

Tuomas Luojus

# USABILITY AND ADAPTATION OF REACT HOOKS

# ABSTRACT

Tuomas Luojus: Usability and Adaptation of React Hooks  
M.Sc. Thesis  
Tampere University  
Master's Degree Programme in Software Development  
April 2021

---

In this study, usability and adaptation of a new feature to React framework – hooks – is inspected. The usability inspection is done according to API usability evaluation principles based on earlier research. The analysis of adaptation of hooks is based on the experiences of professional React developers and how their projects have adapted to using hooks.

Web as a software platform has been increasingly popular in recent times. This has been enabled by improvements in browser and computer performance. Web applications are attractive to users due to their lack of need for installation and capability to work on most devices available to users. As the popularity of the web has increased, so has the number of solutions available for developers. This has led to short lifecycles of development tools, constant emergence of new technologies, and rapid shifts in industry standards. Therefore, many developers have felt overwhelmed by the number of solutions. For the regular developer, there is a need for standard, high quality, and high usability solutions. This thesis argues that the solution for this is thoughtful development and usability evaluation of web development software.

One impactful new solution is React hooks. Hooks were introduced to React JavaScript UI framework in 2018 as a part of React moving from less compact class components to seemingly more clear function components. Hooks allow function components to have local state and other procedural features whereas before, only class components could include them. In present time, React development community has largely abandoned class components in favor of function components.

To evaluate the usability and adaptation of hooks as an API, two studies were conducted. Firstly, a case study was done. In the case study, three general purpose components were written as class components and function components. The components were then compared with each other to find usability issues. Secondly, six professional React developers were interviewed in order to find hooks related issues from real experiences of React developers.

It is concluded that the developers have almost entirely replaced class components with function components and hooks their projects. However, while hooks have accomplished their purpose well in most cases, there are still critical cases where they have serious usability issues or do not functionally provide sufficient replacement for class components.

Key words and terms: API usability, hooks, React, function components, state management

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

# Contents

<b>1</b>	<b>INTRODUCTION</b> .....	<b>1</b>
<b>2</b>	<b>WEB SOFTWARE ARCHITECTURE</b> .....	<b>3</b>
2.1	SOFTWARE ARCHITECTURE .....	3
2.1.1	Defining Software Architecture.....	3
2.1.2	Components and Interfaces.....	5
2.1.3	UI Architecture and MVC .....	5
2.2	SOFTWARE STATE.....	6
2.2.1	State in Software Development .....	7
2.2.2	Modeling Software State .....	7
2.3	WEB APPLICATIONS.....	8
2.3.1	Web Application Architecture Overview .....	9
2.3.2	Static and Dynamic Web Sites .....	10
2.3.3	Web as an Application Platform.....	10
2.3.4	AJAX.....	11
2.3.5	Single Page Application .....	11
2.3.6	JavaScript .....	12
2.3.7	JavaScript UI Frameworks .....	13
<b>3</b>	<b>REACT UI FRAMEWORK</b> .....	<b>16</b>
3.1	REACT OVERVIEW .....	16
3.2	COMPONENTS .....	17
3.2.1	Class Components .....	17
3.2.2	Function Components.....	18
3.2.3	Function Components With Hooks.....	18
3.3	COMPONENT LIFECYCLE.....	19
3.4	REACT STATE MANAGEMENT .....	21
3.4.1	Local State .....	22
3.4.2	Global State Management with Redux .....	22
3.4.3	Other Global State Management Libraries .....	23
<b>4</b>	<b>RESEARCH METHODS</b> .....	<b>25</b>
4.1	CASE STUDY .....	26
4.2	DEVELOPER INTERVIEWS .....	26
4.2.1	Participants .....	27
4.2.2	Data Collection .....	28
4.2.3	Data Analysis.....	29
<b>5</b>	<b>CASE STUDY</b> .....	<b>31</b>
5.1	COMPONENT 1: DATETIMEDISPLAY .....	31
5.2	COMPONENT 2: LOGINVIEW .....	33
5.3	COMPONENT 3: USERPREFERENCESWINDOW .....	35
5.4	CASE STUDY FINDINGS .....	36
<b>6</b>	<b>DEVELOPER INTERVIEWS</b> .....	<b>38</b>
6.1	REACT AND OTHER FRAMEWORKS .....	38
6.1.1	React Overall .....	38
6.1.2	React Architecture .....	39
6.1.3	React Performance.....	39
6.1.4	Global State Management with React .....	40

6.1.5	Function and Class Components .....	42
6.2	FUNCTIONALITY OF HOOKS .....	42
6.2.1	Built-in hooks .....	43
6.2.2	Lifecycle Management .....	43
6.2.3	Custom and Third Party Hooks .....	44
6.3	HOOKS BASED REACT DEVELOPMENT.....	44
6.3.1	Readability.....	44
6.3.2	Ecosystem and Collaboration .....	44
6.3.3	Testability.....	45
6.4	ADAPTATION TO HOOKS .....	46
6.4.1	Learning Hooks .....	46
6.4.2	Transition to Hooks .....	46
6.4.3	Refactoring Class Components.....	47
6.5	INTERVIEW FINDINGS.....	47
<b>7</b>	<b>FINDINGS AND DISCUSSION .....</b>	<b>48</b>
7.1	FINDINGS .....	48
7.2	RELIABILITY .....	49
7.3	FUTURE RESEARCH.....	50
<b>8</b>	<b>CONCLUSION.....</b>	<b>51</b>
	<b>REFERENCES.....</b>	<b>53</b>

## Appendices

## 1 Introduction

The rapid growth of the web as a software platform [Taivalaari and Mikkonen 2017] has increased the need for highly usable web development tools and solutions for professional application development. During the last decade, web development has experienced short cycles of industry standard development tools rising and falling. Every time a new tool rises to industry standard, developers have to spend time learning them and adapting to them only to have to abandon them again after a few years [Taivalaari and Mikkonen 2017]. The current industry standard frontend framework for JavaScript is *React* [Greif and Benitte 2021]. React is an open source library originally created by developers at Facebook [Facebook Open Source. 2021a].

React has two types of components: class components and function components. Before hooks were introduced to React, it was not possible to include procedural features such as state and lifecycle management in function components natively. Therefore, if a stateful component had to be made, it had to be written as a class component. Meanwhile, the interviewed developers agreed that class components were deemed difficult to understand, compartmentalize, and manage. [Facebook Open Source. 2021b]

In 2018, a new solution – *hooks* – was introduced to React. Hooks brought state and other features from class components to function components. According to React developers interviewed for this study, the React developer community has gradually adapted to hooks-based React development. This study aims to inspect how that adaptation process has been for React developers, and how successful are hooks as a replacement for earlier class-based solutions from a usability perspective. Since the introduction of hooks, class components have been gradually replaced by function components in React development.

Few studies have been done on specific web development frameworks or solutions such as React. Most academic research on software and application architecture focuses on theoretical ideas of designing and analyzing software instead of analyzing specific existing frameworks. As the currently most used JavaScript frontend framework, React ought to be studied in more detail.

Traditionally, comparisons of programming frameworks and languages have been done by comparing their performance and technical details, but since computers and browsers are getting increasingly faster and more efficient, usability evaluation of software solutions for developers has become increasingly meaningful. Hooks were thus a good fit for this study, since their main motivation is to make development easier and increase the usability of React for developers. There is little effect on performance when using function components with hooks compared to class components.

The purpose of the study is to add support in favor of thoughtful usability-based development and usability evaluation of technology solutions. Especially in the fast-moving

web ecosystem, deliberate and attentive care ought to be used when introducing new software solutions to the development ecosystem. This way, the ecosystem will be steadier leading to fewer difficulties and less risky decisions for developers and better software.

Both the code-specific aspect of hooks and actual experiences of React development with hooks are analyzed in order to get a holistic view of the usability and adaptation of hooks. The research questions are: (1) *What usability benefits and drawbacks do hooks bring to React?* And (2) *How have project teams adapted to React hooks?*

The study was conducted by first reviewing the existing literature on web software architecture. Then, a case study was conducted by comparing two implementations of a program: one written with class components, one with function components and hooks. By this comparison, technical aspects of hooks were inspected. Finally, an interview of React developers were conducted, where real life experiences and opinions of new hooks-based React development were analyzed.

From the study, ten positive and six negative usability issues of hooks were recognized. The overall usability of hooks was recognized to be positive in typical use cases compared to class-based React, while some serious usability problems were present in more specific cases.

Chapter 2 presents important concepts and ideas as a context for React hooks and the study. In Chapter 3, core aspects of the React framework are explained. Chapter 4 covers the methodologies of the case study and the developer interviews in more detail. Chapters 5 and 6 cover the results of the case study and the developer interviews, respectively. Chapter 7 combines the findings of the two studies, presents the finally found usability issues, and offers some discussion on the research done. Finally, the conclusions of the research are drawn in Chapter 8.

## **2 Web Software Architecture**

In this chapter, the needed background information on web software architecture and web applications will be presented as a base for the study. The chapter will start from basics of software architecture and move gradually towards web architecture and finer details of web development. Important terms will be defined, and essential ideas will be presented based on literature of the field of software architecture.

The structure of the chapter is as follows. Firstly, the term *software architecture* will be inspected and defined based on the literature of the field. Then, the notion of *state* will be inspected. Finally, different aspects of web applications, such as predominant models and architectures in web application development will be inspected.

### **2.1 Software Architecture**

Software architecture does not differ much from “regular” construction architecture in principle. An architect designing a sauna building has to first recognize the smaller parts that are included in the concept of a sauna: the door, the walls, the roof, the chairs, and the stove. Then they design how to connect them in relation to each other, often drawing a model of it. A software architect divides a large software into smaller pieces, and designs how to connect them, often producing some type of visual model.

Software architecture as a distinct field of study started in the 1990’s [Kruchten et al. 2006], after which it has been adapted as a part of regular software development process. However, recently there has much discussion in the software development field whether software architecture is needed in the era of modern agile development where it has been nicknamed BUFD (Big Up-Front Design) [Gruhn and Rüdiger 2018, 178]. Despite this, software architecture is still an important part of agile software development – it has only transformed from large documentation and long planning phases to iterative refactoring, frequent communication, and reference to design decisions [Abrahamsson et al. 2010; Keeling 2015].

In different contexts, the term software architecture gets often used for various different situations. In everyday discussion, it can sometimes mean any higher-level concept of a software product, so some specificity is needed when studying the topic.

#### **2.1.1 Defining Software Architecture**

Software Architecture has been defined in several ways, often differently by different parties. In this chapter, a few definitions of software architecture will be inspected, and at finally combined into a definition of software architecture which will be utilized for this study.

IEEE [2000] has a standard for architectural description. It defines architecture as an organization of a system that consists of components, the components’ relationships with each other and the environment, and different principles guiding the architectural design

and evolution. Notable in their definition is that it not only includes dividing the system into smaller pieces, but also considers relationships, stakeholders, and the temporary evolution of the software product during its life cycle from requirements definition to termination of use.

Bass et al. [2013, Ch. 1] have defined software architecture as a *bridge between business goals and the resulting system*. They write that software architecture of a system includes the structures of that system, which includes software elements, their relations, and their properties. Notably, they argue that software architecture only includes information that is important outside of a single element. Thus, the architecture is not concerned with the internal workings of an element. As such, software architecture is an abstraction that includes relevant information about the system and omits irrelevant information. They especially assert that the software architecture is what allows teams of software developers to work on different parts of a product simultaneously regardless of organizational, geographical, or time-zone restrictions. The individuals or groups do not need to know how the others' software works; they only need to know what kind of interface it offers with which they can interact.

Gruhn and Rüdiger [2018, Ch.1] have also characterized software architecture as ordering a large system to smaller elements and descriptions of their connections and behaviors in relation with each other via *interfaces*. They emphasize that architecture is not what is implemented in the code but a set of structures and rules that guide the design of the software product until completion, leading to code that represents the architecture.

The definition of Wills [1998 p. 482] also includes software's parts and their essential external qualities, and the relationship of those parts. They emphasized that architectures were not only drawings on paper, but rigid structures and rules of the natures, roles, and relationships of the parts of a system.

Soni et al. [1995] found four categories of architecture by inspecting software architecture in industrial context: (1) conceptual architecture, which describes the system by its functional components and their interfaces, (2) module architecture, which describes the ideal structure of the system, (3) execution architecture, which describes the dynamic structure of the system by its run-time elements, and (4) code architecture, which describes the structure of the source code base.

Martin [2017, Ch. 1] argues that there is no difference at all between design and architecture. He looks at it from the point of view of productivity: The faster the product gets done, the better the architecture. For the definition of architecture, he follows the others: the division of a system into components, the arrangement of those components, and their communication between each other [Martin 2017, Ch. 1; Ch. 15].



In conclusion, while some fine details differ, most of the literature defines software architecture as dividing a system into smaller elements or components and their relationships. Typically, their relationships are described by the interfaces they offer, and by the interfaces of other components they use themselves. Good software architecture can be seen as a counter measure to complicated, entangled, or monolithic software. It enables groups of people to work on a same project without having to know all the specificities of code written by others – only the interfaces. This overview does not consider what the potential systems are or their context. For example, they can be inspected from the viewpoint of functions or software components. [Gruhn and Rüdiger 2018, Ch. 1]

### 2.1.2 Components and Interfaces

Systems can be decomposed into components (also known as subsystems) that are connected by their interfaces. A component is an independent software piece that offers some functionality to the system via an offered interface so other components can use it. Components can be modified by altering their state, interfaces, or by creating new components that inherit them. [Gruhn and Rüdiger 2018, Ch. 1]

Ever since the concept of software components was conceived, there has been a vision for a type of software development, where the developers would have to write minimal software themselves while most of the development process would include structuring and combining existing software components to create new software products. While the use of components-based architecture and modal software has increased, this vision has not yet fully realized. [Caldiera and Basili 1991; Holzmann 2018]

Interfaces, or APIs connect the components to each other. Gruhn and Rüdiger [2018, Ch. 1] defined three types of interfaces: *export interfaces* that offer the components' functionality to the system, *import interfaces* which define what the component requires from the system, and *assumption/commitment interfaces* which describe assumptions that the component has of the system, and commitments if the assumptions hold (Figure 1).



Figure 1. System of two components connected by their interfaces. [Gruhn and Rüdiger 2018, Ch. 1]

### 2.1.3 UI Architecture and MVC

Several toolkits and interface builders have been available for UI development since the 1970's. Since modern application UIs can usually be divided into different visual and

functional components quite intuitively, a component-based approach has been a natural choice for UI architecture. [Myers et al. 2000]

*MVC* or *model-view-controller* is a common architecture for developing user interfaces presented by Krasner and Pope [1988]. MVC makes a division between three parts of a user interface: **model**, which is the application's domain-specific software simulation or the central structure of the application, **view**, which displays the state of the application for the user, and **controller**, with which the user can control the model and the view (Figure 2). Building user interfaces of applications in this manner, the developers of one part does not have to have total understanding of the other two parts, making the development more efficient. MVC architecture is used in a great number of user interface toolkits, and is well known by UI developers, as it is general enough to fit most UIs.

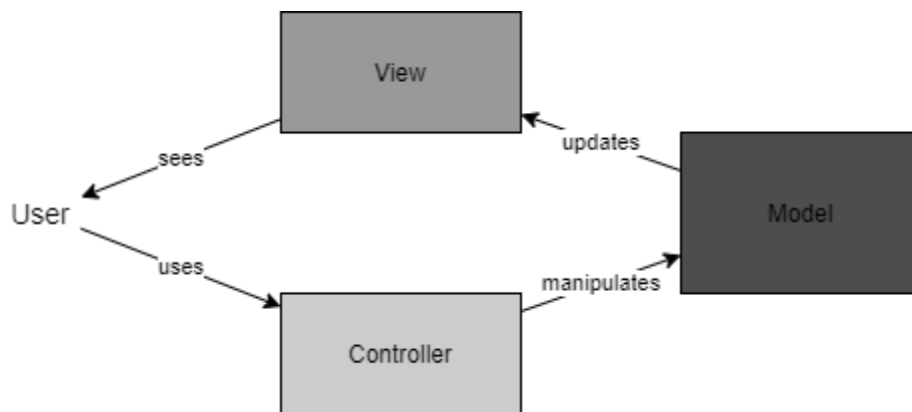


Figure 2. MVC pattern [Krasner and Pope 1988]

For example, in a typical MVC implementation of a checkbox control, the model is the Boolean data whether the checkbox is selected or not. The control is the event handler that handles the mouse click from the user's device and manipulates the model. The view is then the visible box and checkmark rendered on the screen of the user's device with pixels lighting up.

## 2.2 Software State

*State* in software refers to any configuration, context or history related information saved in the memory of an application, typically as a variable. Software uses its state to alter its behavior. Thus, a *stateful application* might have different outputs depending on the state of the application whereas a *stateless application* will always return the same output when given the same input. Happe et al. [2014] identified three types of state categories: component-specific, system-specific, and user-specific state. These categories can be further divided into subcategories depending on whether they work in *runtime*, *deployment time*, or *instantiation time*.

*Component state* means the state encapsulated and accessible inside a specific component. The component state is furthermore divided into four subcategories: Protocol state, internal state, allocation state, and configuration state. *System state* is the state of the whole software product shared between different components. Finally, *User-specific state* is any data stored for specific users of the system. [Happe et al. 2014]

### 2.2.1 State in Software Development

There are a few considerations to be taken into account when developing stateful software. Firstly, quality assurance becomes difficult when a software can have indefinite number of possible state combinations. Whereas stateless software is usually quite easy to test, writing automated test cases for every possible state combination of a stateful software is highly difficult if not impossible. For example, if a software has a user modifiable UI component with tens of thousands of possible state combinations, it is not usually feasible to write tests for all those state combination possibilities. One proposed way of testing stateful software are *exploratory test agents*; automated actors that explore the system under testing to expose faults and report them to the developers [Karlsson 2019].

This issue of testing stateful software is particularly relevant with UI software. While manual human testing is perhaps the most reliable way to test application UIs, it is costly in time and resources, whereas automated testing is often either lacking in test coverage or too difficult to implement. There have been solutions proposed such as AppFlow that have been somewhat successful mitigating this issue with automated testing of stateful applications. [Hu et al. 2018]

Another issue with stateful software development is that when state is included in a software, it becomes difficult to transfer state between sessions. For example, if a user closes a web browser running a web application, all of the session-specific state information is lost. In the case of web, there are solutions of persisting state between sessions such as storing session information to the backend server and accessing it with cookies or other user information at the beginning of the following session.

Some web application developers have opted for building applications with stateless user interfaces, where the user-related information is saved to and accessed from the cache or the backend with network requests each time there is changes to the state of the application. This is known as *Remote Session* architecture [Fielding 2000, Ch. 3]. With good caching and thoughtful data flow management, it is possible to achieve more scalable software with stateless architecture.

### 2.2.2 Modeling Software State

One way to model software state is the *finite-state machine (FSM)*. An FSM is an abstraction that models a machine that can be in exactly one state at a time. As its name implies, an FSM has a finite number of possible states. For example, a simple elevator can only

be in one of three states: (1) going up, (2) going down, or (3) being still. The change of state in FSMs is called a *transition*. Furthermore, a more modern elevator can be in a state defined by the sequence of stops requested by its users. Similarly, stateful software can be modeled by the finite number of its possible states at any time.

*Statecharts* are an extension to finite state machines with added hierarchical properties [Harel 1987]. With statecharts, finite state machines can be represented in a modular and hierarchical way, with which it is simpler to see the state logic at a glance. While infinite state machines and statecharts are a rather old concept, they are still used in complex software state management architecture now. Figure 3 visualizes an Statechart representation of an FSM model of a button that activates only when the above checkbox in checked.

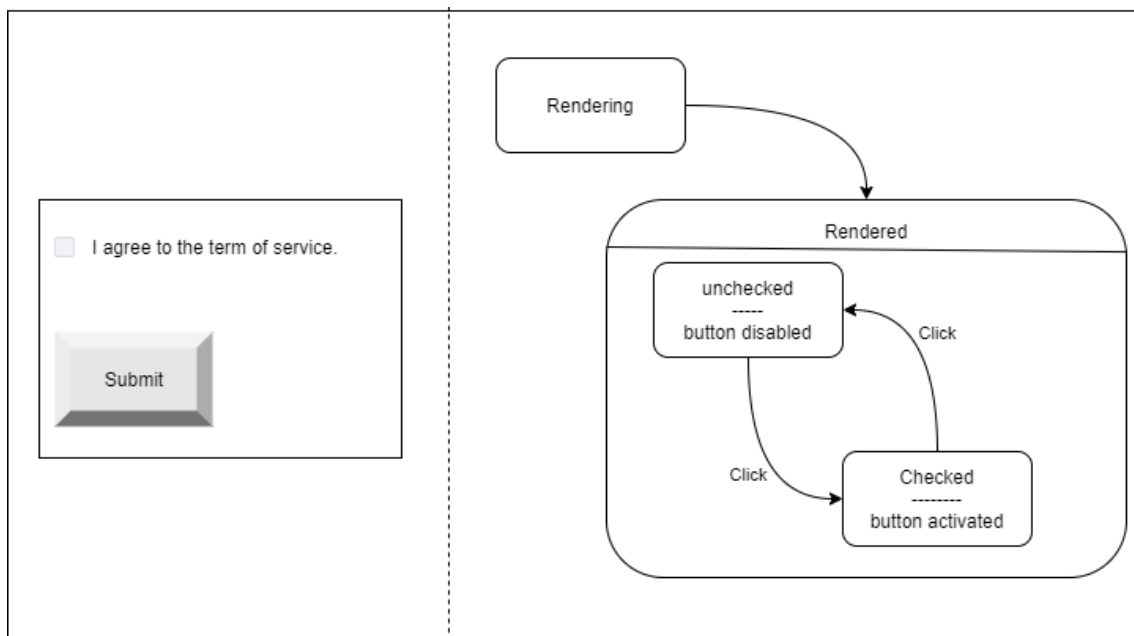


Figure 3 Statechart representation of a common UI element as an FSM model

### 2.3 Web Applications

Web application is a software application that is accessible through web via an internet browser. Web applications do not have to be installed separately, and they work with most devices that have a modern browser. In recent years, the computational power of browsers has increased significantly, which has led to a large focus shift towards web applications.

The web has moved on from its original function of sharing documents and simple multimedia content. Already in 2011, Taivalsaari et al. [2011] documented that the web

had become the predominant application platform. They also foresaw the increase of programming capabilities and interfaces in the web browser and the continuation of the trend towards web-based software. As key characteristics that drove the application development towards the web, they saw the lack of requirement for manual installation or manual upgrades, instant worldwide deployment, and open application formats that enable combinations of content or *mashups*.

Another reason of the rise of the web has been the increased performance of the browser and the JavaScript language, which have highly increased the user experience of web applications [Taivalsaari and Mikkonen 2017; Wagner 2017].

Despite the recent growth of web-based applications, there have been worries on the future of the viability of web applications. Mikkonen et al. [2019] have expressed worries about the *cornucopia*, or abundance of features available for web development. All the different APIs that provide abstractions to applications often contain very different development styles and paradigms. This problem, if not treated appropriately, could even lead to software application development moving away from web-based solutions.

### 2.3.1 Web Application Architecture Overview

Web application architecture includes the browser, the network, and the web server. The browser sends HTTP requests to the server via the network infrastructure, to which the server responds, again via the network (Figure 4). Thus, in the MVC model, the browser takes care of the view, whereas the controller and the model can be distributed in any number of ways between the client and the server. [Conallen 1999]

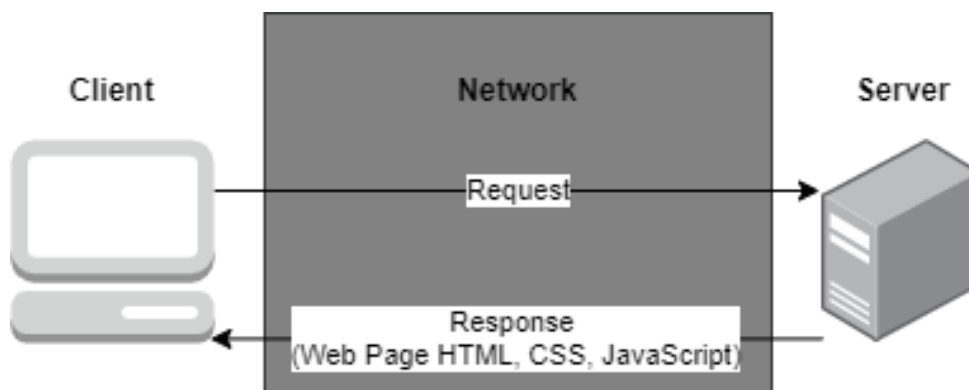


Figure 4. Basic web architecture

Modern web applications consist of three parts, each with their own responsibilities: **HTML** (Hypertext Markup Language) that defines the semantic structure of the page in a tree structure also known as the DOM (Document Object Model), **CSS** (Cascading Style Sheets) that provides the visual style, and **JavaScript** that allows programmatic functionality of the web application.

### 2.3.2 Static and Dynamic Web Sites

Web sites can be divided into *static* and *dynamic sites* based on their functionality. A web site is static if the content of the page is fixed with no changes to the HTML. The web browser requests a specific web site from the server, and the server provides that specific web site's HTML content possibly with CSS and JavaScript. The content of the document object model (DOM) that the browser creates from the HTML does not change at any point of use.

Dynamic web sites, on the other hand, as the name implies, provide the user with content that is dynamic; the content might change depending on, for example, the user, the user's interaction with the website, or the time of day. Dynamic web sites can use either server-side scripting or client-side scripting for generating dynamic content. Server-side scripting means that when a server receives a HTTP request from a browser, it uses different parameters to generate a web site HTML file – possibly with a template system – that is then served to the browser. Client-side scripting enables the manipulation of the DOM tree, for example when a user clicks on an UI element. [Shklar and Rosen 2009, Ch 6]

While static web sites provide only plain information in graphical or textual means, dynamic websites offer interactivity and dynamic content based on different parameters such as the user who is currently using the web site, newest content, the location of the user, or the time of day.

### 2.3.3 Web as an Application Platform

In the turn of the millennium, web development reached a turning point. More web developers started treating the web as a platform for social, interactive, dynamic, and eloquent applications rather than a collection of static web sites. This turning point was coined *web 2.0*. [O'reilly 2009]

As web 2.0 started becoming the predominant use case of the internet, a term called *rich internet applications (RIA)* was coined. The term refers to dynamic web sites that work like desktop applications. Compared to the simple world of web 1.0, RIAs offered more interactivity and richer, more satisfying user experience by scripts running in the browser client. The research on RIAs peaked in the 2009 and has been in a steady decrease since. In recent times, many of the features that have traditionally been described as RIA features have become the norm in web applications. Therefore, the use of the term RIA has largely decreased as well. [Casteleyn et al. 2014]

A more recent but similar term – *progressive web applications (PWAs)* – have received hype in the web development community. PWAs are web applications that behave similarly to native applications thanks to features like offline support, background synchronization, and home screen installation. One main benefit of PWAs is that instead of creating a separate native application for each different platform, the developers could

build one responsive web-based application that would work on any platform with tools used in web development – HTML, CSS, and JavaScript. With PWAs, the end user can use the application without installation with any device like a PC, mobile phone, or a smart watch like they would a traditional installable application [Biørn-Hansen et al. 2017]. While large software companies such as Google has encouraged the use of PWAs, they have raised concerns with issues related to security and privacy [Lee et al. 2018].

### 2.3.4 AJAX

AJAX (Asynchronous JavaScript and XML) is a client-side scripting technique that allows the client browser to send HTTP requests to the server and modify the web page content dynamically without reloading the whole page every time there is changes to the page. With AJAX, after the initial HTTP request, the client could send requests in a form of an *XmlHttpRequest* or *JSON* in the background, without any interruptions visible to the user [Paulson 2005; Shklar and Rosen 2009]. AJAX web application architecture is detailed in Figure 5.

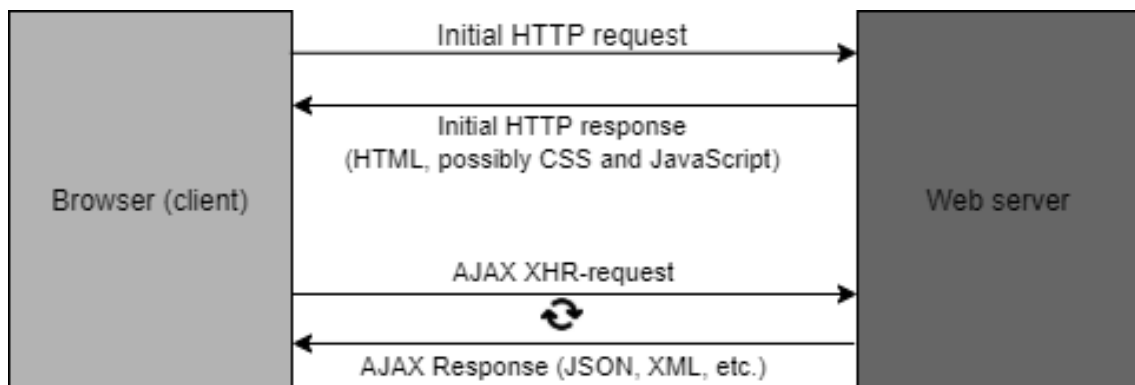


Figure 5. AJAX web application model

With AJAX, it became possible to show changing content to the user without having to refresh the whole page. Web sites could have UI elements that reacted to the input of the user considerably faster compared to older techniques of reloading the page every time there was any changes needed.

Although the term AJAX is not as often used anymore, the idea behind AJAX is a cornerstone of modern web applications, and most modern web applications use AJAX in some form. Perhaps the reason for the decrease of the term's usage could be that it is so universal that it does not need to be specifically mentioned when discussing web development.

### 2.3.5 Single Page Application

With the emergence of AJAX, web applications gained potential for rarely needing to do a complete refresh after the initial server request. A web application that only has to be

loaded once from the server and after that only uses techniques like AJAX for interactivity and content generation is called a single page application or SPA. SPAs are built from multiple parts that can be updated independently as to not require a full refresh of the application. [Mesbah and van Deursen 2007]

The main purpose of SPAs is making web applications rely less on bandwidth speed and server delay in order to make the applications faster and more interactive. The increase in computing power of web browsers and the general rise of web application popularity are essential reasons for the rise of SPAs. [Jadhav et al. 2015]

SPAs do not usually simply follow the MVC pattern. Since the idea of SPAs is to move as much functionality as possible from the server to the client, so is a lot of the model and controller functionality moved to the client as well. Often SPAs follow the Fractal MVC or FMVC model, which simply means that different parts of the application contain their own MVC models while the application itself also has its own MVC model. [Mikowski and Powell 2013, Ch 4]

In SPAs the application state dictates what the UI shows the user. When the user clicks on a tab for example, the state changes so the tab content becomes visible. There are many solutions to state management in SPAs and each development framework has different ways of managing application state. One of the most used state management models currently is Flux, which is developed by Facebook specifically for the React JavaScript framework [Tay 2019]. Flux is further presented in chapter 3.3.2.

### **2.3.6 JavaScript**

While other client-side programming languages have been introduced over the past decades, JavaScript has maintained its place as the most used frontend language. JavaScript is not the only language that runs on browsers, and some alternatives like WebAssembly [WebAssembly 2021] have gained traction in the web development community. However, as JavaScript still is without a doubt the most popular client-side language, it is highly relevant to study its current use. JavaScript is typically used according to the ECMA (European Computer Manufacturers Association) standard *ECMAScript* [ECMA-international 2021].

Whereas previously JavaScript was seen as a simple language to only write simple scripts no longer than few lines, nowadays most websites rely fully on JavaScript. Not only is JavaScript the dominant programming language in the UI side, recently it has also expanded to server side as well with the Node.js framework [Node.js 2021]. This has resulted in JavaScript becoming one of the most used programming languages in the world [Taivalsaari and Mikkonen 2017; Mikkonen et al. 2019].

When web applications were still in their infancy, reusable components were already recognized as one of the most useful and potent tools for web user interface (UI) development [Myers et al. 2000]. Most web applications have started using component-based



architecture, where different components are connected via interfaces. Web site UIs are often constructed from several reusable components and sub-components as building blocks to compose the whole UI [Daniel et al. 2012]. With JavaScript, this has recently been done with frameworks that provide a component APIs for building granular UIs.

### 2.3.7 JavaScript UI Frameworks

JavaScript is rarely used by itself. Often it is used with libraries like jQuery, which is the most popular JavaScript library [w3techs 2021] that provides easier DOM tree traversal and manipulation. While libraries like jQuery help bring interactivity to web applications, there has been a growing trend towards more holistic JavaScript frameworks which not only make the JavaScript development easier and faster, but also give structure and rigidity to the web applications.

The terms *library* and *framework* are used somewhat interchangeably in everyday conversation and academic literature, but generally the word library will be used for importable systems which provide different helpful functionalities – something a developer can plug into their code whereas a framework is a more structural system – something a developer can plug their code into. However, there is no consensus whether a system like ReactJS is a library or a framework. On one hand it does not force the developer to any rigid frame for development and is quite unopinionated on how the developer structures their code, but on the other hand, it offers declarative solutions and inversion of control for many common JavaScript functionalities. In this study, most libraries that offer more than simple functionality – like ReactJS – will be referred to as “frameworks” for consistency.

There have been some attempts at academically evaluating JavaScript frameworks by authors such as Gizas et al. [2012], who compared the performance of seven JavaScript frameworks in terms of size metrics, complexity metrics, and maintainability metrics. Graziotin and Abrahamsson [2013] pointed out the importance of practitioner needs when evaluating JavaScript frameworks. They argued that while metrics such as validation, quality, and performance are seen as important for researchers, actual practitioners typically value metrics such as documentation, community, and pragmatics which ought to be studied as well.

According to the State of JS 2020 survey [Greif and Benitte 2021], three JavaScript frontend frameworks were noticeably more used than the rest: React, Angular, and Vue. The survey was answered by 23,765 developers. Here will be given an overview of the three frameworks.

ReactJS, also known as React, is a frontend framework developed by Facebook [Facebook Open Source 2021a]. React is a somewhat unopinionated framework as it does not force the user to use any specific state management or architectural style. React uses its own HTML-like syntax for building the DOM tree called JSX (JavaScript XML). JSX

is similar to a templating language, but it is in fact only syntactic sugar for React that gets transformed to regular JavaScript by the React engine. React applications can be divided into components with each component having the possibility to have its own state. React uses its own virtual DOM to update the HTML of its components. React does not have an enforced way of global state management, so the developer must make a decision to use, for example, libraries like Redux or MobX or sharing local states with the context API. Thus, React is mainly concerned with the view part of MVC.

Angular [Google 2021] is a JavaScript framework led by its team at Google. When Angular 2 was released in 2016, it had changed so much that the developers decided to make it its own product instead of a new version of the original Angular. This resulted in the existence of two distinct Angular frameworks: the original (now known as AngularJS) and Angular 2 (now known as Angular). Angular applications consist of components also. For DOM manipulation, Angular uses HTML extended with additional syntax for templating, and automatically changes the DOM when the component state changes. Compared to React, Angular offers more rigid environment and less configuration. Whereas in React, the developer is free to select any libraries and packages for any product, angular is more opinionated.

Vue.js, or simply Vue [You 2021], is a lightweight JavaScript frontend framework that only focuses on the view part of MVC. As such, Vue is comparatively simple to just plug into an already existing web application. Vue components are built with HTML-based templates, and Vue is claimed to be faster than its alternatives thanks to its light weight and optimization.

Additionally, there are numerous other frontend UI frameworks and libraries that have various amounts of support and popularity. The large amount of web frameworks and their overlap with each other can make it difficult for developers to decide which one to use. This oversupply of frameworks has led some researchers like Mikkonen et al. [2019] to predict a possible decrease in their role as drivers for web application evolution as the frameworks often get replaced by other frameworks and abandoned.

Another tool that is often used with JavaScript is TypeScript [Microsoft 2021]. JavaScript famously has *dynamic typing*, which means that any time a variable is declared, the developer does not have to define the type of that variable because the language infers it. The type of that variable can then later be changed if it is reassigned as the name *dynamic typing* suggests. This can lead to many issues such as difficult to detect bugs and unexpected behavior. While dynamic typing was specifically chosen to make JavaScript development simpler for the developer, the disadvantages of dynamic typing have been felt to be too large compared to the advantages and the developers have begun to move

towards static typing with tools such as TypeScript. TypeScript is an open-source language that enforces static typing in JavaScript by adding type definitions to it. While in JavaScript, a variable is declared simply as:

```
let a = false;
```

In TypeScript with the type declaration is included in the variable declaration:

```
let a: boolean = false;
```

TypeScript validates the JavaScript code by checking for any type errors. TypeScript code is transformed into JavaScript code, which means it can be run anywhere where JavaScript code runs.

### 3 React UI Framework

React was published and open sourced at the JSConf US 2013 conference [Occhino and Walke 2013]. The early development of React was done by engineers at Facebook. The main objective of React was to offer a light UI JavaScript framework that was unopinionated and fast. In 2020, React was the most used JavaScript UI framework [Greif and Benitte 2021].

In this chapter, React is firstly introduced in a general way including the JSX syntax and relevant tools. Secondly, React function components and class components are covered, Then finally React state management is discussed.

#### 3.1 React Overview

React takes care of the view portion of the MVC model. As such, the Model and Control aspects of React applications can be built with any architectural model. The main function of React is to visually present the model of an application UI including dynamic data, and when that data changes, React updates the view accordingly. React UIs are composed of reusable components and their interfaces which they use to communicate [Hunt 2013].

Instead of templates, React uses its own syntax, *JSX*, to present and refresh the data contained in them. While JSX resembles HTML visually, it is actually only *syntactic sugar* for creating JavaScript elements. Therefore, the trivial code examples Example 1 and Example 2 are compiled as identical by React [Facebook Open Source 2021c]. When the data included in the components change, React calculates the difference of the old and new rendered content and updates only the parts of the UI that have changed data in them [Hunt 2013].

---

```
const name = 'Bob';
return (
  React.createElement(
    'div',
    {className: 'greetingContainer'},
    'hello ' + name
  );
);
```

---

Example 1. React element using React.createElement

---

```
const name = 'Bob';
return(
  <div className='greetingContainer'>
    hello {name}
  </div>
);
```

---

Example 2. React element using JSX

React applications are often built with the help of toolchains. React itself provides the *Create React App*, which is a simple toolchain for SPAs, but there are others like *Next.js* and *Gatsby* as well. Toolchains allow simple management and deployment of React applications. They usually include a package manager like *npm* or *Yarn*, a bundler like *webpack*, and a compiler like *Babel*.

The package manager allows the developers to include 3rd party packages to their application, the bundler bundles the modular core of the application into smaller packages for optimization, and the compiler transfers the JavaScript code to a form that works on older browsers [Facebook Open Source 2021d].

React allows *one-way data binding* between the view and the model. When a user interacts with the view, for example by clicking a checkbox, the data regarding the interaction is sent to the model from an event handler function. On the other hand, if something changes inside the state of the model, it can change the state of the view accordingly. *Two-way data binding* – a feature that is used in some other frameworks for binding a part of the application model to its view and updating the view every time the model is updated and vice versa – is not supported by React.

In order to conduct DOM updates, attribute manipulation, and event handling, React uses a pattern called *virtual DOM* (VDOM). VDOM is a virtual representation of the actual DOM that is stored in the application memory. With VDOM, the developer can write DOM manipulation in a declarative way and React handles the actual DOM updates visible in the UI via the VDOM.

## 3.2 Components

React applications comprise of reusable components that have import interfaces and export interfaces. Parent components can send parameters – *props* – to child components. Props can then be used by the child component similarly to how typical functions use parameters. There are two ways of writing components: class-based and function-based [Facebook Open Source 2021e].

Each component has a *lifecycle* consisting of mounting, updating, and unmounting [Facebook Open Source 2021e]. Mounting happens at the initial render. Most data fetching and other side effects typically happen right after mounting. Updating happens when some data state inside the component changes and the component has to be rerendered. Unmounting happens when a component is closed. Aborting ongoing network request calls and other cleanup is typically done at unmount time.

### 3.2.1 Class Components

Class components used to be the only way to include state and lifecycle methods in React components since function components did not have those features until somewhat recently. React class components are written as regular ES6 classes. They may contain

functions, variables, props, and state accessible with the *this*-keyword. Example 3 shows a typical React class component syntax.

---

```
class Greeting extends React.Component {
  render() {
    <div>hello {this.props.name}<div>
  }
}
```

---

Example 3. React class component

### 3.2.2 Function Components

Traditionally function components were used for simple components that did not require interactivity or other complicated functionality. With the introduction hooks, function components have become practically de facto standard way of writing components [Facebook Open Source 2021f]. Function components are seen as simpler with less boilerplate code and no need for *this*-keyword. In example 4, functional component composition is presented. Even in this trivial example, the simpler form of function component can be observed (cf. example 3).

---

```
function Greeting(props) {
  return <div>hello {props.name}<div>
}
```

---

Example 4. React function component

Since the introduction of *hooks* to React, function components have gained most of the functionality of class components like state and lifecycle management. As a result of this and the fact that class components are seen as more complicated than function components, function components have become the preferred component type [Facebook Open Source 2021f].

### 3.2.3 Function Components With Hooks

In function components, increasingly important features of single page web applications like state and lifecycle management used to not be possible. On the other hand, while these features were present in class components, they often had increased complexity and length due to required boilerplate code. Also, *this*-keyword and handler binding in class components was seen as complicated and undesirable. This made it difficult to write compact and decoupled components that were able to manage state and lifecycle.

React solved this issue by presenting a feature called *hooks* to React in October 2018. Hooks were a way to easily add state management, lifecycle management and other procedural features to function components, effectively bringing the beginning to the end for class components [Facebook Open Source 2021f].

Hooks are functions that add more features to React functional components by “hooking into” some functionality, which the function component then uses to render a view for the user. Hooks always start with *use*-keyword [Facebook Open Source 2021f]. With basic React comes ten built-in hooks (Table 1) and a possibility of writing own custom hooks.

Hook	Purpose
useState	Saving and updating a stateful value
useEffect	Running effectful code after every completed render or when certain values change
useContext	Accessing a context value of a context object
useReducer	useState alternative with reducer functionality
useCallback	Creating callbacks with memoization optimization
useMemo	Creating values with memoization optimization
useRef	Storing mutable values
useImperativeHandle	Customizing instance value exposed to parent components when using ref
useLayoutEffect	useEffect alternative that fires synchronously after all DOM mutations
useDebugValue	Displaying a label for custom hooks in React DevTools

Table 1 React built-in Hooks

If the developer needs some additional, more complex, functionality to their components, they have the possibility to write custom hooks. Custom hooks take advantage of built-in hooks to let the developers write, for example, side effects or stateful logic that can be shared between components. Several 3rd party libraries have also started offering hooks to use as interfaces to simply hook into the functionality they provide.

### 3.3 Component Lifecycle

In class components, the component lifecycle is handled by defining what happens at each point of the component’s lifecycle with different lifecycle methods (Figure 6). The lifecycle methods can be divided into three categories: mounting, updating, and unmounting. These methods run at particular times in the component lifecycle process. For example, in the *componentDidMount* method, all the functionalities that happens immediately after a component is mounted would be written. In addition to the lifecycle methods presented in Figure , three additional lifecycle methods – *componentWillMount*, *componentWillUpdate*, and *componentWillReceiveProps* – exist, but have been deemed unsafe and recommended to avoid while still kept in the framework as legacy methods.

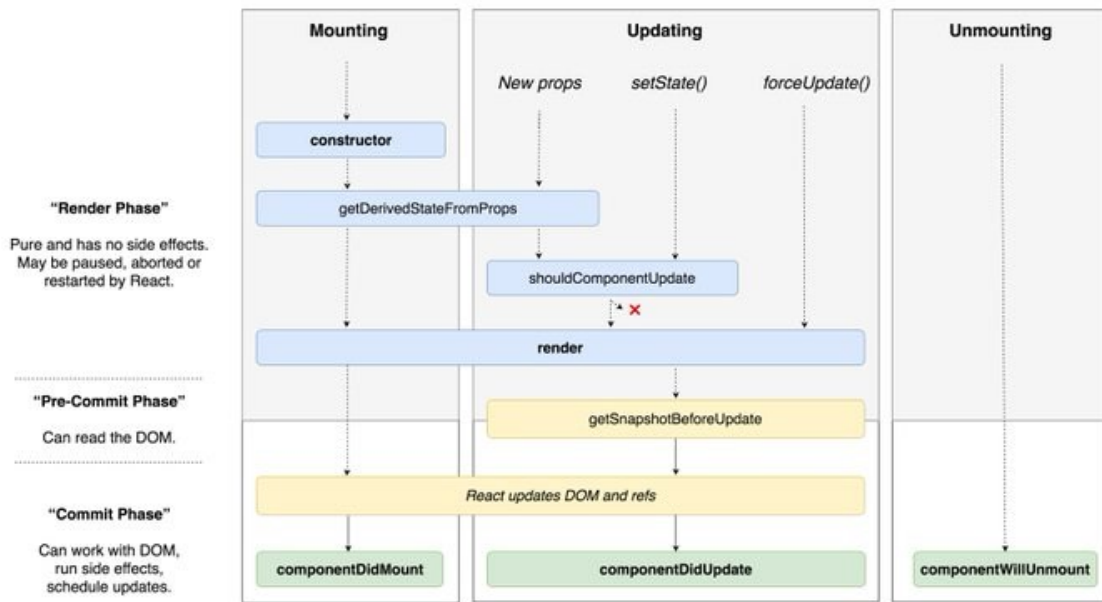


Figure 6 Class component lifecycle [Abramov 2018]

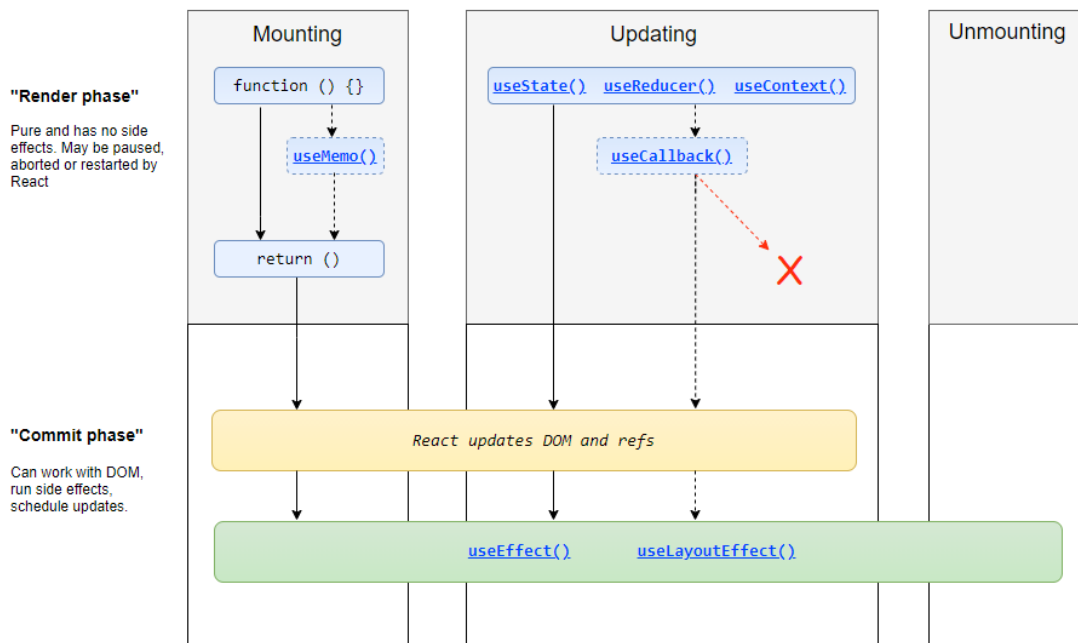


Figure 7 Function component lifecycle with hooks [Margalit 2021]

With hooks, on the other hand, the lifecycle of a component is managed with just two hooks that are similar to each other: `useEffect` and `useLayoutEffect` (Figure 7). Code given to them runs at the mounting of the component as well as every time its dependencies change, and then performs the given clean-up function when the component unmounts. Two identical components are presented, first written as a class component, second as function component in Example 5.



---

```
// ### Example class component ###
class ExampleClassComponent extends React.Component {
  componentDidMount() {
    console.log("component mounted");
  }
  componentWillUnmount() {
    console.log("Component unmounts.");
  }
  render() {
    return <h1>Hello World</h1>;
  }
};

// ### Example function component ###
const ExampleFunctionComponent = () => {
  useEffect(() => {
    console.log("component mounted");
    return () => {
      console.log("Component unmounts.");
    }
  }, []);
  return <h1>Hello World</h1>;
};
```

---

Example 5. Lifecycle comparison of class and function components

### 3.4 React State Management

The idea behind SPAs is supported by state management. SPAs would lose almost all their interactivity if there were no state management. Two main questions have risen when building stateful applications in React: How to hold state information locally inside a component, and how to share state between components.

While Happe et al. [2014] divided software state into three categories: component-specific, system-specific, and user-specific state, Frontend software state is usually divided into just local state and global state which map into component-specific and system-specific, respectively while user-specific state data is typically stored in either local or global state. This mapping gets a little hazy when taking into consideration the fact that local state is shareable between components either from parent component to child component via props or via context from any part of the component tree to any other part. Thus, the question arises whether or not this shared local state is still part of the component-specific state or if it then becomes system-specific state. Typically, in React, local state is used in state data that is scoped only inside one single component, whereas global state includes data that several components depend on. For example, the state of a text field component would most likely be only stored in the local state of the component whereas a theme selection affecting the colors and other appearance factors of the whole application would be saved in the global state.

### 3.4.1 Local State

In class components, each component has a single local state object accessible by `this.state` and mutable by `this.setState` function. The state must be assigned in the constructor method of a class component in order to be usable in that component. State is stored in a form of a JavaScript key-value object.

Before hooks were introduced, it was not possible to save local state in a function component. The introduction of the `useState` hook allowed function components to have local state. The `useState` hook returns a value and a setter function for that value, which the program can call to change the state of that specific state value (typically triggered by an event or a `useEffect`-hook) [Facebook Open Source 2021g]. This is more compact compared to class functions where the state has to be managed as one single object for the whole component.

### 3.4.2 Global State Management with Redux

While component's local state management is relatively straight-forward, SPAs' increasingly growing need for application's shared state presents a more complicated question. Afterall, components need to communicate with each other via some type of interface – not only parents with children and vice versa, but sibling to sibling and largely separated components as well. Originally the main method of shared state in React was parent components sending stateful data to children via props. This often resulted in a large number of props being sent down the DOM tree to child components. If the components were far away from each other in the DOM, they had to be sent through many layers of child components before finally reaching the component that used that data. This was known as *prop drilling*.

As the development philosophy of React is to strictly manage the view part of MVC in an unopinionated manner, it does not offer any native solutions to global state management. Perhaps due to this, there are a large number of external libraries and solutions for state management today, most prominently *Redux*, *MobX*, *Recoil*.

The global state management library Redux utilizes the *Flux*-pattern developed originally at Facebook [Abramov 2021]. In Flux, the application has one *store*, where the state is saved. This store is then accessed by *actions* via a *dispatcher* and finally the view is updated according to the changes in the store state. Furthermore, when a view is changed in certain ways, an action can be sent to the dispatcher to change the state again (Figure 8). Due to the rigid nature of Flux, developers have to write a relatively high amount of code: for every stateful action, developers must write an action creator, an action, and a reducer that calculates a new state whenever an action is dispatched to the store with all the included boilerplate code.

Redux does not natively support asynchronous operations as part of dispatching actions. Popular solution for this issue is the library *redux-thunk* [Hanh 2021]. With *redux-*

think, asynchronous operations such as data retrieval can be sent as a part of the action to the dispatcher, making state management with Redux more flexible for different uses.

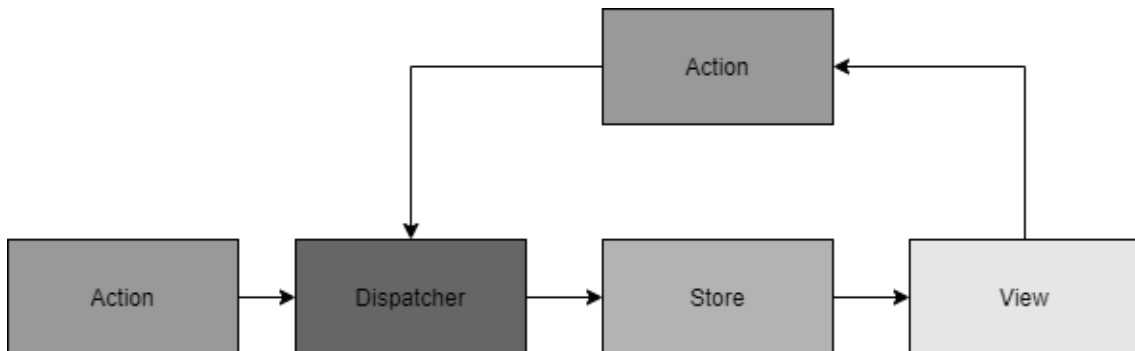


Figure 8 Flux architecture

Some third party data fetching libraries also have state management capabilities. For example, *Apollo*-library uses *GraphQL* to fetch data from a remote server and saves it into *cache* as a type of state [Apollo 2021]. Similar cache-based data fetching/state management libraries for React include React-Query [Linsley 2021a] and SWR [SWR 2021], which work with REST and Promise queries as well as GraphQL [Linsley 2021b].

### 3.4.3 Other Global State Management Libraries

*MobX* [MobX 2021] uses similar model to Redux but is claimed to be simpler and more compact while requiring less boilerplate code. On the other hand, it has a smaller community, and it supports impure functions, which can lead to unpredictable problems.

**Recoil** is a newer solution for global state management from Facebook [Facebook Open Source 2021h]. Instead of having many complicated parts like Redux does, Recoil comprises only of *atoms* and *selectors*. Atoms are pieces of state that can be read from and written to from any component. Any time an atom updates, any component that subscribe to it will be rerendered. Selectors are state functions that derive the state from another state modified by some pure function.

One solution for state sharing between components is using **context API** for sharing local states. Context is a built-in solution for React to share variables and other data between components that are not necessarily related to each other. Using context for global state management is not strictly speaking global state management since it does not store or manage any separate state data by itself; it is merely a transport mechanism. However, in many applications global state management is wholly replaced by local state management sharing with context.

One particularly curious state management solution that has gained popularity lately is **react-query** [Linsley 2021c]. React-query separates *client state* from *server state*. Most global state management tools are able to handle client state quite well, but are not

equipped to manage server state, or the changing data that is saved in the server. The data has to be retrieved with asynchronous operations and updating the changing data is challenging. With solutions like react-query, the developer can simply tell the library what data needs to be fetched and the location of that data via the library API, and the library handles keeping that data up to date with the server state.

**XState** [2021] is another state management solution that has received attention during recent times. XState is a JavaScript library that utilizes finite state machines and statecharts to handle state in JavaScript applications. XState itself works on several platforms including React. It has been claimed to be helpful when describing the behavior of an application and implementing it according to that model.

One might think the above list of global state management solutions is already quite extensive but in reality, it is only scratching the surface of the numerous state management options available for developers. Despite the number of solutions available, the recent trend of global state management in React seems to steer towards simpler solutions that enable developers to write code faster without bothering with boilerplate code and unneeded logic that can be abstracted away. Since the introduction of hooks to React, almost every active state management library has started providing hook-based APIs to accommodate to new function component focused React development.

## 4 Research Methods

Myers and Stylos [2016] argue that API usability is a crucial element that might dictate whether an API will be successful or not. Myers and Stylos suggest that APIs with poor usability are difficult to change later down their lifecycle since it might affect existing code bases that use those APIs, which is why APIs ought to be published only after their usability has been tested and evaluated.

The aim of this study is to evaluate the usability of React hooks as an API from different perspectives. Myers and Stylos [2016] maintain that a good way of evaluating the usability of APIs is to use the popular Nielsen's *heuristic evaluation* method, which is typically used in user interface evaluation. In this study, Nielsen's [1994] ten heuristics (Table 2) will be used for supporting the usability evaluation of React hooks.

#	Usability Heuristic	Example in API context
1	Visibility of system status	Ease of checking state; feedback for mismatches between state and operations
2	Match between system and the real world	Logical method names and organization into classes
3	User control and freedom	Aborting and resetting operations
4	Consistency and standards	Consistency throughout the API
5	Error prevention	Good default operations; coherency
6	Recognition rather than recall	Autocomplete-friendliness
7	Flexibility and efficiency of use	Efficient use of API
8	Aesthetic and minimalist design	Small number of classes, methods
9	Help users recognize, diagnose, and recover from errors	Explanatory error messages
10	Help and documentation	Adequate documentation

Table 2 Nielsen's Usability Heuristics in API context [Myers and Stylos 2016]

The study was conducted in two parts: case study and developer interviews. This was done due to the need to inspect hooks-based state management from two distinct angles: Firstly, the code-specific angle in which the hooks-based model will be examined by simply different aspects of the structuring and logic of the code that uses it according to heuristics, and secondly, actual experiences and opinions of React developers who have had professional experience with hooks-based state management.

This chapter will present the case study and the developer interviews, and their analysis in more detail. For the developer interviews, firstly the purpose of the interviews will be presented, then the selection of the interview participants will be discussed, and finally the interview data analysis will be presented.

## 4.1 Case Study

In the case study part, local state management was inspected by comparing class components to function components with hooks. Two versions of identically working components were created and inspected: one class component and one function component. Then, they were analyzed from different aspects such as readability, code line count, and other factors based on the API usability evaluation methods presented by Myers and Stylos [2016].

The code written for the case study included software components that were part of an enterprise and industrial automation research prototype application developed at Nokia Bell Labs. The application was developed in collaboration with different development teams across the corporation. The main purpose of the application is supervision and surveillance of various enterprise facilities. The parts included in the case study were functionally only generic ones such as basic controls and panels that have been approved for publication for this thesis paper. The case study components were written by the thesis writer during his traineeship at Nokia Bell Labs from September 2020 to March 2021.

The case study included three components and their subcomponents. The three components were (1) `DateTimeDisplay` (Appendix 1 & 2), a component that displays the current time and date to the user, (2) `LoginView` (Appendix 3 & 4), a component, which includes input fields for username and password and some additional functionality, and (3) `UserPreferencesWindow` (Appendix 5 & 6), a panel, in which the user can change basic settings related to their preferences such as the color theme, video quality, temperature units (Celsius or Fahrenheit), and the display language. These components were chosen for this case study because they represent different aspects of possible React components and have different functionalities to present different aspects of class-based and function-based component architecture and their qualities. In addition to React version 16 and JavaScript, the components used for the analysis use TypeScript for strict static typing and Redux for global state management.

The usability of hooks was evaluated using the application of Nielsen's ten heuristics [Nielsen 1994] for API usability evaluation by Myers and Stylos [2016]. The usability of hooks in the components was inspected from the viewpoint of the ten heuristics and found usability issues were assigned to the relevant heuristics.

## 4.2 Developer Interviews

In the interview part, six React developers were interviewed for their experiences with state management in React. The interviews were exploratory and semi-structured. As a semi-structured interview, the data was qualitative in nature. The aim of the interviews was gaining understanding of the developers' opinions, habits, and programming methods related to state management and functional components. Some studies have been

done on interview-based API usability evaluation. One such study that was also used as an inspiration for this interview was done by Piccioni et al. [2013]. In their research, they interviewed 25 participants on four API usability aspects: *understandability*, *abstraction*, *reusability*, and *learnability*. They found that issues such as naming API features, discovering relations between types of the API, and availability and quality of documentation had an impact on API usability.

Semi-structured interview method was chosen because the interviewer was also experienced in React development and as a result could handle the conversation and ask follow-up questions and clarifications. The purpose of the interviews was to gather information on firstly, the habits and experiences of react developers including organizational and practical issues related to state management, and secondly, opinions and more personal thoughts of the developers on React state management and web application development.

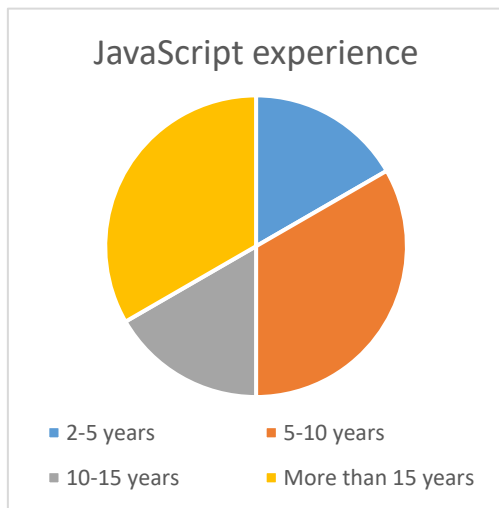


Figure 9 JavaScript experience of the interview participants

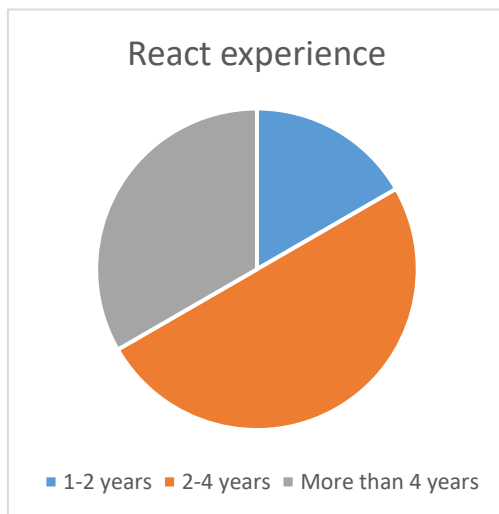


Figure 10 React experience of the interview participants

opment.

As there has been little this type of usability research focusing on JavaScript APIs, the grounded theory methodology was chosen for this research. As defined by Martin and Turner [1986], grounded theory *allows the researcher to develop a theoretical account of the general features of a topic while simultaneously grounding the account in empirical observations or data*. Using grounded theory, the aim of the interviews was to gather data in a holistic manner to create understanding and structure of the underlying elements of the research topic.

#### 4.2.1 Participants

Before looking for the interviewees, the screening criteria was prepared. It was decided that to participate in this interview, the person ought to have professional experience with React since the interviews aimed to gain understanding of the practical working methodology of React developers as well.

The search for interviewees happened in a few different ways. The first method to find interview participants was to ask around the project work team at Nokia Bell Labs, which is how the first two participants were found. After

this, an interview invitation was sent around to different parties such as university lecturers of relevant classes and personal acquaintances. Among a handful of responses, two additional suitable interviewees were found this way. Finally, a message was sent on a Slack group of a local React conference, where two more interviewees were found. The interviewees residing in Finland were given a reward of a 15-euro gift card to a local food delivery service. At the end of each interview, the interviewees were asked if they happened to know any potential interview participants, but finally any additional interviewees were not found with this “snowball sampling” method [Goodman 1961].

The interviewees comprised of React developers aged 26-46 living in Finland, the United States of America, and Austria. They had JavaScript experience ranging from 2 years to more than 15 years (Figure 9) and React experience from 1.5 years to over 6 years (Figure 10). Some participants also held experience in other frontend JavaScript frameworks such as Vue, Angular, Svelte, Preact, and Ember. The interviewees ranged from students who had moderate React experience to professional developers who had been working with React since its release and some who had participated in organizing React conferences and had been deeply involved in the React developer scene.

The interview participants held overall positive attitudes towards React and React hooks according to the pre-interview form. When asked whether they would recommend React to a colleague, three answered “agree” and three answered “strongly agree”. For the claim *React is moving in the right direction*, the categories “can’t say”, “agree”, and “strongly agree” each had two answers. The participants were generally satisfied with React architecture and the way hooks fixed class components in React. For the learnability of hooks, the participants were more divided, with four answering “agree”, one answering “can’t say”, and one answering “disagree”.

#### **4.2.2 Data Collection**

Before the interview, the interview participants were asked to fill a short pre-interview form (Appendix 7) for purposes of auditing and recording the basic characteristics like age, location, and experience levels of the participants as well as recording their views on React development in a general and quantitative manner.

The interviews were conducted as video teleconferences due to the COVID-19 pandemic and geographical distances. The length of the interviews ranged approximately from 40 minutes to 65 minutes. The interviews followed a semi-structured format. An interview guide partly inspired by the API usability interviews by Piccioni et al. [2013] listing the most important themes was prepared before the interviews (Appendix 8), but a large part of the discussion occurred by following up on concepts and ideas that emerged during the interviews. The interviews took place from January 2021 to March 2021.

The interviews begun with a short unrecorded chat between the interviewer and the participant, after which the recording was started. The interview generally started with



basic question about the participant's experiences and opinions of web development and React before moving into more detailed topics. The interviews varied somewhat based on the expertise of the participants. For example, if a participant was especially knowledgeable in the global state management aspect of React development, more time was spent on that topic. It could clearly be seen that most developers preferred to talk about the topics by reflecting on their past projects and based on their concrete experiences of those projects rather than abstract ideas and concepts. By reflecting on past projects, the developers were able to convey their thoughts and opinions and base them on real experiences. At the end of the interview, the participants were asked if they had any additional comment or questions, which sometimes resulted in very important information that was not covered by the actual interview.

#### **4.2.3 Data Analysis**

The interviews were recorded and transcribed to a text form for detailed analysis. The transcription happened by the interviewer by listening to the recordings and transcribing them into a text document. Since the focus of the interviews was not on the manner of speech of the interview participants, filler words and pauses were not transcribed, and focus was placed on accurately transcribing the actual ideas expressed during the interview without making any changes to the speech content.

The analysis was done by extracting ideas that emerged during the interview. After transcription, the interviews were transformed into an Excel spreadsheet and coded according to their themes as detailed by Meyer and Avery [2009]. The categories for the coding were generated by browsing through the transcripts and writing down categories that came up frequently during the interviews (Table 3). These categories partly lined up with the themes that were prepared beforehand, but also were formed by the concepts that emerged from the interviews organically. Each concept that came up in the interview was categorized into one, two, or three categories.

Several specific usability issues were mentioned by the interview participants. Usability issues mentioned by the interviewed developers were mapped to the API usability application of Nielsen's ten usability heuristics [Nielsen 1994] of Myers and Stylos [2016] in order to see what heuristics the mentioned usability issues represented and what aspects of usability were weakened or benefitted the most by hooks

<b>Category</b>	<b>n</b>	<b>Explanation</b>
<b>Personal (pe)</b>	41	Personal experiences with web development
<b>react overall (ro)</b>	58	High-level React experiences and opinions
<b>other frameworks (of)</b>	32	Experiences with other JavaScript frameworks
<b>react architecture (ra)</b>	31	Component architecture of React
<b>react performance (rp)</b>	14	Computational performance and optimization of react
<b>hooks overall (ho)</b>	41	High-level experiences and opinions of hooks
<b>Lifecycle (lc)</b>	61	Function and class component lifecycle management
<b>function components (fc)</b>	60	Experiences on function component -based React development
<b>custom hooks (ch)</b>	21	Experiences with custom hooks
<b>Third party hooks (3p)</b>	19	Experiences with third party hooks
<b>readability and memory (rm)</b>	25	Readability and memory load requirement of hooks-based React
<b>Collaboration (co)</b>	8	Collaboration with hooks
<b>learning &amp; documentation (ld)</b>	38	Learning and documentation of React and hooks
<b>Ecosystem (es)</b>	24	React ecosystem, effect on use of hooks
<b>frontend testing (ft)</b>	34	Effect of hooks-based development on testing
<b>hooks adaptation (ha)</b>	25	Adaptation to hooks-based development
<b>refactoring components (rc)</b>	32	Experiences with refactoring React applications, effect of hooks on refactoring
<b>Global state (gs)</b>	44	React global state management solutions

Table 3 Interview categories and number of appearances

## 5 Case Study

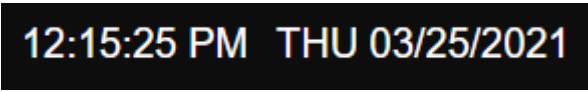
In this chapter, the results of the case study will be presented and analyzed. The case study includes three UI components: the *DateTimeDisplay*-component, the *LoginView*-component, and the *UserPreferencesWindow*-component. All of them were written with the style of class components and the more modern style of function components using hooks. The components and their source lines of code (SLOC) are presented in Table 3, where the additional length and complexity of class components can already be seen.

Component	Class SLOC	Function SLOC
<b>DateTimeDisplay</b>	56	45
<b>LoginView</b>	198	176
<b>UserPreferencesWindow</b>	318	287

Table 4 Relevant components and their source lines of code

### 5.1 Component 1: DateTimeDisplay

The function of the *DateTimeDisplay*-component (Figure 11) is to show the current time and date to the user. The time is displayed in current time and it rerenders once a second to show the current time by the second to the user.



12:15:25 PM THU 03/25/2021

Figure 11 *DateTimeDisplay*-component

Right after the component is first defined, the local state of the component is initiated including the current time and date, and the width of the time string for presentation purposes. In the class-based version of the component, the state is one object located within the component and accessible with the *this*-keyword. It is initialized in the *constructor* method, which is run before the initial render and additionally binds the incoming props to the *this*-keyword with the function *super(props)*. Most of the typing information for the code snippets is declared as *any* for ease of reading.

---

```
export class DateTimeDisplay extends React.Component<any, any>{
  constructor(props: any) {
    super(props);
    this.state = {
      dateTime: this.formatDateTime(new Date()),
      widthOfString: ''
    };
  };
  // [...]
}
```

---

Snippet 1 State initialization in the class-based *DateTimeDisplay* component

In the function version of the `DateTimeDisplay` component, it is possible to initialize the two state values as their own variables with the built-in `useState` hook. The hook takes the initial state value as a parameter and returns a pair of values: the state value itself, and a function that updates it. For example, in the first `useState` call in Snippet 2, the hook is called with the parameter `formatDateTime(new Date())`, which is a function returning the date in a desired string format. This is placed as the initial state. The hook then returns values `dateTime`, which is the current state, and `setDateTime`, which is the function that can be called to change the state.

---

```
export function DateTimeDisplay(props: any): JSX.Element {
  const [dateTime, setDateTime] = useState(formatDateTime(new Date()));
  const [widthOfString, setWidthOfString] = useState('');
  // [...]
}
```

---

#### Snippet 2 State initialization in the function based `DateTimeDisplay` component

After the state initialization, and the initial render of the component has happened, an interval function will be set. The interval function runs once per second and it updates the time and date states as needed. Furthermore, the interval must be cleared if the component unmounts for any reason so that it will not continue running in the background. In the component function (Snippet 3), this will be achieved with the class component lifecycle methods `componentDidMount` and `componentWillUnmount`.

In the function component (Snippet 4), the same is achieved with just one hook: `useEffect`. `useEffect` takes as parameter two values, firstly a function and secondly an array of dependable values. The function will then be run any time there is changes to any of the dependent values as well as after the initial rendering of the component. In this case, the dependent array is left as an empty array `[]`, which indicates that the function shall be run only after the initial render. The returned function will be run on component unmount, similarly to the `componentWillUnmount`-method in the class component.

---

```
private intervalId: NodeJS.Timeout | undefined;

componentDidMount() {
  this.intervalId = setInterval(() => {
    this.setState({
      dateAndTimeStrings: this.formatDateAndTimeStrings(new Date()),
    });
  }, 1000);
}

componentWillUnmount() {
  if (this.intervalId) {
    clearInterval(this.intervalId);
  }
}
```

---

#### Snippet 3 Setting up an interval in the class-based `DateTimeDisplay` component

---

```
useEffect(() => {
  const intervalId: NodeJS.Timeout = setInterval(() => {
    setDateAndTime(formatDateAndTimeStrings(new Date()));
  }, 1000);

  return () => clearInterval(intervalId);
}, []);
```

---

#### Snippet 4 Setting up an interval in the class-based DateTimeDisplay component

Finally, the date and time is presented in the return-clause of the component in JSX syntax. The JSX itself is identical in function and class components except for the use of `this`-keyword in the state object in the case of the class component.

---

```
return (
  <div
    <div style={ { minWidth: this.state.widthOfTimeString } } >
      { this.state.dateTime.time }
    </div>
    <div>{ this.state.dateTime.date }</div>
  </div>
);
```

---

#### Snippet 5 `this.state` declaration needed in returned value of class components

Two issues have been identified in the `DateTimeDisplay` component. Firstly, the management of the single local state value that is bound to `this` keyword in class components requires more code than the `useState` hook approach. Secondly, the lifecycle management of the component with the `useEffect` hook requires less code and the mounting and unmounting functions are all done in the same block, whereas in the class component the mounting and unmounting functionality is tied to their own lifecycle methods and the `intervalId` variable must be declared as a class variable instead of a local constant as in the function component.

## 5.2 Component 2: LoginView

The `LoginView`-component is a very typical UI component that the user of the application uses to input their credentials and access the application if the credentials are correct (Figure 12).

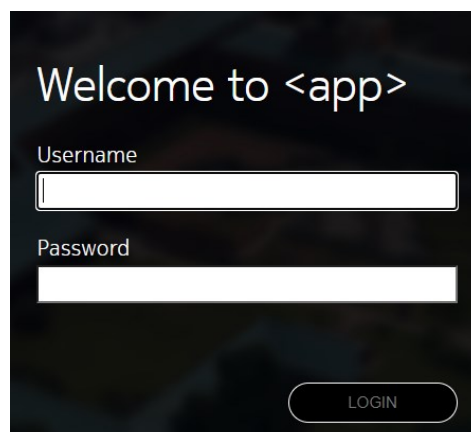


Figure 12 LoginView component

The component has four state values: credentials, error, loading, and disabled (Table 5), which are defined in the same way as in the DateTimeDisplay component (Snippet 1, Snippet 2). One main difference of the LoginView is that it has two event handler functions, *handleInputChange* and *handleSubmit*. In the function component, they can easily be implemented as regular functions, whereas in the class version, they are implemented as class methods.

---

State variable	Explanation
<b>credentials</b>	Object containing the inputted username and password
<b>error</b>	String error message shown to the user if not empty
<b>loading</b>	Boolean value whether the system is loading, shows animation if true
<b>disabled</b>	Boolean value whether button disabled (when field empty)

---

Table 5 State values in LoginView

There are a few important things to be pointed out. Firstly, in the Snippet 6, the added boilerplate code due to the binding of the methods to the class via the *this*-keyword is visible. This is not necessary in function components. Secondly, the binding of Redux state and actions to the component props via the *connect*-function (Snippet 7). In function components, the dispatch function can be directly called without needing to map it to props.

---

```
class LoginView extends React.Component<any, any>{
  constructor(props: any) {
    super(props);
    this.state = {
      [...]
    };
    this.handleInputChange = this.handleInputChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  async handleSubmit(e) {
    [...]
  }
  // [...]
  this.props.preferenceUserSet(response.userName)
  // [...]
  const i18n: I18n = this.props.i18n;
  // [...]
}

const mapStateToProps = (state: RootState) => ({
  i18n: state.preferenceSettings.i18n
});
const mapDispatchToProps = (dispatch: AppDispatch) => ({
  preferenceUserSet: (u: any) => dispatch(preferenceUserSet(u)),
  appMessageDisplay: (m: any) => dispatch(appMessageDisplay(m)),
  preferenceSet: (p: any) => dispatch(preferenceSet(p))
});

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

---

Snippet 6 Class component: Binding and using methods, state, and dispatch actions

```
export function LoginView(): JSX.Element {  
  const dispatch = useDispatch();  
  const i18n: I18n = useSelector((state) => state.preferenceSettings.i18n);  
  [...]  
  dispatch(preferenceUserSet(response.userName));  
  [...]  
  const handleSubmit = async (e) => {  
    [...]  
  }  
}
```

### Snippet 7 Function component: No need to bind anything to the component

Lastly, like in the `DateTimeDisplay` component, local state in class components is managed by the `setState` function that is bound to the `this`-keyword. In function components, on the other hand, the `useState` values can individually be edited with their specific function returned by the hook with less boilerplate code.

In the `LoginView` component, the added boilerplate code of class components due to having to bind class methods and Redux functions to the class explicitly can be clearly observed. In the function version, the Redux functions and component functions can be used directly, without binding it to `this` reducing the amount of code needed for the same component.

### 5.3 Component 3: UserPreferencesWindow

The `UserPreferencesWindow` (Figure 13) is a component that lets the user select different use settings according to their preferences. The component includes two tabs: *General* and *Language and Region*. The user can change settings like the color theme of the application, the video quality of the embedded video streams and others. The interactions are instantly dispatched to the Redux-store and visible to the user. For example, if the user changes the color theme of the application, the new theme will be instantly dispatched to the Redux store and the server-side user profile.

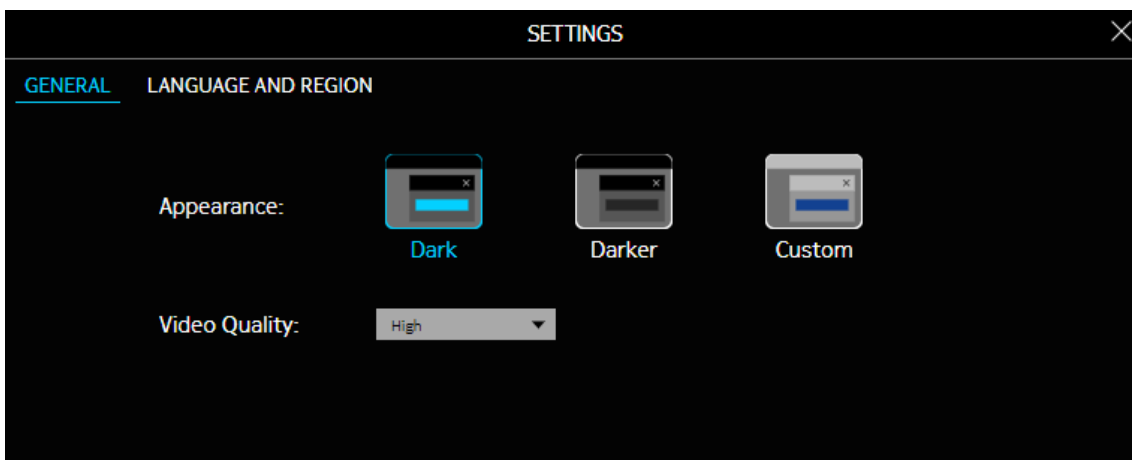


Figure 13 UserPreferencesWindow

The same issues that were visible in class-based `DateTimeDisplay` and `LoginView` components are also present in the class version of the `UserPreferencesWindow`. The added boilerplate code due to method binding and Redux dispatch mapping to props, the confusing use of *this*-keyword (49 appearances in the source code) was also present in the `UserPreferencesWindow` component.

Perhaps the most interesting bit of the functional version of the `UserPreferencesWindow` was the inclusion of one custom hook `usePreference` (Snippet 8). This custom hook takes no parameters and returns a state value including the currently chosen user preference settings and a function that changes that value similarly to the `useState` hook. Additionally, the custom hook sends the changed state to the redux store and the backend server any time changes are made to the preferences state. The hook uses built-in hooks `useState` and `useEffect` to achieve this functionality. This hook than then be reused by different components that need to alter or retrieve the user preferences. In the class component, on the other hand, these operations had to be done as part of event handlers. As such, they can be difficult to make reusable, and are scattered in different parts of the source code file.

---

```
function usePreference() {
  const prefsFromStore = useSelector(state => state.prefSettings.userPrefs);
  const [prefs, setPrefs] = useState(prefsFromStore);
  const dispatch = useDispatch();

  useEffect(() => {
    if (prefs && prefs !== prefsFromStore) {
      dispatch(preferenceSet(prefs));

      // Dispatching changes to backend server
    }
  }, [prefs]);

  return [prefs, setPrefs];
}
```

---

Snippet 8 Custom hook for dispatching changed preference data to Redux and backend

## 5.4 Case Study Findings

By comparing the class component and function component implementations of common UI elements, some clear differences have emerged. Firstly, function components seem to be able to offer more concise, more expressive, and more granular component implementations than class components. With less boilerplate code, function components can be divided into smaller reusable parts.

Another seemingly insignificant but rather noteworthy difference is the lack of *this*-keyword in function components. In class components, props, state, class variables, and methods are all accessed via *this*-keyword. This caused confusion as well as decreased code readability.



Based on the case study, local state management with the `useState` hook is effective and simple. Compared to class components' state management bound to the class with *this*-keyword, `useState` can be used flexibly and efficiently.

The class components' lifecycle methods do not simply map to the `useEffect` hook. While the lifecycle methods offer the developer a clear view of what happens in different times in the component's lifecycle, the `useEffect` hook is more concerned with its dependencies. For example, it does not seem to be possible to map a lifecycle method such as `componentDidUpdate` to `useEffect`. Rather, the development view must be transferred from temporal logic to variable- or dependency-specific outlook.

Custom hooks were found to be a clear way of creating reusable state logic that fits the need of simple UI elements. The structure and interfaces of custom hooks of the components included in the case study were sufficient and adequate for the cause.

In conclusion, function components offer more compactness and clarity compared to class components, but they cannot be simply mapped to each other, especially as a result of their different approaches to component lifecycle management.

## 6 Developer Interviews

In this chapter, the findings of the developer interviews will be presented. The aim of the interviews was to gain state-related information on the opinions and experiences of professional React developers. This chapter is divided into four parts that form cohesive themes around the ideas and topics that were discussed during the interviews: (1) React and other frameworks, (2) functionality of hooks, (3) hooks based React development, and (4) adaptation of hooks.

### 6.1 React and Other Frameworks

In this first part, results on React overall, React architecture, React performance, and global state management with React will be presented. When discussing different aspects of React, developers tended to compare it with other frameworks they had experience with.

#### 6.1.1 React Overall

Overall, the participants had a fairly positive opinion on React. As an especially favorable trait of React, simplicity of the base library was mentioned by almost every participant. The fact that React itself does not enforce any opinions or rigid structures to any project was seen as a clearly positive trait of the framework. The ease of setting up a React project with create-react-app and the numerous built-in performance and accessibility improvements that React offers were mentioned as a clear advantage of React.

React was mentioned to abstract away a large part of functionality for ease of use. While this was seen as very helpful for developers, it was pointed out that one still needs to understand the underlying mechanisms and principles of JavaScript to use those abstractions efficiently.

The large size of the ecosystem of React was mentioned multiple times. According to the participants, while the large ecosystem has considerable advantages on development it has sometimes been difficult to navigate the quickly-changing and large ecosystem of React. The number of libraries available for, for example, state management and styling were seen as overwhelming at times. Most of developers (5/6) at some point had been working on a project that had started with some technology that was new and suitable at the time only to switch to a newer technology in the middle of the project only some time later.

*[P5] The developer might have started with Redux and Styled Components and then changed to Emotion but didn't drop the old Styled Components, and later might decide to change to Recoil instead of Redux.*

Compared to other frontend UI frameworks, React was seen to get updated slower. Especially recently when new frameworks have been appearing constantly and updated

quickly, React has been moving more slowly according to the developers. An example mentioned was concurrent mode, which was already announced three years ago, but still has not been released into React, while other frameworks seem to be moving at a faster pace. Another worry was that React was focusing on presenting simple solutions to both small-scale and large-scale projects, which have vastly different needs.

*[P6] They're trying to solve a spectrum of problems as opposed to only focusing on the main problems of the majority of the community.*

In addition to React, other frameworks mentioned during the interviews included jQuery, BackboneJS, Angular, AngularJS, Vue, Svelte, Preact, and Hyperapp. When asked about the developers' general thoughts on React, they often compared React to these other frameworks. The advantages of React, compared to the others, were mostly seen to be the flexibility and quickness of the React API, the large ecosystem, and similarity to native JavaScript. Most (4/6) of the interviewed developers were working mainly with React, while some (2/6) were working with up to two UI frameworks in parallel. Most (5/6) developers expressed that they were passively monitoring other UI frameworks as well even if not actively using them.

### **6.1.2 React Architecture**

One particularly strong advantage of React was the architecture. According to the participants, splitting the source code into React components was a natural way of creating user interfaces. Sharing data between the component tree via props was seen as a sufficient way of handling the component interfaces. On the other hand, some participants (2/6) expressed disapproval of the context API; sharing data between components via context was seen as a frequent cause of problems and not always a sufficient solution for *prop drilling* – sending large number of props down several layers of the component tree.

One participant expressed worries that there were not enough developers with skills in designing React architecture. Since React does not provide any architectural guidance of how components ought to be arranged in the component tree or the system directory structure, the developers have to do all the architectural decisions themselves. This was seen as a problem since often developers did not have the necessary skills in architectural design when building React applications which had led to code base that was confusing and difficult to manage.

### **6.1.3 React Performance**

Performance of React was a frequent topic especially among the more experienced React developers. Overall, the performance of React was seen as largely positive with some drawbacks. The VDOM concept was a large advantage for performance of React. The developers were pleased with the automatic optimization done by the React engine. The

automated optimization of React helped the developers to be “lazy” when developing React applications, since they did not have to consider the optimization aspect themselves and could focus on the view and business logic parts of the application.

*[P2] [VDOM optimization] makes it really easy and nice for us to relax a little [...] and rely on fact that in the end it gets optimized and figured out what gets changed in the end.*

While the developers were satisfied with the performance of React overall, they expressed discontent with the performance of the context API and slow release of concurrent computing solutions into the framework. Also, the recent hype of SPAs and their performance compared to more traditional styles of web sites was criticized. One developer said that while the performance of SPAs is better, the development time of them is also much larger and often not worth the increased performance.

#### **6.1.4 Global State Management with React**

The more experienced interview participants held strong opinions on global state management solutions. It is important to recognize these views as global state management along with local state management compose all state management in React. Although the interviewer did not bring up any global state management libraries themselves, the participants were keen to discuss the differences of all state management libraries they were familiar with.

The one global state management library mentioned by every interview participant was Redux. Redux was seen as the default global state management library that other libraries were compared with. Redux itself did get criticism mainly for the amount of boilerplate code needed to use it, which is why most (4/6) developers had felt dissatisfaction towards using it recently and had started looking at other options for global state management. Redux was sometimes used with tools such as RxJS and redux-observable, which allow additional functionality like composing asynchronous operations with Redux.

One developer explained that in a project they had been a part of, Redux was dropped altogether and replaced by thoughtful data retrieval any time server data was needed while keeping simple state data like the user information in the context where it could be quickly accessed. That is, instead of retrieving all the needed data from the server after the first render, they would retrieve smaller pieces of data whenever they needed it. Abandoning Redux had a positive impact on the clarity and maintainability of the project according to the developer.

Some developers (4/6) also had experience with using Apollo’s built-in caching tools to handle the state of an application. The idea is that Apollo is used to retrieve data from the server and the retrieved data is saved in the cache. Every time when that data is then

needed, Apollo can either retrieve the cached version of that data or retrieve the newest data from the server. This way the state data does not need to be saved in any external state store.

MobX and Recoil were other global state management libraries that were mentioned by several (5/6) developers. Slight shift from the dominance of Redux towards the simpler solution Recoil had been observed by some of the developers. MobX, on the other hand, was seen as a tool “for specific cases” and slightly falling behind in popularity mostly mentioned as an alternative to Redux that many developers knew but had not tried.

One experienced developer described that they had built their own state management tool based on JavaScript’s mutation observer, which can be used inside HTML markup. The reason was that their project management had prohibited the use of React in their project, so they decided to build their own state and component management tools based on React logic.

One additional solution for transferring state related information between the application components was doing it via the URL of the web application. With libraries such as react router, the application edits the URL according to the state of the application and then in different components the state data would be parsed from the URL directly. This also allowed copying the URL from a specific application state and sharing it to anyone with the state information persisting.

Context API with local state management hooks was seen as a state management style that had to be used with caution since it was slow. Since the whole context tree had to be refreshed every time context had any changes, it was only used for data that changed infrequently. One solution was mentioned for this problem: use-context-selector [Kato 2021]. With use-context-selector, the developers can access and edit just certain parts of the context, so that every component using context does not have to be rerendered.

Another up and coming state management solution that was mentioned by a few (2/6) developers was XState. XState was used for applications with complex stateful logic by incorporating infinite state machine and state chart -based thinking in the state management.

*[P4] I see as the biggest problem for me, in React, that there are so many patterns and libraries for state management and none of them are React’s official ones.*

The large number of solutions for React state management was mentioned by several developers. Overall, the large ecosystem had led to difficulties choosing proper state management tools for projects, but it was also seen as a sign of constant improvement and forward movement. From the interviews, it was quite clear that most of the lesser experienced and full stack -focused developers struggled with the amount of state management

libraries, while the more experienced developers focused mainly on front end and React saw it as a positive that the React ecosystem was constantly coming up with new solutions for all the different use cases while expressing compassion with newcomers who had to learn all the different solutions from the beginning.

Two highly experienced participants, P5 and P6, said that state management tools can be categorized, and then the selection becomes much easier. P6 divided the state management tools into tools that keep the state close to the components, tools that keep the state away from the components and state machines. P5, on the other hand, described a two-dimension categorization dividing state management tools into firstly direct and indirect, and secondly single and multi-store tools.

### **6.1.5 Function and Class Components**

All the participating developers had experience with writing both, class components and function components. Developers held overwhelmingly positive opinions on function components compared to class component while having some criticism as well.

Before hooks, the developers had to write both class components and function components. Class components were written for components that included some complex logic or state logic while function components were written for components that had little logic and handled simple views. After hooks were introduced to React, developers have been only or mostly writing function components making the code of React components more unified.

*[P5] When I was writing my book [before the introduction of hooks], I constantly thought this would be much more pleasant if the whole thing could be done with functions.*

Compared to class components, function components were seen as more compact and “clean” with less boilerplate code allowing for dividing the applications into smaller components and reusing component logic. On the other hand, half (3/6) of the developers purposefully still had class components in their projects for easier monitoring of complex lifecycles and debugging since in class components, the component is rendered in a clear lifecycle. Most (5/6) of the developers mentioned having a few class components in their projects for this purpose while one developer said they had no class components and no apparent need for class components in their projects.

### **6.2 Functionality of Hooks**

One of the main purposes of this study was to find out the different qualities of React hooks, and how they were able to replace class components in the view of the developers. Overall, the developers held positive opinions on hooks-based state management with a

few caveats such as the lifecycle management of components being more difficult in certain cases.

### 6.2.1 Built-in hooks

The developers held mostly positive opinions on hooks, referring to the ability to easily write reusable logic with hooks, the small amount of boilerplate code, and the ease of using them. In addition to the most popular built-in hooks *useState* and *useEffect*, the developers had used *useContext*, *useReducer*, *useCallback*, *useMemo*, and *useRef* extensively, while *useImperativeHandle*, *useLayoutEffect*, and *useDebugValue* had been used considerably less.

Hooks did receive some criticism as well, mainly on their level of abstraction. On one hand, hooks were perceived to be too “magical” – meaning that their functionalities were too abstract and not instantly visible to the developers using them. In many cases, the developers had had difficulties figuring out how the hooks actually work “under the hood”. On the other hand, some (2/6) developers felt that the abstraction level was not high enough, since the developers had to decide themselves when to use tools such as *useEffect* as opposed to *useLayoutEffect* and felt that the decision could be handled by the hooks themselves.

### 6.2.2 Lifecycle Management

The difference between lifecycle handling in class components and function components with hooks was perhaps the main criticism of hooks. Whereas in class components, the lifecycle of components was handled by lifecycle methods, which ran at specific parts in the component’s lifecycle such as on mount, on props update, and on unmount. With hooks, this was replaced by effects, which run at component mount as well as when their dependencies change whereas the returned function is run on component unmount.

*[P1] It works in about 90 percent of cases, but the rest 10 or 20 percent where there is more complex state management is not trivial enough to ignore.*

Most (5/6) of the developers mentioned this lack of direct mapping of lifecycle methods to effects as one of the largest shortcomings of hooks. Several (4/6) of the developers had had difficulties with using effects and had even considered reverting certain components back to class components because of the problems with the effect hook. Especially components where it was seen as important to track its lifecycle were often kept as class components.

The developers’ opinions with lifecycle management with hooks were not exclusively negative. One of the developers expressed exclusively positive views on the effect based lifecycle management, praising the better division of functionality into separate effects following single responsibility principle. One developer expressed neutral opinions, and

the majority (5/6) expressed generally positive opinions with the drawback of tracking complex lifecycles being more difficult.

### **6.2.3 Custom and Third Party Hooks**

The developers' habits and experiences using custom hooks varied somewhat. Half (3/6) of the developers said that they had never written custom hooks, some (2/6) said they sometimes write custom hooks for specific cases, and one said they use custom hooks often. Custom hooks were mostly used with side effects such as data retrieval.

While custom hooks were seen as a good way to reuse and share functionality between components, they were also criticized for having to write several custom hooks for very similar issues since hooks were difficult to customize.

Since the introduction of Hooks, most third party libraries now also support hooks. Especially useful were the different state management and data retrieval related hooks. Different third party libraries that provided hooks mentioned by the developers were Redux, Apollo, and React Router. Hooks-based third party library interfaces were talked about in a positive manner by all developers.

## **6.3 Hooks Based React Development**

When discussing hooks, the participants were keen to express their opinions on different aspects of hooks such as their readability, ecosystem, collaboration, testability, and ease of learning.

### **6.3.1 Readability**

Every interviewed developer agreed that in most cases, function components with hooks were easier to read and understand than class components. One developer pointed out that just the good naming of hooks had led to easier to read code since the developer could just read the name of the hook and infer the functionality in a general way. All developers also said that developing with hooks require either less memory load from the developer or about the same as class components.

*[P2] Them being functional components makes [reading them] easier. It not having this-keyword makes it easier. There is less unnecessary text and it's grouped better in useEffect.*

### **6.3.2 Ecosystem and Collaboration**

Every participating developer agreed that the large size of the React ecosystem was one of the most influential advantages of React. What this meant for hooks based React development was that according to the developers, it had been easy to find solutions and answers to any questions they had about hooks, and they had been able to find libraries to support their hooks based development without difficulties. Additionally, most of the



community had adapted to hooks and there was little confusion due to the change from class components to function components.

One of the clear advantages of hooks was that of any component based architecture: the client of a component does not need to know how the component works, only what the interface is like, and what it produces. Similarly, regarding hooks it was said that collaboration was easy since the developers who use them only need to know what each hook takes as parameters and what it produces and not go too much into details on the inner workings of them. This was usually achieved via rigid documentation. Therefore, the developers had had no problems with collaborative programming with hooks, and some had felt that hooks based development was even more easy to do collaboratively than class based development.

### **6.3.3 Testability**

Every developer agreed that frontend testing was very important in maintaining the quality and functionality of the application code base. The pragmatic experiences and more fine opinions, however, varied slightly between the developers. On what to test, one developer said that only the most critical core components ought to be unit tested while another developer said that they unit test almost 100 percent of their components.

Some (2/6) developers said that while they think that writing frontend tests is important, there was rarely time or budget for that. In the end, most (4/6) developers agreed that the amount of testing was mostly a matter of budgeting and organizational or managerial decision making. One developer pointed out that it was the most important to decide what to test. They pointed out different testing styles and testing libraries such as unit testing, acceptance testing, and performance testing, which could be chosen based on the specific use case and the purpose of testing the application.

On testability of hooks specifically, the developers thought it was either easier compared to class components or about the same. One advantage of function components with hooks approach compared to class based approach was writing unit tests with correct state information. One developer explained that in their project they had a standard way of setting up the state of the application for testing, which all the unit tests could use, making testing easier. For testing styles other than unit testing, there was no difference between class components and function components since the visual output of the function and class components are the same in the end.

One difficulty with hooks was regarding custom hooks. Since custom hooks were seen as a kind of “hidden” dependency, it was difficult to know when mock hooks had to be written and how in unit tests. For example, when using a custom hook that accesses the context API, the developer has to wrap the component in a provider for unit testing, which was not an easy thing to notice.

## **6.4 Adaptation to Hooks**

Since the introduction of hooks to React in 2018, the React community has started moving towards hooks based React development away from class based development. The interview participants were asked about their own experiences on this transition and how do they think about refactoring old class components into function components with hooks.

### **6.4.1 Learning Hooks**

Most (5/6) of the interviewed developers said that the official documentation of React was the first place where they started learning about hooks. The official documentation was regarded highly, one developer even saying it is “the best documentation you can find”. One developer did mention of a case of the official documentation not having clear enough explanation of one aspect of the useState hook, but otherwise the explanations and examples presented in the official documentation web site was seen as very useful in helping the developers understand different aspects of hooks and hooks based React development.

One developer found out about hooks when talking with a family member who mentioned them. After hearing about them, that developer started to argue in favor of class components with their family member while looking at informal videos and blogs comparing class components to function components. After some time, that developer also started favoring function components after finding out more about them, but they mentioned that had they started their studying from the official sources it would have been more efficient compared to reading from informal sources.

One additional way of learning more about hooks that was mentioned during the interviews was React conferences. In these conferences, the developers had been able to hear about different hooks and their functionalities from other prominent figures of the web development community.

### **6.4.2 Transition to Hooks**

When adapting to new technologies, developers usually had to estimate issues such as the size of the ecosystem, maturity, the reliability of the provider, and the learning curve in order to make a decision whether to include the new technology to a project. One developer mentioned having internal processes such as sending requests for comments to other team members and having sometimes lengthy discussions taking different things into consideration.

In the case of hooks however, the developers had no problems adapting them to projects. Since hooks were already a part of the React core library and could be used alongside old code, adding them into current projects was not a difficult decision at all. However, the developers mostly did not jump into using hooks right away after they were

introduced, but rather waited a while to see whether they were accepted by the React developer community before starting to use them in their own projects themselves.

### **6.4.3 Refactoring Class Components**

Refactoring old class components into function components was seen as something that had some value and something that had to be done in some cases but not all. While the developers brought up advantages of refactoring – increased readability and maintainability, accommodating new developers not familiar with old styles, and more learning resources – their view of actually refactoring components from class components to function components mostly was that they lacked time and budget for that.

Although refactoring class components to function components specifically was not seen as a very time-consuming itself, it was done only in cases where it caused considerable benefits to the product. One developer expressed that they did not see any benefits of refactoring and had “made peace with class components”.

### **6.5 Interview Findings**

While the developers were mostly satisfied with hooks based React development, Most (4/6) had found cases in their development process where hooks were not a sufficient replacement to class components. Additionally, almost all (5/6) developers interviewed had had some usability problems with hooks, either with using or learning them.

Despite the shortcomings of hooks, all developers said that they had embraced hooks based development and abandoned class components except in the few cases where they could not be replaced by hooks, in which cases they would keep the class components.

## 7 Findings and Discussion

In this chapter, the combined findings of the two studies regarding usability of React hooks are presented. Firstly, the positive (Table 6) and negative (Table 7) usability issues found will be presented and classified by Nielsen's [1994] usability heuristics. Then, the findings concerning adaptation of hooks will be presented. After that, the reliability of the study will be covered, and finally, potential future research topics will be discussed.

### 7.1 Findings

Although the number of negative usability findings was close to the number of positive ones, the positive ones were seen as more important for most use cases by the developers. The developers were quick to praise hooks at first, and they said that hooks were sufficient and appropriate in most cases. The problems started to show in irregular cases where the basic functionality of hooks was not be sufficient for various reasons.

The positive usability findings were mostly regarding *flexibility and efficiency of use, aesthetic and minimalist design* with one finding each in *consistency and standards, recognition rather than recall, and help and documentation*. The negative findings were regarding *error prevention, visibility of system status, consistency and standards, recognition rather than recall, and flexibility and efficiency of use* as presented in Table 8.

Positive Usability Finding	Heuristic
<b>+Hook names consistently start with <i>use</i></b>	Consistency and standards
<b>+Hook naming mostly recognizable</b>	Recognition rather than recall
<b>+Ability to declare arbitrary number of local state variables with <i>useState</i></b>	Flexibility and efficiency of use
<b>+Combining and reusing state logic</b>	Flexibility and efficiency of use
<b>+Good third party library support</b>	Flexibility and efficiency of use
<b>+Quick to refactor from class components to function components with hooks</b>	Flexibility and efficiency of use
<b>+Separation from core component (<i>this-key-word</i>)</b>	Aesthetic and minimalist design
<b>+Better code structure with effect hooks</b>	Aesthetic and minimalist design
<b>+Compact state and lifecycle management</b>	Aesthetic and minimalist design
<b>+Large ecosystem and stellar documentation leading to ease of learning</b>	Help and documentation

Table 6 Positive usability findings

Negative Usability Finding	Heuristic
<b>-Too “magical” – functionality not instantly visible to developers</b>	Visibility of system status
<b>-No direct mapping from class lifecycle methods</b>	Consistency and standards
<b>-Leaky abstraction – Similar hooks where only difference is optimization</b>	Error prevention
<b>-Hooks are a hidden dependency when testing</b>	Error prevention
<b>-Some hook names not recognizable</b>	Recognition rather than recall
<b>-Effect based lifecycle management not sufficient in all use cases</b>	Flexibility and efficiency of use

Table 7 Negative usability findings

Heuristic	Positive findings	Negative findings
<b>Visibility of system status</b>	0	1
<b>Match between system and the real world</b>	0	0
<b>User control and freedom</b>	0	0
<b>Consistency and standards</b>	1	1
<b>Error prevention</b>	0	2
<b>Recognition rather than recall</b>	1	1
<b>Flexibility and efficiency of use</b>	4	1
<b>Aesthetic and minimalist design</b>	3	0
<b>Help users recognize, diagnose, and recover from errors</b>	0	0
<b>Help and documentation</b>	1	0

Table 8 Found Usability Issues Categorized by Usability Heuristic

The interview participants had adapted to hooks well by the time of the interviews. The adaptation did not happen directly after the introduction of hooks, but rather gradually as the hooks became increasingly more favored by the React developer community. Some developers had replaced class components completely with function components and hooks, while some developers still saw class components as a useful tool for certain situations.

## 7.2 Reliability

The reliability of the research was considered in various ways. Firstly, the interview participants chosen for the interviews had to have enough information and experience of React development. Therefore, only developers who had professional React experience

were chosen. Few potential participants who had some non-professional React experience were found as well during the participant search phase but had to be declined. Furthermore, the chosen participants all had varying amounts of experience, as seen in figures 9 and 10, which added different perspectives to the analysis of the subject matter. Secondly, the themes that were selected for the interviews were chosen as not to include any opinions of the interviewer themselves and to give a holistic overview of the inspected areas. Thirdly, the interviews were recorded and transcribed in a way which allowed the recording of specific thoughts of the participants. Lastly, the analysis of the data was done in a manner which considered every interviewee's opinion. During the interviews, every participant was given the chance to give their opinion on all the pre-determined themes.

For the greater reliability of the case study, three different components from a real application were selected. All the inspected components were part of the same application but had largely different functions inside the application: some only included local state, some included Redux API, and one included a custom hook. All the components were written by the same person.

### **7.3 Future Research**

This study has, most of all, added support for the need for thoughtful API development and evaluation in the libraries included in the industry standard. While the writer in no way wants to hinder rapid development and innovation that is characteristic to modern development of web development tools by implementing long evaluation and design periods, the regular developer who builds web applications will definitely appreciate stable and highly usable tools in their everyday development.

This study has also shown that there is need for more research on API usability. APIs have traditionally been studied regarding their performance and degree of use, but recent trends of improved speed of computers and browsers have revealed a need for study on the usability aspect of APIs.

In the future, research on the usability of APIs will be essential. Until recently, most research on web technologies has been regarding the performance aspect of development solutions while the usability of them has been studied only little. As the computational power of computers and environments such as web browsers increase, the significance of usability over performance will see an increase as well.

More extensive evaluation of hooks specifically would certainly reveal problems this study was not able to find. For the development of React and React hooks, usability is one of the aspects which will determine whether React will still be the principal web user interface framework in the future.

## 8 Conclusion

In this chapter, the study done for this thesis will be inspected from a larger perspective. Then the research questions presented in the introduction will be addressed. After that, the place of this type of study in this research field will be inspected, and finally some final comments about the topic will be given.

The rise of the web as an application platform has been observed over the last years. As applications are moving to the web platform, there has been an increase of available web technologies and ways of development. The velocity of new styles, frameworks, and libraries appearing in the web development ecosystem has caused many developers to experience difficulties in choosing and adapting new technologies to their projects. This could lead to developers making suboptimal decisions of technology selection and changing the used technologies in the middle of a project causing much unnecessary work and costs. To support a steadier ecosystem, it is crucial to ensure that new technologies released have been developed and evaluated in a thoughtful manner.

This study aimed to evaluate the usability and adaption of one significant new feature of the most popular web user interface development framework – React hooks. Hooks were inspected by creating a case study and conducting developer interviews of web development professionals with the purpose of recognizing different issues that hooks had on state management in the React framework. Issues on usability and hooks adaptation were recognized and analyzed. While the results of the study can be regarded as appropriate, it must be recognized that the scope of the study was limited, and not all aspects of hooks based state management were found. For example, React is known to have good error messaging, which did not come up in any of the interviews.

The answer to the research questions presented in the introduction chapter are now presented.

*Question 1: What usability benefits and drawbacks do hooks bring to React?* – Hooks were found to offer several usability benefits with a few drawbacks. Overall, the developers held positive opinions on the usability of hooks in most cases, while expressing that they had severe problems in less common use cases. Ten positive usability issues and six negative usability issues presented in Chapter 7 were found.

*Question 2: How have project teams adapted to React hooks?* – In the few years since the introduction of hooks, most developers have started using them. The quick adopters often were ones who kept up with the React developer community regularly, while most developers had begun using hooks gradually and not immediately after their introduction. Waiting for hooks to be accepted and adapted by the larger developer community before adapting them to software projects themselves was common among the developers. Now developers still use class components in specific situations instead of fully substituting them with function components with hooks.

This study has shown the importance of thoughtful usability evaluation and development of new web technologies. In the rapid-moving world of web development, developers are increasingly in need of standard and steady web development techniques. While the fast movement of the web development ecosystem is unquestionably a prerequisite of innovation, the majority of the web development still happens in the real software projects where stability is a necessity.



## References

- Abrahamsson, P., M. A. Babar, and P. Kruchten. 2010. "Agility and Architecture: Can they Coexist?" *IEEE Software* 27 (2): 16-22. doi:10.1109/MS.2010.36.
- Abramov, Dan. 2021. "Redux - A Predictable State Container for JavaScript Apps. | Redux.", accessed Mar 1, 2021, <https://redux.js.org/>.
- Abramov, Dan. Twitter post. April 5, 2018, 4:56 a.m., [https://twitter.com/dan\\_abramov/status/981712092611989509](https://twitter.com/dan_abramov/status/981712092611989509).
- Apollo. 2021. "Managing Local State.", accessed Mar 1, 2021, <https://www.apollographql.com/docs/react/local-state/local-state-management/>.
- Bass, Len, Paul Clements, and Rick Kazman. 2013. *Software Architecture in Practice*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley.
- Biørn-Hansen, Andreas, Tim A. Majchrzak, and Tor-Morten Grønli. 2017. "Progressive Web Apps: The Possible Web-Native Unifier for Mobile Development." SCITE-PRESS.
- Caldiera, G. and V. R. Basili. 1991. "Identifying and Qualifying Reusable Software Components." *Computer (Long Beach, Calif.)* 24 (2): 61-70. doi:10.1109/2.67210.
- Casteleyn, Sven, Irene Garrig'os, and Jose-Norberto Maz'on. 2014. "Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond." *ACM Transactions on the Web* 8 (3): 1-46. doi:10.1145/2626369.
- Conallen, Jim. 1999. "Modeling Web Application Architectures with UML." *Communications of the ACM* 42 (10): 63-70. doi:10.1145/317665.317677.
- Daniel, Florian, Stefano Soi, Stefano Tranquillini, Fabio Casati, Chang Heng, and Li Yan. 2012. "Distributed Orchestration of User Interfaces." *Information Systems (Oxford)* 37 (6): 539-556. doi:10.1016/j.is.2011.08.001.
- ECMA-international. 2021. "ECMA-262: ECMAScript® 2020 Language Specification.", accessed Apr 8, 2021, <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>.
- Facebook Open Source. 2021a. "React – A JavaScript Library for Building User Interfaces.", accessed Feb 24, 2021, <https://reactjs.org/>.
- Facebook Open Source. 2021b. "State and Lifecycle – React.", accessed Feb 25, 2021, <https://reactjs.org/docs/state-and-lifecycle.html>.
- Facebook Open Source. 2021c. "JSX in Depth – React.", accessed Feb 24, 2021, <https://reactjs.org/docs/jsx-in-depth.html>.
- Facebook Open Source. 2021d. "Create a New React App – React.", accessed Feb 24, 2021, <https://reactjs.org/docs/create-a-new-react-app.html>.
- Facebook Open Source. 2021e. "Components and Props – React.", accessed Feb 25, 2021, <https://reactjs.org/docs/components-and-props.html>.

- Facebook Open Source. 2021f. "Introducing Hooks – React.", accessed Feb 25, 2021, <https://reactjs.org/docs/hooks-intro.html>.
- Facebook Open Source. 2021g. "Hooks API Reference – React.", accessed Feb 26, 2021, <https://reactjs.org/docs/hooks-reference.html>.
- Facebook Open Source. 2021h "Recoil.", accessed Mar 1, 2021, <https://recoiljs.org/>.
- Fielding, Roy T. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine.
- Gizas, Andreas, Sotiris Christodoulou, and Theodore Papatheodorou. 2012. "Comparative Evaluation of Javascript Frameworks." *Proceedings of the 21st International Conference on World Wide Web*.
- Goodman, Leo A. 1961. "Snowball Sampling." *The Annals of Mathematical Statistics*: 148-170.
- Google. 2021. "Angular.", accessed Apr 6, 2021, <https://angular.io/>.
- Graziotin, Daniel and Pekka Abrahamsson. 2013. "Making Sense Out of a Jungle of JavaScript Frameworks." *International Conference on Product Focused Software Process Improvement*. Springer, Berlin, Heidelberg, 2013.
- Greif, Sacha and Raphaël Benitte. 2021. "State of JS 2020.", accessed Feb 23, 2021, <https://2020.stateofjs.com/en-US/>.
- Gruhn, Volker and Rüdiger Striemer. 2018. *The Essence of Software Engineering*. 1st ed. Cham: Springer International Publishing. doi:10.1007/978-3-319-73897-0.
- Hanh, Evan. 2021. "Redux-Thunk.", accessed Apr 6, 2021, <https://github.com/reduxjs/redux-thunk>.
- Happe, Lucia, Barbora Buhnova, and Ralf Reussner. 2014. "Stateful Component-Based Performance Models." *Software and Systems Modeling* 13 (4): 1319-1343. doi:10.1007/s10270-013-0336-6.
- Harel, David. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8 (3): 231-274.
- Holzmann, Gerard J. 2018. "Software Components." *IEEE Software* 35 (3): 80-82. doi:10.1109/MS.2018.2141034.
- Hu, Gang, Linjie Zhu, and Junfeng Yang. 2018. "AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests." *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- Hunt, Pete. 2013. "Why did we Build React? – React Blog.", accessed Feb 24, 2021, <https://reactjs.org/blog/2013/06/05/why-react.html>.
- IEEE. 2000. *IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE.

- Jadhav, Madhuri A., Balkrishna R. Sawant, and Anushree Deshmukh. 2015. "Single Page Application using Angularjs." *International Journal of Computer Science and Information Technologies* 6 (3): 2876-2879.
- Karlsson, Stefan. 2019. "Exploratory Test Agents for Stateful Software Systems." *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- Kato, Daishi. 2021. "Use-Context-Selector.", accessed Apr 8, 2021, <https://github.com/dai-shi/use-context-selector>.
- Keeling, Michael. 2015. "Lightweight and Flexible: Emerging Trends in Software Architecture from the SATURN Conferences." *IEEE Software* 32 (3): 7-11. doi:10.1109/MS.2015.65.
- Krasner, Glenn E. and Stephen T. Pope. 1988. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System." *Journal of Object Oriented Programming* 1 (3): 26-49.
- Kruchten, P., H. Obbink, and J. Stafford. 2006. "The Past, Present, and Future for Software Architecture." *IEEE Software* 23 (2): 22-30. doi:10.1109/MS.2006.59.
- Lee, Jiyeon, Hayeon Kim, Junghwan Park, Insik Shin, and Soeul Son. 2018. "Pride and Prejudice in Progressive Web Apps: Abusing Native App-Like Features in Web Applications." *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.
- Linsley, Tanner. 2021a "React Query.", accessed Mar 1, 2021, <https://react-query.tanstack.com>.
- Linsley, Tanner. 2021b "Comparison | React Query Vs SWR Vs Apollo Vs RTK Query.", accessed Mar 1, 2021, <https://react-query.tanstack.com/comparison>.
- Linsley, Tanner. 2021c "React Query.", accessed Mar 24, 2021, <https://react-query.tanstack.com/overview>.
- Margalit, Gal. GitHub repository. 2021. <https://github.com/Wavez/react-hooks-lifecycle>
- Martin, Patricia Yancey and Barry A. Turner. 1986. "Grounded Theory and Organizational Research." *The Journal of Applied Behavioral Science* 22 (2): 141-157.
- Martin, Robert. 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design, First Edition*. 1st ed. Boston: Prentice Hall.
- Mesbah, A. and A. van Deursen. 2007. "Migrating Multi-Page Web Applications to Single-Page AJAX Interfaces." *11th European Conference on Software Maintenance and Reengineering (CSMR'07)* IEEE. doi:10.1109/CSMR.2007.33.
- Meyer, Daniel Z. and Leanne M. Avery. 2009. "Excel as a Qualitative Data Analysis Tool." *Field Methods* 21 (1): 91-112. doi:10.1177/1525822X08323985.
- Microsoft. 2021. "TypeScript: Typed JavaScript at any Scale.", accessed Apr 8, 2021, <https://www.typescriptlang.org>.

- Mikkonen, Tommi, Cesare Pautasso, Kari Systä, and Antero Taivalsaari. 2019. *On the Web Platform Cornucopia*. Cham: Springer International Publishing. doi:10.1007/978-3-030-19274-7\_25.
- Mikowski, Michael, and Josh Powell. 2013. *Single Page Web Applications. 1st ed.* Manning Publications.
- MobX. 2021. "README · MobX.", accessed Mar 1, 2021, <https://mobx.js.org/index.html>.
- Myers, Brad A. and Jeffrey Stylos. 2016. "Improving API Usability." *Communications of the ACM* 59 (6): 62-69.
- Myers, Brad, Scott Hudson, and Randy Pausch. 2000. "Past, Present, and Future of User Interface Software Tools." *ACM Transactions on Computer-Human Interaction* 7 (1): 3-28. doi:10.1145/344949.344959.
- Nielsen, Jakob. 1994. "10 Usability Heuristics for User Interface Design.", accessed Apr 5, 2021, <https://www.nngroup.com/articles/ten-usability-heuristics/>.
- Node.js. 2021. "Node.Js.", accessed 18.2., 2021, <https://nodejs.org/en/>.
- O'reilly, Tim. 2009. What is Web 2.0 " O'Reilly Media, Inc."
- Occhino, Tom and Walke, Jordan. 2013. "JS Apps at Facebook.", accessed Feb 24, 2021, <https://www.youtube.com/watch?v=GW0rj4sNH2w>.
- Paulson, L. D. 2005. "Building Rich Web Applications with Ajax." *Computer (Long Beach, Calif.)* 38 (10): 14-17. doi:10.1109/MC.2005.330.
- Piccioni, M., Furia, C. A., & Meyer, B. 2013. "An empirical study of API usability." *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 5-14)*. IEEE.
- Shklar, Leon and Rich Rosen. 2009. *Web Application Architecture: Principles, Protocols and Practices*. 2nd ed. John Wiley & Sons, Inc.
- Soni, Dilip, Robert Nord, and Christine Hofmeister. 1995. *Software Architecture in Industrial Applications* ACM. doi:10.1145/225014.225033.
- SWR. 2021. "SWR: React Hooks for Data Fetching.", accessed Mar 1, 2021, <https://swr.vercel.com>.
- Taivalsaari, A., T. Mikkonen, M. Anttonen, and A. Salminen. 2011. "The Death of Binary Software: End User Software Moves to the Web." 2011 *Ninth international conference on creating, connecting and collaborating through computing*. IEEE. doi:10.1109/C5.2011.9.
- Taivalsaari, Antero and Tommi Mikkonen. 2017. *The Web as a Software Platform: Ten Years Later* SCITEPRESS - Science and Technology Publications. doi:10.5220/0006234800410050.
- Tay, Yangshun. 2021. "In-Depth Overview | Flux.", accessed Feb 22, 2021, <https://facebook.github.io/flux/docs/in-depth-overview>.
- w3techs. 2021. "Usage Statistics of JavaScript Libraries for Websites.", accessed Feb 22, 2021, [https://w3techs.com/technologies/overview/javascript\\_library](https://w3techs.com/technologies/overview/javascript_library).

Wagner, Jeremy L. 2017. *Web Performance in Action: Building Fast Web Pages* Manning Publications Company.

WebAssembly. 2021. "WebAssembly.", accessed Apr 8, 2021, <https://webassembly.org>.

Wills, Alan Cameron. 1998. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Reading (Mass.): Addison-Wesley.

XState. 2021. "XState.", accessed Apr 6, 2021, <https://xstate.js.org/>.

You, Evan. 2021. "Vue.js.", accessed Feb 23, 2021, <https://vuejs.org/>.

### **DateTimeDisplay component source code (class version)**

```
export class DateTimeDisplay extends React.Component<any, any> {
  constructor(props: any) {
    super(props);
    this.state = {
      dateTime: this.formatDateAndTimeStrings(new Date()),
      widthOfTimeString: ''
    };
  }
  private intervalId: NodeJS.Timeout | undefined;

  componentDidMount() {
    this.intervalId = setInterval(() => {
      this.setState({
        dateTime: this.formatDateAndTimeStrings(new Date()),
        widthOfTimeString: `${ this.state.dateTime.time.length }ch`
      });
    }, 1000);
  }
  componentWillUnmount() {
    if (this.intervalId) {
      clearInterval(this.intervalId);
    }
  }

  formatDateAndTimeStrings(date: Date): DateAndTimeStrings {
    const localization: Localization = Localization.get();
    const dateString: string = localization.formatDateDowMmDdYyyy(date).replace(',', ' ');
    const timeString: string = localization.formatDateHhMmSs(date);
    return ({ date: dateString, time: timeString });
  }

  render() {
    return (
      <div className={ css.root }>
        <div className={ css.time } data-testid={ "timeField" } style={ { minWidth:
this.state.widthOfTimeString } } >
          { this.state.dateTime.time }
        </div>
        <div className={ css.date } data-testid={ "dateField" }>
          { this.state.dateTime.date }
        </div>
      </div>
    );
  }
}
```

### DateTimeDisplay component source code (function version)

```
export function DateTimeDisplay(): JSX.Element {
  const [ dateTime, setDateTime ] = useState(formatDateAndTimeStrings(new Date()));
  const [ widthOfTimeString, setWidthOfTimeString ] = useState('');

  useEffect(() => {
    const intervalId: NodeJS.Timeout = setInterval(() => {
      const now: Date = new Date();
      setDateTime(formatDateAndTimeStrings(now));
    }, 1000);
    setDateTime(formatDateAndTimeStrings(new Date()));
    setWidthOfTimeString(`${ dateTime.time.length }ch`);
    return () => clearInterval(intervalId);
  }, []);

  function formatDateAndTimeStrings(date: Date): DateAndTimeStrings {
    const localization: Localization = Localization.get();
    const dateString: string = localization.formatDateDowMmDdYyyy(date).replace(', ', ' ');
    const timeString: string = localization.formatDateHhMmSs(date);
    return ({ date: dateString, time: timeString });
  }

  return (
    <div className={ css.root }>
      <div className={ css.time } data-testid={ "timeField" } style={ { minWidth:
widthOfTimeString } } >
        { dateTime.time }
      </div>
      <div className={ css.date } data-testid={ "dateField" }>
        { dateTime.date }
      </div>
    </div>
  );
}
```

### LoginView component source code (class version)

```
class LoginView extends React.Component<any, any>{
  constructor(props: any) {
    super(props);
    this.state = {
      credentials: { username: '', password: '' },
      error: '',
      loading: false,
      disabled: true
    };
    this.handleInputChange = this.handleInputChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  private nameInput = React.createRef<HTMLInputElement>();
  handleInputChange(e: React.ChangeEvent<HTMLInputElement>): void {
    const { name, value } = e.target;
    this.setState({ credentials: { ...this.state.credentials, [name]: value } },
      () => this.setState({ disabled: (this.state.credentials.username === '' ||
this.state.credentials.password === '') })
    );
    this.setState({ error: '' });
  }
  async handleSubmit(e: React.MouseEvent<HTMLElement>): Promise<void> {
    e.preventDefault();
    this.setState({
      error: '',
      loading: true,
      disabled: true
    });
    let response = null;
    try {
      response = await this.auth(this.state.credentials);
      this.props.preferenceUserSet(response.userName);
    } catch (e) {
      this.setState({
        error: e,
        disabled: false
      });
    } finally {
      this.setState({ loading: false });
      if (this.nameInput.current) {
        this.nameInput.current.focus();
      }
    }
  }

  private productName = i18n.labels.productName;
  private textCut = this.productName.length < 10
    ? ' '
    : <br />;

  auth(credentials: Credentials) {
    // <removed>
  }

  render() {
    logger.debug("render()");
    const i18n: I18n = this.props.i18n;
    // <removed>

    return (
      <div className={ css.root } >
        <img src={ blLogo } className={ css.blLogo } alt="Nokia Bell-Labs Logo" />
        <div className={ css.version }>ver. { process.env.REACT_APP_VERSION }</div>
        <div className={ css.loginContainer } >
          <div className={ css.loginPanel } >
            <div className={ css.body } >
              <h1 className={ css.loginTitle } data-testid={ 'loginTitle' } >
                { i18n.labels.welcomeTo } { this.textCut }
                <span className={ css.productName }>{ this.productName }</span>
              </h1>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
```



```
<div className={ css.textFields }>
  <div className={ css.labelAndInput }>
    <div className={ css.label }>
      { i18n.labels.userName }
    </div>
    <input
      className={ css.textField }
      type='text'
      name='username'
      data-testid={ 'usernameInput' }
      onChange={ this.handleInputChange }
      value={ this.state.credentials.username }
      disabled={ this.state.loading }
      autoFocus={ true }
      ref={ this.nameInput }
    />
  </div>
  <div className={ css.labelAndInput }>
    <div className={ css.label }>
      { i18n.labels.password }
    </div>
    <input
      className={ css.textField }
      type='password'
      name='password'
      data-testid={ 'pwInput' }
      onChange={ this.handleInputChange }
      value={ this.state.credentials.password }
      disabled={ this.state.loading }
    />
  </div>
  { this.state.loading
    ? <div className={ css.status }> { i18n.labels.loggingIn } </div>
    : <div className={ css.error } data-testid={ 'error' }>{
this.state.error }</div> }
  </div>
  <div className={ css.buttonContainer }>
    <button className={ css.loginButton }
      onClick={ this.handleSubmit }
      disabled={ this.state.disabled }>
      { i18n.labels.loginButton }
    </button>
  </div>
</form>
</div>
</div>
</div>
</div>
);
}
}

const mapStateToProps = (state: RootState) => ({
  i18n: state.preferenceSettings.i18n
});
const mapDispatchToProps = (dispatch: AppDispatch) => ({
  preferenceUserSet: (u: any) => dispatch(preferenceUserSet(u)),
  appMessageDisplay: (m: any) => dispatch(appMessageDisplay(m)),
  preferenceSet: (p: any) => dispatch(preferenceSet(p))
});

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

### LoginView component source code (function version)

```
export function LoginView(): JSX.Element {
  logger.debug("render()");
  const [ credentials, setCredentials ] = useState<Credentials>({ username: '', password: '' });
  const [ error, setError ] = useState<string>('');
  const [ loading, setLoading ] = useState<boolean>(false);
  const [ disabled, setDisabled ] = useState<boolean>(true);
  const i18n: I18n = useSelector((state: RootState) => state.preferenceSettings.i18n);
  const dispatch = useDispatch();
  const userInput = useRef<HTMLInputElement | null>(null);
  const productName = i18n.labels.productName;
  const [userPreferenceQuery, { data: userData }] = use-
  LazyQuery(API_BASE_MAP.GET_USER_INFO, {
    fetchPolicy: "no-cache"
  });
  const textCut = productName.length < 10
    ? ' '
    : <br />;

  useEffect(() => {
    if (userInput.current !== null){
      userInput.current.focus();
    }
  }, [loading]);

  useEffect(() => {
    setDisabled(credentials.username === '' || credentials.password === '');
  }, [credentials]);

  const handleInputChange = (e: React.ChangeEvent<HTMLInputElement>): void => {
    const { name, value } = e.target;
    setCredentials({ ...credentials, [name]: value });
    setError('');
  };

  const handleSubmit = async (e: React.MouseEvent<HTMLElement>): Promise<void> => {
    e.preventDefault();
    setError('');
    setLoading(true);
    setDisabled(true);
    let response = null;
    try {
      response = await auth(credentials);
      dispatch(preferenceUserSet(response.userName));
    } catch (e) {
      setError(e);
      setDisabled(false);
    } finally {
      setLoading(false);
    }
  };

  if (userData) {
    if (userData.getUserProfile.error) {
      dispatch(appMessageDisplay({
        type: "error",
        message: i18n.messages.failedSavePreferences,
        details: userData.getUserProfile.error
      }));
    }
    if (userData.getUserProfile?.profileJson && userData.getUserProfile?.lastUpdatedTS)
  {
    dispatch(preferenceLastTsSet(JSON.parse(userData.getUserProfile.lastUpdatedTS)));
    const profileSettings = JSON.parse(userData.getUserProfile.profileJson);
    delete profileSettings.lastUpdatedTS;
    dispatch(preferenceSet(profileSettings));
  }
  }
  return (
    <Route>
      <Redirect to="/map" />
    </Route>
  );
}
```

```
);
}

return (
  <div className={ css.root }>
    <img src={ blLogo } className={ css.blLogo } alt="Nokia Bell-Labs Logo" />
    <div className={ css.version }>ver. { process.env.REACT_APP_VERSION }</div>
    <div className={ css.loginContainer }>
      <div className={ css.loginPanel }>
        <div className={ css.body }>
          <h1 className={ css.loginTitle } data-testid={ 'loginTitle' }>
            { i18n.labels.welcomeTo } { textCut }
            <span className={ css.productName }>{ productName }</span>
          </h1>
          <form>
            <div className={ css.textFields }>
              <div className={ css.labelAndInput }>
                <div className={ css.label }>
                  { i18n.labels.userName }
                </div>
                <input
                  className={ css.textField }
                  type='text'
                  name='username'
                  data-testid={ 'usernameInput' }
                  onChange={ handleInputChange }
                  value={ credentials.username }
                  disabled={ loading }
                  autoFocus={ true }
                  ref={ userInput }
                />
              </div>
              <div className={ css.labelAndInput }>
                <div className={ css.label }>
                  { i18n.labels.password }
                </div>
                <input
                  className={ css.textField }
                  type='password'
                  name='password'
                  data-testid={ 'pwInput' }
                  onChange={ handleInputChange }
                  value={ credentials.password }
                  disabled={ loading }
                />
              </div>
              { loading
                ? <div className={ css.status }> { i18n.labels.loggingIn } </div>
                : <div className={ css.error } data-testid={ 'error' }>{ error }</div>
              }
            </div>
            <div className={ css.buttonContainer }>
              <button className={ css.loginButton }
                onClick={ handleSubmit }
                disabled={ disabled }>
                { i18n.labels.loginButton }
              </button>
            </div>
          </form>
        </div>
      </div>
    </div>
  );
}
```

**UserPreferencesWindow component source code (class version)**

```

class UserPreferencesWindowC extends React.Component<any, any> {
  constructor(props: any) {
    super(props);
    this.onClose = this.onClose.bind(this);
  }

  onClose() {
    const user: string | undefined = this.props.userName;
    const lastUpdatedTS: number | undefined = this.props.lastUpdatedTS;
    const preferences = this.props.userPreferences;
    const [sendUserPrefs] = useMutation(API_BASE_MAP.PUT_USER_INFO);

    if (user) {
      const kvl = Object.entries(preferences).map(([key, value]) => ({ key, value }));
      const userProfileInput: UserProfileInput = { userId: user, lastUpdatedTS:
lastUpdatedTS, kvl: (kvl as KeyValuePair[]) };
      const i18n: I18n = this.props.i18n;
      sendUserPrefs({ variables: { userProfileInput } })
        .then((resp: any) => {
          this.props.preferenceLastTsSet(resp.data.putUserProfile.lastUpdatedTS);
          const getPrefError = resp.data.putUserProfile.error;
          if (getPrefError) {
            logger.error("Error sending preferences:", getPrefError);
            const errorCode = resp.data.putUserProfile.errorCode;
            const errorDetails = errorCode === 'UPE-0001'
              ? i18n.getMsg("errorCode_" + errorCode) as keyof Messages
              : getPrefError;

            this.props.appMessageDisplay({
              type: "error",
              message: i18n.messages.failedSavePreferences,
              details: errorDetails
            });
          }
        })
        .catch((error: any) => {
          logger.error("Error sending preferences:", error.message);
          this.props.appMessageDisplay({
            type: "error",
            message: i18n.messages.failedSavePreferences,
            details: error.message
          });
        });
    }

    this.props.preferenceWindowToggle();
  }

  render() {
    const i18n: I18n = this.props.i18n;
    const prefWindowOpen: boolean | undefined = this.props.preferencesWindowOpen;
    const preferences = this.props.userPreferences;
    const setPreferences = (p: any) => this.props.preferenceSet(p);
    const availableLocales: AvailableLocalesMap = this.props.availableLocales;

    const tabs: Tab[] = [
      {
        id: "general", title: i18n.labels.generalTitle,
        component: <GeneralTab i18n={ i18n } key={ "generalTab" } preferences={ prefer-
ences } setPreferences={ setPreferences } />
      },
      {
        id: "languageRegion", title: i18n.labels.languageAndRegionTitle,
        component: <LocaleTab i18n={ i18n } availableLocales={ availableLocales } key={
"localeTab" } preferences={ preferences } setPreferences={ setPreferences } />
      },
    ];

    if (!prefWindowOpen) {
      return null;
    }
  }
}

```

```
return (
  <Draggable bounds="parent" handle=".dragHandle">
    <div className={ css.root }>
      <div className={ css.titleContainer + " dragHandle" }>
        <div className={ css.leftSide }></div>
        <div className={ css.title }>{ i18n.labels.settingsTitle }</div>
        <IconButton
          name={ "Close" }
          tooltip={ i18n.labels.closeTooltip }
          iconSvg={ <Close/> }
          onClick={ this.onClose }
          testId={ "userPref-close-button" }
          borderless={ true }/>
      </div>
      <div className={ css.body }>
        <TabPanel tabs={ tabs } toolbarButtons={ [] } />
      </div>
    </Draggable>
  );
}
}

class GeneralTab extends React.Component<any, any> {
  constructor(props: any) {
    super(props);
  }

  onVideoQualityChange(value: any){
    this.props.setPreferences({ ...this.props.preferences, videoQuality: value.value });
  }

  render() {
    const videoQualityOptions = [
      { value: 'high', label: this.props.i18n.labels.highValue },
      { value: 'medium', label: this.props.i18n.labels.mediumValue },
      { value: 'low', label: this.props.i18n.labels.lowValue },
    ];
    const themeControl = <ThemeControl
      preferences={ this.props.preferences }
      setPreferences={ this.props.setPreferences }
      i18n={ this.props.i18n }
    />;
    const videoQualityControl = <DropDownControl
      id={ 'videoQualitySelection' }
      options={ videoQualityOptions }
      selected={ this.props.preferences.videoQuality }
      onChange={ (v: any) => this.onVideoQualityChange(v) }
    />;

    return (
      <div className={ css.generalContent }>
        <LabelAndControl label={ this.props.i18n.labels.appearanceLabel } control={
themeControl } />
        <LabelAndControl label={ this.props.i18n.labels.videoQuality } control={ vide-
oQualityControl } />
      </div>
    );
  }
}

class LocaleTab extends React.Component<any, any> {
  constructor(props: any) {
    super(props);
  }

  private temperatureOptions = [
    { value: 'C', label: '°C - Celsius' },
    { value: 'F', label: '°F - Fahrenheit' }
  ];
  private languageOptions: BasicControlOptions[] = [];

  componentDidMount(){
    const availableLocales: AvailableLocalesMap = this.props.availableLocales;
    Object.entries(availableLocales).forEach (
      ([key, value]) => this.languageOptions.push({ value: key, label: value })
    );
  }
}
```

```
    }

    onTemperatureUnitChange(value: any) {
      this.props.setPreferences({ ...this.props.preferences, temperatureUnit: value });
    }
    onLanguageChange(value: any) {
      this.props.setPreferences({ ...this.props.preferences, locale: value.value });
    }
  }

  render() {
    const temperatureControl = <RadioControl
      id={ 'temperatureUnitSelection' }
      options={ this.temperatureOptions }
      selected={ this.props.preferences.temperatureUnit }
      onChange={ (v: any) => this.onTemperatureUnitChange(v) }
    />;
    const languageControl = <DropDownControl
      id={ 'languageSelection' }
      options={ this.languageOptions }
      selected={ this.props.preferences.locale }
      onChange={ (v: any) => this.onLanguageChange(v) }
    />;
    return (
      <div className={ css.generalContent }>
        <LabelAndControl label={ this.props.i18n.labels.language } control={ language-
Control } />
        <LabelAndControl label={ this.props.i18n.labels.temperatureUnit } control={ tem-
peratureControl } />
      </div>
    );
  }
}

function LabelAndControl({ label, control }: {label: string, control: JSX.Element}):
JSX.Element {
  return (
    <div className={ css.preferenceContainer }>
      <div className={ css.preferenceName }>
        {label}:
      </div>
      { control }
    </div>
  );
}

class ThemeControl extends React.Component<any, any> {
  constructor(props: PreferenceHookTypes) {
    super(props);
  }
  render() {
    const themeToSvg = {
      'dark': <DarkTheme/>,
      'darker': <DarkerTheme/>,
      'custom': <CustomTheme/>,
      'light': <CustomTheme/>,
      'nokia': <CustomTheme/>
    };
    return (
      <div className={ css.preferenceControl }>
        {availableThemes.map((theme) => (
          <div className={ css.themeSelection } key={ theme }>
            <IconButton
              className={ css.themeSvg }
              testId={ `${ theme }-themeSelection` }
              name={ theme }
              iconSvg={ themeToSvg[theme] }
              tooltip={ theme }
              selected={ this.props.preferences.theme === theme }
              onClick={ () => this.props.setPreferences({ ...this.props.preferences,
theme: theme }) }
              borderless={ true }
              withoutBackground={ true }
            />
            <div className={ this.props.preferences.theme === theme ? css.select-
edThemeName : css.themeName }>
```

```
        {this.props.i18n ? (this.props.i18n.labels as any)[`themeValue_${ theme
}]` : null}
      </div>
    </div>
  )))
</div>
);
}
}

function DropDownControl(props: BasicControlProps) {
  return (
    <div className={ css.preferenceDropdown }>
      <DropDownBox
        id={ props.id }
        value={ props.options.find( ({ value }) => value === props.selected) }
        onChange={ props.onChange }
        options={ props.options }
        placeholder={ props.selected }
      />
    </div>
  );
}

function RadioControl(props: BasicControlProps) {
  return (
    <div className={ css.radioControlContainer }>
      {props.options.map((value: any) => (
        <div key={ value.value } className={ css.radioControl }>
          <RadioButton
            name={ value.value }
            selected={ props.selected === value.value }
            onSelect={ () => props.onChange(value.value) }
            label={ value.label }
          />
        </div>
      ))}
    </div>
  );
}

const mapStateToProps = (state: RootState) => ({
  i18n: state.preferenceSettings.i18n,
  preferencesWindowOpen: state.viewportSettings.preferencesWindowOpen,
  userName: state.preferenceSettings.userName,
  lastUpdatedTS: state.preferenceSettings.lastUpdatedTS,
  userPreferences: state.preferenceSettings.userPreferences,
  availableLocales: state.preferenceSettings.availableLocales
});

const mapDispatchToProps = (dispatch: AppDispatch) => ({
  preferenceWindowToggle: () => dispatch(preferenceWindowToggle()),
  preferenceSet: (p: any) => dispatch(preferenceSet(p)),
  preferenceLastTsSet: (ts: any) => dispatch(preferenceLastTsSet(ts)),
  appMessageDisplay: (m: any) => dispatch(appMessageDisplay(m))
});

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

**UserPreferencesWindow component source code (function version)**

```

export function UserPreferencesWindow(): JSX.Element | null {
  const i18n: I18n = useSelector((state: RootState) => state.preferenceSettings.i18n);
  const prefWindowOpen: boolean | undefined = useSelector((state: RootState) =>
state.viewportSettings.preferencesWindowOpen);
  const dispatch = useDispatch();
  const [preferences, setPreferences] = usePreference();
  const user: string | undefined = useSelector((state: RootState) => state.preference-
Settings.userName);
  const [sendUserPrefs] = useMutation(API_BASE_MAP.PUT_USER_INFO);
  const lastUpdatedTS: number | undefined = useSelector((state: RootState) =>
state.preferenceSettings.lastUpdatedTS);

  const onClose = () => {
    if (user) {
      const kv1 = Object.entries(preferences).map(([key, value]) => ({ key, value }));
      const userProfileInput: UserProfileInput = { userId: user, lastUpdatedTS:
lastUpdatedTS, kv1: kv1 };
      sendUserPrefs({ variables: { userProfileInput } })
        .then((resp: any) => {
          dispatch(preferenceLastTsSet(resp.data.putUserProfile.lastUpdatedTS));
          const getPrefError = resp.data.putUserProfile.error;
          if (getPrefError) {
            logger.error("Error sending preferences:", getPrefError);
            const errorCode = resp.data.putUserProfile.errorCode;
            const errorDetails = errorCode === 'UPE-0001'
              ? i18n.getMsg("errorCode_" + errorCode) as keyof Messages
              : getPrefError;

            dispatch(appMessageDisplay({
              type: "error",
              message: i18n.messages.failedSavePreferences,
              details: errorDetails
            }));
          }
        })
        .catch((error: any) => {
          logger.error("Error sending preferences:", error.message);
          dispatch(appMessageDisplay({
            type: "error",
            message: i18n.messages.failedSavePreferences,
            details: error.message
          }));
        });
    }

    dispatch(preferenceWindowToggle());
  };
  const tabs: Tab[] = [
    {
      id: "general", title: i18n.labels.generalTitle,
      component: <GeneralTab key={ "generalTab" } preferences={ preferences } setPrefer-
ences={ setPreferences } />
    },
    {
      id: "languageRegion", title: i18n.labels.languageAndRegionTitle,
      component: <LocaleTab key={ "localeTab" } preferences={ preferences } setPrefer-
ences={ setPreferences } />
    },
    // {
    //   id: "timeZone", title: i18n.labels.timeZoneTitle,
    //   component: <TimeZoneTab />
    // },
  ];

  if (!prefWindowOpen) {
    return null;
  }

  return (
    <Draggable bounds="parent" handle=".dragHandle">
      <div className={ css.root }>

```



```
<div className={ css.titleContainer + " dragHandle" }>
  <div className={ css.leftSide }></div>
  <div className={ css.title }>{ i18n.labels.settingsTitle }</div>
  <IconButton
    name={ "Close" }
    tooltip={ i18n.labels.closeTooltip }
    iconSvg={ <Close/> }
    onClick={ onClose }
    testId={ "userPref-close-button" }
    borderless={ true }/>
  </div>
  <div className={ css.body }>
    <TabPanel tabs={ tabs } toolbarButtons={ [] } />
  </div>
</div>
</Draggable>
);
}

function usePreference(): [UserPreferences, React.Dispatch<React.SetState-
Action<UserPreferences>>] {
  const preferencesFromStore: UserPreferences = useSelector((state: RootState) =>
state.preferenceSettings.userPreferences);
  const [preferences, setPreferences] = useState<UserPreferences>(preferencesFromStore);
  const dispatch = useDispatch();

  useEffect(() => {
    if (preferences && preferences !== preferencesFromStore) {
      dispatch(preferenceSet(preferences));
    }
  }, [preferences]);

  return [preferences, setPreferences];
}

function GeneralTab(props: any) :JSX.Element {
  const i18n: I18n = useSelector((state: RootState) => state.preferenceSettings.i18n);
  const videoQualityOptions = [
    { value: 'high', label: i18n.labels.highValue },
    { value: 'medium', label: i18n.labels.mediumValue },
    { value: 'low', label: i18n.labels.lowValue },
  ];

  const onVideoQualityChange = (value: any) => {
    props.setPreferences({ ...props.preferences, videoQuality: value.value });
  };

  const themeControl = <ThemeControl
    preferences={ props.preferences }
    setPreferences={ props.setPreferences }
  />;
  const videoQualityControl = <DropDownControl
    id={ 'videoQualitySelection' }
    options={ videoQualityOptions }
    selected={ props.preferences.videoQuality }
    onChange={ onVideoQualityChange }
  />;

  return (
    <div className={ css.generalContent }>
      <LabelAndControl label={ i18n.labels.appearanceLabel } control={ themeControl } />
      <LabelAndControl label={ i18n.labels.videoQuality } control={ videoQualityControl
    } />
    </div>
  );
}

function LocaleTab(props: PreferenceHookTypes) :JSX.Element {
  const i18n: I18n = useSelector((state: RootState) => state.preferenceSettings.i18n);
  const availableLocales: AvailableLocalesMap = useSelector((state: RootState) =>
state.preferenceSettings.availableLocales);
  const languageOptions: BasicControlOptions[] = [];
  Object.entries(availableLocales).forEach (
    ([key, value]) => languageOptions.push({ value: key, label: value })
  );

  const onLanguageChange = (value: any) => {
```

```
    props.setPreferences({ ...props.preferences, locale: value.value });
  };

const onTemperatureUnitChange = (value: any) => {
  props.setPreferences({ ...props.preferences, temperatureUnit: value });
};

const temperatureOptions = [
  { value: 'C', label: '°C - Celsius' },
  { value: 'F', label: '°F - Fahrenheit' }
];

const languageControl = <DropDownControl
  id={ 'languageSelection' }
  options={ languageOptions }
  selected={ props.preferences.locale }
  onChange={ onLanguageChange }
/>;
const temperatureControl = <RadioControl
  id={ 'temperatureUnitSelection' }
  options={ temperatureOptions }
  selected={ props.preferences.temperatureUnit }
  onChange={ onTemperatureUnitChange }
/>;

return (
  <div className={ css.generalContent }>
    <LabelAndControl label={ i18n.labels.language } control={ languageControl } />
    <LabelAndControl label={ i18n.labels.temperatureUnit } control={ temperatureCon-
  trol } />
  </div>
);
}

function LabelAndControl({ label, control }: {label: string, control: JSX.Element}):
JSX.Element {
  return (
    <div className={ css.preferenceContainer }>
      <div className={ css.preferenceName }>
        {label}:
      </div>
      { control }
    </div>
  );
}

function ThemeControl(props: PreferenceHookTypes) {
  const i18n: I18n = useSelector((state: RootState) => state.preferenceSettings.i18n);

  const themeToSvg = {
    'dark': <DarkTheme/>,
    'darker': <DarkerTheme/>,
    'custom': <CustomTheme/>,
    'light': <CustomTheme/>,
    'nokia': <CustomTheme/>
  };
  return (
    <div className={ css.preferenceControl }>
      {availableThemes.map((theme) => (
        <div className={ css.themeSelection } key={ theme }>
          <IconButton
            className={ css.themeSvg }
            testId={ `${ theme }-themeSelection` }
            name={ theme }
            iconSvg={ themeToSvg[theme] }
            tooltip={ theme }
            selected={ props.preferences && props.preferences.theme === theme }
            onClick={ () => props.setPreferences({ ...props.preferences, theme: theme }) }
          >
            borderless={ true }
            withoutBackground={ true }
          </>
          <div className={ props.preferences.theme === theme ? css.selectedThemeName :
            css.themeName }>{(i18n.labels as any)[`themeValue_${ theme }`]}</div>
        </div>
      ))}
    </div>
  );
}
```

```
);
}

function DropDownControl(props: BasicControlProps) {
  return (
    <div className={ css.preferenceDropdown }>
      <DropDownBox
        id={ props.id }
        value={ props.options.find( ({ value }) => value === props.selected) }
        onChange={ props.onChange }
        options={ props.options }
        placeholder={ props.selected }
      />
    </div>
  );
}

function RadioControl(props: BasicControlProps) {
  return (
    <div className={ css.radioControlContainer }>
      {props.options.map((value: any) => (
        <div key={ value.value } className={ css.radioControl }>
          <RadioButton
            name={ value.value }
            selected={ props.selected === value.value }
            onSelect={ () => props.onChange(value.value) }
            label={ value.label }
          />
        </div>
      ))}
    </div>
  );
}
```

## Pre-interview Form

### Pre-interview form

\* Required

Name \*

Your answer \_\_\_\_\_

Age \*

Your answer \_\_\_\_\_

Country of residence \*

Your answer \_\_\_\_\_

For how long have you used JavaScript? \*

0-2 years

2-5 years

5-10 years

10-15 years

More than 15 years

How familiar are you with following JavaScript frameworks? \*

	Not familiar at all	Somewhat familiar	Familiar
ReactJS	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Vue.js	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Angular	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Preact	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Ember	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Svelte	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

ReactJS

For how long have you used ReactJS \*

- Less than 1 year
- 1-2 years
- 2-4 years
- More than 4 years

Pick your opinion on following claims regarding ReactJS \*

	Strongly disagree	Disagree	Can't say	Agree	Strongly agree
I would recommend React to a colleague.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
React is moving in the right direction.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Hooks successfully fixed the issues of class components.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
React architecture works well for me.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Hooks are intuitive and easy to learn.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Which built-in React hooks are you familiar with? \*

	Not familiar at all	Somewhat familiar	Familiar
useState	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useEffect	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useContext	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useReducer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useCallback	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useMemo	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useRef	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useImperativeHandle	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useLayoutEffect	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
useDebugValue	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

## Interview Guide

- Personal history with JavaScript and React
- React and React architecture in general
  - Overall impressions of React
  - Is component and props -based architecture of React suitable for you?
  - Future vision of React; is React moving in the right direction?
- Hooks-based state management
  - Usability of hooks as a state management tool
  - Transition to 100% function components? Or do Class components have a place in modern React
  - Built-in hooks, custom hooks, and 3<sup>rd</sup> party custom hooks; your experiences and opinions
  - Different pragmatic aspects of hooks (readability, collaboration, ecosystem, testability, learning, documentation)
- Global state management
  - Choosing your state management library and architecture
  - Viability of local state sharing with context
  - How do you handle the jungle of state management options?
- Adapting new technologies
  - How quickly did you adapt to hooks?
  - What goes into your thought process of choosing a framework, library, etc. for a project?
  - Long-term viability of current JavaScript ecosystem
  - Fast-moving JS: value of refactoring, maintaining, and adapting new technologies during development.
- Any questions or comments?
- Do you know anyone who could participate in this interview?