# FFI-RAPPORT

20/00354

# webSAF

## – front-end architecture and design

Madeleine Josephine Selvig Hallén
Kristian Selvaag

# webSAF
# – front-end architecture and design

Madeleine Josephine Selvig Hallén
Kristian Selvaag

Norwegian Defence Research Establishment (FFI) 10 February 2020

**Keywords**
Modellering og simulering
Grafisk brukergrensesnitt
Webteknologi
Simuleringsverktøy
Stridssimuleringer
Datagenererte styrker

**FFI report**
20/00354

**Project number**
1508

**Electronic ISBN**
978-82-464-3138-3

**Approvers**
Dan Helge Bentsen, *Research Manager*
Halvor Ajer, *Director of Research*

*The document is electronically approved and therefore has no handwritten signature.*

# Summary

webSAF is an easy-to-use, web-based tool for simulation-supported, two-sided wargaming. It consists of a server that communicates with a graphical user interface (GUI) in the browser using WebSocket. It currently has functionality for controlling indirect fire entities, maneuver entities and air defence entities. webSAF is designed to be independent of the simulation tools in use, and can be used to control entities in a federation of different simulation tools.

The web-based user interface (UI) is written in TypeScript, a superset of JavaScript, and is based on a selection of web technologies, the most important being React, Redux and OpenLayers. This report describes the technologies, decisions that were made and experiences related to development of the GUI.

# Sammendrag

webSAF er et brukervennlig, web-basert verktøy for simuleringsstøttede, tosidige krigsspill. Det består av en server som kommuniserer med et grafisk brukergrensesnitt i nettleseren ved hjelp av WebSocket, og har foreløpig funksjonalitet for å kontrollere indirekte ild-entiteter, manøverentiteter og luftvernentiteter. webSAF er designet for å være uavhengig av hvilke simuleringsverktøy som benyttes, og kan brukes til å kontrollere entiteter i en føderasjon av ulike simuleringsverktøy.

Det web-baserte brukergrensesnittet er skrevet i TypeScript, et supersett av JavaScript, og er basert på en rekke ulike webteknologier hvor de viktigste er React, Redux og OpenLayers. Denne rapporten beskriver teknologiene, beslutninger som er tatt underveis og erfaringene knyttet til utvikling av brukergrensesnittet.

# Contents

# 1 Introduction

webSAF is an easy-to-use, web-based tool for simulation-supported, two-sided wargaming. It consists of a server that communicates with a graphical user interface (GUI) in the browser using WebSocket [1]. It currently has functionality for controlling indirect fire entities, maneuver entities and air defence entities. webSAF is in principle independent of the simulation tools in use, and can be used to control entities in a federation of different simulation tools. At FFI we have mainly used webSAF with Virtual Battlespace (VBS), but air defence units are simulated in VR-Forces.

The webSAF GUI is based on a handful of different web technologies such as React, Redux, OpenLayers and more. React was launched in 2013, and was still in its infancy when the webSAF project started in 2016. Its popularity has grown greatly, and it was both the most loved and most wanted web framework in Stack Overflow's annual survey for 2019 [2]. OpenLayers was at version 3 in 2016 and still lacking some features. Development of OpenLayers has progressed together with webSAF, and new features have been implemented continuously.

This report describes the technologies used in the webSAF user interface (UI) and how they were used. In addition, we try to give some insight to some of our experiences using these technologies. For more information about webSAF, see "webSAF – An easy-to-use, web-based graphical user interface for controlling semi-automated forces" [3].

# 2 Game design

The webSAF application is inspired by strategy games. Interacting with webSAF is much like using Google maps or any other sliding maps application. A map fills the entire background, with unit symbols and other graphics overlaid on top of it. The player selects a unit by left clicking it either in the map or in the *Order of Battle (OOB)*, and right clicks in the map or on the unit in order to make that unit perform actions such as *Move*, *Assault* or *Mount*. Some actions require more clicks to perform, like selecting areas to defend or setting target type and effect when requesting fire support. Notifications appear in the top right corner when something of significance occurs, for example when units encounter enemy forces. The unit symbol changes depending on the operational status of the unit. *Not operational* and *Annihilated* units are indicated by a yellow and red bar, respectively. Because webSAF contains quite a bit of functionality, this report will not go into detail on how to actually use webSAF. However, a user guide is available that explains the gameplay, and is updated along with changes to webSAF. In addition, the FFI-report "webSAF – An easy-to-use, web-based graphical user interface for

controlling semi-aumtomated forces” [3] addresses all aspects of webSAF, not only the front-end.

The game design has undergone continuous improvement in order to make webSAF better and more user friendly. Figure 2.1 shows webSAF at the time of writing.


*Figure 2.1 webSAF at the time of writing. An infantry squad is selected.*

In the original design, the player could select several units simultaneously, and any order they were given would go to the subset of selected units that were able to perform this order. The multi-unit selection design was inspired by real-time-strategy games, where most units are limited to either “move” or “attack”. During the course of development, the list of possible orders grew, and most of them had several parameters that needed to be set, making them much more complex than simple “move” or “attack” orders. At the same time, predicting which units would be doing what after an order was given to an arbitrary selection of units was not always easy for the player. The design was therefore changed to single unit selection. The list of available orders is determined from the type of unit selected, and the execution is predictable: if a command is given to the selected unit, then that unit as a whole will execute the command. Graphics in the map indicate what command the unit is currently performing.

In early webSAF versions, only one command could be issued to a unit at a time, and the player had to wait for the unit to finish that command before giving it a new order. But as development continued, it was clear that it would be a big advantage for the player to be able to queue orders.

Thus, the order queue was implemented. During the course of further development, additional improvements were made, such as highlighting the corresponding order graphics in the map, making it easy to delete orders in the queue.

The status summary for the selected unit is another example of the iteration-driven design approach. The original idea was to right click a unit, and request “unit information”. This was cumbersome and required several clicks to request along with a round-trip to the server to receive, so the idea was dropped in favor of the status summary: whenever a unit is selected, its status summary is displayed in the bottom right corner, with information about child units in the bottom left corner. However, the method of right clicking a unit and requesting information from the server is used when requesting logistics information.

# 3 User interface

While some toolkits for creating cross-platform desktop GUIs offer some viable options, such as Qt [4] and GTK+ [5], browsers have grown powerful and web-based applications have one major advantage: the ease of distribution. In particular, no installing of software is required; an advantage that is important in a high-security network environment typical to military research where installing new software can be problematic. The application web page can instead be served on the organization’s intranet, making it accessible via a regular browser.

webSAF is not FFI’s first attempt at creating a map-based web-GUI: Simulation Supported Wargaming for Analysis of Plans (SWAP) is a demonstrator system that aims to show how simulation can enhance the military decision making process [6]. The focus of that project was on fuel consumption, routes and timing in the planning of force movement. The technology used in SWAP is similar to that of webSAF; the front end is web-based with a map as the central element. webSAF owes some of its design decisions to the experiences of the SWAP team. However, SWAP is considerably different from webSAF, meaning that building on SWAP would have been inefficient. Controlling forces on both red and blue side, for example, is not considered in the design of SWAP.

# 4 Design principles

The GUI design philosophy for webSAF is based on balancing the following aspects, roughly prioritized in descending order:

1. *Time efficient user interaction*. The user should be able to do a lot, fast.

2. *Familiar user interaction*. The system should be easy to use.

3. *Ease of implementation*. The work needs to be completed within time and budget constraints.

4. *Ease of maintenance and extension*. The project is anticipated to expand in the future. The software architecture therefore needs to be amenable to changes and extensions.

5. *Aesthetics*. Visual style should not hinder adoption by users.

One of the main drivers behind this work was to save operator time during constructive simulation, in order to reduce the number of operators required. The GUI was therefore designed to help the operator reach his or her goals with the least amount of manual effort. Operator efficiency takes precedence over other design aspects.

Familiarity was the driving aspect for other design decisions. The “sliding maps” way of panning and zooming in and out in the map view, for example, is very similar to widely used online map services. The right-click context menu is also a way of minimizing UI clutter, while still keeping useful interactions just a click away. The context menu is complemented by a few static menus, mainly the order of battle and the notifications. The user has the option to hide these, and when not hidden, they are kept to the sides as to not clutter the map. Apart from the thin bars at the top and bottom of the screen, the map fills the entire GUI area. Menus and available interactions appear on-screen when an action fires, for instance the logistics menu upon requesting logistics information. The disadvantage for a new user is that it is not immediately obvious what interactions are available. On the other hand, it can be overwhelming to see all the available options simultaneously.

Another aspect that might discourage new users is poor aesthetics. However, aesthetics is rarely prioritized in internal software projects, and slow adoption of new software and routines is a persistent problem across organizations [7]. Thus, the visual design of the GUI was given some consideration. Perceived performance is closely linked to aesthetics: Slow user interfaces can fatigue users. Performance tuning has not yet been given significant development time, but it was considered during software architecture design and when selecting external software libraries.

# 5 Technology stack

The webSAF project started in late 2016. This is important to keep in mind because web technologies evolve and change so fast. Back in 2016, the biggest challenge for the project was considered to be the data flow from the web-GUI to the underlying simulation engine, Virtual BattleSpace (VBS) [8], and back to the GUI. For this reason, a quick prototype was made using pure JavaScript and HTML together with OpenLayers 3 [9] and WebSocket [1]. After successfully establishing communication between the web-GUI prototype and VBS, work began on designing the front-end so that it would achieve the envisioned features while adhering to the design principles.

Experience from developing the prototype made it apparent that better tools were needed to create an application of the envisioned scale: code splitting and dependency management, for example, was not solved gracefully on the prototype web platform. Because scripts are included in the HTML document sequentially using the *script* tag, each script adds functions or variables to the global namespace. If one script depends on a function or variable from another script, the developer has to make sure to include them in the correct order. A possible disaster scenario is if a script redefines a variable declared by another, creating errors that are hard to debug.

There was also the problem of browser support for standardized JavaScript (ECMAScript [10]). The ECMAScript standardization process has resulted in quality-of-life improvements to the JavaScript language, like lambda-syntax and a module system, but web browser support has been lagging behind. In 2016, the latest edition of the language with consistent support across browsers was ECMAScript 5$^{th}$ edition (ES5) [11]. To avoid having to write browser specific code, web developers resort to *polyfills*, libraries that implement the quality-of-life improvements that ES5 lacks, or compilers that compile newer JavaScript to ES5 (these compilers are also known as *transpilers*). There are many web development challenges to overcome, but there are even more solutions to choose between. The most important tools used in webSAF are presented in this chapter.

## 5.1 Package manager

webSAF uses Node.js [12] and Node Package Manager (npm) [13]. Node.js is a cross-platform JavaScript runtime built on Chrome's V8 JavaScript Engine for executing outside the browser (i.e. as a traditional scripting language). It is designed to build scalable applications, and through npm, it provides access to a massive registry of JavaScript code packages. Npm makes development easier because hundreds of thousands pieces of code is shared within the community. Developers can thereby avoid rewriting basic components, libraries or frameworks. Each piece of code may in turn depend on other pieces of code, and these dependencies are managed by package managers. Examples of packages used in webSAF include Moment, Redux, MilSymbol and many, many more.

## 5.2 Build system

A typical website project setup using npm consists of script source files in a language that compiles to ES5, along with resource files such as cascading style sheets (CSS), images and an *index.html* file. A *build* process is required, in which all files (JavaScript files, image files and other assets) are compiled and bundled into one huge file that actually gets served to visitors of the website.

Several build systems (or task runners) were available in 2016: Grunt [14], Gulp [15], Webpack [16], Browserify [17] among others. The choice fell on Webpack because it seemed easy to set up. It was quite new, but there were plenty of tutorials, and several project templates were using Webpack already. Webpack also provides a development server that continuously serves the latest version of the web page at all times; whenever changes are made to the source code (i.e. whenever you save your changes), the development server triggers a page refresh in the web browser.

## 5.3 Choice of user interface library

The main purpose of JavaScript on a web page is to interact with the Document Object Model (DOM), the tree structure that represents the visual layout of a web page. The standard DOM-API is not to everyone's liking, and so a number of libraries and so-called "web frameworks" have emerged. Examples include jQuery [18], Angular [19], React [20], Vue [21] and many others. These libraries promise familiar design patters, ease of development and/or performance improvements. Three user interface libraries were considered for webSAF: Angular, Vue and React.

Angular is a platform and framework for building client applications in HTML and TypeScript [22]. It is open source under the MIT license and maintained by Google, and in late 2016, version 2 had just arrived. An Angular application is made up of modules, components and services. Modules provide a context for a collection of components, while components correspond to views, the things you actually see in the browser. Components depend on services to provide application logic that is not directly related to the view. A component typically consists of an HTML template, a CSS style sheet and a TypeScript file that specifies component lifecycle methods and component metadata. The HTML template is extended with *directives* and *binding markup*. Directives are special script-like labels that provide simple logic. A directive can be invoked in the template either as a tag, an attribute, in a comment or by passing it to the class-attribute. The *ng-repeat* directive, for example, repeats a tag once for each item in a collection. Binding is a mechanism that keep the changes in a component's variables and the UI in synch.

React on the other hand, is focused purely on the view part of a web-application. It is described by its developers as *a JavaScript library for building user interfaces* [23]. React is maintained by Facebook. React is all about *components*, which are roughly analogous to Angular-components, with one major difference: the HTML template and component logic are mixed in

one JSX (JavaScript XML) file, or TSX file when TypeScript is used. This may sound like a violation of the principle of separation of concerns, but on the other hand, most HTML templates contain logical directives (in HTML), which could be much more elegantly handled by a proper scripting language. In JSX syntax, React element tags can be used mostly like other variables in JavaScript.

Vue, like Angular, uses HTML templates with logical directives, but is less verbose in the setup of components. It is marketed as a *progressive framework* for building user interfaces, meaning that it is easy to adopt incrementally into an existing web application [24]. Vue also boasts impressive optimizations and low memory footprint, and in general looked like a good alternative. The only downside back in 2016 was limited support for TypeScript. TypeScript support has improved since then, with official type declarations and the release of Vue version 2.5 in October 2017 [25].

Table 5.1 shows a comparison between the different interface libraries.

| | **Angular** | **React** | **Vue** |
|---|---|---|---|
| Scope | Full MVC framework. Rendering, state, routing, controllers, models etc. | Rendering and state. | Rendering and state. |
| Templates | HTML templates with script-like *directives*. | JavaScript extended with HTML-like tags. | HTML templates with script-like attributes. |
| Data binding | Two-way. | One-way (down). Uses callbacks for propagating input. | Two-way. |
| TypeScript integration | Native. | Good via typing/TSX. | Somewhat lacking (in 2016). |

*Table 5.1 Comparison of user interface libraries Angular, React and Vue.*

Between Vue's limited TypeScript support at the beginning of the project, and a stylistic preference for React's HTML-in-JavaScript approach, React was chosen as the tool for creating the user interface. Because React was chosen as user interface library, the next section establishes some terms and concepts from the React library.

## 5.4 React

React is a JavaScript library for building user interfaces [20]. From its launch in 2013 it has grown immensely in popularity. It is used by large apps such as Facebook, Netflix and Instagram. According to Stack Overflow's Developer Survey for 2019, React was both the most loved and the most wanted web framework by developers. The Developer Survey is the largest and most comprehensive survey of coders from around the world with nearly 90 000 developers answering the 20-minute survey in 2019. This year React also passed Angular for the first time when asked which framework web developers actually use [2]. The following sections provide a

review of React and important terms related to it. However, to ease the reader's burden, Table 5.2 lists a set of commonly used terms related to React.

| Term | Description |
|---|---|
| *Component* | A class that inherits the *react component* base class or a pure *rendering* function that maps an arbitrary *props* argument to a *react node*. It can also be a function that uses *Hooks*. |
| *Props* | A component's props are parameters that is passed to it in the parent's *render* function or method. |
| *State* | A component's state is data that can change, independently of the parent component, through the *setState* method, or using *Hooks*. |
| *React component base class* | A class that is built into React. It contains lifecycle methods, utility methods like *setState*, and requires derived classes to implement the *render* function. In TypeScript it takes the types of *props* and *state* as type parameters. |
| *Class component* | A class that inherits the *react component* base class |
| *Functional component* | A function that maps an arbitrary *props* argument to a *react node. Functional components* can also use *Hooks* to access React state and use lifecycle features. |
| *Event callback* | A function that is passed as props to a JSX element. It is typically bound as an HTML event listener, and calls the *setState* method in the callback function body. |
| *React node* | Any data type that can be converted to HTML: A JSX element, a string or a number or *null*. |
| *JSX element* | An inline XML tag that represents an HTML element or a React component. |
| *React hooks* | Functions that "hook into" React state and lifecycle features from function components. |

*Table 5.2 Commonly used terms related to React.*

### 5.4.1 React components

A React app is made up of a hierarchy of React components. The hierarchy is not an inheritance hierarchy, nor is it an aggregation or composition hierarchy from object oriented design theory, but rather a *rendering*-hierarchy. Component A is the "parent" of component B if component A *renders* component B. React components are created either by writing a class that inherits the *react component* base class, or through a pure[1] *rendering* function that maps an arbitrary *props* argument to a *react node* [26]. Simple react components can be seen in Figure 5.1.

[1] A pure function is a function that, for the same input, always produces the same output. In addition, a pure function cannot produce side-effects such as mutating a variable outside its scope (including the input variable), or performing input or output actions.

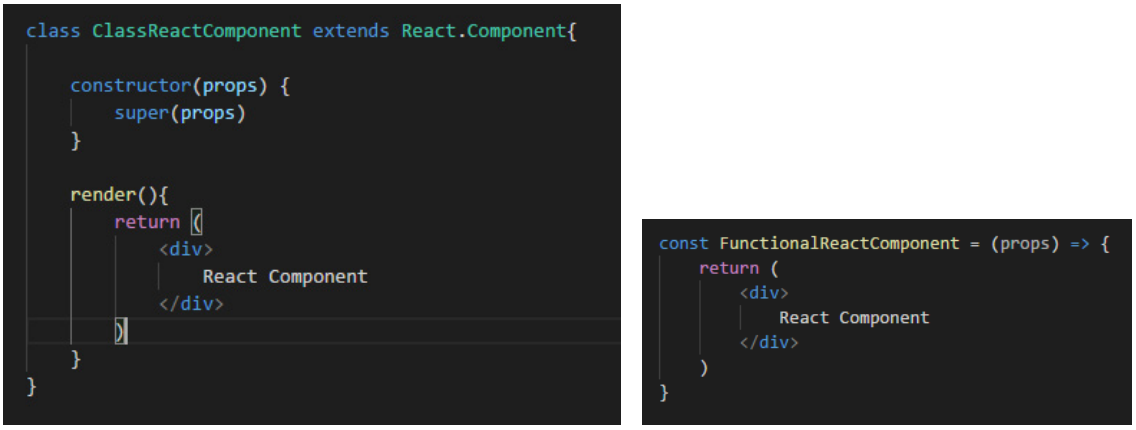



*Figure 5.1 Left: React class component. Right: React functional component.*

*Props* (short for properties) are parameters that are needed to render a component, and is analogous to *attributes* of normal HTML tags. In the example in Figure 5.1, none of the components are actually using props, as both only return some HTML. Note also that while a constructor is not required in a React class component, the render method is. The render method and other lifecycle methods are described more thoroughly in section 5.4.3.

### 5.4.2 State

React components have a state object where property values that belong to the component are stored [27]. A state object can change independently of parent components. Every time the state object changes, the component re-renders. The state object is initialized in the constructor using *this.state* as can be seen in Figure 5.2. State is changed through the *setState* method. This method is inherited from the *react component* base class and takes either a new state, or a function that maps the old state to the new, as argument, and merges the old state with the new. The *handleClick* method in Figure 5.2 is a custom method that handles clicking the button: each time the button is clicked, the state is changed, triggering a re-rendering of the component.

In TypeScript, the *react component* base class takes the types of props and state as type parameters. The type system, paired with a good editor, will show errors and diagnostics within the editor if the wrong types of props are passed to the component, or the *setState* method is called with the wrong argument.

```
class Counter extends React.Component{

    constructor(props) {
        super(props);

        this.state = {
            count = 0
        };
    }

    handleClick(){
        this.setState(({ count }) => ({
            count: count + 1
        }));
    }

    render(){
        return (
            <div>
                <p>Count: {this.state.count}</p>
                <button onClick={() => this.handleClick()}>Click me!</button>
            </div>
        );
    }
}
```

*Figure 5.2 An example of a React component for a simple counter.*

In February 2019, the React team released version 16.8.0. New to this version was the introduction of *Hooks* [28]. Hooks make it possible to use state and other React features without having to write classes [29]. Because React classes are quite verbose, and also not optimal for compilers, Hooks are a great addition. Figure 5.3 shows the same counter component as in Figure 5.2, but using the *useState* hook.

```
const CounterFunc = () => {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {this.state.count}</p>
            <button onClick={() => setCount(count + 1)}>Click me!</button>
        </div>
    )
}
```

*Figure 5.3 Same component as in Figure 5.2, but using hooks instead.*

Hooks are currently not widely used in webSAF because it is quite new, but some components do utilize Hooks. This works very well and it is certainly something to use more in the future.

### 5.4.3 Lifecycle methods

The *render* method is the only required method in a React class component. It is a description of what you want to see on the screen [27]. It outputs a R*eact element*, whether it is a string, number, JSX or anything that can be converted to HTML. This means that a component can be rendered by another component by returning it as JSX from the *render* method, where the attributes of the JSX tag are the props of the rendered component. The render method should be pure, meaning that it does not modify component state and returns the same result each time it is invoked.

The *render* method is a *lifecycle method*. Lifecycle methods can be overridden to run code at a particular time during the lifecycle of a component. When a component is mounted, that is, when it is rendered for the first time, some important lifecycle methods are:

- constructor()
- render()
- componentDidMount()

The *componentDidMount* method is a good place to use for example the *setInterval* window method. Because *setInterval* will continue calling a function or evaluate an expression with a given time interval until *clearInterval* is called, it is important to remember to perform any clean up tasks as well. The *componentWillUnmount* method is the only lifecycle method called right before the component is removed from the DOM, and is a good place to perform such clean up tasks.

As previously stated, a component re-renders when changes are made to its state or props. Important lifecycle methods during re-render is:

- render()
- componentDidUpdate()

There are more lifecycle methods, but these are rarely used.

## 5.5 JSX

JSX is a syntax extension to JavaScript [31] and is used extensively throughout webSAF. JSX produces React elements. It is basically just “syntactic sugar” for the *React.createElement(component, props, ...children)* function. Because JSX compiles to a React function, the React library must be in scope when using JSX. The function or class you are calling using the JSX tag must of course also be in scope. Let us assume that we use the *CounterFunc* component from Figure 5.3 inside another React component. This is as easy as including a <CounterFunc /> tag as can be seen from Figure 5.4. If our counter component had

needed any *props*, those would be passed down to the CounterFunc component as parameters. It is important to note that a component have to start with a capital letter in order to use it in JSX.

```
const App = () => {
    return (
        <div>
            <h1>This is my Counter</h1>
            <CounterFunc />
        </div>
    )
}
```

*Figure 5.4 Using JSX in React is very easy.*

JSX elements and JavaScript can be nested within each other. Opening a tag switches the parsing context to “HTML mode”, while opening curly braces within the “HTML mode” switches the context back to JavaScript.

## 5.6 Compiling and type checking

Programming languages that compile to ES5 are abundant. An obvious choice was to use the newest ECMAScript standard together with a transpiler, but there were several options that offered even more convenience: TypeScript [32] and Flow [33], backed by tech-companies Microsoft and Facebook respectively. TypeScript is a typed superset of JavaScript, while Flow is a static typechecker for JavaScript. While TypeScript provides its own compiler, Flow does not. Hence, a separate compiler would be required as well when using Flow. Ultimately, the choice fell on TypeScript.

Figure 5.5 shows the Counter component from Figure 5.2 written in TypeScript. Even though it is more verbose, the type checking provides useful aid when debugging. Notice how we have added a *prop*, so we can set our own start value for the counter. If no start value is provided, the Counter component will have a default start value of 0.

```tsx
type CounterProps = {
    counterStartValue?: number
}

type CounterState = {
    count: number
}

class Counter extends React.Component<CounterProps, CounterState>{

    constructor(props: CounterProps) {
        super(props);

        this.state = {
            count: props.counterStartValue || 0
        };
    }

    handleClick(){
        this.setState(({ count }) => ({
            count: count + 1
        }));
    }

    render(){
        return (
            <div>
                <p>Count: {this.state.count}</p>
                <button onClick={() => this.handleClick()}>Click me!</button>
            </div>
        );
    }
}
```

*Figure 5.5 The Counter component from Figure 5.2 written in TypeScript.*

Experiences using TypeScript have been good in general, but some parts of React does not work that well with TypeScript . The code sometimes requires some additional adjustments such as typecasting that otherwise would be unnecessary as seen in Figure 5.6 [34].

```tsx
const timerStyle = {
    whiteSpace: "nowrap"
}  as React.CSSProperties;
```

*Figure 5.6 Example of additional casting required by TypeScript.*

## 5.7 Map Library

Two map libraries were considered at the beginning of the webSAF project: OpenLayers [35] and Leaflet [36]. The main difference being that OpenLayers focuses on a rich set of features out of the box, while Leaflet is smaller and adopts a plugin architecture. The map functionality envisioned for webSAF included the ability to draw and modify shapes, render icons, aggregate

icons as well as render several map overlays. Implementing these features would have been possible using either of the two libraries, but OpenLayers was chosen because of its all-in-one philosophy. Minimizing the number of external plugins is convenient and potentially safer when developing in a closed network environment.

However, working with OpenLayers has been challenging at times. Version 5.0.0, released in June 2018, contained a completely reworked library, now under a new npm package name (*ol* instead of *openlayers*). This meant that in order to keep up with newer versions, the old *openlayers* package had to be uninstalled, and the new *ol* module installed. Additionaly, quite a lot of code had to be rewritten because of the changes. The new library has a better developer experience, but it took some time to get everything up-to-date.

There has also been issues with some functionality changes in OpenLayers. At one point, the support for *mousedown*, *mouseup* and *mousemove* events was lost in OpenLayers [37]. This led to the UI losing some functionality without any information as to why. After some debugging, the issue was fixed in the UI by using *pointer* events, but as of September 2019, OpenLayers still did not support the aforementioned events. Updating OpenLayers to a newer version with breaking changes almost always produces errors. Fixing errors can be frustrating because OpenLayers provide very little feedback, if any, as to what is wrong. Sometimes it might be as little as setting a flag to “true” that had to be set to “false” in the previous version for the map area to render properly. Sometimes we have gone into the OpenLayers source code and written our own error handlers in order to figure out where something is breaking.

Another issue with OpenLayers is that the documentation is somewhat lacking. For instance, we wanted a function that offsets a position on the map in a given amount of meters in a given direction. This functionality exists in OpenLayers, but it is not mentioned in their documentation (in October 2019). However, it was in the source code.

A fourth issue with OpenLayers is performance optimization. If the framerate drops, it can be hard to figure out why. One layer was dropping the framerate drastically, and it only contained a polygon with four edges. Why this simple geometry caused this large drop in frame rate is unknown, but the issue was fixed by making lines instead of a polygon.

Because the UI has been seeing a drop in performance due to OpenLayers styling, some tips for improving performance are [38][39]:

- Cache entire icon styles in order to reuse them. Then use different style *set* methods such as *setText*, *setFill*, *setStroke* etc.

- Use *layer style functions* whenever possible instead of storing the style on the feature, because feature styles consume quite a bit of memory.

- If a layer is causing performance issues, and a lot of features are added using the *addFeature* method, try creating an empty array of features, and push the features to it. Then use the *setSource* method and build a new source that renders all the features as seen in Figure 5.7.

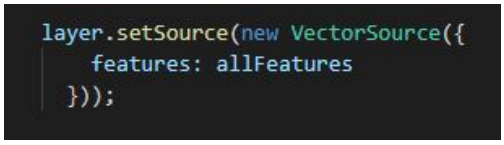



*Figure 5.7 Here, layer is an OpenLayers vector layer, and VectorSource is an OpenLayers vector source. allFeatures is an array of OpenLayers features.*

## 5.8 Symbols and icons

The *Milsymbol* library was used for rendering military map icons [40]. It enables the creation of SVG (Scalable Vector Graphics) symbols from MIL-STD-2525C [41] and MIL-STD-2525D-codes. The Milsymbol library makes the interchange between the different codes very easy as it handles both. The first consists of letters while the second of numbers. This means that the GUI has to do some checks when rendering the different symbols for *Not operational* and *Annihilated* units, but Milsymbol handles the rest. Other graphics were created using Adobe Illustrator, such as icons for ruler, compass, map etc.

## 5.9 Client-server communication

The web client sends updates to, and receives updates from, the simulation server. Updates must be frequent, so that the operator has a real-time picture of what happens on the battlefield. Many of the updates are also server driven, i.e. simulation updates that need to be sent from the server to the client even though there is no operator input. The standard Representational State Transfer (REST) architecture for accessing web resources was not suitable for the frequent and push-driven updates needed for webSAF [42]. WebSocket, on the other hand, is better with high loads and is bi-directional. WebSocket is a communication protocol that provides an interface to create a two-way TCP (Transmission Control Protocol) connection between the web client and simulation server [43]. This connection was easy to set up and fast enough to send data with high frequency.

# 6 Software architecture

The front-end architecture was to a large degree inspired by the libraries employed: UI-applications are not something new and plenty architectures have been proposed and deployed over the years. In this case, with a specific platform (the web), and a handful of libraries to choose from, it was only natural to adopt an architecture that suited the available tools.

## 6.1 Event system

The overall GUI system can be divided roughly into three areas of responsibility: the menus, the map, and synchronizing state by communicating with the server. Each of these areas is tightly coupled to the respective libraries chosen to cover them: React, OpenLayers and WebSocket. React is rooted in the functional paradigm, while OpenLayers, and WebSockets expose a typical object-oriented API. The initial architecture was inspired by the object-oriented approach, and was based around an event system as seen in Figure 6.1. If a menu needed to know when a shape was edited in the map, the map would be responsible for broadcasting a *ShapeUpdate* event, and the menu would subscribe to that event with a callback function to handle the *ShapeUpdate* event. In this way, the different modules; menus, map and WebSocket client all knew about the event system, but they did not know about each other.

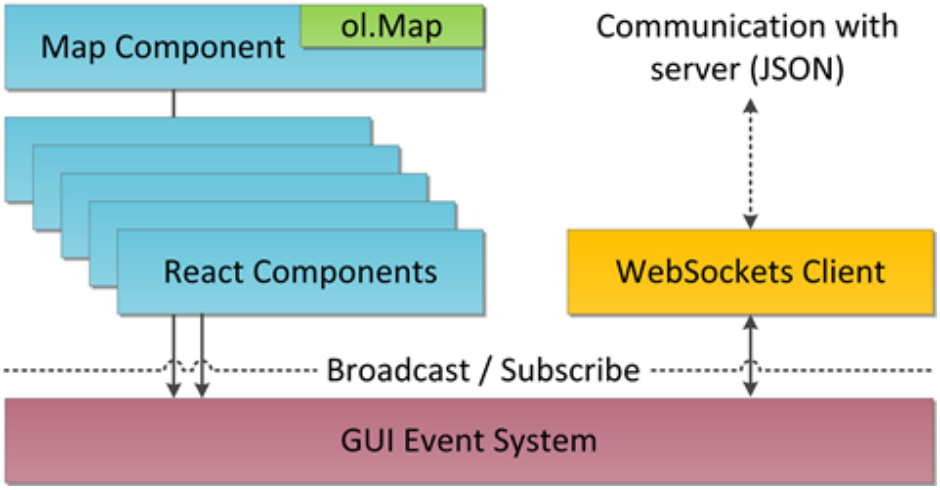



*Figure 6.1 Event system architecture for WebSAF. React components communicate with the server via a WebSockets client through the event system.*

The event system was easy to integrate and worked well for a while, but shortcomings soon became evident. Events in themselves represent *changes* in the application. Menus and maps, however, do not render *changes*, they render *state*; the data that is the result of changes accumulated over time. The map and the menu each had to keep track of its own state. Whenever a change occurred, an event had to be explicitly broadcasted. If the state of the map was changed, and an event is not broadcasted, the state of the menu would be out of sync with the state of the map, and vice versa: if something changed in the menu, and the map was

updated, state would again be out of sync. This means that for every interaction that is added to a menu (or to the map), the programmer has to remember to *explicitly* broadcast events, or things will go wrong. Combine this burden with the fact that a lot of the state is actually shared between the map and menu components (the order of battle for example), and the merits of an event-driven approach look weak.

## 6.2 The reactive approach

Reactive UI programming [44], the paradigm that underpins React, offers a compelling alternative to the event architecture outlined in the previous section. There is no authoritative definition of reactive programming, but the spirit of it is roughly that the UI is a function of the state. Whenever the state changes, that function is re-evaluated. Interaction with the UI produces events, or triggers callbacks, which in turn change the state and causes the UI to update. The challenge in this kind of architecture is to connect subsections of the state to subsections of the UI, so that everything does not have to be recalculated every time something changes in the state. React solves this by only making state changes possible through a component's *setState* method discussed in section 5.4.2. The state lives inside each component. If two components need to share state, the state is typically contained inside a component higher up in the hierarchy[2], and then it is passed as *props* to the subcomponents as Figure 6.2 shows. Input is handled by callbacks that are defined in the same component as the state they modify. These callbacks are then passed to subcomponents through props, where they are called. The concept is often referred to as *Flux.* It is an architectural pattern for controlling and managing the state of an application.

[2] In React, hierarchy refers to data flow, not an object oriented inheritance-hierarchy. If a component A returns a component B from its render function, then component A is higher up in the hierarchy than component B.

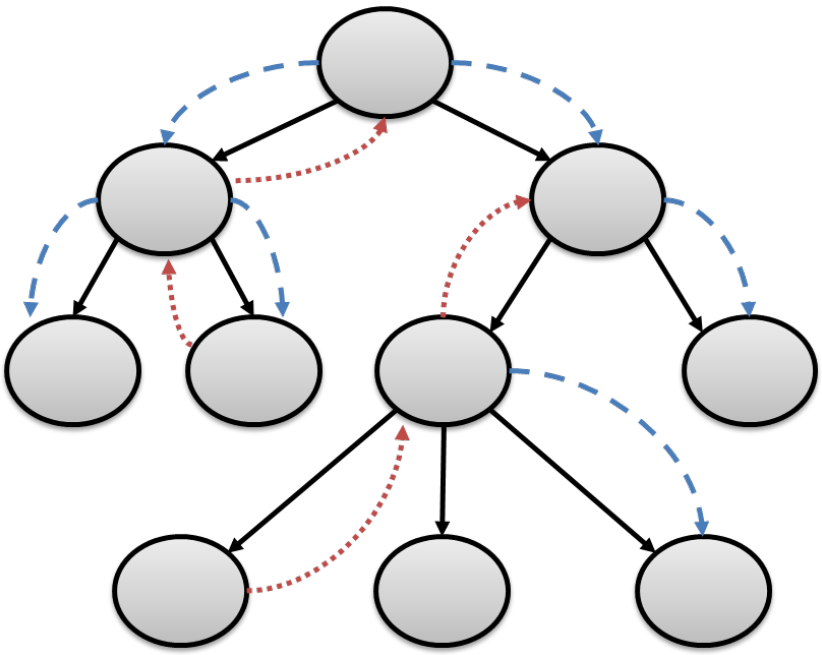


*Figure 6.2 Architecture of a React application. The ovals represent React-components, and the arrows represent the relationship between them. The solid arrows (black) represent the "renders" relationship. The downwards-pointing dashed lines (blue) represent the passage of props. The upwards-pointing dashed lines (red) indicate that the component calls the callback functions it is passed through props.*

A large part of the state in webSAF is shared between the menus and the map. This means that this state must live high up in the component hierarchy, and be passed down to subcomponents. Let us call this *global state*. Threading the state through component props in this way can be cumbersome and verbose. However, the *Redux* library provides a way to connect components directly to the global state, independently of component hierarchy as illustrated in Figure 6.3 [45].

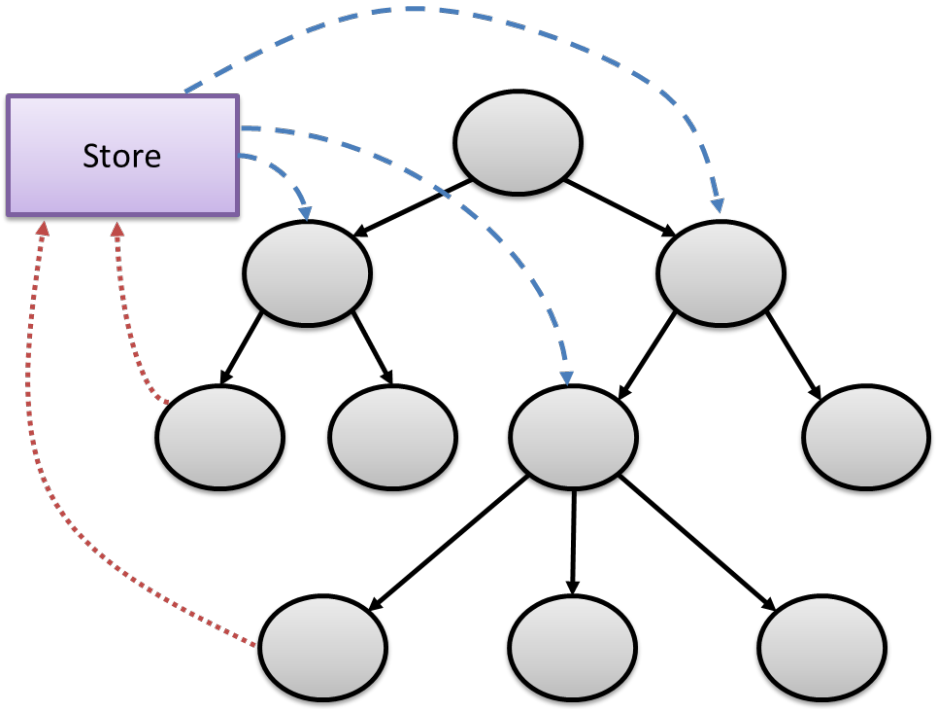



*Figure 6.3 Architecture of a Redux application. The ovals represent React-components that have been connected to the store. The solid arrows (black) represent the "renders" relationship. The downwards-pointing dashed lines (blue) represent the mapping from state to component props. The upwards-pointing dashed lines (red) indicate that the component dispatches actions to the store.*

Instead of passing state down as props to subcomponents, there is a central *store* for the global state. The React-Redux library is implemented in such a way that only UI components that are affected by the new state get updated. Using this binding library, each component specifies which parts of the global state they need to read through a *mapStateToProps* function [46]. Note that *mapStateToProps* has to be a function that returns an object. An example of this is seen in Figure 6.4. This example is from the *Logistics* component in webSAF, and listens to a part of the global state called "ammo".

```
const mapStateToProps = (state: GlobalAppState): LogisticsStateProps =>({
    ammoResponse: state.ammo
});
```

*Figure 6.4 Example of the mapStateToProps function.*

Input is sent directly to the central state store through a *mapDispatchToProps* function. Rather than using callbacks, input is handled by dispatching *actions*. The dispatcher passes actions to a *reducer*, a pure function that simply takes the action and previous state as input, and produces the next state as output. Figure 6.5 shows the *mapDistpatchToProps* function for the webSAF Logistics component. This particular component dispatches two actions: one that closes the

Logistics menu, and one that deletes ammo information from the state. These functions call a reducer that returns a new global state.

```
const mapDispatchToProps = (dispatch: (action: Action) => Action | void): LogisticsDispatch =>
{
    return {
        close: () => GlobalState.dispatchAction(CloseMenu(LogisticsMenuKey)),
        deleteAmmo: () => GlobalState.dispatchAction(LogisticsAction.DeleteAmmo())
    }
}
```

*Figure 6.5 Example of mapDistpatchToProps.*

The *connect* function is the final piece in a React-Redux component and is illustrated in Figure 6.6 by the connect function for the Logistics component in webSAF. This function connects a React component to the Redux store. It does not modify the component class that was passed to it, but returns a new, connected component class that wraps the component it was passed. Note that while *mapStateToProps* and *mapDispatchToProps* can be renamed however you wish, sticking with this naming convention ensures clarity and consistency.

```
export const LogisticsMenu = connect(
    mapStateToProps,
    mapDispatchToProps
)(Logistics)
```

*Figure 6.6 Example of the connect-function.*

The system architecture of webSAF using Redux is shown in Figure 6.7.

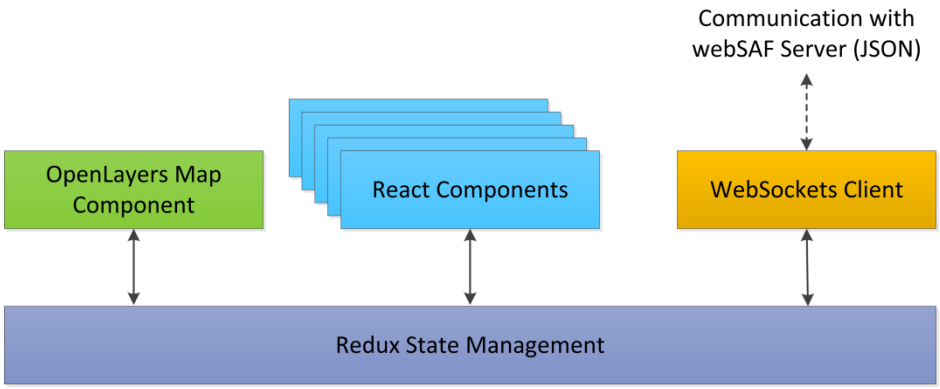



*Figure 6.7 System architecture for webSAF using Redux*

Redux menu components are defined by a render function: a pure function that maps *props* to React elements. Props could be anything, but is typically just an object containing the data that is relevant for displaying in the GUI component. The props are recalculated from the *mapStateToProps* function each time the state updates. Why is this function needed? Why could not just the component take the *state* as input directly? One way in which Redux determines

what parts of the UI needs to update, is by computing a component's *props* and compare these with the *props* computed for the *previous state*. If the *props* have changed, then the render function is re-evaluated with the new *props*. If the props have not changed since the last render, then there is no point in re-evaluating the rendering function: the input has not changed, which means the output has not changed either. We can make this assertion because the render function is pure.

There is no React-Redux equivalent for OpenLayers. The connection between the state store and the map is managed in a more manual way through events. The Redux *store* has the option of subscribing to state updates. Whenever the state has changed, the subscribed callback function is invoked with the new state as argument. This is how the map state is kept in sync with the overall state. The callback function itself checks which parts of the state has been updated and only updates the map as needed.

React and Redux work smoothly together, but it does require quite a lot of boilerplate[3] code. The new React-Redux version 7.1.0 released in June 2019 provides its own set of *hooks*, with methods like *useSelector* and *useDispatch* [47]. The *useSelector* function takes the place of *mapStateToProps*, allowing us to hook directly into the Redux store, or our global state. It takes the entire store as argument, and returns whatever part of the store we want to access, so we can save it as a variable in the component. The *useDispatch* hook will return a function that we can use to dispatch actions to the Redux store. These new methods are so far, in October 2019, rarely used in webSAF, but the reduction in boilerplate code makes it worth using in future components.

# 7 Discussion

The iteration driven test and development cycle has been fruitful for improving the features of webSAF. The test cycles also revealed inevitable bugs, which were for the most part fixed as soon as they were uncovered. The two causes of bugs that stood out in the beginning were corrupt or outdated state (typically in the map) and server-client communication mismatch. Straight-out crashes have been very rare, and the symptoms of the aforementioned bugs were usually that the GUI was displaying something unexpected, or displaying nothing where something was expected. TypeScript's type checking and the Redux architecture can take a lot of the credit for this relatively crash free development experience. Type checking eliminates the "type mismatch" category of bugs. Although TypeScript has fulfilled its purpose, Flow might have been a better choice when working with React because of how well the two work together. Because Flow is developed by the same people as React, it has advanced built-in support for

[3] Boilerplate code are sections of code that have to be included in many places with little to no alteration. Boilerplate is often used about verbose programming languages because developers must write a lot of code to do smaller jobs.

React. However, support for React in TypeScript is improving continually. Simultaneously, Redux goes a long way in mitigating state synchronization bugs by containing the state in a central store and automatically updating anything that depends on it. The aim of webSAF, however, was not to explore development methodologies, so there is no opportunity for comparison and it might have gone just as smoothly with other tools.

The libraries chosen have, for the most part, delivered what they promise. React is quite easy to learn. However, it is certainly an advantage to learn the basics of JavaScript before taking on React. The component approach helps organize the code and make code bits reusable. React is very much a library, and this can be both an advantage and a disadvantage, depending on your preferences. On one hand, although React has a lot of features, it does not provide everything needed for building an application. This means that if you want a framework that includes everything, React is not the way to go. On the other hand, because React is a library, you can choose which technologies you want to use, and easily include React in any project.

React has worked well as a UI library, but one downside to React is the one-way data flow – you can only pass *props* down to child components, but not up to parent components [30]. This can result in slightly verbose and cumbersome passing of callbacks through the hierarchy, from the parent component that contains the state, to the child component that receives the event that causes the state to update. However, state management using Redux solves this issue, even though the Redux approach is quite verbose as well. However, the new React and Redux Hooks might cut down boilerplate code substantially.

Integrating Redux in webSAF has been useful as it introduces structure and discipline into an application by forcing updates to go through the reducer function. On the other hand, when a new action is needed, there is a lot of code that needs to be edited in different places: the action needs to be defined and a sub-reducer function is needed to process it. That sub-reducer function needs to be connected to the main reducer, and the action needs to be dispatched from somewhere, typically a React component. If the action affects some new part of the state, then the state declaration has to be updated as well, and given a default value. This would not have been so bad if all state-synchronization bugs could be preemptively eliminated. But the fact that the application had already been in development for some time when Redux was introduced, meant that it was not trivial to recast all application interactions as Redux actions. The result is that the application state resides for the most part in the Redux store, but some is still maintained in some React components, and in the map. The benefits of Redux could possibly have been achieved with a more disciplined state management strategy using React only, by hoisting shared state into parent components, instead of relying on a global event system. There would, however, have been plenty of callback functions that needed to be passed through-out the component hierarchy. All in all, Redux has made state management easy.

OpenLayers version 3 was still in development at the time of the project start and was lacking some features, like animating the view. But the OpenLayers development has gone quickly in parallel with webSAF, and features have been integrated along with OpenLayers updates. Some issues still remain though. The performance issues in the UI have mainly been related to styling in OpenLayers. Suggestions on how to increase performance when styling features are outlined

in section 5.7. However, because of many issues related to OpenLayers, it might be preferable to try Leaflet in future map applications if it is sufficient for the project. If you on the other hand are developing a complicated Geographic Information System (GIS) application, OpenLayers is more suited.

# 8 Further work

webSAF is under continuous development. The need for new functionality is seen frequently, and features are added gradually. There is also the need for improvements in webSAF. Because of the continuous development of the webSAF UI, the code base is somewhat messy. As previously mentioned, webSAF started out with an event system, and as the complexity grew, Redux was implemented. This means that some parts of webSAF still rely on the event system, while other parts depend on Redux. It would be beneficial to have only one system for state management. While the flux architecture is fairly straightforward when working with React components, it gets more complicated when working with other JavaScript libraries such as OpenLayers. There are some guides online that outline different ways of using OpenLayers together with React and the flux application architecture [48], but it is uncertain how much this would improve the system. Cleaning up the code base will require a lot of work, and it might not be worth the time spent.

# 9 Conclusion

webSAF is an easy-to-use, web-based tool for simulation-supported, two-sided wargaming. webSAF consists of a server that communicates with a graphical user interface (GUI) in the browser. The technologies used in the webSAF GUI has delivered as promised for the most part. Experiences using React together with Redux have been very good. It makes state management easy, but does require a bit of boilerplate code. Writing the code in TypeScript has reduced errors during development and made crashes a minimal concern. The biggest development issues have been related to OpenLayers. Future map application projects should therefore consider Leaflet unless it is a very complex GIS application and the full extent of OpenLayers is required.

webSAF is still a work in progress, and improvements and new features are added continuously.

# References


[1] WebSocket, https://www.websocket.org/, accessed October 2019.

[2] Stack Overflow, Developer Survey Results 2019, https://insights.stackoverflow.com/survey/2019, accessed November 2019.

[3] Evensen, Per-Idar et al., *webSAF– An easy-to-use, web-based graphical user interface for controlling semi-automated forces*, Norwegian Defence Research Establishment report 19/01622, 2019.

[4] The Qt Company, https://www.qt.io/, accessed October 2019.

[5] The GTK+ Project, https://www.gtk.org/, accessed October 2019.

[6] S. Bruvoll *et al.*, *Simulation-supported Wargaming for Analysis of Plans (SWAP)*, Norwegian Defence Research Establishment 16/00524, 2016.

[7] B.H. Hall & B. Khan, *Adoption of New Technology*, National Bureau of Economic Research, Cambridge, Massachusetts, 2003.

[8] Bohemia Interactive Simulations, VBS3, https://bisimulations.com/products/virtual-battlespace, accessed October 2019.

[9] Openlayers 3, https://openlayers.org/en/v3.20.1/doc/, accessed October 2019.

[10] ECMA International, https://www.ecma-international.org/, accessed July 2018.

[11] ECMAScript compatibility table, https://kangax.github.io/compat-table/es5/, accessed July 2018.

[12] Node.js, https://nodejs.org/en/, accessed October 2019.

[13] Npm, Inc., https://www.npmjs.com/, accessed October 2019.

[14] Grunt, The JavaScript Task Runner, https://gruntjs.com/, accessed July, 2018.

[15] Gulp, https://gulpjs.com/, accessed July, 2018.

[16] Webpack, https://webpack.js.org/, accessed July, 2018.

[17] Browserify, http://browserify.org/, accessed July, 2018.

[18] jQuery, https://jquery.com/, accessed July 2018.

[19] Angular, https://angular.io/, accessed July 2018.

[20] React, https://reactjs.org/, accessed October 2019.

[21] Vue.js, https://vuejs.org/, accessed July 2018.

[22] Angular, Architecture Overview, https://angular.io/guide/architecture, accessed October 2019.

[23] React, Getting Started, https://reactjs.org/docs/getting-started.html, accessed October 2019.

[24] Vue.js, Introduction, https://vuejs.org/v2/guide/, accessed October 2019.

[25] Vue.js, TypeScript support, https://vuejs.org/v2/guide/typescript.html, accessed July, 2018.

[26] React, Components and Props, https://reactjs.org/docs/components-and-props.html, accessed October 2019.

[27] React, State and Lifecycle, https://reactjs.org/docs/state-and-lifecycle.html, accessed October 2019.

[28] Github React, V 16.8.0, https://github.com/facebook/react/releases/tag/v16.8.0, accessed October 2019.

[29] React, Introducing Hooks, https://reactjs.org/docs/hooks-intro.html, accessed October 2019.

[30] Chahal, Sakshi (May 26th 2019), "Passing Data Between React Components – Parent, Children, Siblings", https://towardsdatascience.com/passing-data-between-react-components-parent-children-siblings-a64f89e24ecf, accessed October 2019.

[31] React, Introducing JSX, https://reactjs.org/docs/introducing-jsx.html, accessed October 2019.

[32] TypeScript, https://www.typescriptlang.org/, accessed October 2019.

[33] Flow, https://flow.org/en/, accessed October 2019.

[34] Wan, Benny, "Comparing Flow with TypeScript", https://medium.com/the-web-tub/comparing-flow-with-typescript-6a8ff7fd4cbb, accessed October 2019.

[35] OpenLayers, https://openlayers.org/, accessed October 2019.

[36] Leaflet, https://leafletjs.com/, accessed October 2019.

[37] Mousedown, mouseup, mousemove events are missing in latest version (May 24th 2018), https://github.com/openlayers/openlayers/issues/8219, accessed October 2019.

[38] Github, "Performance dropping steeply by raising number of styled features", https://github.com/openlayers/openlayers/issues/8392, accessed October 2019.

[39] Github, "Efficiently rendering hundreds of unique vector map features", https://github.com/openlayers/openlayers/issues/8514, accessed October 2019.

[40] Spatial Illusions, Milsymbol, https://www.spatialillusions.com/milsymbol/index.html, accessed October 2019.

[41] Department of Defense, *MIL-STD-2525C Common Warfigthing Symbology*, 2008.

[42] Fielding, Roy, "Chapter 5: Representational State Transfer (REST)". *Architectural Styles and the Design of Network-based Software Architectures* (Ph.D.). University of California, Irvine, 2000.

[43] The WebSocket Protocol, https://tools.ietf.org/html/rfc6455, accessed October 2019.

[44] Patel, Keval (December 12th 2016), "What is Reactive Programming", https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382, accessed October 2019.

[45] Redux, https://redux.js.org/, accessed October 2019.

[46] React Redux, https://react-redux.js.org/, accessed October 2019.

[47] Gonzáles, Max (July 25th 2019), "Clean Up Redux Code With React-Redux Hooks", https://medium.com/swlh/clean-up-redux-code-with-react-redux-hooks-71587cfcf87a, accessed October 2019.

[48] Callsen, Taylor (May 13th 2017), "Using OpenLayers with React", https://taylor.callsen.me/using-reactflux-with-openlayers-3-and-other-third-party-libraries/, accessed October 2019.

# Abbreviations

| | |
|---|---|
| CSS | Cascading style sheets |
| DOM | Document Object Model |
| ES5 | ECMAScript 5 |
| GIS | Geographic Information System |
| GUI | Graphical User Interface |
| HTML | HyperText Markup Language |
| JSX | JavaScript XML |
| MGRS | Military Grid Reference System |
| npm | Node Package Manager |
| OOB | Order of Battle |
| REST | Representational State Transfer |
| SAF | Semi-Automated Forces |
| SVG | Scalable Vector Graphics |
| SWAP | Simulation Supported Wargaming for Analysis of Plans |
| TCP | Transmission Control Protocol |
| UI | User Interface |
| VBS | Virtual Battlespace |
| XML | Extensible Markup Language |

## About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

### FFI's MISSION

FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

### FFI's VISION

FFI turns knowledge and ideas into an efficient defence.

### FFI's CHARACTERISTICS

Creative, daring, broad-minded and responsible.

## Om FFI

Forsvarets forskningsinstitutt ble etablert 11. april 1946. Instituttet er organisert som et forvaltningsorgan med særskilte fullmakter underlagt Forsvarsdepartementet.

### FFIs FORMÅL

Forsvarets forskningsinstitutt er Forsvarets sentrale forskningsinstitusjon og har som formål å drive forskning og utvikling for Forsvarets behov. Videre er FFI rådgiver overfor Forsvarets strategiske ledelse. Spesielt skal instituttet følge opp trekk ved vitenskapelig og militærteknisk utvikling som kan påvirke forutsetningene for sikkerhetspolitikken eller forsvarsplanleggingen.

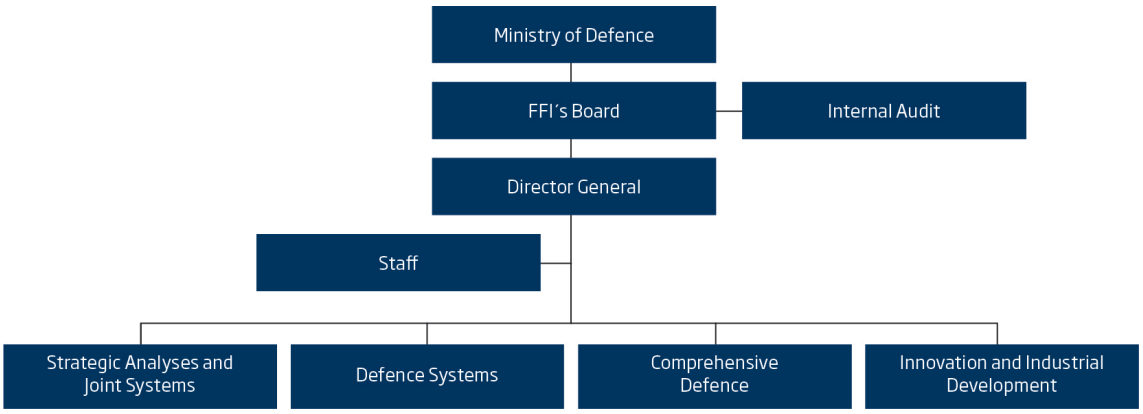
### FFIs VISJON

FFI gjør kunnskap og ideer til et effektivt forsvar.

### FFIs VERDIER

Skapende, drivende, vidsynt og ansvarlig.

## FFI's organisation



**Forsvarets forskningsinstitutt**
Postboks 25
2027 Kjeller

Besøksadresse:
Instituttveien 20
2007 Kjeller

Telefon: 63 80 70 00
Telefaks: 63 80 71 15
Epost: ffi@ffi.no

**Norwegian Defence Research Establishment (FFI)**
P.O. Box 25
NO-2027 Kjeller

Office address:
Instituttveien 20
N-2007 Kjeller

Telephone: +47 63 80 70 00
Telefax: +47 63 80 71 15
Email: ffi@ffi.no

FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment