRONTEND PERFORMANCE WITH REACTJS AND COMPARING IT TO KNOCKOUT

Damien Morizur
Bachelor's Thesis
Spring 2019
Degree Programme in Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author: Damien Morizur
Title of the bachelor's thesis: Enhancing Magento Frontend Performance with ReactJS and Comparing It to Knockout
Supervisor: Veijo Väisänen
Term and year of completion: Spring 2019 Number of pages: 50 + 8 appendices

---

Magento 2 is a PHP-based framework used for building e-commerce solutions. It enables developers to easily extend, customize and create pages and features in order to build the best possible e-commerce solutions. This thesis was commissioned by Vaimo Finland Oy. The company previously developed a module for Magento 2 that replaced the product search functionality with a new user interface using KnockoutJS. While providing with a better and more efficient solution than the default Magento search, it still lacked in performance, particularly in older browsers such as Internet Explorer. The implementation done for this thesis aimed at replacing this solution with a new module using ReactJS as the framework for the search user interface.

For this thesis, a new module for Magento 2 was built. It fetches search results asynchronously and uses ReactJS as the library for rendering the user interface. Performance testing was conducted to assert whether the new user interface had more performant rendering than the previous one created with Knockout.

The implementation resulted in a proof of concept that in the future will undergo further development and replace the user interface from the current module. The testing done showed that the new user interface had improved rendering performance compared to the previously created solution.

Keywords: ReactJS, Magento 2, Frontend, Search

CONTENTS

VOCABULARY

API: Application Programming Interface.

CSS: Cascading Style Sheets

DOM: Document Object Model

Less: Preprocessor used for compiling CSS

REST: Representational State Transfer

SKU: Stock keeping unit

UI: User Interface

URL: Uniform Resource Locator

XML: Extensible Markup Language

1 INTRODUCTION

I have been studying in Oulu University of Applied Sciences for nearly 4 years, learning the basis of working as a web developer. Now working for a company named Vaimo, I started this project eager to apply all I have learned both during studies and working life.

Background

Vaimo was founded in 2008 and it is the global leader in delivering award-winning digital storefronts. With 400 employees over 12 different countries around the globe, Vaimo's focus is to drive digital commerce success for its brand, retail and manufacturer clients. (12)

Vaimo uses the Magento platform to build most of its e-commerce solutions, which allows developers to easily expand on an extensive array of features to build unique stores for visitors to browse and buy products from.
Always looking for the best possible solutions for businesses, the company has also created many new features that either expand on, or replace, existing features from the Magento platform, with the goal of added value for the stores.

One such feature, which was the base for this project is the product search functionality. Searching products on a webstore needs to be as seamless an experience as possible so that customers can quickly find the products they want. This is the reason why Vaimo has previously created a module for Magento that aimed at improving how quick and easily users can find products that they want.

By default, the Magento search works so that when performing a search, as can be shown in the figure 1, it will load a dedicated page where results matching the search will be rendered. Every new search requires the user to navigate to this page, thus leaving the one they were on.

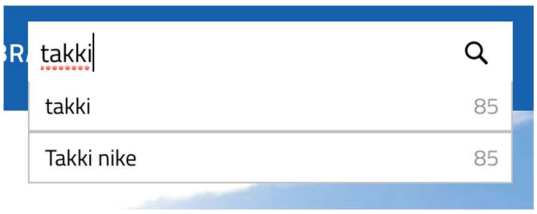

FIGURE 1. An example of typing a search with Magento's default search

In comparison, the module previously created by Vaimo creates a brand-new user interface for the product search functionality in the form of a popup that opens when performing a search on any page and that asynchronously fetches results with Ajax and render them using Knockout. Provided with advanced multi-option filtering, the user interface allows the user to search, sort and filter products without the need for page loads. (figure 2)

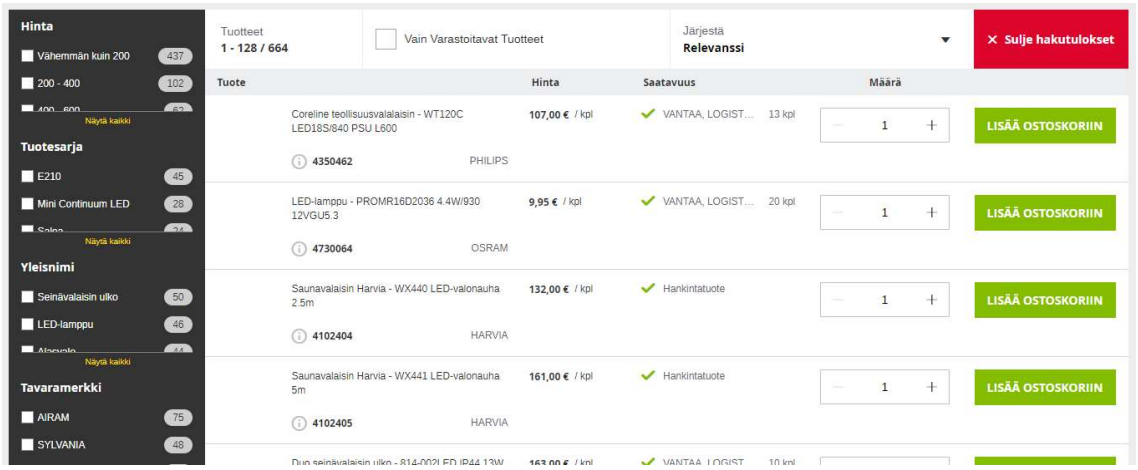

FIGURE 2. A screenshot of the module's search UI.

While being a great solution, it appeared that rendering performance of the module was not the best it could be when dealing with many products to render on the page, particularly with older browsers such as Internet Explorer. These performance bottlenecks were attributed to the use of Knockout JavaScript library for rendering the user interface, as it seemed not optimal for handling large amounts of dynamic content changes.

Aim

The aim of this thesis project was to implement a new module for the Magento platform. It would aim at replacing the current user interface of the module with a new solution made with ReactJS. ReactJS was chosen for its known great performance and the fact that future versions of Magento will also use that popular library, making it the next big thing for Magento frontend developers. The aim of the implementation was to be more performant in comparison to the previous one, meaning performance tests would be required at the end.

This thesis report will focus on setting the background of the implementation, with the use of Magento and React, then go through the implementation of the new module created and finally the testing done to assert whether the new implementation using React was a more performant solution than the current one using Knockout.

2 MAGENTO

Magento is an open-source PHP-based platform for building e-commerce solutions. Built by the Magento company (now part of Adobe), it is used by over 350,000 developers all over the world. (1) It enables the creation of highly customizable digital storefronts for Business-to-Customer and Business-to-Business purposes.

2.1 General overview

Magento's platform is built upon PHP and MySQL. During its lifetime of 10 years (the version 1.0 released in March 2008 and the version 2.0 in November 2015 (2)), it has undergone changes in terms of structure and development patterns, and is now in its second major version, Magento 2.
As the Magento version 2.2 was used for this thesis, the development flow talked upon in this document will be of Magento 2 when referred-to as Magento.

Magento's structure is comprised of two main parts, one being the back-end, with the database and MySQL, and Model, Data and Service interfaces, as can be seen in the figure 3. These are directly connected and used in Magento's Blocks, Layouts and Templates, which would be defined as the front-end of the application.

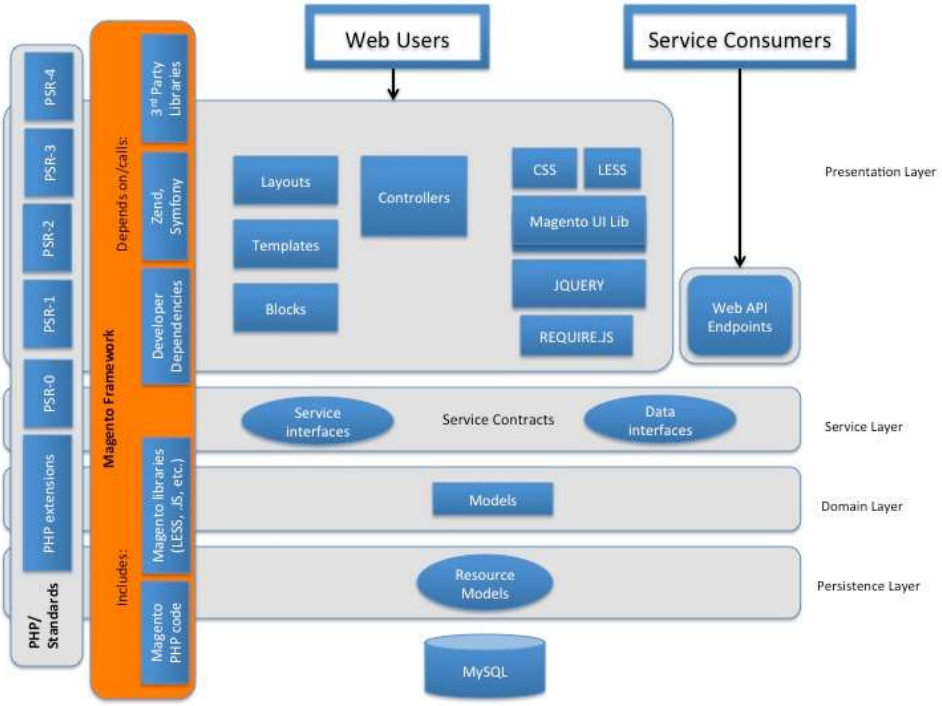

FIGURE 3. Representation of Magento's layered architecture. (3)

When installing Magento for a brand-new project, a basic webstore is created that the developers can then modify to fit the needs of the client to create a customized user experience.

2.2 Product searching

One of the prominent features on any webstore, and the focus of this thesis work, is the product search feature, which allows users to type in terms to search products by. Once search results are gathered, the user can usually sort or filter through those by categories or product attributes.

In Magento, the search usually uses either MySQL, Solr or Elasticsearch for indexing the catalog. Catalog indexing determines what results are returned to the storefront when entering a new search term or changing filtering options from already gathered results.

While not directly tied to this thesis work, it was important to understand the technologies involved on the backend. For this thesis, the catalog indexing solution used on the backend was Elasticsearch.

2.2.1 Elasticsearch

Elasticsearch is a free and open-source search-engine built on Apache Lucene. It is particularly used when dealing with large amounts of data and when needing to quickly retrieve that data. When using MySQL, querying for data can take very long when having a lot of data. Thus, the main selling point of Elasticsearch, is to multiply querying speeds by the hundreds. It works by indexing data into documents that have keys and values and it comes with an API that allows for retrieving these indexed documents. (4)

When used on a Magento store, the catalog is indexed in an Elasticsearch node. It is easily configurable by just enabling it and specifying from the admin panel of the store, the IP address and port of the host where the Elasticsearch node is located.

2.2.2 Search API

Magento has a fair amount of REST API endpoints, of which one that can be used for searching products. This search API endpoint takes as data sent an array called searchCriteria, which has the following structure (figure 4):

```
searchCriteria[filter_groups][<index>][filters][<index>][field]=<field_name>
searchCriteria[filter_groups][<index>][filters][<index>][value]=<search_value>
searchCriteria[filter_groups][<index>][filters][<index>][condition_type]=<operator>
```

FIGURE 4. The structure of the searchCriteria array. (5)

To make a request, the client must send a GET request to a specific path appended to the base URL of the store. The path is /rest/V1/search/? to which the search criteria are appended.

While returning all matching products given to these search criteria, the search API does not return products information such as product name, SKU, price. Another request to a different API then should be done to retrieve such product data by providing product IDs. This means that to retrieve full products information (such as name or price) for a specific search, several requests to

the API are necessary. This is not efficient as making several API requests takes time.

This limitation was addressed in the module previously created by Vaimo. It customized this search API to make it return all the data needed rather than just product IDs, thus making the search more time efficient.
The new module created for this thesis, while creating a brand-new user interface (UI), made use of this custom API when fetching search results using Ajax, and rendering them with React Components.

3 REACT

For this thesis, the author used React as the primary tool for implementing the new search interface that will replace the one from the previously made module, which used Knockout as the JavaScript library. In this section, the author will clarify what React is and why it was chosen for the implementation based on how it works, its relationship with Magento framework and Vaimo as a company.

3.1 React as a tool

React is a JavaScript library created by Facebook and designed for building user interfaces. It relies on the use of components and states for rendering dynamic content on web pages without the need for page loads. It is one of the most popular and supported JavaScript libraries for UI interfaces these days, as is shown with NPM downloads data in the figure 6. This means a better overall support, which is a great help for development.

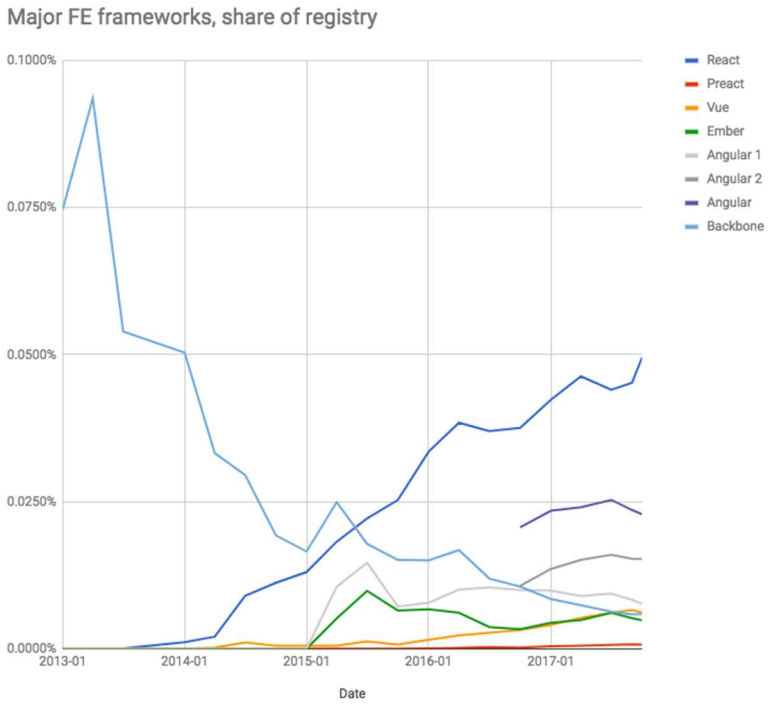

FIGURE 5. JavaScript frameworks popularity. (7)

Unlike older JavaScript solutions, which involved manually triggering changes on the page, React “knows” when the UI needs to be updated and re-rendered on the page when the states of the components change.

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

FIGURE 6. An example of a React component (15)

The figure 6 shows an example of a simple React component. It is a JavaScript class that extends the React.Component class. The render function inside the component returns a description of what the component will show on the page and is most of the time written using the JSX syntax. Such component can be added as a child to other components, i.e. in their render methods by declaring it as: <ExampleComponent />. When doing so, props can be passed to it. They are accessible from within that component (as can be seen in the figure 6), allowing for the component to be used several times with different values for those props.
In addition to props, components can also have states, which are specific to them, and which can be used and modified to update components with new data. These states can also be passed to child components as props.

Going further with using such libraries as React for building complex Uis, where components can be dependent on each other, usually involves the use of a state management library alongside React, of which the two main choices are Redux and MobX. They help to manage the states of the app in the scope of the app rather than the component itself.

The main idea that differentiates React from other libraries or frameworks (e.g. AngularJS, Knockout) is the use of a “virtual DOM”, which enhances performance by reducing the amount of changes happening on the DOM.

3.1.1 React’s virtual DOM

React’s virtual DOM is technically what it sounds like. It is a DOM that is created when components render and when states change. The approach used is to create a new virtual DOM when some element needs re-rendering and compare it to another virtual DOM representation of the previous state of the app, making the “diff” using an algorithm and re-rendering in the DOM only the elements that need changing (Figure 7). (6)

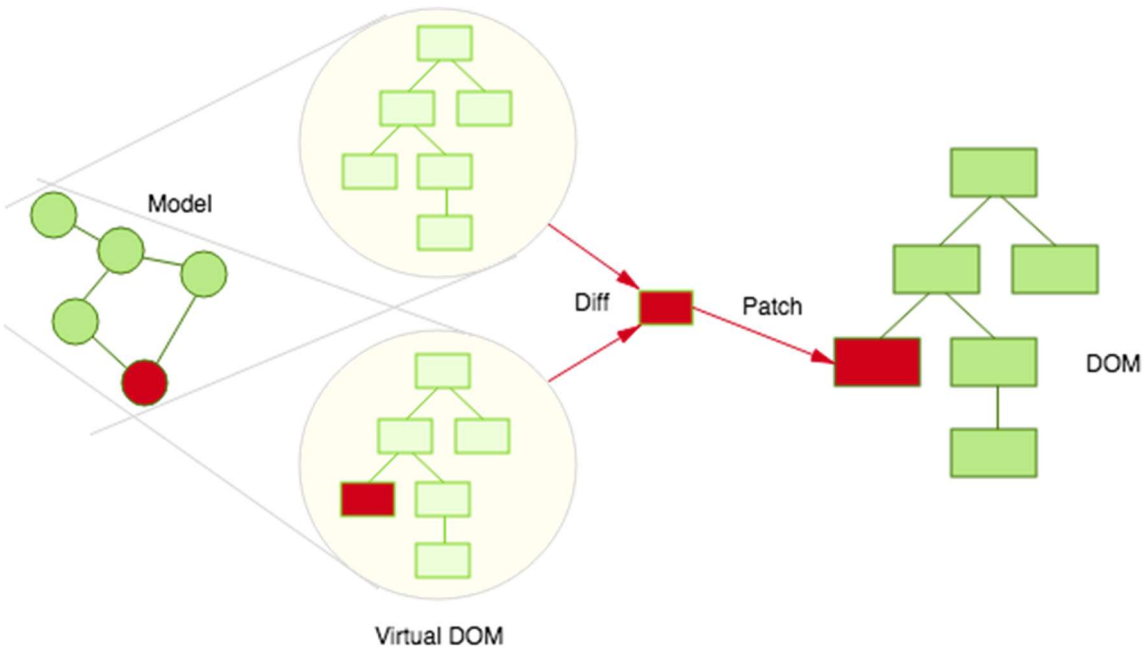

FIGURE 7. Representation of how React changes the DOM (6)

The virtual DOM is a full representation of the DOM without being an actual DOM, meaning that the process of creating a new virtual DOM is much faster than creating the DOM of the actual page. Consequently, since React creates a new virtual DOM every time something needs re-rendering, it does not actually need to know what has changed. This means that the developers do not need to control the rendering process, since React will then change the DOM of the

page only when an actual change is needed. Performance wise, it is one of the best and smartest methods to re-render part of the DOM that needs changing.

3.1.2 State management library

When using a JavaScript library, such as React, developers create components which in most cases are stateful, meaning that each component has states that can change and that trigger component updates. When the application becomes large with many interconnected components, it can become cumbersome and states may need to be passed from parents to children and vice versa. To go around this issue, state management libraries exist. They allow to keep the states of the application in one place for ease of state management. The two most popular libraries used alongside React are Redux and MobX.

Although they both achieve the same goals, differences exist between the two. For this thesis project, it became quickly clear that using a state management library would make a difference, notably to handle all states and data into one place, so that the components react to changes in the data.

MobX was chosen as the state management library for the following reasons:

- It makes use of observables, meaning that the app would keep track of the changes in the data without the need to manually track the changes. (8)
- The author of the thesis was familiar with KnockoutJS, which also makes the use of observables for UI changes, meaning that the mindset for development was easier to attain.
- MobX has a steady learning curve. (8)
- MobX was previously used in Vaimo for another project (Redux was not used at the time), meaning a better support during the development.

3.2 React and Magento

Magento is developing a new tool called PWA Studio. It is a Progressive Web Application tool that will allow developers to work with Magento in a “headless” way, meaning that the frontend is not directly tied to the backend and

communication is done through APIs.
Up until now, Magento user interface was created by using what are called “blocks” which have a template (phtml file) and a PHP block. Data is often rendered on the page load through these blocks and models and interfaces coming from the backend.

PWA Studio uses React, Redux and webpack, meaning that developers working on solutions using PWA Studio will need to be familiar with these technologies.

3.3 React and Vaimo

Although most current Vaimo projects will not be using PWA Studio, new projects will likely make use of it in the future. This is the reason why early on, Vaimo has set its eyes on React in a development perspective. A few features from current projects have also recently been implemented with React, mostly for performance reasons.
This thesis project started at a time when the company decided to start proposing these kinds of solutions.

4 IMPLEMENTATION

The implementation phase of the project concentrated on the creation of a new module for the Magento platform. It should effectively replace the built-in search functionality by a custom search interface for searching, browsing and filtering products, while making use of the custom API implemented in the module previously created.

The main aim of this implementation was to re-create most of the UI elements and features present in this module and to assert whether the rendering performance was improved by using ReactJS instead of KnockoutJS. Therefore, while recreating most of the features present in the current module, the author had to keep performance in mind.

Since the main aim of the creation of the new module was to replace the front-end from the currently used module, one of the very first steps was to find out how the front-end from that module worked in order to have a good base for starting, especially since the backend from that module was to be kept and used for the new module.

4.1 Creation of the module

Prior to starting to create the Store and UI components, a few things needed to be done:

- Installing a new Magento project that has the current module installed as dependency. This was important so that the backend from that module could be used from within the project. This project was used all throughout the Thesis for development and testing.
- Disabling the UI from the current module but keeping the backend working to be able to use the custom search API.
- Creating the skeleton of the new module.
- Creating a new Magento block and template that would be present on every page of the webstore.

- Adding React, React-dom, MobX-react and MobX dependencies (figure 8). React and MobX-react are added as dependencies to the React components, while MobX is added to the Store component.

FIGURE 8. Screenshot of React and MobX dependency files and folders added.

- Creating a JavaScript file that creates a jQuery widget which is instantiated on the template. It defines the MainApp React component which will be the parent for all other React components. It also requires the store component and passes it as a prop to the MainApp. (Figure 9)

```
define([
    'jquery',
    'jquery/ui',
    "react",
    "react-dom",
    "MainApp",
    "store"
], function ($, ui, React, ReactDOM, MainApp, Store) {
    'use strict';
    var store = new Store();
    $.widget('vaimo.searchReact', {

        _create: function () {
            ReactDOM.render(
                React.createElement(MainApp, {
                    store: store
                }),
                $(this.element)[0]
            )
        }
    });

    return $.vaimo.searchReact;
});
```

FIGURE 9. Instantiating the MainApp component.

- Adding and setting up compiling dependencies. React files are not directly usable by web browsers. They need to be compiled into valid JavaScript files. The workflow used for this thesis was to use Babel as the compiler and Gulp as the task runner that runs the compiler.
- Creating a Less file where components styles can be added. For each component, during their implementation, styles were added to give each component, and therefore the UI, a basic appearance.

The structure of the UI of the new module can be divided into two parts, as shown in the figure 10:

- The UI components, which are all the React components and are for the most part stateless, meaning that their states are handled by the Store component.
- The Store component, containing all the observables and functions that modify the states, as well as the functions that both prepare the data for the API calls and parse the data received back to be in a usable format for the UI components.

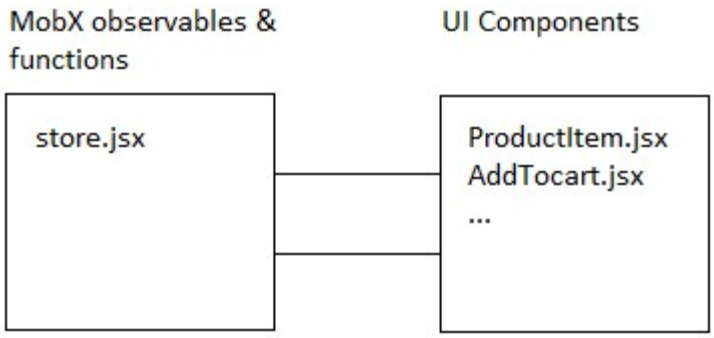

FIGURE 10. Structure of the new module.

4.2 Store component

The Store component could be defined as the “brain” of the module. It is a JavaScript Class which uses MobX as dependency.

It contains:

- Variables and arrays used within the Store
- Observables. These are variables, arrays or objects that have the @observable tag in front of the name when being declared (as shown in the figure 11). They contain the data that will dynamically change in the app and, when used in React components, that will trigger updates when the data is changed. For React components to react to changes in these observables, an @observer tag is added before their class declarations. Consequently, MobX will track the changes in these observables and

relevant React components will update when data is changed, thus handling the states of the app.

- Functions that either directly or indirectly affect the state of the App and update UI components by changing observable values.
- Functions that handle creating the search criteria and sending the search requests in a format usable by the API, as well as recovering and parsing the search results to a format usable by the React components.

```
@observable searchResultsReady = {
    products: [],
    filters: []
};
@observable searchResultsItems = [];
searchResultsItemsTemp = [];
@observable resultsLoading = false;
@observable searchContainerStatus = 'hidden';
@observable currentPage = 0;
```

FIGURE 11. An example of observable variables from Store class.

4.3 UI components

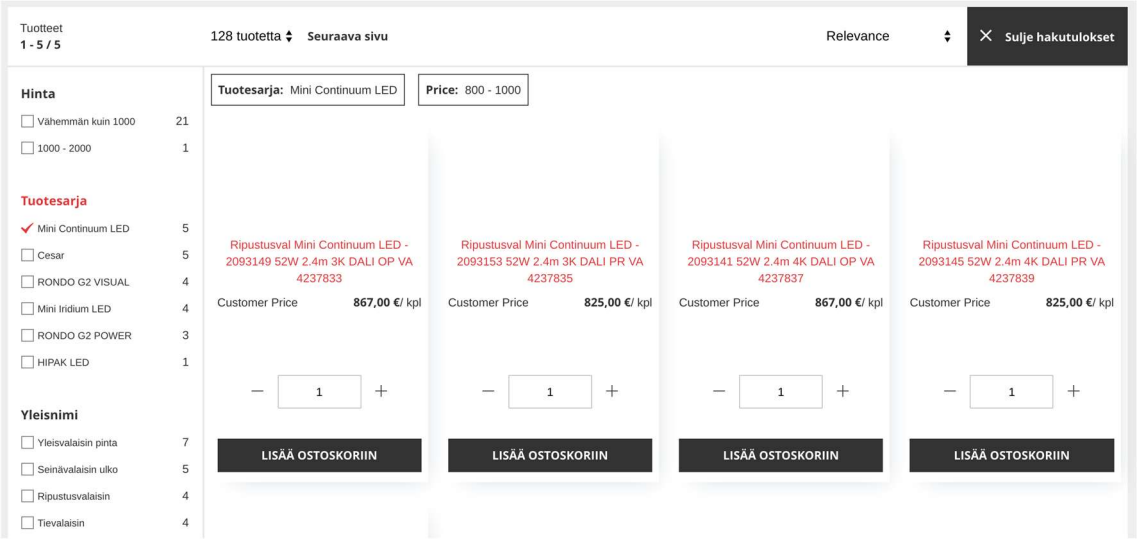

FIGURE 12. An overview of UI components of the new module.

The figure 12 shows the components implemented during this thesis project.

The UI components created for this thesis are a total of 22, structured in groups and hierarchy. The main component of the app (MainApp) is parent to the four main components, which consequently are parents to the remaining 17 components. Those four main components are:

Toolbar

Contains different tools for the search, such as closing the search, going to the next/previous results page, viewing products amounts and sorting results as well as changing the number of products per page.

Filters

All the options that products can be filtered by, with values grouped by filters and filters grouped by filter groups (if any). They can be selected to add search criteria to a new search request.

Product Items

Contains all the product “cards” of the search results as well as add-to-cart / remove-from-cart actions.

Filter tags

Currently active filters that contain options to remove a filter from active filters.

4.3.1 First React component

To test a preliminary rendering of React components for search items, the author initially setup a function in the Store component that triggered an Ajax request using the customized Magento search API to fetch search results, given a fixed search term as an argument. The aim was to use the response to map and display product names one after the other using a React component.

An observable was created in the Store component to receive the items once fetched. The MainApp component would render an initial ProductsList component that mapped the items from the observable.

The first setup and working component was crucial as getting started can be daunting, especially when using React with Magento. Once the first React component was working, and the Store component was set and ready to have observables and functions added, it was easier to then just start implementing the UI components one by one.

4.3.2 Product Items

After successfully creating the first “test” component, the logical next step was to flesh out the product items components to display all necessary product information in what is called a product card.
Following the logic of component hierarchy, in which everything that is reusable or repeatable should most likely be its own component, the initial component that mapped and rendered product names was changed to map product items from the search results and render a ProductItem component for each product (figure 13).

This component would in turn render five different components:

- ProductImage: Product image.
- ProductInfo: Product name and SKU.
- ProductPrice: Product price (or prices).
- ProductQuantity: Product quantity input and incremental and decremental selectors.
- ProductAddToCart: Adding a product to the cart and removing it from the cart actions handling.

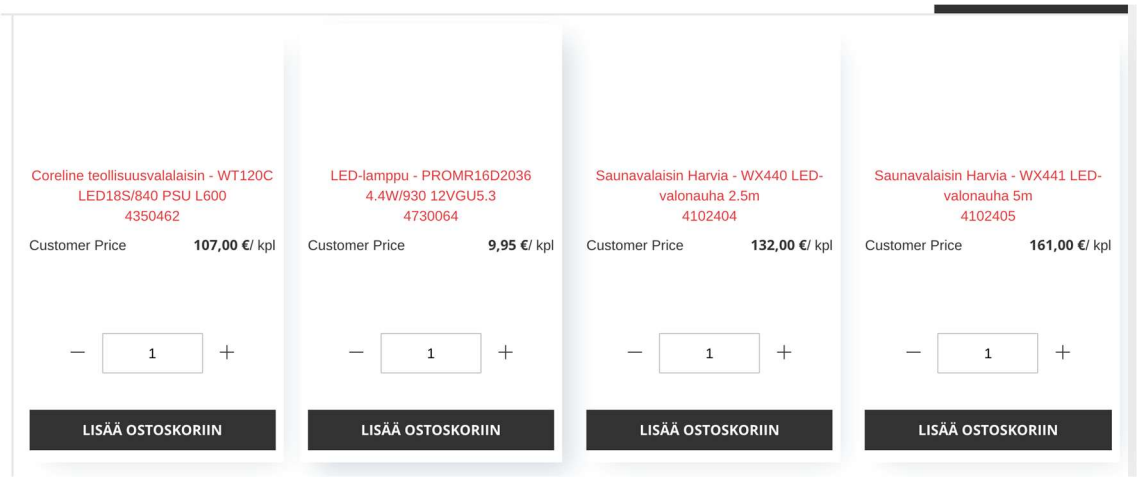

FIGURE 13. Product items from the search results

This compartmenting of components became relevant for two main reasons:

- Performance in terms of rendering. If some data inside a component needs to change on the page, only this component should be updated to speed up rendering.
- Extendibility. Components can be either re-used or moved to different places, adding a level of customization to the implementation.

4.3.3 Quantity change and add-to-cart

FIGURE 14. Quantity input and an Add to the cart button.

The figure 14 shows the quantity change and add-to-cart components. These components are children of the product item component. They were particularly interesting to create in that they are the only components inside the product item component that have actions associated to them, and therefore they were the first dynamic actions implemented.

The ProductQuantity component is composed of an input field for the quantity of that item to add to the cart along with selectors that increment or decrement the quantity value.

ProductAddToCart component has a button that either adds or removes the item from the cart depending if the item is present in the shopping cart.

4.3.4 Search input and opening the search popup

To behave as a single-page feature that can be used on any page of the webstore, the new search UI needed to open when a user performs a search. To achieve this, a listener to a typing event from the search input (present usually on the top of Magento store's header) was added. It opened the search UI when typing more than 3 letters on the input and used the value as a search term.

4.3.5 Loader

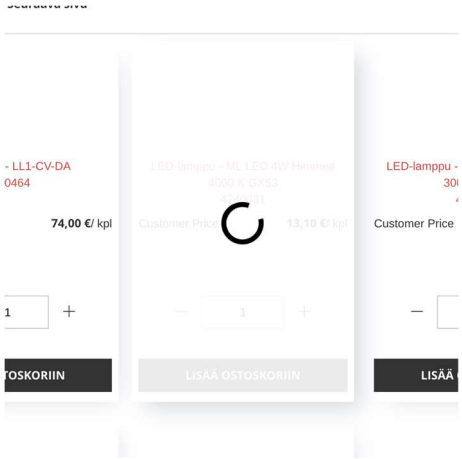

FIGURE 15. A loader showing when adding product to the shopping cart.

This component was crucial to the application since whenever a user performs an action that takes some time to execute, a loader should be shown on the UI to inform the user that the application is processing the request.

React really shined in this case since it allowed to create one component for all the loaders to use on the page, that would be rendered in different places, on different conditions:

- On top of item list, when fetching new items.
- On top of one item, as shown in the figure 15, when adding it to / removing it from the shopping cart.

In terms of performance, using a CSS animation seemed the best choice, particularly if using the CSS transform property, since it does not trigger any repainting on the page when running (9). Paintings can be expensive, particularly on older browsers.

Since CPU power is in use when the loader is running (to fetch and process the data), there is a trick to force the CSS animation to use GPU instead, by adding a 3D property to the animation styles. In this case translateZ(0) is added. It does not affect the actual animation but forces the use of GPU to process it, thus removing the need for extra CPU power. (10)

4.3.6 Search Toolbar

FIGURE 16. Search results Toolbar components.

The toolbar is a component that renders several smaller components which are the different options related to results paging and sorting.

In this case the components are (in order from the figure 16):

- Products amount (per page / total)

Shows the current range of products showing on the page and how many products in total exist for the current search. It changes when new results come and calculates the range of products using the page size and the current page values.

- Page size selector

Added late in the project to help with testing. Allows a user to change the page size, meaning the number of products showing on the page. It was implemented so that the value is saved to the localStorage of the browser and it would be kept between different browsing sessions.
Available values were defined in the Store component.

- Pager (next / previous page)

Allows the user to go to the next page or the previous page depending on the current page. The last page is defined by calculating using the page size, current page and total amount of products found. The current page number is saved and when changing the page, a new value is saved before triggering a new search.

- Products sorting

Allows the user to sort the results by different sort orders. Changing the value triggers a new search with that value as search criteria.
Available values are defined and saved in the Store component.

- Search close

Closes (hides) the search popup and allows the user to get back to browsing the page they were on before performing a search.

Each of these ‘tool’ components are all within one parent React component to allow for customizations such as moving the toolbar to different positions or duplicating it to the bottom of the page (without having to re-do the whole UI element). It is one of the four main components on the page, along with the Multi-option filters component, the product items component and filters tags component.

The figures 17 and 18 below respectively show the previous and next page buttons, and the sort order select input with its different options.

FIGURE 17. Pager action buttons

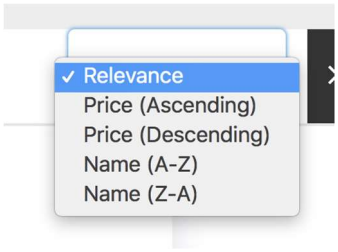

FIGURE 18. Sort order select input.

4.3.7 ErrorBoundary

The use of an Error Boundary component is mainly to be able to catch errors when they occur within their child components without crashing the whole app, allowing for displaying a fallback UI. (14) This means that, i.e. if the ProductItem component has an error, ordinarily the whole app would crash, but now the fallback UI would be showing within the product items list component, while not breaking other components.

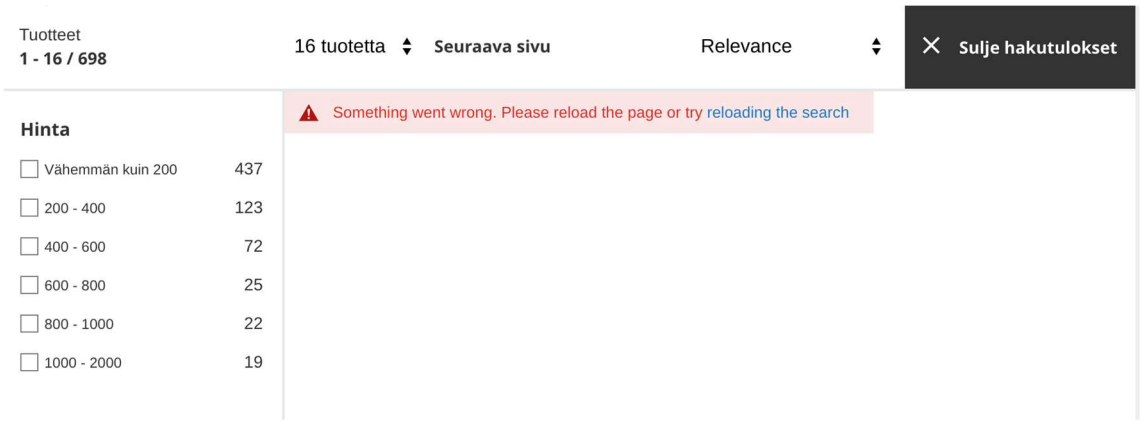

FIGURE 19. A fallback UI when having an error in the ProductItem component.

The ErrorBoundary component created for this implementation contained a user-friendly error message along with a link that allows the user to reload the search, as can be shown in the figure 19.

4.3.8 Multi-option filters

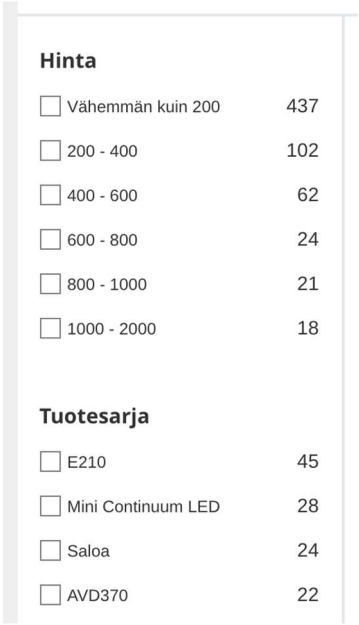

FIGURE 20. Multi-option filters.

The component for filters list was quite obviously an important feature to implement. It lists, grouped by type, all the options that search results can be filtered with. The list of filters and filter options available is provided with the search results fetched through the search API. (Figure 20)

Like the products list component, the filters list is composed of child components, coming all the way down to selectable filter items:

- FiltersContainer: Maps and renders filter groups if any.
- FilterGroup: Maps and renders filter options (e.g. Price or Brand).
- FilterOption: Maps and renders filter items.
- FilterItem: One filter item that is selectable.

When selected, filter items are added to active filters and to the search criteria, before triggering a new search API call with the added filters as arguments.

4.3.9 Filter tags

Filter tags are closely related to filters so that they show on the search results page the currently active filters. (Figure 21)

FIGURE 21. Filter tags.

Although the filters were implemented in such a way that it is visible whether they are active, the filter tags were important to implement so that all active filters could be seen in one place, with the possibility to quickly remove filters by clicking on said tags.

Filter tags are saved in an observable array in the Store component and are appended or removed from the array when a filter is selected from the filters list or when clicking on a filter tag to remove it. Doing so removes the tags and performs the same action that clicking a selected filter from the filters list would, thus removing the filter from active filters and triggering a new search.

4.3.10 Search state in URL

This feature works in two ways:

- Updating the URL of the page to contain the current state of a search in the form of the current search term and the active filters. (Figure 22)
- Retrieving the search term and filters from the URL (if any) and loading the search UI as well as fetch products using those criteria.

When typing on the search bar, the text is also copied to the URL of the page.

FIGURE 22. An example of search term and filter values appended to URL

This was an important feature since when the user goes to a product page from the search, they need to be able to go back to the search in the same state as it was before leaving. Product pages are different pages so the best way to keep the previous search is to save them in the URL, after the base address, in a format that is easily retrievable and can be parsed.

4.4 Optimizations

For this implementation, some optimizations were done to get the best possible performance out of the new module. While many optimizations were seamlessly made during the implementation of the components, by trying different ways of doing things, some optimizations were done afterwards, when the module was ready for performance testing.

Two such optimizations were done, all of which had rendering performance in mind:

- Grouping all data that is updated in the UI into one single observable object.

After receiving the data from the API when performing a search, both filters and products are handled separately, getting the respective data ready for being assigned to observable arrays that are used in the React components.

Early on, both filters and products were in separate observables that were updated at separate times, meaning that MobX and React needed to process

and make DOM changes twice in a row, increasing the time it took to render new data to the user interface.

The solution found was to save both products and filters to two temporary arrays. When both were ready to be rendered, they were added to one single observable object which is then used on both filters and products related components, effectively lowering the time taken for rendering search results. (Figure 23)

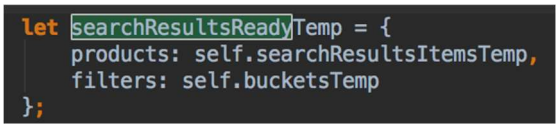

FIGURE 23. A screenshot of saving products and filters into one object.

- Using React's "production" build as dependency

During development, the added React dependencies were the ones that can be used for development and they allow added functionalities that can be used for debugging React applications during development.

React also has a "production" build that has minified code and does not have those extra features, making it more performant and thus adding more performance to the application.

4.5 Extendibility and generic-ness of the new module

Modules created for Magento can be project specific, meaning that they will customize, modify and/or add a new feature that is likely to be needed only for that specific project. If the new feature can or should be used for different projects, the module then needs to be created as a separate dependency which can have its own repository and is added to the composer dependencies of the projects.

In the case of this thesis project, the module needed to be as generic as possible so that it could be installed on different projects and different clients

could benefit from it, while allowing developers to customize the user interface as easily as possible.

There are several ways to address generic-ness and extendibility when implementing the new module:

- Small React components. Each component can be declared and used in any other component, making it easy to change components positions if need be.

- Generic CSS styles and added HTML classes, which allow most elements to either inherit styles from the project where the module is added (e.g. buttons or inputs), while enabling the developers to easily modify or add styles by selected HTML classes.

- The Store component has many configurable variables and arrays, all grouped at the beginning of the file, to help with modifying certain configurations (e.g. sort order values or amounts of products per page).

In most Magento modules, components or blocks can easily be configured by changing certain values in their layout configurations (in xml files), making it easy to extend at a project level. This was not easily possible with the new module. Since React is not integrated into Magento, React components cannot be declared using xml declarations. This was a big drawback when comparing the extendibility between the new module and the current one, making the latter better in that regard.

5 PERFORMANCE TESTING

Once the implementation reached a state of “proof of concept”, and most features present in the currently used module were functional in the new module, it was ready for extensive performance testing and comparisons. Although some testing was conducted during the implementation process to make some optimizations and to insure that the module was working properly on multiple browsers, the core of the testing had to occur when all performance heavy features were ready, so that the new module could be fairly compared to the current one in terms of rendering performance.

Additionally, since the new module changes the UI without affecting the fetching of the data, the performance improvements were expected to be mostly for rendering the UI elements, thus performance testing and comparison was limited to rendering. To be able to test both modules on the same webstore, a test environment was used where the new module would be used on all pages apart from one where the current module would be used and the new one disabled. Since that page was otherwise empty, another similar page was created on which the new module could be tested, so that both modules would have the same testing conditions.

5.1 Preliminary testing

One of the key components of this thesis was to create the new module so that it would eventually replace the frontend from the previous module done with Knockout. Logically, the idea of the project came from the fact that Knockout was not performing well for a large amount of dynamic content to render on the page, particularly for older browsers, and that using React would be more performant.

While the first observations during and after the implementation of the new module were showing very promising results, it was crucial to be able to prove that the new module was indeed more performant. That could be done once the implementation was in a state of possessing the same number of features and components than the previous implementation.

The first logical step was to manually test performance with the tools at hand, in this case the developer tools from Chrome browser, particularly the Performance tab, that allows for recording and analyzing runtime performance. (Figure 24)

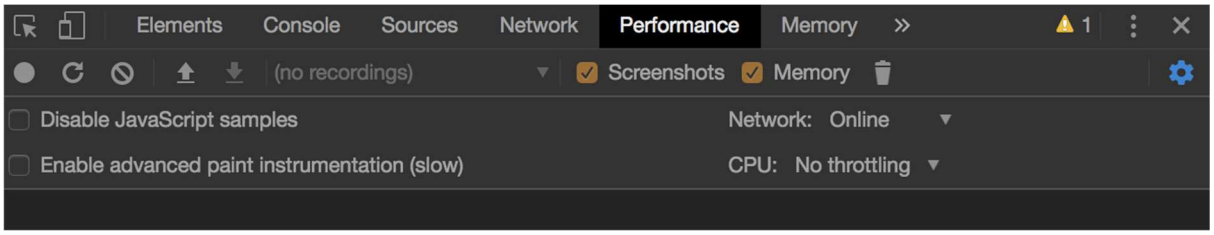

FIGURE 24. An overview of the Performance Tab from Chrome Dev Tools.

When pressing the record button, it starts recording what happens on the page. And once stopping the recording, it will display a timeline showing all kind of data relating to performance.

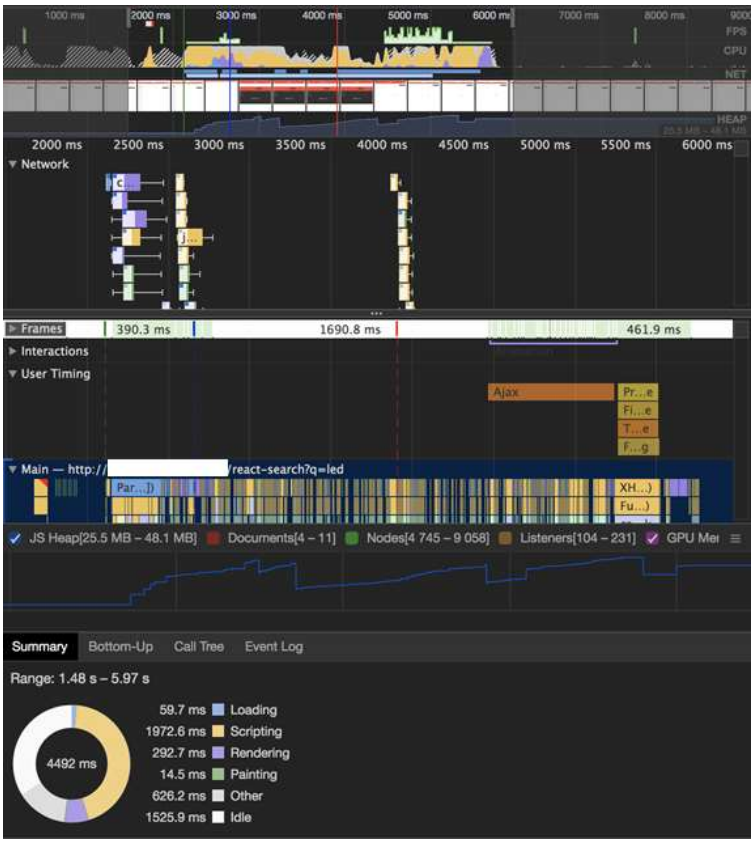

FIGURE 25. The timeline and summary from the run-time analysis.

The first and second rows in the figure 25 show the frames per second, and the CPU usage, which both are good indications for performance and are the first things that were looked at. The color code used helps to identity what is being processed, and the summary at the bottom shows, for a selected timeframe, the amount of time used for each type of process.

One of the first steps for measuring performance was to pinpoint the rendering part of the timeline (after products are loaded and are going to be displayed on the page) and see how long it took. While this is a good indication of performance, it was going to be inefficient to gather meaningful metrics for both modules and multiple browsers by recording and analyzing the runtime manually for every case.

5.2 Gathering metrics

To be able to gather metrics on rendering performance, it was important to be able to define what constitutes rendering, as well as what can and cannot be measured in an automated way.

In the case of this project, when a search is performed it goes through the following steps:

- Creating the API request
- Making the API call
- Waiting for the response
- Processing response data
- Updating observables

After this, MobX and React libraries take care of updating the components on the page with the new data that the observables have, updating the DOM tree and finally painting the new tree to the page for the user to see. Logically, rendering is then everything that relates to updating the components on the page with the new data. It is the part that interested the author during this project because of the differences between Knockout and React in how component changes are handled.

Following this thought, it seemed logical that rendering performance should be measured from the moment the observables are updated, up until the page is visually changed before the eyes of the viewer.

When using the Chrome dev tools, it could be observed that the results were visually present on the page after:

- The DOM tree changed
- The browser processes the layout and styles of the page
- The browser paints the new DOM changes.

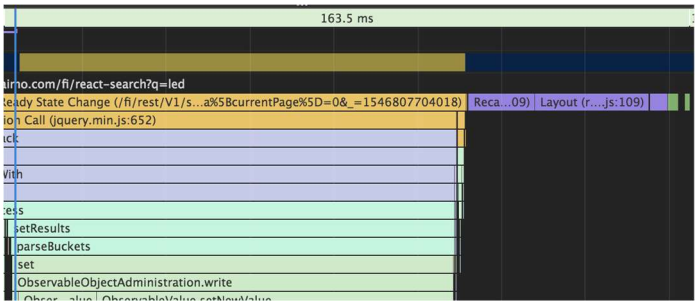

FIGURE 26. The timeline of the rendering of products on the page.

The figure 26 shows the timeline from the moment the observable is changed until the results show on the page. As can be seen, a large portion of the time is used for scripting, which in this case revolves around MobX and React processing changes to the DOM and updating the DOM tree. This is followed by the browser recalculating layout and styles (in purple color), then painting the changes (green color).

To be able to mark and measure different moments in the timeline of the loading and rendering of the search results, the Performance API of the browser was used. This allowed e.g., to mark “moments” in the execution of the code.

An example of how to use the performance API in JavaScript is shown below:

```
performance.mark(‘first_example_mark’);
…
performance.mark(‘second_example_mark’);
```

performance.measure('Name of the measurement', 'first_example_mark', 'second_example_mark');

This will measure the exact time that passed between the two marks and it will also be accessible from the browser console as well as being visible from the runtime analysis User Timings section.
In the case of measuring rendering performance for the new module (and the current one, for comparison), the first mark was added right before the line that sets the new value(s) for the observable(s).

The difficult part was to be able to mark the end of rendering, as the author wanted to record the DOM tree changes as well as the layout and styles calculations, and the painting, since the page visually changes only after the painting is done. After researching on the topic, no way was found that would programmatically detect the end of the painting that follows the DOM changes, leaving a possible rendering performance measurement to the time it takes for the DOM to be updated.

Initially, the way found to detect the end of DOM changes was to record the "componentDidUpdate" event from the React component (i.e. ProductItemsList component), but a more accurate way was found by using mutationObserver and recording the very next time a "mutation" occurred after the ProductItemsList component was updated. The Mutation observer is a web browser API that allows to detect when changes are made to the DOM tree. (13) The same markings and measurements were added to the Knockout module to have exactly the same way of measuring on both modules.

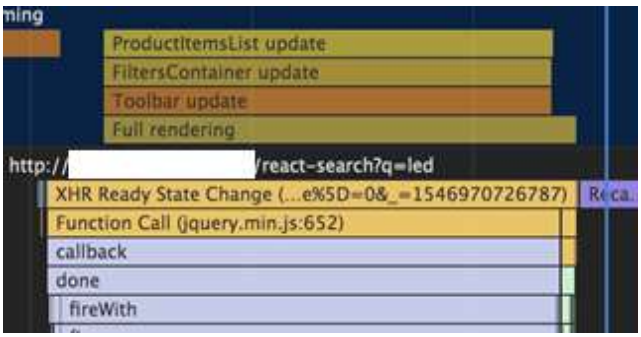

FIGURE 27. Different measures of rendering in the new module.

The figure 27 shows the difference between measures made with marking the componentDidUpdate event of the components compared with marking with the mutation observer (marked “Full rendering”). It measured right up to the end of the browser DOM tree changes.
The advantage of using the performance API to make the measurements is that markings and measures are accessible from the browser’s console at any time after measures are done, without the need to have the developer tools opened during the measurements. This way, measurements are truer to the reality, since using developer tools may temper with page performance (particularly observed on Internet Explorer).

The next step was to perform tests on multiple browsers and gather metrics for performance. As relevant metrics involve testing on multiple browsers, for a different amount of search results and for two different modules, it was suggested to use an automated testing framework to gather metrics without too much hassle.
After looking at different options, WebdriverIO for Selenium was chosen, mostly since it makes use of nodeJS, which the author thought would feel mode familiar to use.

Setting up the automated testing framework

Here are the steps that were taken to create and run the test scripts:

- Downloading and installing webdriverIO as instructed in the official documentation (13)
- Running the test runner configuration script with default parameters except when asked if the user wants to add a service, then Selenium Standalone is picked.

This was practical because Selenium Standalone is a Selenium server that is ran and shutdown when running tests, and it already has most browser drivers included.

- Defining browser capabilities in the configuration file.

- Writing the test scripts as JavaScript files under the /test/specs folder
- Running the scripts

As can be seen in the figure 28, the rendering measurement done using the Performance API can be read from the script by executing JavaScript on the browser during the test. Similarly, the number of products per page can be changed by modifying the localStorage value that the new module saves and uses for fetching products, and that was added to the current module for testing purposes.

```
browser.url('http://                /knockout-search?q=led');
$(waitForElementKnockout).waitForExist(20000);
browser.pause(3000);
const measureKnockout16items = browser.execute(function() {
    return performance.getEntriesByName('Full rendering')[0].toJSON().duration;
});
const setKnockoutLocalStorage32 = browser.execute(function() {
    localStorage.setItem('search-page-size', '32');
    return true;
});
```

FIGURE 28. A snippet of the script for test case 1.

When tests are running, the scripts make the browsers open and perform the actions that the script defines (i.e. loading a page, selecting an element, waiting for some condition) and take the measures using the previously shown method. At the end the script instructs the console to give the measures as output (figure 29) the values of which (in milliseconds) are saved in tables as shown in appendices 1 & 2.

```
------------------------------------------------------------
[0-0] KNOCKOUT:
16: 194.40004516028966
32: 306.39993147005043
[0-0] 64: 505.6997101293655
128: 1102.5000404626717
------------------------------------------------------------
[0-0] REACT:
16: 75.09990332382222
[0-0] 32: 84.90015983742342
64: 106.50012099325795
128: 162.4001904311517
```

FIGURE 29. A screenshot of an example test result for Test case 2 on Microsoft Edge

Each test was performed 10 times for each browser. The average rendering time from compiled results (see appendices) was calculated and used as a test result to create the following charts:

Test case 1: The search is made on a page load with a search term loaded from the URL. (Figures 30, 31, 32, 33 and 34 shown below)
A lower value means faster rendering and better performance.

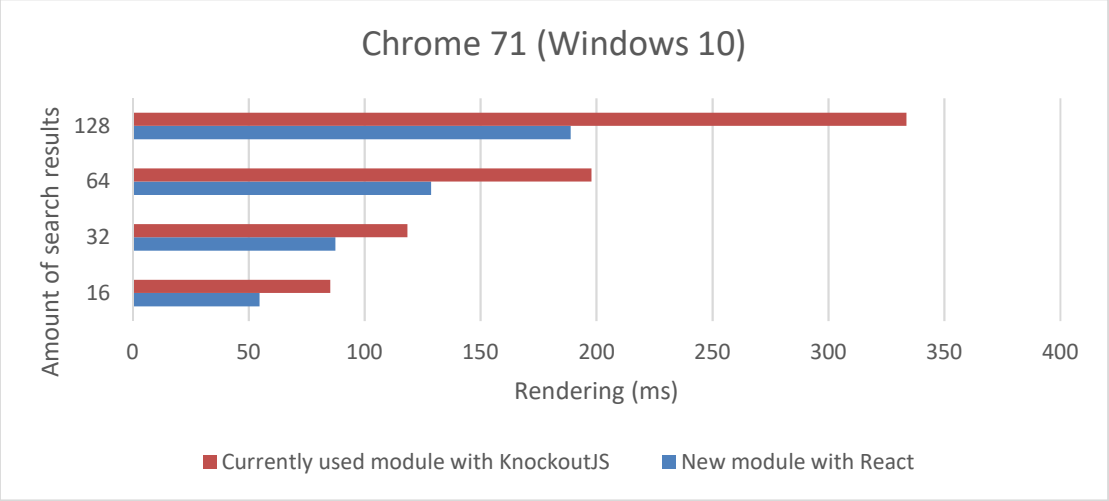

FIGURE 30. Rendering performance comparison on Chrome 71 / Windows 10

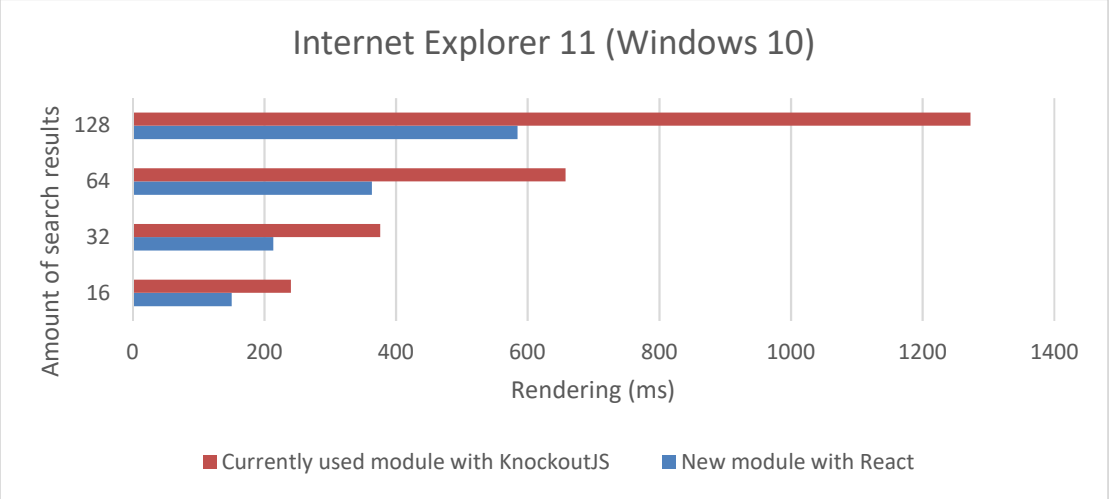

FIGURE 31. Rendering performance comparison on Internet Explorer 11 / Windows 10

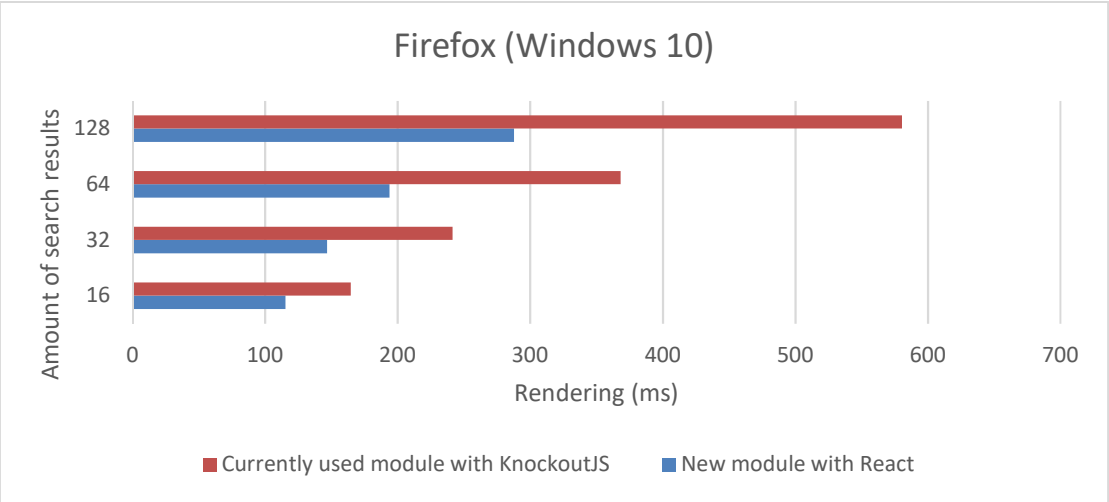

FIGURE 32. Rendering performance comparison on Firefox / Windows 10

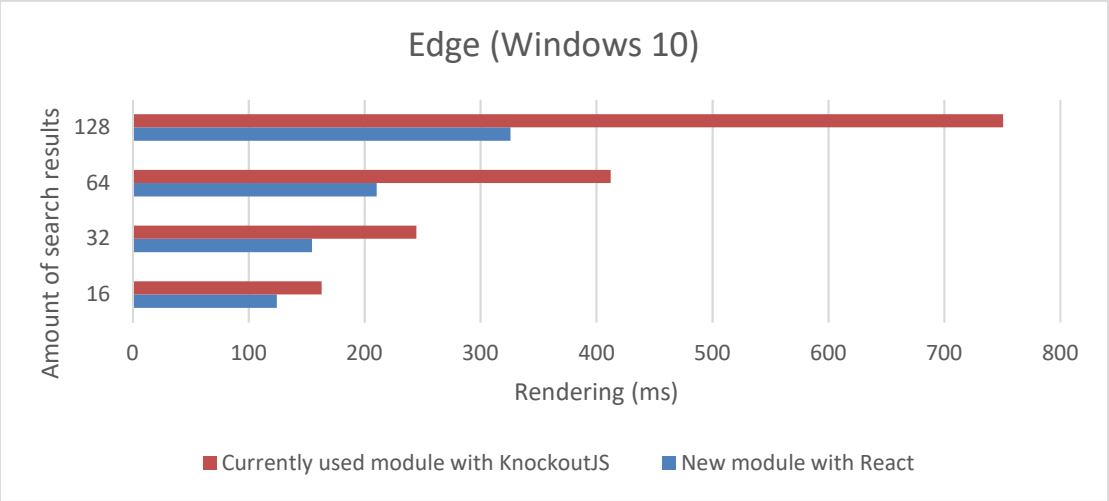

FIGURE 33. Rendering performance comparison on Microsoft Edge / Windows 10

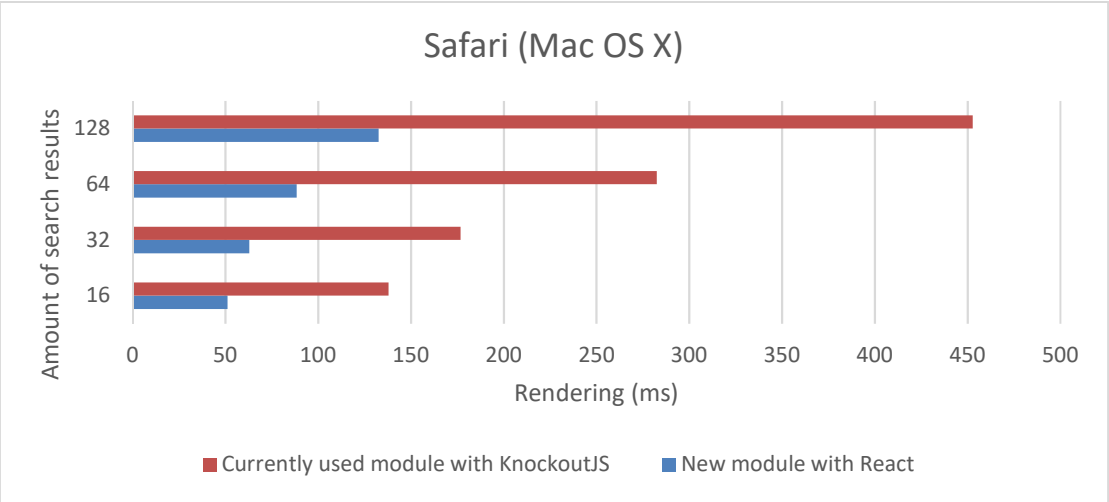

FIGURE 34. Rendering performance comparison on Safari / MacOS X

Results from the figures 30 to 34 show that, for the same testing conditions, the time it takes for the new module to render the components and products on the search interface is shorter than for the current module.

Test case 2: The search is opened and search results are showing. The user changes the sort order from “relevance” to “price (cheaper first)”. (Figures 35, 36, 37, 38 and 39 shown below)

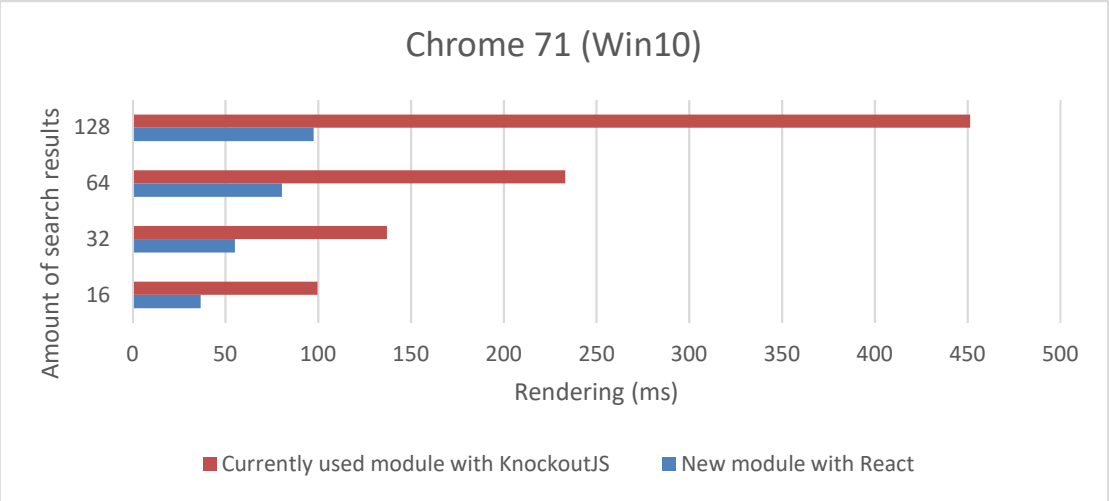

FIGURE 35. Rendering performance comparison on Chrome 71 / Windows 10

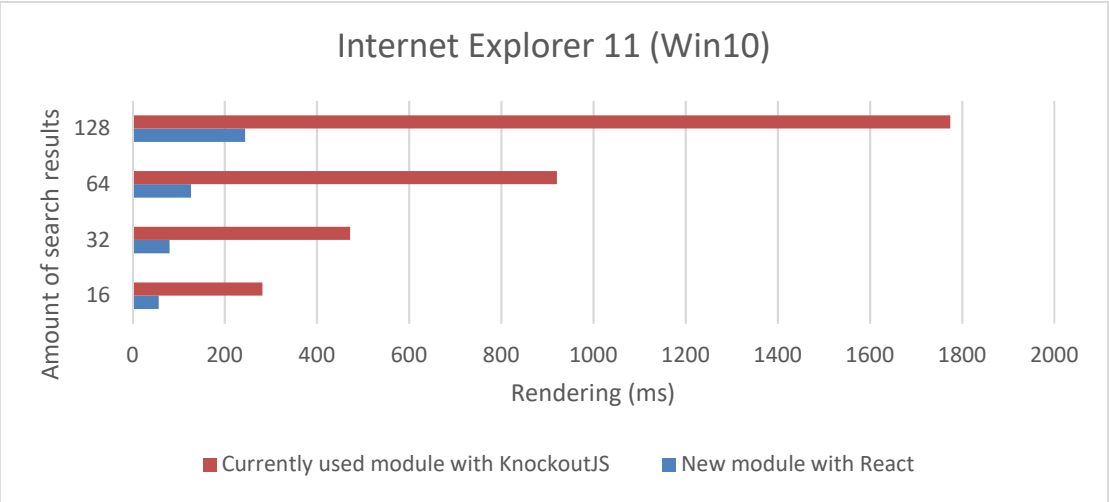

FIGURE 36. Rendering performance comparison on Internet Explorer 11 / Windows 10

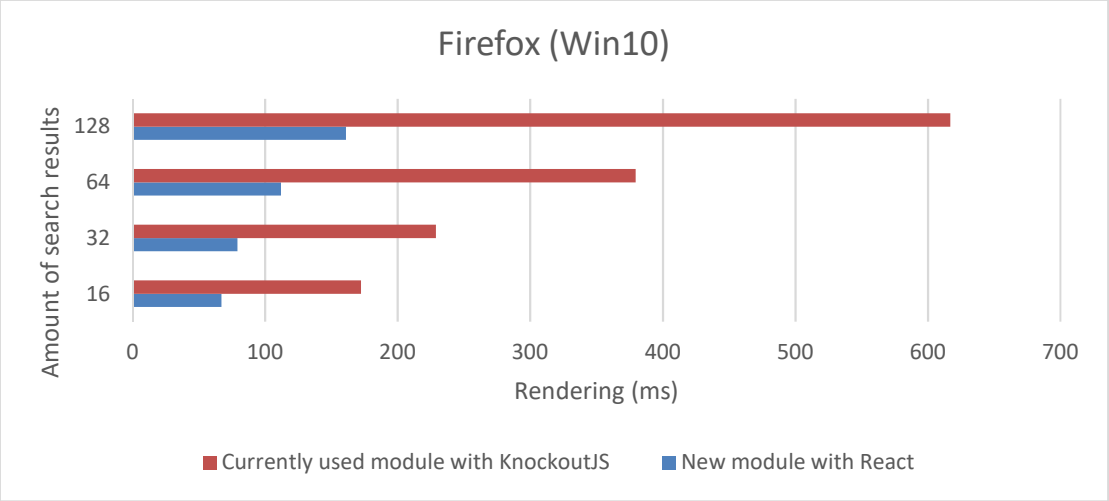

FIGURE 37. Rendering performance comparison on Firefox / Windows 10

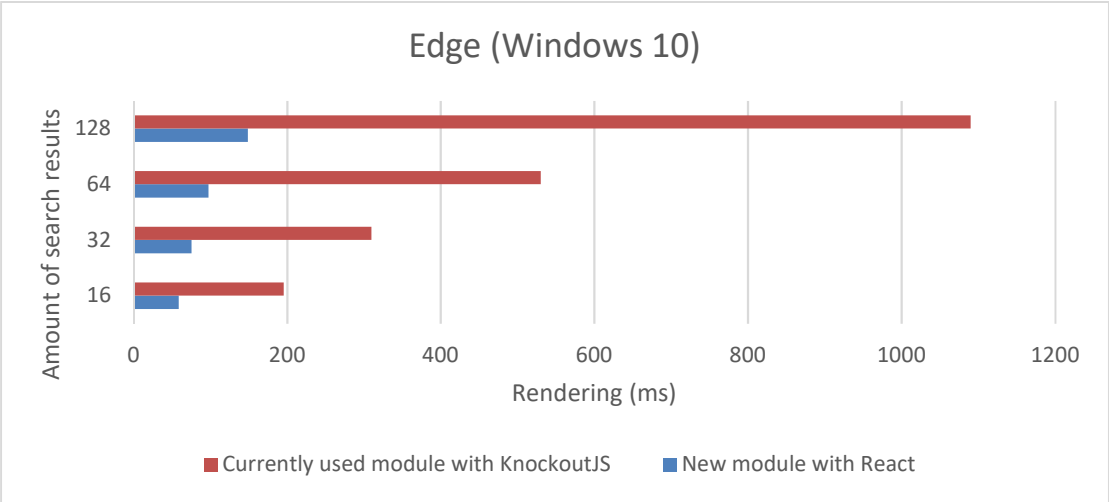

FIGURE 38. Rendering performance comparison on Microsoft Edge / Windows 10

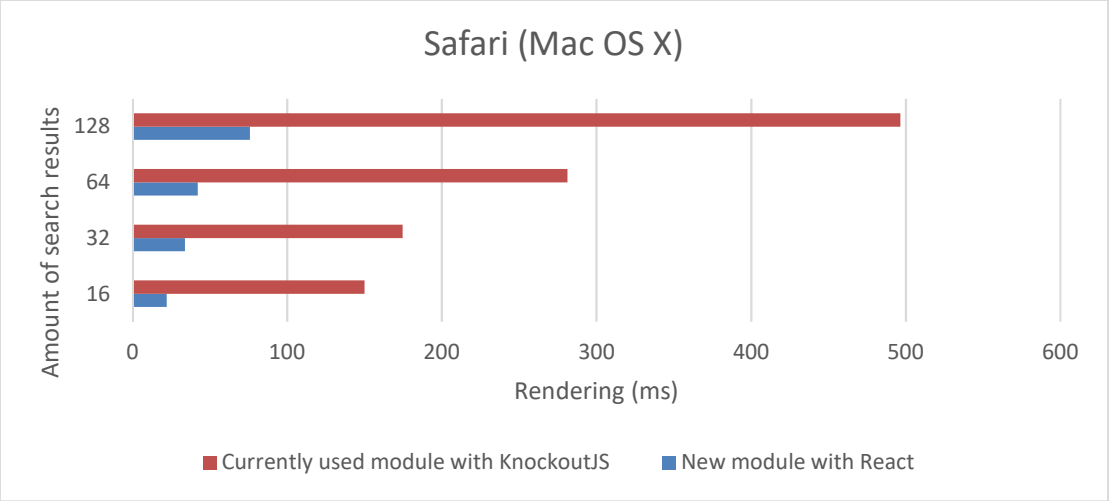

FIGURE 39. Rendering performance comparison on Safari / Mac OS X

The figures 35 to 39 show that for this test case, the difference is even more visible. This is a case where React really shines, since it effectively updates the parts of the DOM that need changing, which in this case is only product information, such as the name, SKU and price on the product cards.

Overall these tests helped to assert that this implementation using React and MobX had better rendering performance than the currently used module which was done using Knockout. This can be explained both by the fact that React has overall more performant rendering than Knockout, but also by having

performance in mind during the implementation, as well as the optimizations made to the components.

6 CONCLUSION

The main aim of this thesis was to create a brand-new module for Magento 2. It would replace a previously made module which was using Knockout as the library for the dynamic user interface. The module was asynchronously fetching product search results using a modified Magento search API.
Prior to starting this project, the observation was made that the module, while functional, was not very performant in older browsers such as Internet Explorer. React was assumed to be more performant and a perfect replacement for the frontend part of the module, to be made into a new module.

With performance in mind, a new user interface was implemented, making use of the backend and custom API from the current module but creating a brand-new user interface. Currently in a state of proof of concept, the new module implements most of the current module's UI functionalities into new React components.

Although it became quickly clear that the new module felt "faster", it was important for this thesis to be able to prove that the new module had more performant rendering, which the tests performed showed clearly.
The proof of greater performance will also be important in the future for showing to potential clients that would be interested in adding this new module to their website.

The module will undergo further development in the future, and it was an invaluable project for the author, in that it allowed to view the implementation of a user interface from the performance point of view and to learn about testing rendering performance of a website.

REFERENCES

1. Magento - The Magento Advantage. Date of Retrieval 21.10.2018
   https://magento.com/advantage

2. Wikipedia - Magento. Date of Retrieval 21.10.2018
   https://en.wikipedia.org/wiki/Magento

3. Magento DevDocs - Architecture. Date of Retrieval 21.10.2018
   https://devdocs.magento.com/guides/v2.1/architecture/archi_perspectives/arch_diagrams.html

4. QBOX - What is Elasticsearch. Date of Retrieval 2.12.2018
   https://qbox.io/blog/what-is-elasticsearch

5. Magento DevDocs - Performing Searches. Date of Retrieval 13.8.2018
   https://devdocs.magento.com/guides/v2.1/rest/performing-searches.html

6. Tero Parviainen - Change and its detection in JavaScript frameworks. Date of Retrieval 14.12.2018
   https://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

7. NPM - State of JavaScript frameworks 2017. Date of Retrieval 15.12.2018
   https://www.npmjs.com/npm/state-of-javascript-frameworks-2017-part-1#

8. Codeburst - MobX vs Redux with React: A noob's comparison and questions. Date of Retrieval 15.12.2018
   https://codeburst.io/mobx-vs-redux-with-react-a-noobs-comparison-and-questions-382ba340be09

9. CSS Tricks - Browser painting and considerations for web performance. Date of Retrieval 3.12.2018 https://css-tricks.com/browser-painting-and-considerations-for-web-performance/

10. Treehouse - Increase your site's performance with hardware accelerated CSS. Date of Retrieval 4.12.2018 https://blog.teamtreehouse.com/increase-your-sites-performance-with-hardware-accelerated-css

11. WebdriverIO – Getting started. Date of Retrieval 13.2.2019 https://webdriver.io/docs/gettingstarted.html

12. Vaimo.com – About Us. Date of Retrieval 14.2.2019 https://www.vaimo.com/about-us/

13. MDN Web docs - Mutation observer. Date of Retrieval 14.2.2019 https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver

14. ReactJS - Error Boundaries. Date of Retrieval 16.2.2019 https://reactjs.org/docs/error-boundaries.html

15. ReactJS – Tutorial: Intro to React. Date of Retrieval 28.2.2019 https://reactjs.org/tutorial/tutorial.html

APPENDICES

Appendix 1 Metrics from Test Case 1 rendering performance testing

Appendix 2 Metrics from Test Case 2 rendering performance testing

OS:	**Browser**				Knockout						
Win10	Chrome 71										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	85	87	83	88	85	87	84	92	82	79	85.2
32	115	115	119	118	119	126	120	119	117	117	118.5
64	194	197	194	191	194	193	218	194	211	192	197.8
128	326	333	334	331	329	334	332	347	332	338	333.6

OS:	**Browser**				React						
Win 10	Chrome 71										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	55	54	63	53	54	54	54	53	53	54	54.7
32	89	90	93	83	86	91	85	86	84	87	87.4
64	131	126	142	130	128	126	125	129	123	127	128.7
128	188	191	184	194	189	189	193	184	180	197	188.9

OS:	**Browser**				Knockout						
Win10	Internet Explorer 11										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	230	236	233	235	231	248	257	235	253	248	240.6
32	361	367	369	371	366	372	396	381	382	383	376.3
64	669	656	649	649	657	663	670	644	670	649	657.6
128	1286	1303	1266	1310	1267	1268	1235	1254	1289	1248	1272.6

OS:	**Browser**				React						
Win10	Internet Explorer 11										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	142	141	155	189	177	143	142	134	141	140	150.4
32	205	204	240	239	240	201	206	202	198	202	213.7
64	381	394	348	384	387	338	372	359	341	333	363.7
128	606	620	560	600	614	599	558	567	563	562	584.9

OS:	**Browser**				Knockout						
Win10	Firefox										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	170	165	160	163	163	167	166	165	157	170	164.6
32	241	244	233	236	244	233	241	253	247	242	241.4
64	375	365	365	376	375	364	360	366	366	371	368.3
128	573	557	577	575	571	589	565	633	576	588	580.4

OS:	**Browser**				React						
Win10	Firefox										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	115	121	120	112	125	109	111	105	125	112	115.5
32	142	147	145	138	152	150	148	150	147	148	146.7
64	198	192	187	188	189	186	194	196	208	201	193.9
128	261	265	252	317	468	263	261	259	268	264	287.8

OS:	Browser				Knockout						
Win10	Microsoft Edge										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	164	163	165	159	164	163	161	166	161	165	163.1
32	253	238	239	245	255	249	242	246	240	240	244.7
64	429	403	403	419	411	421	412	413	411	402	412.4
128	767	762	738	730	754	747	728	759	755	766	750.6

OS:	Browser				React						
Win10	Microsoft Edge										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	102	144	143	153	149	98	121	108	95	130	124.3
32	136	152	128	170	139	149	146	147	182	200	154.9
64	200	201	204	209	203	225	218	214	220	212	210.6
128	321	328	312	320	307	316	334	339	343	339	325.9

OS:	Browser				Knockout						
Mac OS X	Safari										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	90	82	192	169	155	95	177	166	87	166	137.9
32	229	225	122	197	140	134	181	199	213	127	176.7
64	296	296	208	293	299	300	289	309	259	277	282.6
128	468	439	459	446	469	450	456	439	462	440	452.8

OS:	**Browser**				React						
Mac OS X	Safari										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	50	50	54	49	49	49	57	57	49	48	51.2
32	63	62	62	62	64	60	58	77	59	63	63
64	84	88	113	82	83	81	82	83	100	88	88.4
128	131	135	129	133	129	132	120	132	129	156	132.6

OS:	**Browser**				Knockout						
Win10	Chrome 71										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	112	99	101	95	101	103	96	93	101	96	99.7
32	136	136	146	134	136	139	137	141	134	132	137.1
64	239	246	235	229	223	245	225	234	222	234	233.2
128	447	457	475	442	451	445	420	451	457	469	451.4

OS:	**Browser**				React						
Win 10	Chrome 71										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	41	35	39	35	37	35	36	36	36	37	36.7
32	54	54	57	56	51	53	57	55	52	62	55.1
64	76	85	81	86	78	83	77	84	69	86	80.5
128	102	108	105	88	100	103	86	97	82	105	97.6

OS:	**Browser**				Knockout						
Win10	Internet Explorer										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	274	280	284	278	277	280	284	287	298	278	282
32	476	474	491	472	476	464	464	470	464	472	472.3
64	946	1017	890	933	957	875	880	956	878	879	921.1
128	1736	1770	1812	1746	1739	1749	1809	1779	1822	1785	1774.7

OS:	**Browser**				React						
Win10	Internet Explorer										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	69	49	69	73	70	47	49	47	48	49	57
32	92	92	76	93	68	72	73	69	70	100	80.5
64	119	178	118	120	117	115	118	135	135	118	127.3
128	286	205	215	212	207	208	247	286	295	282	244.3

OS:	**Browser**				Knockout						
Win10	Firefox										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	189	175	175	174	180	172	150	170	168	170	172.3
32	213	269	213	212	288	225	216	215	214	224	228.9
64	342	332	420	411	339	424	343	416	350	419	379.6
128	682	669	579	560	684	600	669	578	573	575	616.9

OS:	**Browser**				React						
Win10	Firefox										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	66	68	69	69	59	68	59	70	75	69	67.2
32	87	74	66	74	91	77	90	84	68	81	79.2
64	98	115	124	116	114	93	129	92	119	120	112
128	169	162	176	160	164	173	172	152	120	163	161.1

OS:	**Browser**				Knockout						
Win10	Edge										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	205	199	195	195	193	197	198	196	189	189	195.6
32	301	354	297	310	317	311	300	301	301	304	309.6
64	507	536	547	546	523	525	523	540	523	534	530.4
128	1074	1095	1102	1070	1106	1137	1102	1060	1066	1088	1090

OS:	**Browser**				React						
Win10	Edge										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	47	45	49	45	64	69	76	72	60	61	58.8
32	84	78	91	61	79	65	87	75	62	73	75.5
64	95	95	98	90	93	111	93	103	93	105	97.6
128	148	149	146	151	147	150	153	154	146	147	149.1

OS:	**Browser**				Knockout						
Mac OS X	Safari										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	119	397	128	125	115	133	124	126	116	117	150
32	169	181	185	173	165	191	170	164	178	171	174.7
64	282	277	285	285	286	288	297	288	274	251	281.3
128	484	494	501	495	493	538	500	495	480	486	496.6

OS:	**Browser**				React						
Mac OS X	Safari										
Amount of products	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
16	22	20	21	25	30	21	18	21	20	23	22.1
32	30	30	76	27	27	31	27	29	34	29	34
64	40	44	41	43	42	35	46	44	41	45	42.1
128	76	77	75	75	70	77	70	72	102	65	75.9