Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Migrating Angular-based web apps to Web Components - A case study at 30MHz

Author: Sander Ronde (2639938)

1st supervisor: Ivano Malavolta
daily supervisor: Dara Dowd (30MHz)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

June 24, 2021

Abstract

Since 2018, a set of technologies together referred to as Web Components are supported in all major browsers. Web Components are a set of technologies that enable support for the creation of custom HTML elements, thereby allowing for encapsulation of functionally and semantically related code. In this respect, they are similar to JavaScript (JS) frameworks, with the exception that these created elements (also called Web Components) have no dependencies and do not require any additional code to function. As such, any component such as a button, checkbox, or switch written using Web Components functions regardless of the JS framework used on a given page. This contrasts with code written using a JS framework, which generally requires that framework to be loaded on the page.

There are various reasons for developers to migrate existing an existing set of components (generally referred to as a component library) from a JS framework to Web Components. In this paper, we present a case study performed at the software company 30MHz, tackling such a scenario for one of these JS frameworks, namely the migrating of a set of Angular components to Web Components. Angular is one of the more popular JS frameworks for building web applications. It also suffers from the previously mentioned issue of components written in Angular not being usable in other JS frameworks. The migration to Web Components presents a solution to this issue.

In evaluating the quality and performance of the resulting Web Components, we find the load time of the JS code to be roughly twice as long, with render times of individual components being about 5ms slower for a single component. The resulting render times remain competitive with the render times of various other component libraries. Additionally, we find indications that the impact of the performed case study on the maintainability of the codebase containing the source components is minimal. These findings together lead us to conclude that the migration of Angular components to Web Components is feasible.

This migration process presents a time-saving method for developers wishing to create a cross-framework component library based on an existing Angular component library.

Contents

List of Figures

List of Tables

1

Introduction

Web Components [1] are a set of technologies recently added to the web platform that allow for the definition of custom HTML elements. These custom HTML elements allows for encapsulation of functionally and semantically related code. In this purpose, these Web Components are similar to JavaScript (JS) frameworks such as ReactJS [2], Angular [3], Svelte [4], and Vue [5]. What separates Web Components from JS frameworks is the fact that Web Components are native to web browsers and do not require external code to function. As of 2018, Web Components are supported in all major browsers [6], marking the moment at which Web Components are supported across all platforms and browsers without any additional code being required. This places Web Components in a special position where any individual Web Component can be added to a web page with the guarantee that it will work. This intercompatibility allows developers to create a single Web Component ranging from simple components such as buttons and checkboxes to complex components such as charts and video players. This is in contrast to components that are created using a JS framework. These will generally only work if the component's framework is the same framework the web page uses. This lack of compatibility significantly reduces the pool of components that developers have access to, thereby reducing the community's ability to share code with each other in the form of components.

One solution to this problem is providing the ability to migrate components that have been written in a JS framework to Web Components. This effectively frees developers from

[1] `https://www.w3.org/TR/2013/WD-custom-elements-20130514/#about`
[2] `https://reactjs.org/`
[3] `https://angular.io/`
[4] `https://svelte.dev/`
[5] `https://vuejs.org/`
[6] `https://caniuse.com/?search=webcomponents`

the constraints of a JS framework and allows the created Web Components to be used anywhere. There are various reasons to migrate a component (or multiple components) to Web Components. One of which is the ability to have the same components be re-used across different teams that each use a different JS framework. Another reason is the ability to provide created components to the open-source community, thereby allowing other developers to make use of these components. In this paper, we present a case study targeting another reason for this migration process, namely the migration of a design library to Web Components for use by a third party. This reason is introduced below.

1.1 Reason for migrating to Web Components

Many companies apply a unified design language to their products. A design language is an overarching style that guides the design of products it applies to, generally being spread over a company's products. Its purpose is to provide users of products with a unique but consistent look and feel across all products. In order to maintain this design language across apps created on their platform, companies tend to provide third-party app developers with such a design language to use for their apps. Examples of such a design language being provided to developers are Google's Android [1], Apple's iOS [2], and Zendesk's Garden [3]. In order to aid developers in creating apps that use this design language, they are often provided with a set of basic components that follow this design language. Examples of UI components include buttons, inputs, layouts, and switches. Such a set of components is commonly called a UI library or design library. In addition to containing just UI-related components, these UI libraries can also contain components that focus on for example API access, accessibility, or analytics. Since the design language a company provides is generally applied to its own products as well, the overlap between its provided UI library and its internally used UI library is relatively large. As such, it would save a lot of time if the UI library that is provided to third parties can be generated from the internally used UI library (given that it can not be provided to developers as-is).

An example of such a scenario is the one that is present at 30MHz. 30MHz is a technology company in the agriculture industry looking to provide third parties with a UI library. To save time, both now and in future maintenance, it would be best to generate this UI library from the internally used UI library. This UI library is unable to be provided to third

[1] `https://material.io/`
[2] `https://developer.apple.com/design/`
[3] `https://garden.zendesk.com/`

parties as-is. Both because it is interwoven with the rest of the codebase (which should not become publicly available) and because it is written in the Angular JS framework.

Angular is a JS framework for building single-page web applications. Angular is one of many JS frameworks. A few of the most popular JS frameworks as of 2020 [1] include ReactJS, Vue, Svelte, and the previously mentioned Angular. As mentioned before, code written in one framework is generally not usable by other frameworks. They are essentially written in different programming languages. Locking third-party developers to a single JS framework (in this case, Angular) will provide for a worse development experience as they are given less freedom of choice. Since the popularity of other JS frameworks is increasing, this problem is likely to worsen over time, with developers preferring to use a different JS framework over Angular. To get around both the issue of the code being interwoven and the issue of the source code being written in Angular, a solution to these problems has to be devised.

In this thesis, we attempt to find this solution through a case study at 30MHz. Our approach is to migrate the existing Angular components to the previously mentioned Web Components [2]. Because of their being usable by every JS framework, Web Components provide a perfect target format for this UI library. In order to migrate the Angular components to Web Components, we use Angular Elements [3]. Angular Elements is a JS package that aides in the migration of Angular components to Web Components. After creating this Web Component UI library, we create wrappers for JS frameworks that do not natively support Web Components yet. Additionally, we generate documentation and individual component demo pages for developers to use.

In this paper, we describe this migration process and evaluate its effectiveness. This evaluation is done through the collection of various metrics. These are collected on both the original Angular components, the Web Components library, and the various wrappers, as well as a set of popular JS component libraries. We then compare the created Web Components library to the internal 30MHz UI library and other component libraries in the field, allowing us to assess the feasibility of this migration process.

The contributions of this paper are the following:

- We present a case study where we perform the migration of Angular components to Web Components, documenting issues faced along the way. These issues are likely to be faced in similar projects.

[1] `https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/`
[2] `https://www.w3.org/TR/2013/WD-custom-elements-20130514/#about`
[3] `https://angular.io/guide/elements`

- We present a checklist going over the steps required to perform this migration.
- We evaluate the quality and performance of the resulting Web Component UI library and its wrappers. In order to evaluate the impact of this migration process on performance, we compare the Web Component UI library with the original UI library. Additionally, we compare the Web Component UI library with other UI libraries in the field, allowing us to evaluate its relative performance and quality.
- We evaluate the feasibility of applying this migration process for businesses. Firstly we measure the time taken to perform this case study, getting an understanding of the cost of the case study. Secondly, we assess the impact on both the existing codebase and other developers we get a view of how disrupting this project is.
- We provide a GitHub repository that contains the code used for performing the measurements [1]. Additionally, it contains the resulting data, visualizations, and the code used to generate these visualizations.

[1] `https://github.com/sanderronde/master-thesis`

2

Background

This case study was performed at 30MHz, specifically within the context of their software platform. This section describes the company (30MHz), their software platform (the dashboard), and the problem solved by the case study.

2.1 The Company

30MHz is a technology company in the agriculture industry. They offer sensors that collect various types of data, all within the context of agriculture. Examples of types of data include temperature, humidity, and air pressure. 30MHz also provides their customers with a dashboard that allows them to view the collected data. An example of this dashboard can be seen in Figure 2.1. This dashboard is a web app that, as of this case study, is using Angular 10. Data is fetched from a backend, and the various types of data are displayed in different ways using so-called widgets. There are currently three types of widgets:

- *Chart:* A chart widget displays the value of the data over time and provides a good overview of the history of the data up to a given point. An example of a chart can be seen in Figure 2.1 in all but the top-right section.
- *Gauge:* Gauge widgets display the current value of a sensor in a given range. It allows the user to see whether the current value is still within the correct range. An example of a gauge widget can be found on the top-right in Figure 2.1.
- *Image:* Image widgets display the value of a sensor on a specific location of an image. This widget can be used to, for example, display the temperature at various sites on a map. An example of an image widget can be found in Figure 2.2.

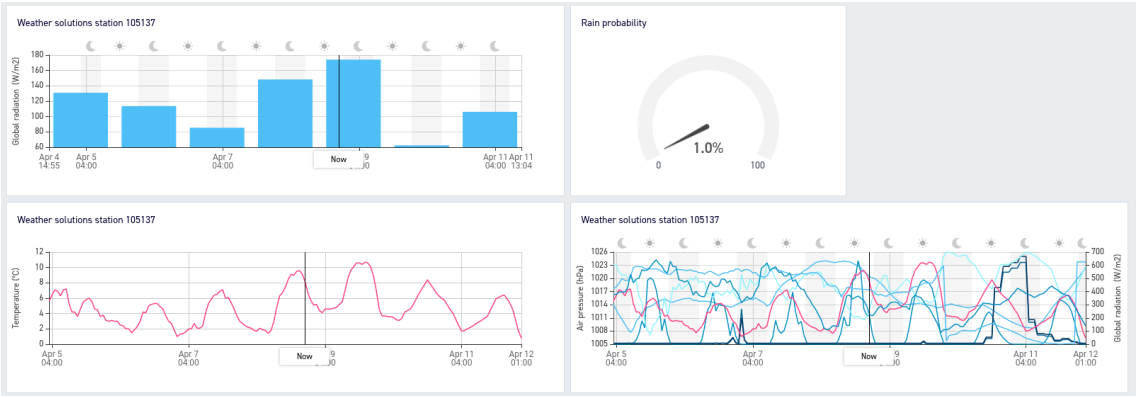


Figure 2.1: Widgets in the 30MHz dashboard

Figure 2.2: An image widget in the 30MHz dashboard

2.1.1 Apps

The data collected by 30MHz can be utilized in many ways. Companies with domain knowledge and expertise in certain areas (such as third parties) can provide customers with new insights and information that simple graphs can not. Because 30MHz itself does not have this domain knowledge and does not have the resources to create every single possible implementation of this knowledge (in the form of a widget or page in the dashboard), 30MHz decided to allow third-party developers to develop them instead. There are currently two implementations:

- *Widgets:* A Widget in the dashboard takes data from one or more sensors and displays it. These are made to provide information at a glance and are fairly small when it comes to screen space, as can be seen in Figure 2.1. An example would be a new way to display the amount of light a plant is getting by showing a sun icon if the plants can grow (it is daytime) and a moon icon if they can not (it is nighttime).
- *Apps:* Apps are full pages in the web app. These fill the entire screen (bar some 30MHz branding) and provide richer and more interactive experiences. An example would be a page where users can tune parameters (such as the number of crops, amount of watering) and see a prediction of their revenue. This prediction can be based (in part) on sensor data.

Since these apps will essentially be pages in the 30MHz dashboard and will feel like part of the platform, it is important that they follow the same design as the rest of the dashboard. A consistent design ensures that users are familiar with the apps and that visual consistency across the platform is not broken. This concept has been applied on Google's Android through Material Design [1] and Zendesk Garden [2] among others. Importantly, these companies all provide app developers with a set of components to help them maintain the intended design language. Such a set of components is generally referred to as a UI library. Similarly, 30MHz wants to provide their third-party app developers with a UI library. In this paper, we will refer to the UI library 30MHz will provide to third parties as the Cow Components UI Library (or CC UI Library). It is named after the logo of 30MHZ, a cow. There already is an internal UI library that covers the basic set of UI components (among others buttons, an input, a date picker), but since it is interwoven with other internal code, its source code can not just be provided to third-party developers.

[1] `https://material.io/`
[2] `https://garden.zendesk.com/`

Additionally, they have been written in Angular [1], meaning that any developers who wish to develop their app in a different JavaScript (JS) framework cannot do so. Looking at the most popular web frameworks in the latest Stack Overflow Developer Survey [2] (2020 as of the writing of this paper), we can conclude that the chance that a developer wishes to use a different JS framework is quite large. In order to still provide developers with a CC UI library, there are two options.

- Write components from scratch in a framework-agnostic format and provide them to developers. Then keep them up to date with the internal set of components by changing one as the other changes.
- Set up automatic migration from the set of internal components to a framework-agnostic format.

The immediately apparent problem with the first option is that developers are maintaining two separate copies of very similar code. This causes several issues. Firstly, the time spent maintaining a component is doubled. Additionally, feature differences between the Angular framework and the framework-agnostic format we choose will lead to added engineering time. Some things that make use of the Angular framework might need workarounds in the other format and the other way around. Another issue with the first option is that the components have to be written entirely from scratch. While writing components from scratch would be manageable for simple components such as buttons, it is unfeasible for more complex components. One such component for which the rewriting process would prove difficult is the 30MHz chart component. This component is vital to the 30MHz design library, seeing as it displays the sensor data. The source code for the chart is tightly coupled with the rest of the platform, referencing about half of the source files in the dashboard through its dependencies. Rewriting all of this code in another framework is wholly unfeasible and not worth the effort, leading us to explore the second option.

While the second option is not an easy one and will likely be a very complex process to set up, it will scale a lot better. Once it is set up, any new components will be automatically migrated, and any changes will be propagated automatically. In the long run, this automatic migration should save time. This option is the one 30MHz eventually decided on. Next, a framework-agnostic format needs to be chosen to facilitate this process.

[1] `https://angular.io/`

[2] `https://insights.stackoverflow.com/survey/2020`

2.2 Web Components

When it comes to choosing a framework-agnostic format for a UI library, there are very few options. Looking at the literature, we find Quid (1), a program that allows code written in a domain-specific language (DSL) to be used to generate components in various frameworks. It currently supports the generating of Web Components [1], Stencil [2], Angular and Polymer [3] components. The authors do mention it should only be used for rapid prototyping. Since it only supports a fairly small set of supported frameworks, and it has the problem of requiring a DSL which the Angular code would have to be migrated to, we choose not to use Quid as the target format.

This brings us to another option, namely Web Components. Web Components (also known as Custom Elements) are a technology proposed in 2013 [4] and implemented in major browsers in 2018 [5]. It allows for the creation of custom HTML elements using JavaScript. These elements can then be used like regular HTML elements. Since every JS framework has support for native HTML elements and almost every framework has full support for Web Components [6], we can cover most JS frameworks by using Web Components as our target format.

2.3 Angular Elements

To perform the migration of Angular components to Web Components, we use Angular Elements [7]. Angular Elements is a JS package that allows for the migration of Angular components to Web Components. It does this by changing the way in which components are mounted. Angular apps are typically mounted to the page by the user through a call to the `bootstrapModule` function. Consequently, the bootstrap component is mounted to the page. This bootstrap component is responsible for containing the rest of the application. After it is mounted, child components are mounted and rendered within its root recursively. Angular Elements works slightly differently. Components registered as Web Components through Angular Elements are instead rendered whenever an HTML element with the registered tag is added to the DOM (document object model). The DOM is the

[1] https://developer.mozilla.org/en-US/docs/Web/Web_Components
[2] https://stenciljs.com/
[3] https://www.polymer-project.org/
[4] https://www.w3.org/TR/2013/WD-custom-elements-20130514/#about
[5] https://caniuse.com/?search=webcomponents
[6] https://custom-elements-everywhere.com/
[7] https://angular.io/guide/elements

model containing the elements on a page and how they are positioned relative to each other. Adding an element to the DOM adds it to the webpage. When such a component is rendered, a new root is created in place of this new HTML element. Instead of a single root in which everything is rendered (as is the case in a typical Angular app), components are all rendered in their own local root. We use Angular Elements for the migration to Web Components in this case study since it appears to be the only package providing this capability.

Figure 2.3: An example of a component that provides type hints (these hints are also referred to as intellisense)

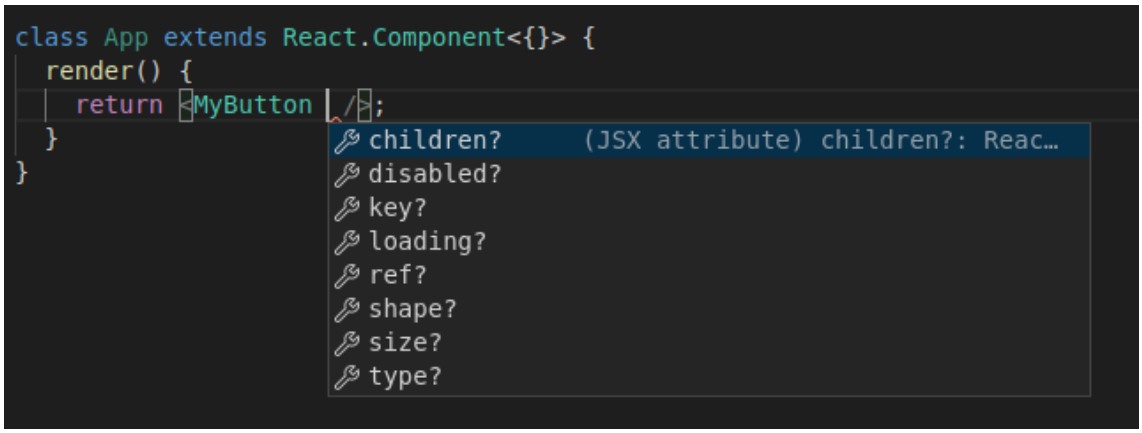
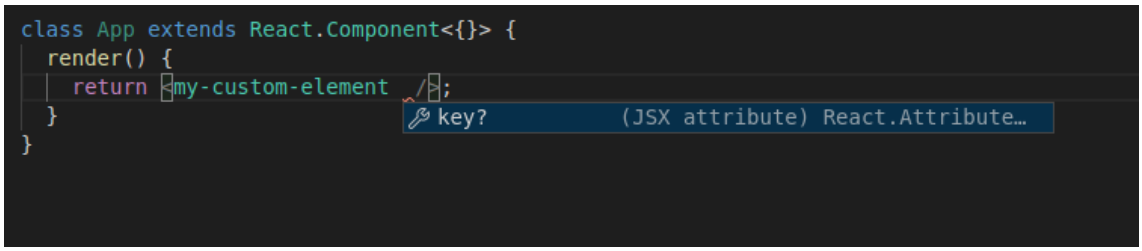


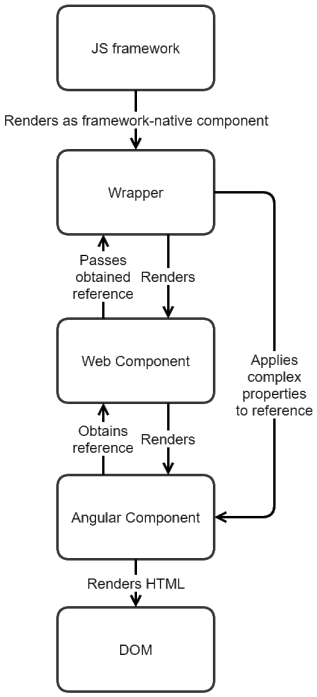
Figure 2.4: An example of a component without type hints



2.4 Javascript frameworks

While migrating components to Web Components makes them usable in most JS frameworks, they do not provide a perfect experience. The first reason for this is them not being perfectly compatible with every JS framework. As of the writing of this paper, there are

Figure 2.5: A diagram representing the relationship between the JS framework and Web Components



still some issues preventing them from working entirely in ReactJS [1], a JS framework created by Facebook in 2013. These issues mostly concern the passing of non-primitive data to the components, such as JavaScript Objects, Arrays, and Functions. The second problem is that they are not native to JS frameworks and do not integrate very well with the tooling provided by the framework. One such tool is type hinting, in which an editor suggests possible values to the developer. An example of type hinting provided by the framework and editor can be seen in Figure 2.3. Compared to Figure 2.4, which shows a component with no type hinting, Figure 2.3 provides the developer with much more information and shows them what options are available to them. Instead of searching the web for the available properties, these options are provided by the element's source code and displayed through the framework's tooling and the editor. In order to improve the developer experience, we will provide what we will call a *wrapper* for each framework. This wrapper has two functions. Firstly, it provides the tooling mentioned above. Secondly, it bridges the gap with JS frameworks that do not natively support Web Components yet. This wrapper is native to the framework, being written using the language and library the framework provides. This allows the framework to infer information from the

[1] `https://reactjs.org/`

wrapper's source code. Under the hood, this wrapper still uses the components in the Web Components UI library to render the components. This wrapper serves as glue code between the framework and these components. By combining the two steps of migrating the original Angular components to Web Components, and the Web Components to wrappers for frameworks, we can provide developers with an experience native to their framework, even though the original source code uses Angular. An overview layout of the relationship between the JS framework, the wrapper and the Web Components can be seen in Figure 2.5.

2.5 UI Libraries

As mentioned before, a set of components that adheres to one cohesive design language is generally referred to as a UI library or design library. There are two methods for implementing a design library, with one being based on doing most of the work in JavaScript and the other being based on shifting this work to CSS. The former is also referred to as a design library, with the latter being called a CSS framework. The idea of a CSS framework is to put almost all of the styles a developer will need in a single CSS file. This includes the various variations they could need. For example, a CSS library could include the `.padding-5` selector as well as the `.padding-2` selector for setting the padding of a component. Note that the number of pixels of this padding is included in the selector. This generally leads to relatively big CSS files, which may or may not be tree shaken. This is in contrast to JavaScript-based UI libraries, which generally use per-component stylesheets instead of global stylesheets. They also tend to shift numbers and sizes to JavaScript or HTML. For example the same padding as above could be applied through a property, i.e. `<my-component padding="2"/>` or `<my-component padding="5"/>`. This approach has the advantage of a more per-component focus, more flexibility, and options that are easier to discover. However, compared to CSS frameworks, JS-based UI libraries are significantly slower. The CSS frameworks generally only append an element to the DOM and apply some pre-computed set of classes to them, meaning they only interact with the swift JavaScript APIs that are native to the browser. JS-based UI libraries, on the other hand, have to take care of styling, component interactivity through various event listeners, and changing what is rendered depending on properties. Since this performance difference is important when it comes to measurements, a given library being a UI library or CSS framework, is mentioned later.

3

Related Work

There are various fields in which the related work is important to us in this paper. Namely related work in the area of UI Libraries, Angular Elements (and the accompanying process of migrating to Web Components), related work on Web Components themselves, and related work on the creation of wrappers around Web Components to target JS frameworks.

3.1 UI Libraries

UI libraries are at their most basic level a set of components that follow a common design language. These components can be written using various different JS frameworks. Mráz *et al.* (2) dives into the definition of the component model, documenting the ideas behind it and how it came to be. They then discuss Web Components and their feasibility as a stand-alone replacement for current JS frameworks in component development. They find Web Components to be cumbersome to use, requiring a lot of boilerplate code (repeated sections of code with little to no variety (3)). They argue that this is the reason that other JS libraries are becoming more popular. They provide more abstraction, less boilerplate, and more ease of use. Further, they discuss two specific JS frameworks in detail, namely LitElement [1] (an abstraction of Web Components) and ReactJS [2] (a stand-alone library for creating components). They find them both to be capable frameworks, both filling their own niche. Finally, they note that both are essentially interoperable, with Web Components almost being natively usable in ReactJS [3]. Ky Nam *et al.* (4) makes use of these components to build a UI library. They document the building of a UI library that uses ReactJS as a JS framework. They draw inspiration from other ReactJS UI libraries

[1] `https://lit-element.polymer-project.org/guide`
[2] `https://reactjs.org/`
[3] `https://custom-elements-everywhere.com/`

such as Material-UI [1], React Bootstrap [2], and React virtualized [3]. They eventually decide to split their UI library into both a generic UI library with minimal styling (focusing on the UI) and a Core library (responsible for handling stateful components and communicating with the server). The end result meets their goals, having created a UI library that uses the aforementioned technologies. Furthermore, there are many blog posts documenting the process of creating UI libraries. In these blog posts [4] [5] [6] [7] [8], the authors provide guidance in setting up and creating a UI library. They mainly concern the basics, explaining how to get started with the process but not delving into the creation of complex components. We also find numerous examples from the industry. These include but are not limited to Svelte Material UI [9] (written in Svelte), React Bootstrap [10] (React), Angular Material [11] (Angular), Wired Elements [12] (Web Components), and Onsen [13] and SyncFusion [14] (both multi-framework). For all but SyncFusion, the source code is freely available on GitHub, allowing us to draw inspiration from it and look at how various problems were solved in different UI libraries. A complete list of UI libraries can be found in Table 4.2.

3.2 Angular Elements

Research on the area of Angular Elements is very sparse. Armengol Barahona *et al.* (5) uses Angular Elements for the rendering of form components. They create a form component in Angular that is able to change the input elements it renders dynamically. It does this by migrating Angular components to Web Components by using Angular Elements. They are then dynamically appended to the DOM. They find Angular Elements to be a good fit for this task, being easy to set up and easy to work with. Again blog posts on Angular

[1] https://material-ui.com/
[2] https://react-bootstrap.github.io/
[3] https://github.com/bvaughn/react-virtualized
[4] https://www.toptal.com/designers/ui/design-framework
[5] https://dev.to/giteden/building-a-ui-component-library-for-your-startup-4cek
[6] https://www.emergeinteractive.com/insights/detail/how-to-ux-ui-design-system-component-library/
[7] https://codeburst.io/building-an-awesome-ui-component-library-in-2020-a85cb8bec20
[8] https://itnext.io/building-a-scalable-ui-component-library-4607de91955a
[9] https://sveltematerialui.com/
[10] https://react-bootstrap.github.io/
[11] https://material.angular.io/
[12] https://wiredjs.com/
[13] https://onsen.io/
[14] https://www.syncfusion.com/

Elements are numerous. In various blog posts [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12], the authors explain how to set up Angular Elements and how to use it to create a new component library. These blog posts predominantly focus on creating new components or migrating simple components through Angular Elements, not migrating larger and more complex components. They all use new and empty projects, contrary to two other blog posts [13] [14]. The authors use Angular Elements to migrate existing AngularJS (an older version of Angular) components to the newer Angular. They do this by migrating the source code of existing AngularJS components to Angular source code. By itself, this would break since the application root still runs on AngularJS and is unable to handle Angular code. By using Angular Elements to migrate the Angular code into Web Components, the Web Components can run inside the AngularJS root. This is thanks to the low-level nature of Web Components, allowing any framework that can render HTML elements to use them. Through this iterative process, they can migrate components one by one, migrating the root component once all of its children have been migrated.

Unfortunately, we were unable to find any related work on the migration of complex Angular components to Web Components through Angular Elements. Related work seems to focus primarily on small get-started style projects. When they focus on more complex projects, it seems like the only use is the migration from AngularJS to Angular.

[1] https://netbasal.com/understanding-the-magic-behind-angular-elements-8e6804f32e9f

[2] https://medium.com/kitson.mac/wrapping-an-angular-app-in-a-custom-element-web-component-angular-element-in-4-simple-steps-ded3554e9006

[3] https://medium.com/@smarth55/angular-elements-use-them-everywhere-including-your-angular-app-697f8e51e08d

[4] https://blog.piotrnalepa.pl/2020/02/02/how-to-convert-angular-component-into-reusable-web-component/

[5] https://medium.com/swlh/angular-elements-create-a-component-library-for-angular-and-the-web-8f7986a82999

[6] https://www.thirdrocktechkno.com/blog/angular-elements/

[7] https://juristr.com/blog/2019/04/intro-to-angular-elements/

[8] https://studiolacosanostra.github.io/2019/07/19/Build-a-reusable-Angular-library-and-web-component/

[9] https://blog.bitsrc.io/using-angular-elements-why-and-how-part-1-35f7fd4f0457

[10] https://www.techiediaries.com/angular/angular-9-elements-web-components/

[11] https://indepth.dev/posts/1116/angular-web-components-a-complete-guide

[12] https://indepth.dev/posts/1228/web-components-with-angular-elements

[13] https://blog.nrwl.io/upgrading-angularjs-to-angular-using-elements-f2960a98bc0e

[14] https://medium.com/capital-one-tech/capital-one-is-using-angular-elements-to-upgrade-from-angularjs-to-angular-42f38ef7f5fd

3.3 JS Framework Wrappers

We were unable to find any research on JS framework wrappers. JS framework wrappers do not seem to be a problem that has been tackled very often, at least in literature. On the website *custom-elements-everywhere.com* [1], the authors keep track of the current usability of Web Components in various JS frameworks. Notably, the ReactJS framework does not fully support Web Components at the time of writing for this paper. In ReactJS, non-primitive values (such as Objects, Arrays, and Functions) can not be passed to Web Components, along with some other issues. As such, it is the only framework that needs a wrapper for the UI library to function at all. Looking at how to fix this issue, we find some proposed solutions in a blog post [2]. In this blog post, the author explores various options to tackle this problem of passing non-primitive data.

3.4 Metrics

In Chapter 4 we introduce metrics that are used to measure the quality and performance of the created Web Component library. Looking specifically at evaluating the quality of Web Components, a candidate for such a set of metrics is proposed by Martinez-Ortiz *et al.* (6). They propose a set of metrics that together give a good overview of the quality of a Web Component. These metrics are structural complexity (the number of import statements for a component), cyclomatic complexity (a quantitative measure of the number of linearly independent paths through a program's source code (7)), maintainability (a derivative based on complexity, lines of code, and Halstead volume (8)), completeness (how complete the information displayed to the user is as a percentage), latency (the time between when a request is made and when its content is received), and consistency (a time metric reflecting how long it takes an update to take effect across different replicas of the same component). The first three metrics are intrinsic metrics based on the source code of the Web Component. They take into account the quality of the source code. The last three metrics aim to capture the quality of a component as perceived by a user. They validate their metrics in user studies, finding that the metrics correlate strongly with the results of the user studies.

[1] `https://custom-elements-everywhere.com/`

[2] `https://itnext.io/handling-data-with-web-components-9e7e4a452e6e`

3.5 Load Time

A similar metric for measuring the quality of a web page is the load time. Gao *et al.* (9) dives into metrics that describe the load time of a page. They find that common metrics such as *onLoad* and *Time To First Byte* fail to accurately describe the load time as perceived by users of the page. Instead, they introduce a learning model that explains the majority of user choices with 87% accuracy. Similarly, Nathan *et al.* (10) find that current metrics do not describe the perceived load time of a page well. They define a new metric called *Ready Index*, aimed to capture interactivity explicitly. They then compare their metric to prior load time metrics, finding that they underestimate or overestimate the true load time of a page by between 24% and 64%. Van Riet *et al.* (11) builds upon this work, also performing a case study at 30MHz in which they achieve a 97.56% reduction in the time for the First Contentful Paint on mobile devices. The First Contentful Paint is a metric that describes the time until the first element on the page is rendered.

4

Study Design

4.1 Research questions

Our goal in this paper is to evaluate the effectiveness of migrating Angular components to Web Components. Based on this goal, we devise a single research question:

RQ1: How technically viable is the process of migrating Angular components to Web Components?
In answering this research question, we assess whether the migration process is possible at all, what a possible performance impact could be, and how the resulting migrated components relate to other component libraries in the field.

4.2 Metric definitions

In order to answer the above research questions, we need to define metrics. These allow us to compare the resulting Web Component library both to the UI library and to other UI libraries. This allows us to get a better understanding of the quality and performance of the created Web Component library relative to other UI libraries. We can divide the used metrics into two categories. The first is measuring the quality of the resulting CC UI library. In order to do measure the quality, we compare the components in the CC UI library to the Angular components they originate from, as well as to various UI libraries. We perform this comparison using various metrics divided into three groups. These groups are *complexity*, *size*, and *performance*. We perform this comparison both at component

granularity and at UI library granularity. A full list of these metrics, as well as a brief description, can be seen in Table 4.1. A detailed explanation of these metrics follows.

ID	Group Metric	Granularity	Description	
SC	Complexity	Structural complexity	Component	The number of import statements for a component. Collected for a source file and all of its dependencies for up to two iterations
CC	Complexity	Cyclomatic complexity	Component	A quantitative measure of the number of linearly independent paths through a program's source code (7)
LOC	Size	Lines of code	Component	The number of lines of code in a given component's source file
SI	Size	Size	UI Library	The file size of the bundled up library
MA	Performance	Maintainability	Component	A derivative based on complexity, lines of code and Halstead volume (8)
RT	Performance	Render Time	Component	The render time of a given component
LT	Performance	Load Time	UI Library	Parsing and running time of the bundled up library in the browser (without download time)
NOC	Performance	Number of Components	UI Library	The number of components in a UI library
FC	Performance	First Paint	UI Library (cow-components only)	First paint event of the browser
FCP	Performance	First Contentful Paint	UI Library (cow-components only)	First paint event of the browser that includes content for the user (text, images, etc.)

Table 4.1: Metrics used in this study

4.2.1 Source code metrics

The first set of metrics, namely structural complexity, cyclomatic complexity, lines of code, and maintainability, are metrics that are recommended by Martinez-Ortiz *et al.* (6) as described in Section 3.4. We use these metrics to compare the quality of our Web Components to other Web Components. We follow almost all recommendations by the paper, including collecting the structural complexity up to a depth of two. Note that we do things slightly differently from the paper. We also keep track of the lines of code metric, which the authors do not. We do not use the lines of code metric to compare the quality of Web Components, but instead we use it to get a rough overview of the complexity of various UI libraries. Note that we are also not using all metrics recommended by the authors. The metrics we are not using are the metrics completeness (i.e. how complete the information displayed to the user is) and consistency (i.e. how long it takes for data to update across different replicas). We are not using completeness because it does not apply at the level at which the CC UI library operates. All of our components are 100% complete, as well as the components of the UI libraries we compare the CC UI library to. As such, it does not make for a very interesting metric. This metric is very effective when

more complex components such as entire pages are concerned, but that is not the case here. We also do not use the consistency metric. The reason for this is relatively simple, namely that we do not have any components with the ability to update across different replicas. The same goes for the UI libraries with which we compare the CC UI library.

4.2.2 Size

The size metric aims to measure the theoretical impact of loading the UI library over the network. We measure this at UI library level granularity since the contributions of individual components are very hard to measure. This should serve as a good indication of the relative network loading time of UI libraries without introducing the variable of network speed. In order to differentiate between a relatively large library and a library that has many components, we also keep track of the number of components metric.

4.2.3 Load Time

The load time metric aims to provide a measure of the real impact of a UI library on the page by measuring the real-world load time. The load time we measure is the load time of the main JS bundle. This contains the code needed to register the components to the page, as well as the code that performs the rendering. Note that we only measure the parsing time and running time of the JavaScript bundle. We explicitly exclude the download time from this metric since this is already captured in SI. A more in-depth definition of how this metric is captured is laid out in section 5.7.

4.2.4 First Paint & First Contentful Paint

The first paint metric, along with the first contentful paint metric, are only collected for the CC UI libraries. These metrics give us an indication of the real-world load time of a page containing the CC UI library and the original components. We use these metrics to evaluate how the paint time of the UI libraries has changed after its migration to Web Components and its later migration to JS framework wrappers. While these metrics do not serve as a perfect way to measure the perceived load time of a page, as discussed in Section 3.5, they should serve as an excellent comparison between the various distributions of the CC UI library. Since each of them contains the exact same content and is derived from the exact same source code, imperfections in these metrics are applied to all test subjects equally.

4.2.5 Render Time

Finally, the render time metric aims to capture the duration of the render cycle. We will define this render time as the time between setting the component's visibility to true and the browser completing the rendering process. If the render time of components in the CC UI library is significantly higher than components in other UI libraries or the Angular components they originate from, the performance impact of migrating Angular components to Web Components will be too significant. If it is slightly higher, the same, or lower, we can conclude that the performance impact is minimal.

In order to obtain an objective measurement of the rendering time that is independent of user perception, we chose to measure the render time of individual components. Since the components we use in our comparisons all load in a single stage (they are either not visible or visible), there is no loading state that could cause ambiguity.

4.3 Metric targets

In order to get a sense of the state of the CC UI library, we need to compare it to other UI libraries. To do so, we have gathered a list of various UI libraries targeting the most popular JS frameworks. Four of the most popular frameworks are ReactJS, Angular, Vue, and Svelte [1]. Through this comparison, we can compare the wrapper targeting a specific JS framework with UI libraries that also target that JS framework, allowing us to observe the influence of the framework itself on the various metrics. These UI libraries are gathered by searching for the terms "Design Library", "UI Library", "javascript UI Library", "Svelte UI library", "React UI Library", "Vue UI Library", "Angular UI Library", and "Web Component UI Library" on Google. We then add any UI library to the list that we came across, either by finding it as a direct result or it being mentioned in a blog post or article. A list of the UI libraries we found and the number of stars on their GitHub page can be found in Table 4.2. While this is not a complete list of all UI libraries, we feel like it is an accurate representation of the most popular UI libraries since it contains all of the biggest UI libraries, as confirmed by the numerous blog posts listing them in order. In order to get a reasonably accurate representation of libraries using each JS framework, we select the three UI libraries with the most GitHub stars per JS framework. The list of included UI libraries can also be seen in Table 4.2. In addition to comparing the CC UI library against other UI libraries, we also compare it against the Angular components from which they

[1] `https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/`

originate. We do this by applying our metrics to the 30MHz dashboard and the relevant components within it.

Since the components included in the selected UI libraries vary greatly, we can not make a proper comparison between individual components of the UI library. For example, a button component can not be immediately compared with a date picker component since date pickers tend to be more complex. In this scenario, higher rendering times can not be attributed to the UI library running it but to the component itself. In order to be able to compare every UI library, we have selected a set of basic components that are available in every UI library. These are the Button, Input (also known as TextField), and Switch (also known as Checkbox). Since every UI library we compare against contains all of these, we can compare the metrics for a single component across all UI libraries. We only apply the various metrics to these three components in each UI library. We also include a stripped-down version of the CC UI library in the set of UI libraries of which we gather metrics. This version only contains the three components mentioned above and allows for a fair comparison with other UI libraries. The reason for this is further explained in Section 5.6.

4.4 Analysis of results

In order to compare the collected measurements, we use the median value over a set of measurements. Compared to the average, the median minimizes the impact of outliers in a data set. Since the measurements we collect are likely to have outliers in them due to the nature of time-sensitive measurements, this statistical value is likely to be a better choice.

UI Library	Github Stars	JS Framework	Included	Version	Website
Svelte Material UI	1.6k	Svelte	Yes	2.0.0	https://sveltematerialui.com/
Smelte	889	Svelte	Yes	1.1.2	https://smeltejs.com/
Svelte-MUI	237	Svelte	Yes	0.0.3-7	https://svelte-mui.ibbf.ru/
Svelteit	51	Svelte	No	-	https://docs.svelteit.dev/
Material UI	67.1k	ReactJS	Yes	5.0.0-alpha.28	https://material-ui.com/
React Bootstrap	19.2k	ReactJS	Yes	1.5.2	https://react-bootstrap.github.io/
React Semantic UI	12.2k	ReactJS	Yes	2.0.3	https://react.semantic-ui.com/
Evergreen	10.6k	ReactJS	No	-	https://evergreen.segment.com/
Rebass	7.2k	ReactJS	No	-	https://rebassjs.org/
Grommet	7.1k	ReactJS	No	-	https://v2.grommet.io/
Baseweb	6.2k	ReactJS	No	-	https://baseweb.design/
Ant Design	5.3k	ReactJS	No	-	https://ant.design/
Elemental UI	4.3k	ReactJS	No	-	http://elemental-ui.com/home
Zendesk Garden	858	ReactJS	No	-	https://garden.zendesk.com/
Shards React	649	ReactJS	No	-	https://designrevision.com/docs/shards-react/getting-started
Angular Material	21.3k	Angular	Yes	12.0.0-next.5	https://material.angular.io/
NG-Bootstrap	7.7k	Angular	Yes	9.1.0	https://ng-bootstrap.github.io/#/home
NGX-Bootstrap	5.3k	Angular	Yes	7.0.0-rc.0	https://valor-softw2are.com/ngx-bootstrap/#/
NG-Lightning	886	Angular	No	-	https://ng-lightning.github.io/ng-lightning/#/
Alyle	236	Angular	No	-	https://alyle.io/
Blox Material	143	Angular	No	-	https://material.src.zone/
Mosaic	117	Angular	No	-	https://mosaic.ptsecurity.com/button/overview
Element	49.8k	Vue	Yes	1.0.2-beta.40	https://element-plus.org/#/en-US
Vuetify	30.2k	Vue	Yes	2.4.9	https://vuetifyjs.com/en/
Quasar	18.3k	Vue	Yes	1.15.10	https://quasar.dev/
Ant Design Vue	14.1k	Vue	No	-	https://2x.antdv.com/docs/vue/introduce
Bootstrap Vue	13.1k	Vue	No	-	https://bootstrap-vue.org/
Vue-material	9.3k	Vue	No	-	https://vuematerial.io/
Buefy	8.6k	Vue	No	-	https://buefy.org/
Vuesax	5k	Vue	No	-	https://vuesax.com/
Chakra	1.1	Vue	No	-	https://vue.chakra-ui.com/
Fish UI	867	Vue	No	-	https://myliang.github.io/fish-ui/
Wired Elements	8.5k	Web Components	Yes	1.0.0	https://wiredjs.com/
Clarity Design	6.2k	Web Components	Yes	5.1.0	https://clarity.design/
Fast	5.6k	Web Components	Yes	1.8.0	https://www.fast.design/
Material Web Components	2.5k	Web Components	No	-	https://github.com/material-components/material-components-web-components
UI5	887	Web Components	No	-	https://sap.github.io/ui5-webcomponents/
Vaadin	17	Web Components	No	-	https://vaadin.com/
Onsen	8.3k	Multi-Framework	Yes	2.11.2	https://onsen.io/
Primefaces (Angular)	1.3k	Multi-Framework	Yes	11.3.2-SNAPSHOT	https://www.primefaces.org/primeng/
Primefaces (React)	1.3k	Multi-Framework	Yes	6.2.2-SNAPSHOT	https://www.primefaces.org/primereact/
Primefaces (Vue)	1.1k	Multi-Framework	Yes	3.3.6-SNAPSHOT	https://www.primefaces.org/primevue/
Syncfusion	unknown (not on github)	Multi-Framework	No (paid)	1.0.0	https://www.syncfusion.com/

Table 4.2: Collected UI libraries, the number of github stars and whether they were included in the study

5

Experimental Setup

We now describe how each of the metrics is being captured and what parameters are used.

5.1 Gathering components

The SC, CC, LOC, and MA metrics are captured from the source files of components. In order to gather these source files, we do the following: We set up an automatic script that gathers the source files of components on a per-library basis. Since most UI libraries follow the convention of storing each component in a single folder or file in a source folder (generally called `src/` or `components/`), this process is fairly simple. In order to provide a fair comparison, we always select the largest source file for components as the entry point. Some UI libraries use a simple index file that re-exports the actual source file as the entry point. If this file were to be used as the entry point, it would result in an unrealistic depiction of the component source. Since the UI libraries in this study always contain a single big source file, this did not result in any situations where the entry point was ambiguous.

5.2 Structural Complexity

The structural complexity is gathered by capturing the number of imports in a given source file recursively up to a depth of two, as recommended in (6). To gather these imports, we use the `typescript`[1] JS package. This package is able to generate an AST or abstract syntax tree of the file. An AST is an in-memory representation of source code. It categorizes the semantic meaning of each expression as a tree, denoting relations between

[1] `https://www.npmjs.com/package/typescript`

nodes in the tree. By iterating over this tree, we can find the imports. We then follow these imports and apply the same process, filtering out any duplicates.

Similar to (6), we only apply this process to the JS source code of a component, not the HTML source code. In the case of Svelte and Vue components, we separate the file into its JS code and HTML code and apply the process to the JS code only. Code using ReactJS is written using either plain JavaScript or JSX. JSX is a superset of JavaScript that supports the describing of HTML elements in JavaScript. Since the `typescript` package has built-in support for JSX, we do not need to separate or modify this code.

5.3 Cyclomatic Complexity, Maintainability, Lines of Code

In order to capture the cyclomatic complexity, maintainability metrics, and lines of code metrics, we input the file into the `ts-complex`[1] JS package. This package is able to calculate the cyclomatic complexity, maintainability, and lines of code metrics for a given source file. Note that the lines of code metric does not capture the raw number of lines but instead filters out any comments, aiming to capture just the lines with actual code.

5.4 Machine specifications

All experiments are performed on a machine with an AMD Ryzen 5 4600H six-core processor and 16GB of RAM. This machine is running Linux 5.11.15 using the Arch Linux distribution [2] with the `mitigations=auto` kernel parameter. All experiment data is loaded from an M.2 SSD. Since these experiments are partially timing-specific, the timing-specific experiments are run sequentially and with minimal background tasks. We achieved a state of minimal background tasks by closing off all non-essential tasks found in the process manager. This should eliminate the effect of experiments on each other and ensure the CPU can always dedicate a single core to the running experiment. Since this machine has six cores, it should easily be able to dedicate one of them to the experiments at all times. Finally, since all experiments were run in one go, the test environments for all tests are identical.

[1] `https://www.npmjs.com/package/ts-complex`
[2] `https://archlinux.org/`

5.5 Time-sensitive metrics

For all time-sensitive metrics (metrics that measure time), we take a few steps to improve their accuracy. We first of all artificially slow down the speed of the processor by a factor of five by using the `Emulation.setCPUThrottlingRate` command [1] in the browser. We then divide the measured number by this scale, normalizing the value. Additionally, we perform every time-sensitive measure thirty times. This allows us to get a good overview of the spread of the measured values, as well as reducing the effect of variations in hardware performance and software influences. Finally, we randomize the order in which tests are run. We do this by creating a queue of all to-be-ran time metrics. Every item in the queue is a single-time metric test for a single bundle, meaning every metric bundle combination is in the queue thirty times. This queue is then shuffled, after which the individual items are executed. While we already made sure to reduce the number of running processes on the benchmark computer, this should ensure that any temporary differences in available processing time should be smoothed out.

5.6 Size

In capturing the size metric, we need to pay attention to several influential factors. The first factor is that the source code of files is split up into multiple files, some of which are not actually used at runtime. Examples of this include files used during testing and type definition files. Additionally, it is possible that unreachable code does not make it into the bundle because it will not be executed. The process by which code that is not going to be executed is excluded from the resulting bundle is called *tree shaking* and is discussed later. The fact that the source code contains unreachable code means that the size of the source code is not representative of the code that is actually being used. Additionally, the UI library will have dependencies outside of its source code that also need to be included. To get around these issues, we use a JavaScript bundler. A bundler is a program that bundles all of the source code of a given project into a single file, including dependencies and source files that are being used, and excluding unreachable files or code. We use the `esbuild`[2] bundler for this process.

Another influential factor in determining the size is the way in which the source code is written. A file containing many comments is larger than a file containing no comments. An

[1] `https://chromedevtools.github.io/devtools-protocol/tot/Emulation/#method-setCPUThrottlingRate`

[2] `https://esbuild.github.io/`

increased number of comments in a file does not necessarily indicate increased complexity; if anything, it indicates the opposite. Similarly, longer variable names increase the size as well. To eliminate this factor, we apply *minification* to our bundle. Minification strips out any non-code text from a bundle and reduces the size of the code to the minimum that is needed.

The final influential factor is the number of components and the type of components. A UI library with five components will generally be smaller than a UI library with thirty components. Even when we account for this difference by capturing the number of components, the result will still be influenced by the types of components the UI library contains. For example, if two UI libraries are the same except that one of them contains a complex chart component, the chart heavily skews the average size of a component. This is the case even though any given component in the other library is the same size; the chart library just has different types of components. To get around this issue, we make sure to construct a bundle containing the same three types of components for every UI library. We then make use of *tree shaking* to exclude other components from the bundle. Tree shaking is the process by which unused code is removed from a JS bundle, effectively reducing its contents to just code that is reachable. By including the same three types of components into the bundle for every UI library, we can create bundles containing the same functionality and nothing more. This eliminates the influence of bigger or smaller components that are also available in the UI library since they are excluded from the bundle entirely.

The tree-shaking process is applicable to the UI libraries against which we are comparing the CC UI library. However, it is not applicable to the CC UI library itself. This is the case because every component is registered as a Web Component simply by loading the library, which means that every component is used. Tree shaking is then unable to remove any components from the bundle, leading to a relatively larger bundle. This would lead to the CC UI library having a much larger size than other UI libraries. To provide a fair comparison, we add a stripped-down version of the CC UI library to the set of UI libraries against which we are comparing. This stripped-down version only contains the three basic components, and as such, allows for fairer size comparison against other UI libraries.

After these factors are taken care of, the process of capturing the size metric is as easy as getting the file size of the resulting bundle.

5.7 Load Time

For the load time metric, we capture the parse- and runtime of the JS bundle described in Section 5.6. We explicitly exclude the network load time of the bundle. In order to collect this metric, we use the `puppeteer`[1] JS package. This package allows the running of and programmatic control over a headless instance of the Google Chrome browser [2] (a headless browser is a browser without a graphical user interface). We set up an empty webpage containing just the JS bundle whose load time we wish to measure. We then enable the Chrome profiler [3] for the page. The Chrome profiler is a profiler built into the Google Chrome browser that allows for the measuring of various metrics during page execution. For example, it is able to measure the run time of JavaScript code and various browser-specific tasks such as painting, compositing, and rendering. We now load the page, stop the profiler and collect the results. We then look for the `EvaluateScript` event in the resulting trace events. This contains the time taken evaluating the given bundle. An example of such a trace can be seen in Figure 5.1.

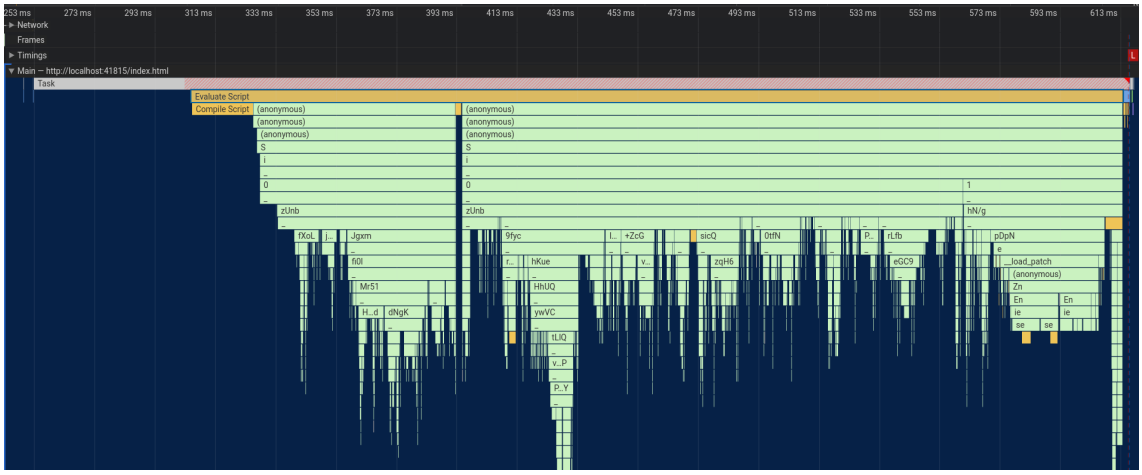


Figure 5.1: An example of a Chrome profiler trace performed during a bundle load. The orange bar labeled "Evaluate Script" indicates the total load time and spans 308.54ms.

5.8 Render Time

We capture the render time metric by using the puppeteer package as well. We prepare a JS bundle containing the three basic components and an exposed function that allows them to

[1] `https://github.com/puppeteer/puppeteer`
[2] `https://www.google.com/intl/en_us/chrome/`
[3] `https://developer.chrome.com/docs/devtools/evaluate-performance/`

be rendered on-demand. This function is JS framework-specific since every JS framework has a different method of conditional rendering. The rendering methods for the various frameworks can be seen in Listings 2,3, 4, 5, 6. We then load the page in a puppeteer browser, enable the profiler, and call the function that renders a given component. We wait for a few seconds, after which we assume the component to be fully rendered. If our assumption turns out to be wrong and the component is not done rendering at time of looking for the end event, we will be unable to find the end event, and metric collecting will fail. We then increase this timeout and try it again. We then iterate through the captured performance trace and look for the time difference between two events. The first event is the calling of the function mentioned above. The second event is the last composite event that has a paint event before it. We repeat this process three times per component (on top of the thirty mentioned in Section 5.5 for a total of ninety measurements), thirty measurements with a single instance of the component, thirty measurements with 10 instances, and thirty measurements with 100 instances. This allows us to measure a more realistic scenario where multiple components are rendered at once, as well as eliminating any performance impacts on just the very first component.

We chose the last composite event that has a paint event before it for the following reason. The chrome browser updates the view through a pipeline process. The complete pipeline can be seen in Figure 5.2. This process always starts with a JS, CSS, or HTML change. Then it performs a different set of pipeline events depending on what changed.

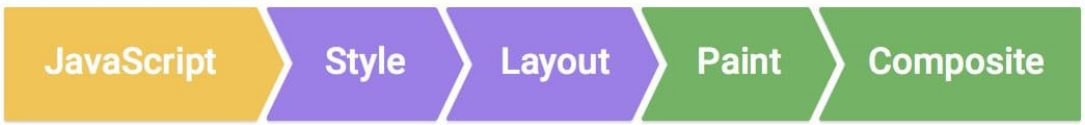


Figure 5.2: The render pipeline in chrome.

Source: `https://developers.google.com/web/fundamentals/performance/rendering`

- *Layout:* If a layout property such as the element's dimensions changes, the entire pipeline is run.
- *Paint:* If a paint-only property such as a color changes, all but the layout stages run.
- *Animation:* If a property that neither layout nor paint changes, only the JavaScript, style, and composite stages run. This pipeline is generally run when an animation is active.

In capturing the render time, we want to capture the time until a component reaches its final state. We need to define this final state for all components. While this state is fairly simple to define and is static for most components, it can also be a dynamic final state. For example, a loading spinner or a component that contains a canvas will at some reach its final state but will still be visually changing. The time between the component not being visible and it reaching its final state are spent in the Layout and Paint stages, while the time after it is spent in the Animation stage. Since we want to capture only the time until the final state, we only care about the Layout and Paint stages. The only difference between these two stages and the Animation stage is that the Animation stage ends with a composite event without a paint event before it, and the other two do not. We use this to our advantage by looking for the last composite event that has a paint event before it. We can not just take the last paint event since the composite event is still part of the pipeline and is technically part of the render stage. When we take the time between the calling of the function that starts the rendering and this event, we are able to capture the time a component takes to render perfectly. A visual representation of this rendering process can be seen in Figure 5.3 and Figure 5.4.

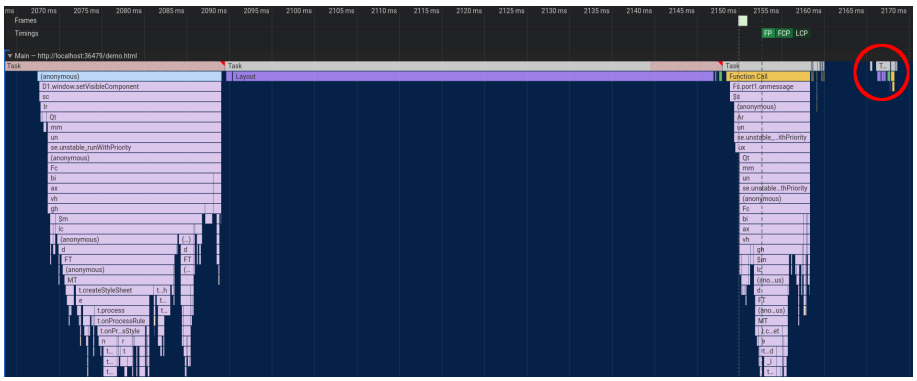


Figure 5.3: An example of a Chrome profiler trace performed during the render of a component. The bar labeled "window.setVisibleComponent" indicates our start time. The end time falls within the red circle, a zoomed in version of which can be seen in Figure 5.4.

5.9 First Paint & First Contentful Paint

In order to measure the first paint and first contentful paint, we construct a page that has the same content across all versions of the CC UI library. This means we construct one for the original Angular components, the Web Components version, and the various JS framework wrappers. We measure this metric by using the browser's built-in

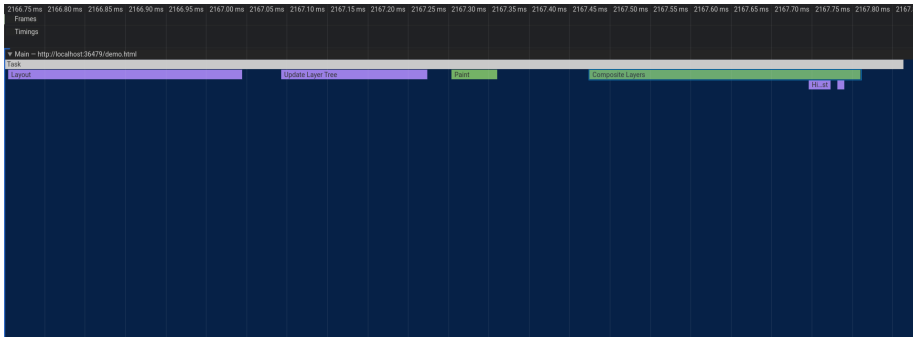


Figure 5.4: An example of a Chrome profiler trace performed during the render of a component. The bar labeled "composite layers" signals our end time. Note that this falls just after a bar labeled "paint", signaling the paint event before the last composite event.

`performance` object. This object keeps track of both the FP and FCP metrics, allowing us to extract them.

5.10 Number of Components

The number of components is captured separately for the UI library as a whole and for the bundle described in Section 5.6. Since the bundles described in Section 5.6 always contain three components, this number will always be three. The only exception is the CC UI library. For the CC UI library and the UI libraries captured as a whole, we gather the number of components by applying the process described in Section 5.1 to gather components, after which we count the number of them.

6

Case Study

In this chapter, we lay out the steps taken and the issues faced during the conversion from Angular components to Web Components. This chapter consists of five sections. The first section provides some information on the build process we are using. The second section describes issues faced that were not specific to Angular or Angular Elements, while the third section contains just Angular-specific issues. The fourth section lists the various optimizations we applied, and the fifth section describes the JS framework wrappers. Note that because of the split between Angular and Web Components issues, the issues are no longer listed out in chronological order. For a chronological overview of the various issues and their relative complexities, see Table 6.1. After the issue sections, we discuss the various JS framework wrappers and how they were created. Lastly, we list optimizations performed along with their effectiveness.

6.1 Build Process

The build process that eventually generates the Web Component library consists of a number of stages. These are the following stages:

- *Pre-build scripts:* Code runs that manipulates the source code and prepares it for the next process. An example of this is the pre-build script that copies the source code to a `build/` folder. This step allows us to transform the source code that is going to be used in the next stage without touching the original files. If we were to change the original files in place, we would also have to undo the replacements. This could prove problematic if a build fails, after which source control is the only way to recover the files.

Section	Section Name	Relative Complexity
6.2.1	Global CSS	simple
6.2.2	Compatibility	simple
6.2.3	Tagname renaming	simple
6.2.4	Theming	simple
6.2.5	Non-string Attributes	medium
6.2.6	Complex Attributes	complex
6.3.1	ng-deep	simple
6.3.2	createCustomElement	simple
6.3.3	EventEmitters	simple
6.3.4	Hierarchical Injectors	hard
6.3.5	ngOnInit	medium
6.3.6	Casing in attribute names	simple
6.3.7	Angular directives	simple
6.3.8	<ng-content>	simple
6.3.9	Angular Attribute Order	medium
6.3.10	Bundling Angular Imports	hard
6.3.11	Angular Ivy	hard
6.4.1	Reduce time searching for CSS	simple
6.4.2	Move CSS searching to initial load	simple

Table 6.1: Sections in chronological order along with their relative complexities

- *Building:* We build the Angular project. In this step, the code in the previously created `build/` directory is built into an Angular bundle. When this bundle is loaded on a page, the Web Component library is initialized, and the various Web Components are ready for use.
- *Bundling:* Build artifacts generated in the previous step are spread out over several files. Since we want to provide developers with a single bundle that contains all of the required code, we bundle these various files up into a single file.
- *Generating of wrappers:* The various wrappers for JS frameworks are generated. This process is described further in Section 6.5.

6.2 Web Component Issues

In this section, we document issues faced that were related to Web Components, as well as our solutions to them. These are issues that are not at all related to the Angular framework and are likely to be faced in other similar projects.

6.2.1 WC1: Global CSS

Problem: Angular components have a property called `encapsulation`[1]. This property determines how CSS styles are applied to the component. It has three possible values:

- *ShadowDom:* Global styles are not applied to the component. Only the component's own styles are applied to it.
- *Emulated (default):* Global styles are applied to the component as well as its own styles. Other components' styles are not applied to it.
- *None:* Global styles, a component's own styles, and other components' styles are applied to this component.

In the 30MHz codebase, the default (or Emulated) value is used, meaning that both global and component-specific styles are applied to it. This is done by putting both of them in a global stylesheet. This stylesheet then has component-specific selectors added to it, making sure that styles are always scoped to a specific component. An example of this process can be seen in Listing 6.1 and Listing 6.2.

[1] `https://angular.io/guide/view-encapsulation#view-encapsulation`

When migrating the Angular components to Web Components, we ensure the components' contents are rendered within a ShadowRoot [1]. A ShadowRoot is a separate root within an HTML document that contains its very own document. This document is entirely separated from the parent document, meaning it is not influenced by global styles in that document. This effectively separates the component from the rest of the DOM, thereby also removing the ability of the global 30MHz stylesheet to be applied to it.

Solution: When a component is rendered, we find the global stylesheet on the page. We then copy it into a Constructable Stylesheet [2] if it has not already been copied. Constructable Stylesheets are a method of creating CSS stylesheets in JavaScript. These stylesheets can then be used together with the `adoptedStylesheets` [3] JavaScript property of a ShadowRoot. Any Constructable Stylesheet placed in the `adoptedStylesheets` array is applied to the ShadowRoot similar to how any `<style>` tag in a document is applied to that document. The advantage to using this method is that Constructable Stylesheets are simply references that can be re-used by the browser, contrary to regular `<style>` or `<link rel="stylesheet">` tags, which are parsed from scratch every time the browser encounters them. This means that every time a new instance of a component is created, instead of copying a `<style>` or `<link rel="stylesheet">` tag and having the browser parse it from scratch (which takes approximately 16ms), we instead add the stylesheet to the `adoptedStylesheets` property of the ShadowRoot and have an already-parsed stylesheet applied to the ShadowRoot. We use this method to apply the global 30MHz stylesheet in the component's own ShadowRoot as well by copying that stylesheet to every component's ShadowRoot. Normally this would incur a heavy performance impact, but because of the use of `adoptedStylesheets`, the performance impact is minimal.

```
1 // my-component.html
2 <my-component></my-component>
3
4 // my-component.css
5 :host {
6   color: red;
```

[1] https://developer.mozilla.org/en-US/docs/Web/API/ShadowRoot

[2] https://developers.google.com/web/updates/2019/02/constructable-stylesheets

[3] https://developers.google.com/web/updates/2019/02/constructable-stylesheets#using_constructed_stylesheets

```
}
```

Listing 6.1: An example of uncompiled source code for a component

```
// my-component.html
<my-component _ngcontent-uix-c290></my-component>

// my-component.css
[_ngcontent-uix-c290] {
  color: red;
}
```

Listing 6.2: An example of compiled code for the component in Listing 6.1.

6.2.2 WC2: Compatibility

Problem: While browser support for Web Components is relatively widespread as of this case study [1], it is not yet universal. Additionally, Safari has chosen not to implement support for *Customized Built-In Elements*[2]. This feature allows for extending built-in HTML elements, allowing developers to extend already-existing elements such as the `HTMLInputElement` and others. Since the 30MHz dashboard makes use of components that extend native elements (in the form of directives[3]), we need this feature to make the CC UI library work.

Solution: We add polyfills to the final JS bundle. These are files that add support for unsupported features by implementing them in JavaScript. If the feature a polyfill intends to provide is already supported by the browser, it will fall back to the built-in version. This ensures that they pose little to no performance impact if a given feature is already supported. In particular we use the `custom-elements`[4] and `custom-elements-builtin`[5] polyfills. These add support for Web Components (aka Custom Elements) to browsers that do not have it. Additionally, they add support for the previously mentioned *Customized Built-In Elements* feature.

[1] `https://caniuse.com/?search=components`
[2] `https://www.chromestatus.com/feature/4670146924773376`
[3] `https://angular.io/guide/attribute-directives`
[4] `https://www.npmjs.com/package/@ungap/custom-elements`
[5] `https://www.npmjs.com/package/@ungap/custom-elements-builtin`

6.2.3 WC3: Tagname renaming

Problem: As per the Web Components specification, all Web Components are required to have a hyphen in their tag name [1]. Angular components, on the other hand, do not have this requirement. Because of this, there are some components in the 30MHz codebase without a hyphen in their name. In order to export them as Web Components, we need to come up with a tag name with a hyphen in it. In this case we decided to prefix every component with `cow-` (for example `<cow-checkbox>`). This renaming, however, leads to an issue with components that are being used inside other components. For example say the `TripleCheckbox` component renders three checkboxes. Example source code for such a component can be seen in Listing 6.3. If such a component is rendered as a Web Component, it will attempt to render the `<checkbox>` HTML tag, not knowing that it has been renamed to `<cow-checkbox>`. The result is an empty component.

Solution: We run a pre-build script that replaces the names of components that will be used in the UI library with their prefixed variant. This pre-build script is run in the first stage of our build process, as described in Section 6.1. We make sure not to replace native HTML elements by matching the found HTML tags against a list of known native HTML elements.

```
1 <checkbox id="checkbox-1"></checkbox>
2 <checkbox id="checkbox-2"></checkbox>
3 <checkbox id="checkbox-3"></checkbox>
```

Listing 6.3: Example source code for a `TripleCheckbox` component.

6.2.4 WC4: Theming

Problem: 3rd party developers have expressed the wish to apply custom theming to their apps. In order to make this possible, we need to find a way to apply a single theme across all components on the page, regardless of ShadowRoots.

Solution: We make use of *CSS Custom Properties*[2]. These are effectively CSS variables that are defined for the whole document, including ShadowRoots. An example of the application of CSS Custom Properties can be seen in Listing 6.4. By changing the styles of the underlying Angular components to use CSS Custom Properties when available, we

[1] `https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-core-concepts`

[2] `https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties`

are able to provide theming. An example of this can be seen in Listing 6.5.

```
function setPrimaryColorTheme(color: string) {
  document.documentElement.style.setProperty('color-primary', color);
}

setPrimaryColorTheme('red');
```

Listing 6.4: Applying CSS Custom Properties to the document

```
my-component {
  /**
    * Tries to use the --color-primary variable if available,
    * falls back to blue when it is not defined.
    */
  background-color: var(--color-primary, blue);
}
```

Listing 6.5: An example of a component making use of CSS Custom Properties

6.2.5 WC5: Non-string Attributes

Problem: As mentioned previously, it is not possible to pass non-string attributes to Web Components using just HTML. This presents an issue since some Angular components expect a non-string attribute to be passed. Examples include but are not limited to a boolean, a number, a JavaScript object with an `alignment` property, and a `Date` instance. While this is a problem that will be solved by our JS framework wrappers and particularly in Section 6.2.6, the CC UI library should be at least usable by itself as well.

Solution: In this scenario, there is no access to JavaScript. This results in us being unable to make use of any solutions to this problem that utilize the setting of attributes or properties through JavaScript. The solution we came up with was to allow developers to optionally pass JSON to components by prefixing the attributes with `json-`. When this is done, the attribute value is parsed through `JSON.parse`, after which the corresponding property is set on the Web Component. This gives developers a way to pass most of the previously presented before. To allow developers to pass `Date` instances, we make it possible to pass strings, which we parse into Dates. While this is not a great workaround to have, especially since it only takes care of the `Date` class and not, for example, the `RegExp` class. The only way to fix this issue for all classes is to provide a method similar to Python `pickle` [1], which, as mentioned on the `pickle` documentation page, is not

[1] `https://docs.python.org/3/library/pickle.html`

secure. We also feel like this problem is not prevalent enough in our case to warrant such a solution.

6.2.6 WC6: Complex Attributes

Problem: Continuing with the same problem, we now need to develop a way to solve the problem for even more situations, this time being able to use JavaScript since the solution will be implemented into the JS framework wrappers. Some of which include the passing of an object reference or an HTML element reference. This will never be possible through JSON since the reference needs to be preserved. Instead, we make use of JavaScript this time. Our goal is to allow a parent component written in the language of the JS framework to be able to pass its properties down to its child. This parent component is essentially just a passthrough component whose set of properties is identical. Its only purpose is to pass on these components and to provide a component native to the JS framework. Another thing to keep in mind is that we need all properties to be defined before the child component performs its first render. This issue is present simply because of the way the original Angular components are written, which assumes that they will only receive attributes once and that they will never change.

Approaches: There are a few approaches to solving this issue. These can be grouped into three categories, an visual representation of which can be seen in Figure 6.1. These categories are the following:

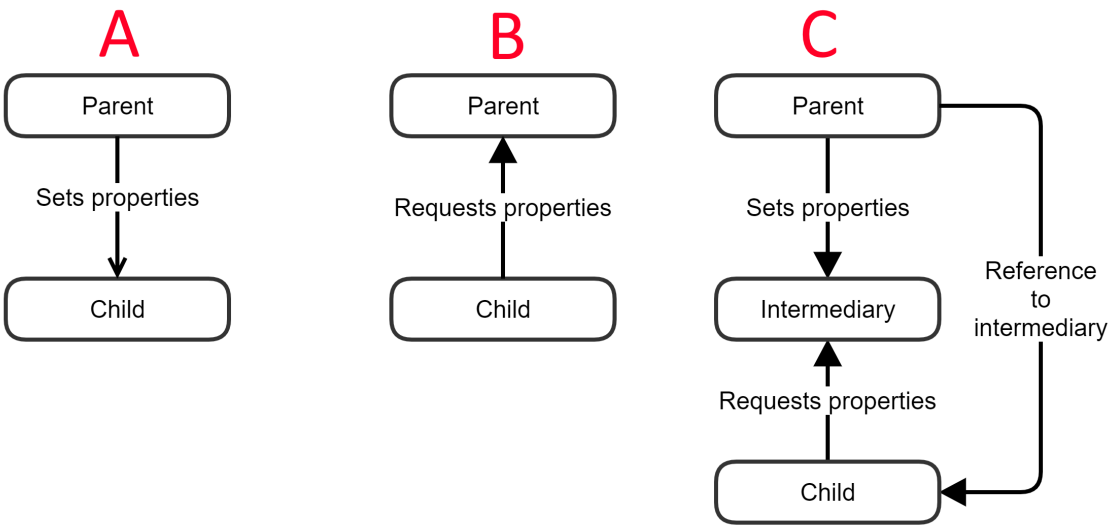


Figure 6.1: Categories into which approaches to passing complex attributes can be grouped

- *Parent → child (marked A in Figure 6.1):* The parent node gets a reference to the child during the rendering process. Then the parent sets the properties on the child.
- *Child → parent (marked B in Figure 6.1):* The child gets a reference to the parent during the rendering process. It then requests its properties from the parent.
- *Parent → intermediary → child, Child → intermediary → parent (marked C in Figure 6.1):* An intermediary takes the properties from the parent. The parent then provides the child with some way to find the intermediary, after which the child can get the properties from the intermediary upon rendering.

The first approach would be the easiest, but this approach is not always feasible. Many JS frameworks do not provide such low-level access to the to-be-rendered component. Instead, they often provide callbacks with a reference to the element after it has been connected to the DOM. Since the properties of our components need to be defined before they have even been rendered, this approach does not work for us.

The second approach presents similar problems. While in some JS frameworks, it is possible to get a reference to the parent component, JS frameworks that use a virtual DOM such as ReactJS and Vue [1] do not have a real parent component instance. Instead, the parent is just an abstract concept.

This leaves us with the last approach—the creation of an intermediary object which holds the properties. The child is then given some way to get a reference to the intermediary (bypassing the problem of the second approach), after which it can get the to-be-applied properties from the intermediary. As long as we make sure the child has a way to find the intermediary access before it has been rendered to the DOM, we are able to fulfill the requirement of defining all properties before the first render.

Implementation: Our implementation consists of several steps. We start by creating a class which we will call *Intermediary*. This class has an instance manager attached to it, which we will call the *IntermediaryManager*. Our JS framework wrapper code will be wrapping around the basic CC UI library. Since the CC UI library does not export the IntermediaryManager, we are unable to get a reference to it from our wrapper (aka the parent). In order to still get a reference to it, we want to store it globally. Because storing such properties on the `window` object can result in collisions and is unreliable, we will be storing it as a property of the defined Web Components. This means that we are able to access the `IntermediaryManager` property on the `customElements.get('cow-checkbox')` object

[1] `https://vuejs.org/`

and get a reference to the IntermediaryManager from both the parent side and the child side.

Now that we have taken care of this issue, we are able to start using it. We make the parent create an instance of an Intermediary. This Intermediary gets a simple string ID. We are then able to look up the ID in the IntermediaryManager and get the corresponding Intermediary. This ID is passed to the child, allowing it to look up this Intermediary.

For passing the actual values, we make use of references. For each of the parent's properties, we pass the value to the Intermediary. The Intermediary then generates a unique string representing that value. If the Intermediary already knows the value, the same string is returned. Internally it maps this string to the value. We then pass this string to the child instead of passing the original complex value (which would not work). The child then receives the value and resolves it back to a complex value by consulting the Intermediary. Through this process, the child component is able to receive complex values from its parent through simple HTML string attributes.

6.3 Angular Issues

In this section, we describe any Angular related issues we faced. These are issues that were specific to Angular and are unlikely to be faced in similar projects when a different JS framework is being targeted.

6.3.1 A1: ng-deep

Problem: Angular provides the `ng-deep` CSS selectors [1]. Where regular CSS selectors stop at the ShadowDom boundary, meaning that a component will never be able to have a selector apply to the DOM of another component, the ng-deep selector does allow for this. This selector is deprecated but still in use in the 30MHz codebase. It is a CSS selector that is implemented in JavaScript by Angular that does not work outside of Angular environments (including the Web Components environment). As such, we need to remove it.

Solution: The fix for this issue was fairly simple. Any instance of ng-deep had to be removed. While there has been some talk around browser support for a similar deep selector [2], with both `::shadow` and `/deep/` making it into Chrome, they have since both

[1] `https://angular.io/guide/component-styles#deprecated-deep--and-ng-deep`
[2] `https://drafts.csswg.org/css-scoping/`

been removed [1]. As such, we had to come up with a workaround. Since the only way to effectively communicate from a component to child components is properties, we changed the code to use properties instead. An example of this change can be seen in Listing 6.6 and Listing 6.7.

```
// parent-component.html
<child-component></child-component>

// parent-component.css
::ng-deep div {
  color: red;
}

// child-component.ts
@Component({
  ...
})
class ChildComponent {
  ...
}
```

Listing 6.6: A component before the ng-deep change

```
// parent-component.html
<child-component red></child-component>

// child-component.ts
@Component({
  ...
})
class ChildComponent {
  @Input() red: boolean;

  constructor(private _elementRef: ElementRef) {
    if (this.red) {
      _elementRef.nativeElement.classList.add('red');
    }
  }
}

// child-component.css
:host[red] {
  color: red;
```

[1] https://developers.google.com/web/updates/2017/10/remove-shadow-piercing

```
21 }
```

Listing 6.7: A component after the ng-deep change

6.3.2 A2: createCustomElement

Problem: The main export of the Angular Elements library is the `createCustomElement` function [1]. This function takes an Angular component and turns it into a Web Component. It does this by extending an `HTMLElement` base class and applying all Angular component features on top of it. However, this function does not offer the ability to change the base class from an `HTMLElement` into anything else. As mentioned before, the 30MHz dashboard makes use of some elements that extend native elements. For example, the 30MHz input field extends the default HTML input field and only adds styling, preserving any built-in accessibility features provided by the browser. When migrating this Angular component to a Web Component, we also wish to preserve these same features. This can be done by extending built-in HTML elements [2]. The `createCustomElement` function does not provide this option, causing us to be unable to create such an element. There is an open feature request for this option at the time of writing of this paper [3].

Solution: We have no choice but to implement this option ourselves. This means we have to copy the entire source code of the `createCustomElement` function, along with many of its dependencies, since very few of them are exported. After this, we change the function to allow us to pass such an option. It should be noted that this does introduce additional difficulties with upgrading Angular Elements. Instead of upgrading the package itself, the copied source code will have to be replaced with the new source code. On the other hand, the referenced issue might be fixed, after which the process of copying source code is no longer necessary.

6.3.3 A3: EventEmitters

Problem: Angular implements `EventEmitters` [4]. These are classes that are able to emit events, as well as being able to be listened to. When the `emit` function is called, the passed value is sent directly to those functions that added an event listener to it. Note that this behavior is different from regular event emitters, which emit a `CustomEvent`,

[1] `https://angular.io/api/elements/createCustomElement`
[2] `https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements`
[3] `https://github.com/angular/angular/issues/19108`
[4] `https://angular.io/api/core/EventEmitter`

which contains the actual value in the `detail` property. A lot of our Angular code relies on the value being directly emitted and it not being wrapped in a CustomEvent.

When a component is migrated to a Web Component, however, this emitted value is wrapped in a CustomEvent. Since the Angular code relies on this value being directly emitted, errors occur. In order to get around this issue, we need to make sure that internal code that listens to such EventEmitters receives the value itself, while external code (such as a 3rd party listening to a Web Component) receives the value wrapped in a CustomEvent.

Solution: We run a script that iterates over the source files, looking for any location where an event listener is being added to such an EventEmitter. Once we find one, we wrap the callback in an unwrapping function that strips away the CustomEvent and returns just the `code` value. An example of this change can be seen in Listing 6.8.

```
1 // before
2 (valueChanged)="myHandler($event)"
3
4 // after
5 (valueChanged)="myHandler(unwrapEvent($event))"
```

Listing 6.8: A change made to an event listener. The definition of `unwrapEvent` can be seen in Listing 6.9

```
1 function unwrapEvent(event) {
2   if (event instanceof CustomEvent) {
3     return event.detail;
4   }
5   return event;
6 }
```

Listing 6.9: The `unwrapEvent` function

6.3.4 A4: Hierarchical Injectors

Problem: Angular makes use of a feature called *Dependency Injection* [1]. This allows a parent module to provide its children with an instance of a particular dependency class. This class instance is shared among the module and its children. Generally, only modules provide their children with dependencies, where modules are simply collections of components that serve some common purpose. This dependency injection feature can also be leveraged to have a given component provide its own instance of a class only to its direct

[1] `https://angular.io/guide/dependency-injection`

children. This means that every instance of that component gets its own separate instance of the dependency, which it then shares with its children, and not one that is shared across all components in the module. This is called *Hierarchical Dependency Injection*[1] and it is a feature that is utilized by 30MHz in some areas.

Angular Elements does not support this feature intentionally [2]. Instead, it only supports the use case where modules provide their components with dependencies. The reason for this is that every component migrated with Angular Elements is mounted to the DOM as its own root. It does not have a concept of a parent component and is unable to look up the injector of its parent. While the pattern of Hierarchical Dependency Injection is not recommended for use with Angular Elements [3], it is still a pattern used by 30MHz, and as such, we need to support it in order for the CC UI library to work.

Solution: Our goal in fixing this problem is to have a component injector inherit from its parent injector, which will facilitate Hierarchical Dependency Injection. To do this, we need to find the parent when the child is being rendered. After this, we can extract its injector, craft a new injector that combines the child and parent injector, and finally supply this new injector to the child. An example of this process can be found on StackBlitz [4].

We first need to find the parent. This process is relatively straightforward. When the child is being rendered, we travel up the DOM tree until we find a node with specific properties that only Angular elements have. We then move on to the next stage of finding its injector.

While the finding of a node's injector is straightforward in development mode since Angular exposes a `window.ng.getInjector` function, this process is a lot more complicated in production mode. To find it, we first need to find the component's hidden Angular properties. These can be found under the component's `__ngContext__` property. Depending on the environment, this can either be an object containing the `tNode` and `lView` properties or an array that contains them at a magic offset. The `tNode` and `lView` are internal representations of a bunch of Angular-specific properties for the component.

We are unable to access the original injector of the parent component since it is hidden in Angular-internal code. Instead, we need to use the `tNode` and `lView` to craft a new injector that will do the same thing as the original injector. However, in order to

[1] https://angular.io/guide/hierarchical-dependency-injection

[2] https://github.com/angular/angular/issues/24824#issuecomment-404399564

[3] https://github.com/angular/angular/issues/24824#issuecomment-404399564

[4] https://stackblitz.com/edit/ngelements-issue-40104?file=src%2Fapp%2Fbar%2Fbar.component.ts

craft this new injector class instance, we need a reference to that same class. While a `Injector` class is exported from the Angular package, this is not actually the injector we want. Angular actually has two types of injectors, one of which is the previously mentioned `Injector` and the other is the `NodeInjector`. This `NodeInjector` is only used internally, and it is the injector we want. To get a reference to it, we access the `injector` property of a fake component created by a `ComponentFactory`. Since the `ComponentFactory` is also not exported, we need to get a reference to it through the global injector. We now finally have a reference to the `NodeInjector` class, which allows us to re-create the parent's injector.

We now merge this injector with the child injector. This is a relatively simple process. When a request for an injected value comes in, we first look for it in the child. If the child does not have it, we look at the parent injector.

We now need to make sure Angular actually uses this injector we just created. To do so, we need to override the component's default element strategy (`NgElementStrategy`). This element strategy is a class provided by Angular that manages the connection between the DOM and the underlying Angular component. Since the `NgElementStrategy` class is also not exported by Angular, we need to find a reference to it somewhere. To do so, we create a fake component and read its `ngElementStrategy` property. We can now extend the class, replace the injector and provide it to the component. A complete code example of this process can be found in Listing 1.

While we attempted to maintain compatibility with the Angular source code by not copying code but by instead getting references to them at runtime, Angular updates might cause the current solution to this problem to break. While the underlying idea for the solution is unlikely to be rendered impossible, the methods we use to achieve it might change, and the code might have to be partially re-written.

6.3.5 A5: ngOnInit

Problem: Angular Elements does intentionally not guarantee the order in which attributes are set on an element (even initial attributes) [1]. This means that attributes can be set on an element both before or after its main init hook (`ngOnInit`) is called. An example of this process can be seen in Listing 6.10. While this is not a problem if attributes are only used to handle visual state, they can cause significant problems when used for

[1] `https://github.com/angular/angular/issues/29050`

component configuration. For example, if a component performs a fetch request to the server and takes a `URL` property that determines the target URL, it is essential that this property be set before the main hook runs. Quite a few components in the 30MHz codebase have a similar setup. As such, we need to guarantee that a component will always have the complete set of initial properties set before its main hook is called.

Solution: We know that, while the order of attribute setting is not guaranteed, we are guaranteed the fact that they will run sequentially. Since JavaScript is a single-threaded language and all attribute setting calls are synchronous, we know that all attributes will be set once the main thread is free again. For this, we use the global `window.requestAnimationFrame` JavaScript function. This function takes a callback and calls it when the main JavaScript thread is free to take on new work. We now firstly replace the component's `ngOnInit` function with an empty function, ensuring that when Angular calls it, the component's main hook is not actually run. We then call `window.requestAnimationFrame` and pass it the original `ngOnInit` function. Now we can guarantee that the `ngOnInit` function is called after all attributes have been set.

```
// HTML source file
<my-element foo="bar" bar="baz" />

// can be transformed into any of the following:
// 1
const element = document.createElement('my-element');
element.setAttribute('foo', 'bar');
element.setAttribute('bar', 'baz');
parent.appendChild(element);

// 2
const element = document.createElement('my-element');
parent.appendChild(element);
element.setAttribute('foo', 'bar');
element.setAttribute('bar', 'baz');

// 3
const element = document.createElement('my-element');
element.setAttribute('foo', 'bar');
parent.appendChild(element);
element.setAttribute('bar', 'baz');
```

Listing 6.10: HTML source code and its Angular Elements equivalent

6.3.6 A6: Casing in attribute names

Problem: In the process of migrating Angular components to Web Components, Angular Elements maps all input properties from camelCase casing to kebab-case. For example the input property `myInputProperty` is set through the `my-input-property` HTML attribute. The reason for this change is that HTML attributes are case-insensitive. To HTML `myInputProperty` is identical to `myinputproperty` and `MYINPUTPROPERTY`. This mapping of input properties presents some issues to us. Internal Angular elements still use the camelCase variant to set properties on their child components. Since the Web Component variants do not recognize the camelCase variant of the property anymore, they are ignored.

Solution: We solve this issue by making sure the Web Components also accept the camelCase variant. As HTML is case insensitive, there is no point in checking the casing of the passed attribute. Instead, we convert it to lowercase and compare it against the lowercase version of the original camelCase input property. In the previous example `myInputProperty`, `myinputproperty`, `MYINPUTPROPERTY`, and `my-input-property` would all refer to the input property `myInputProperty` on the Angular component.

6.3.7 A7: Angular directives

Problem: Angular has two types of elements that appear in the DOM. The first type is the Component, which looks for a given selector or tag name and replaces the original HTML element. For example an `AppRootComponent` with the selector `'app-root'` will look for an `<app-root>` HTML element and replace it with the Angular component instance. The second type is the Directive. Similarly, this looks for a selector, but instead of replacing the original HTML element, this simply mounts to it and runs its own code on it. An example of this would be a `Blink` directive that looks for the `'blink'` HTML class. When mounted, it periodically hides and un-hides the component.

Angular Elements only supports the conversion of Components to Web Components, not the conversion of Directives. Since there are some elements in the 30MHz codebase that use Directives, we need to make sure that these are supported as well.

Solution: While this might sound like a challenging problem since these are entirely different elements, the fix for this issue is surprisingly easy. Under the hood, Angular stores the definition of a Component in the `ɵcmp` property. This is also the property Angular Elements accesses to do the migration from Angular components to Web Components.

Similarly, the definition of Directives is stored in the `ɵdir` property. By simply copying the value of the `ɵdir` property to the `ɵcmp` property, we are able to trick Angular Elements into thinking a directive is a component. Surprisingly, this works, and the directive works perfectly.

6.3.8 A8: <ng-content>

Problem: Angular uses the `<ng-content>` tag for content projection. Content projection is the ability for a component to take a set of children, which it can then place anywhere in its DOM tree. This is effectively the same as the HTML `<slot>` tag [1]. An example of content projection can be seen in Listing 6.11. Content projection works fine in most scenarios, but for unknown reasons, it sometimes does not work. The result is that child elements simply do not show up.

Solution: Our solution is once again quite simple; we take advantage of the fact that the `<ng-content>` and `<slot>` tag do the same thing and append a `<slot>` tag to every occurrence of an `<ng-content>` tag in the source code. This ensures that when the `<ng-content>` tag does not work, the `<slot>` tag takes over instead. Since the browser only allows a given child element to be projected into one spot (*i.e.*, multiple `<slot>` tags do not result in multiple copies of the child element), this approach will not cause any problems in cases where `<ng-content>` does work.

```
// parent-component.html
<child-component>
  <span id="my-span"></span>
</child-component>

// child-component.html
<div id="my-root">
  <ng-content></ng-content>
</div>

// Effective DOM tree
<parent-component>
  <child-component>
    <div id="my-root">
      <span id="my-span"></span>
    </div>
  </child-component>
```

[1] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/slot

```
18 </parent-component>
```

Listing 6.11: HTML source code and its Angular Elements equivalent

6.3.9 A9: Angular Attribute Order

Problem: As previously described in Section 6.3.5, the order in which Angular attributes are set is unknown. While we have fixed this issue for our Web Components in this section, this same issue presents itself again in the writing of an Angular wrapper. This time the problem is that components will be rendered before they have all of their input properties set. This leads to the apparent issue where the wrong contents are rendered.

Another problem stems from our approach in Section 6.2.6. We pass a unique ID to the child that is used to find the intermediary. When the child now receives an attribute that starts with the special reference prefix, it looks up the given reference by finding the intermediary and reading the attribute value. However, if the child receives such an attribute before having received the unique ID of the intermediary, it is unable to resolve the value. Since the order in which Angular attributes are passed is unknown, this situation arises very often.

Solution: We get around these issues by handling the appending to the DOM ourselves. Instead of having our Angular wrapper render the actual Web Components, we instead have them render a *Renderer* component. This component is then passed the tag name of the Web Component. We are now able to precisely control when a component is appended to the DOM, as well as which attributes it gets and in what order. Since the *Renderer* takes the place of the Web Component, it now receives all attributes. We wait until it has received the very last attribute before we decide to create the child component. After creating it, we apply all attributes the *Renderer* has received to the child, after which we append it to the DOM. Since this process allows us such fine-grained control over the rendering cycle, we are able to render the Web Component without any issues.

6.3.10 A10: Bundling Angular Imports

Problem: Angular provides a few ways to build projects. Two of which are in use by us. The first option is to build a project as an application. This bundles everything into a combination of browser-specific bundles. These bundles can then smoothly be loaded by including them in the browser. This is the option we use for the Web Component library. The second option is to build a project as a library. Building a project as a library

preserves all typing information and allows it to be used by another Angular project. Since we are building an Angular project, this is the option we need if we want to be able to provide typings to developers who use our Angular wrapper.

However, we run into an issue after building. 30MHz uses the Font Awesome Pro [1] package for its icons. This is a licensed package that can only be downloaded when a valid key is presented. Since some of these icons are also used in the CC UI library, it needs to be bundled into the library to be able to work. Angular refuses to do this. They recommend using `peerDependencies` [2] instead. While their reasoning for this is valid, it does not apply in our case. Since third-party developers are unable to install the Font Awesome Pro package without a license, they would run into an error when installing the package. Previously, Angular had the `embed` option, which allowed for the embedding of a given set of JS packages. This was eventually replaced with `bundledDependencies` [3], and a little while later, it was deprecated [4]. This leaves us with no native option to bundle our dependency.

Solution: We fix this issue by programmatically changing the build artifacts Angular outputs. There are three types of build formats, the `FESM2015` (or flattened `ESM2015`), `ESM2015`, and `umd` formats. Bundling the Font Awesome Pro source code is fairly trivial for the `FESM2015` and `umd` formats since they are both single files. The `ESM2015` format is a bit different. It consists of a folder that is essentially a clone of the source code, with every TypeScript source file being replaced with a compiled JavaScript version and a `.summary.json` file. This `.summary.json` file functions similar to a `.d.ts` file, in that it provides type information of the file. In fixing our issue, we want to preserve this same folder structure. This means that we can not simply pass the entry point to a bundler and be done. On the other hand, we are also not able to feed every individual JavaScript file to the bundler. This would result in every separate file including its own copy of the Font Awesome Pro library, leading to a significant bundle size. Instead, we create a central folder in the `ESM2015` folder in which we store the Font Awesome Pro JavaScript bundle. This folder functions the same as a `node_modules` folder. We now replace every reference in the source files to point to this central location instead.

Problem: We now run into another problem. When the library that was just built is used in an Angular project, the Angular compiler scans over all of its `.summary.json` files

[1] `https://fontawesome.com/pro`
[2] `https://github.com/ng-packagr/ng-packagr/blob/v10.1.0/docs/dependencies.md`
[3] `https://github.com/ng-packagr/ng-packagr/issues/1106`
[4] `https://github.com/ng-packagr/ng-packagr/commit/0c52486`

to build an AST with type information. It then finds the values that are exported, looks up their type values in the AST, and exports those that correspond to them. In performing this AST building process, the Angular compiler scans our included Font Awesome Pro folder for `.summary.json` files in order to extract definitions. Since those do not exist, an error is thrown.

Solution: The obvious approach to this problem is the following. We generate and bundle along the `.summary.json` files. This will provide Angular with the definitions it needs. The downside to this is that Angular will recursively perform this AST building process, meaning that it will first scan Font Awesome Pro, then its dependencies and their dependencies. All of these files would need to be included in the bundle simply to ensure the Angular compiler does not throw an error.

Instead, the solution is to simply remove any reference to our bundled Font Awesome Pro library from the `.summary.json` files. Now that there is no longer a reference, the Angular compiler will simply skip over it. Since the Font Awesome Pro library is not exported, the Angular compiler never has to export any of its type information, meaning it does not actually need this data.

6.3.11 A11: Angular Ivy

Problem: Angular has released a new version of its compiler called Ivy [1]. This Ivy compiler is set to replace the previous View Engine compiler. Angular does not allow the use of Ivy for projects built as libraries for now [2] both because of compatibility reasons and because Ivy is not seen as stable enough as of the writing of this paper. This forces us to use the View Engine compiler instead. This compiler contains several bugs, one of which is causing the build to fail entirely in our case [3]. This bug is marked as `Fixed by Ivy`; however, we are unable to use Ivy.

Solution: The only other fix to this issue seems to be the disabling of AOT (or ahead-of-time compilation [4] and it is the solution we have applied to this problem. Unfortunately, the disabling of AOT introduces significant overhead during the loading of the built library, an issue that we have not been able to fix. As will be mentioned in the results section, this increases the load time of the Angular wrapper to an unacceptable level. We

[1] `https://angular.io/guide/ivy`
[2] `https://angular.io/guide/ivy#maintaining-library-compatibility`
[3] `https://github.com/angular/angular/issues/25424`
[4] `https://github.com/angular/angular/issues/25424#issuecomment-465643237`

have chosen to still disable AOT compilation as it shifted the issue from a blocking one (the inability to compile the code at all) to a performance issue.

6.4 Optimizations

After finishing the CC UI library, we started looking for performance optimization opportunities. By looking through the Chrome profiler trace, we were able to find some easy performance improvements. These are discussed in detail below.

6.4.1 O1: Reduce time searching for CSS

We provide developers with a CSS file they should include in their final `index.html` file. This file contains the styles required to make the CC UI library work. We refer to this CSS file as the CC CSS file from now on.

As mentioned in Section 6.2.1, we find the CC CSS file on the page and copy it into every component instance. Since there can be many more stylesheets than just the CC CSS file, we need to scan every stylesheet and check whether it is the one. This searching process consists of the following steps.

- Get a list of all stylesheets on the page, this includes both `<style>` tags and `<link rel="stylesheet">` tags.
- Iterate through every stylesheet.
- Iterate through every rule.
- Compare the current rule with a specific marker value that signifies the CC CSS file. If it matches, move on to the next step
- Add this stylesheet to the list of CC CSS files and move on to the next stylesheet.

While we did implement some caching, making sure this process only runs a single time, this process still has a significant performance impact. This performance impact scales linearly with the size of the stylesheet, meaning that a stylesheet with more rules takes longer to scan. It takes about 16ms to scan through a stylesheet of 1667 lines in order to find the marker rule on the machine mentioned in Section 5.4. In addition to scaling with the size of the stylesheet, this number also scales with the number of stylesheets. Since it iterates through every available stylesheet, including a big stylesheet will vastly increase

loading times. A rough formula for this performance impact in milliseconds can be found in the equation below, with SS being a set of all stylesheets and LOC being the number of lines of code.

$$t_{ms} = \sum_{i \in SS} \frac{i_{LOC}}{100}$$

We improve this process in two ways. The first step is to stop scanning after we find the marker simply. Instead of looking for other CC CSS files, we simply stop and return early. Since we know that there is only a single CC CSS file that applies to our Web Components, we can make this change. The performance impact of this change is hard to express in a single number since it largely depends on the stylesheets on the page, but this performance improvement saves about 1ms per 100 rules in stylesheets that are not the CC CSS file.

The second step is to ask the developers to help us in this process. We ask them to add a simple `cow` attribute to the `<style>` or `<link rel="stylesheet">` tag that contains the CC CSS file. Since the developer knows which file contains our supplied CSS file, they should easily add this attribute. At runtime, we check whether any stylesheet tags have a `cow` attribute. If there are, we can skip the entire process of finding the CC CSS file. Since this process was performed as the first component was rendered, this change saves about 16ms on the first component's render time.

6.4.2 O2: Move CSS searching to initial load

While the above fixes provide excellent performance improvements, it may very well be possible that the developer does not add the `cow` tag, preventing our performance improvements from applying. The performance impact of the CSS search is still considerable, and the fact that it is run during the rendering of the first component significantly increases its render time. Since these render times tend to be around 16ms by themselves, a 16ms increase is enormous.

We remove this performance impact from the first render by moving it to the moment the CC UI library is initialized. To ensure we do not add any time to the initial load, we wait until the browser is idle by calling `window.requestIdleCallback`. This ensures that the process of finding CSS is performed while the browser is idle instead of it blocking an important operation such as component rendering.

6.5 JS Framework Wrappers

To improve the developer experience, we wrote JS framework wrappers for a total of three JS frameworks, namely ReactJS, Svelte and Angular. In ReactJS, complex attributes for Web Components do not work natively. A wrapper for ReactJS was required in order to get the CC UI library to work in the first place. Since ReactJS provides good tooling when it comes to components and their properties, we felt it was a good idea to make use of this by providing typings for the CC UI library ReactJS wrapper. The second JS framework is Svelte. While Svelte works perfectly with Web Components out of the box, we created a wrapper to provide typings for the developers similar to ReactJS. The last JS framework is Angular itself. Angular also provides tooling, including built-in checking of component properties, which we felt we had to provide.

We also looked at different JS frameworks but found most of them to have little to no tooling for HTML elements. The UI libraries we looked at were Vue (v2 and v3) [1], Polymer [2], lit-element [3], and wc-lib [4].

In creating these wrappers, we started off by extracting the required component and type data from the components' source files. For example, the list of input properties, as well as their types and descriptions, need to be known. Similarly, all events emitted by the component, their types, and descriptions also need to be known. Finally, we also need to collect some metadata on the component itself, such as the name, tag name, and whether it has child elements. We extracted all this data by using the *typescript* package [5]. This package allows for the parsing of TypeScript and JavaScript code. This code is turned into an abstract syntax tree (AST) with added type information, after which we are able to iterate through it. We extract all this data and turn it into a common format to be re-used by the various scripts that generate JS framework wrappers.

After collecting this data, we are able to generate the various JS frameworks. Generating wrappers was a reasonably smooth process for most JS frameworks. It consisted of iterating through the extracted component data and generating source files written in the language of the JS framework. These source files were then fed into a bundler and bundled into the wrappers. This process went smoothly for both the ReactJS and Svelte

[1] `https://vuejs.org/`
[2] `https://www.polymer-project.org/`
[3] `https://lit-element.polymer-project.org/`
[4] `https://www.npmjs.com/package/wc-lib`
[5] `https://www.npmjs.com/package/typescript`

wrapper. During the process of creating an Angular wrapper, on the other hand, we faced a few challenges. These challenges are described in detail below.

7

Results

After collecting the metrics described in Chapter4 over the created CC UI library, we are able to compare the CC UI library to the original Angular components, the various JS framework wrappers, and various other UI libraries. In the following sections, the various metrics are broken down, and the results are compared between the various libraries.

7.1 Render Time

The render time metric allows us to evaluate the direct performance impact on users once the page has loaded. We first compare the CC UI library to the original Angular components and the other JS framework wrappers. This allows us to evaluate the performance impact added by the process of migration to Web Components, as well as the performance impact added by the JS framework wrappers. After this, we compare the CC UI library to the UI libraries listed in Table 4.2, allowing us to evaluate the performance of the CC UI library relative to UI libraries as a whole.

7.1.1 Cow Components

As mentioned in Chapter 5, we have measured three basic components in particular that every UI library contains. These are the Button, Switch, and Input. We have measured the time needed to render 1 instance, 10 instances, and 100 instances of this component. The various render times for the cow-components UI libraries with these numbers of components can be seen in figures 7.1, 7.2, and 7.3 respectively. We first take a look at the single-component render times. When we compare the performance of the CC UI library with the original Angular components, we find the CC UI library's median render time to be 81% and 100% higher for the Input and Switch components, respectively, with the

mean render time for the Button being 41% lower. Other than the Angular wrapper's Button (which is 100% slower), the wrappers' render times are very similar. The ReactJS and Svelte Button rendering times are 6% and 12% lower, respectively, with both the Input and Switch render times being between 172% and 190% higher. From this, we can conclude that, although there is a full Angular root running for each component, the performance impact for a single component is minimal. Now taking a look at the render times for 10 and 100 components, we start to see some big differences. The Web Components version can still keep up with the original components when it comes to rendering 10 components, being 5% faster, 144% slower, and 111% slower for the Button, Input, and Switch components, respectively. When rendering 100 components instances, however, it is eclipsed by the original components. Render times are 250%, 393%, and 265% slower for the Button, Input, and Switch, respectively. It seems that the impact of creating a new Angular root for each component does become significant with many components. Additionally, the render times for the various JS frameworks start to differ quite a lot. We see a trend of the ReactJS and Svelte wrapper growing further away from the Web Components version, with the average of the component render times increasing by 339% for the ReactJS wrapper and 597% for the Svelte wrapper. The Angular wrapper moves away even further, with the average of its component render times increasing by 735%. It seems that the performance impact for rendering a relatively small amount of components is minimal, while it scales up relatively quickly with a greater number of components. The fact that picking a JS framework to develop in is especially interesting, potentially costing a difference of 150ms over Web Components or 50ms over a different framework.

7.1.2 UI Libraries

We now compare the render times of the various UI libraries. Since the number of UI libraries we are comparing is very high (coming in at 29 total), showing them all in one figure makes for a very cluttered view. Instead, we compare a single component at a time. We have chosen to discuss the Button component in this section, however a complete overview can be found in Figures 1, 2, and 3. The render times of the Button component for the various UI libraries for 1, 10 and 100 components can be found in figures 7.4, 7.5, and 7.6 respectively.

We, first of all, find that there are large differences in render times even within UI libraries that share the same framework. For example the render time for a button in `react-bootstrap` is 57% faster than `material-ui` 64% faster than `semantic-ui-react`.

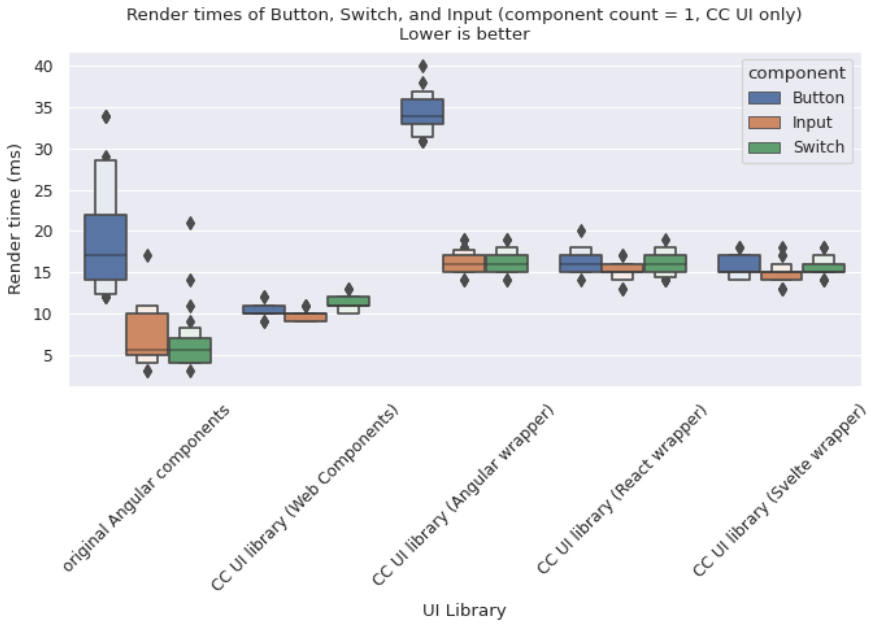


Figure 7.1: Render times of a single Button, Switch, or Input component (CC UI only)

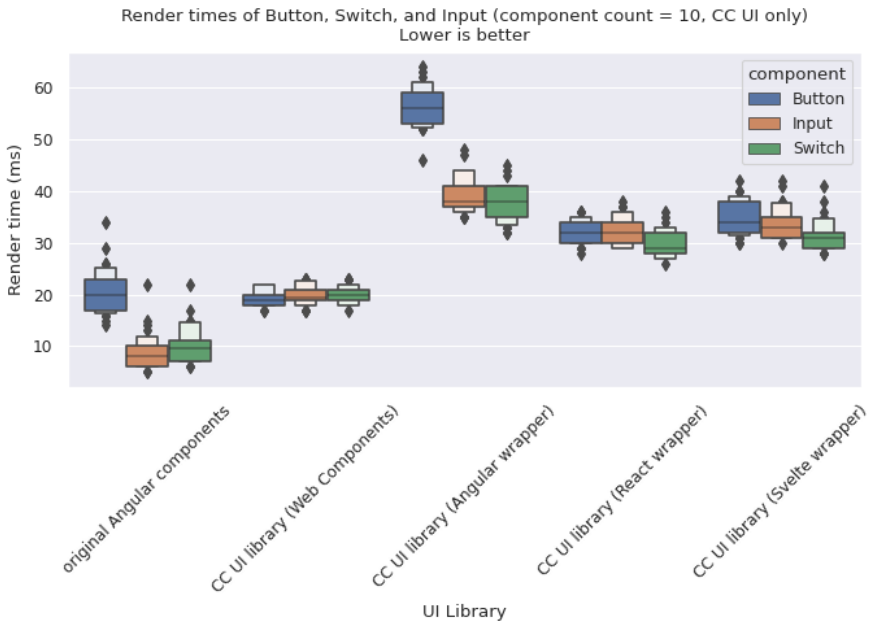


Figure 7.2: Render times of ten Button, Switch, or Input components (CC UI only)

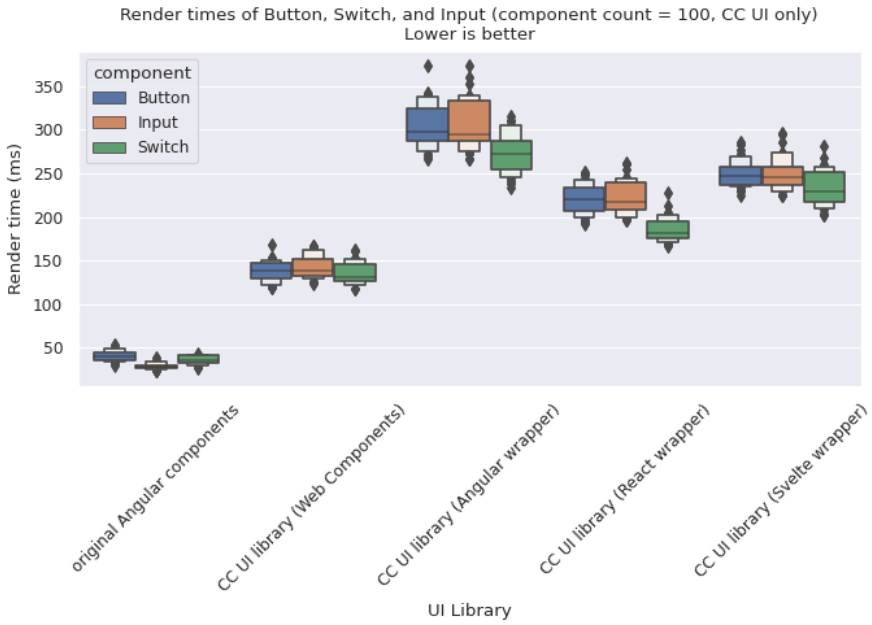


Figure 7.3: Render times of one hundred Button, Switch, or Input components (CC UI only)

In cases where this performance difference is relatively small, this has to do with the libraries themselves, but in a few cases, this has to do with the type of library. These libraries (`react-bootstrap`, `ng-bootstrap`, and `ngx-bootstrap`) make use of a CSS framework, as laid out in section 2.5. As such, they are significantly faster and are essentially in a different category from the CC UI library, which is JS-based.

When we ignore these outliers, we can draw some conclusions on the average render times of the various frameworks. We first take a look at the single-component render times in Figure 7.4. We can see that Svelte UI libraries are generally swift, together having an average render time of 11.6ms. This falls in line with other performance benchmarks [1]. After this, Vue (12ms) and the UI libraries using Web Components (19,8ms) are the fastest. Interestingly, Web Components are slower than UI libraries using Svelte. Since Web Components are a native technology, one would be lead to believe that they would be faster. This might have something to do with how the authors of the UI libraries created their Web Components. It could be that their approach imposes a significant performance impact. The following frameworks when it comes to render time performance are Angular (29ms) and ReactJS (35ms). They are pretty close in performance, both being significantly

[1] `https://rawgit.com/krausest/js-framework-benchmark/master/webdriver-ts-results/table.html`

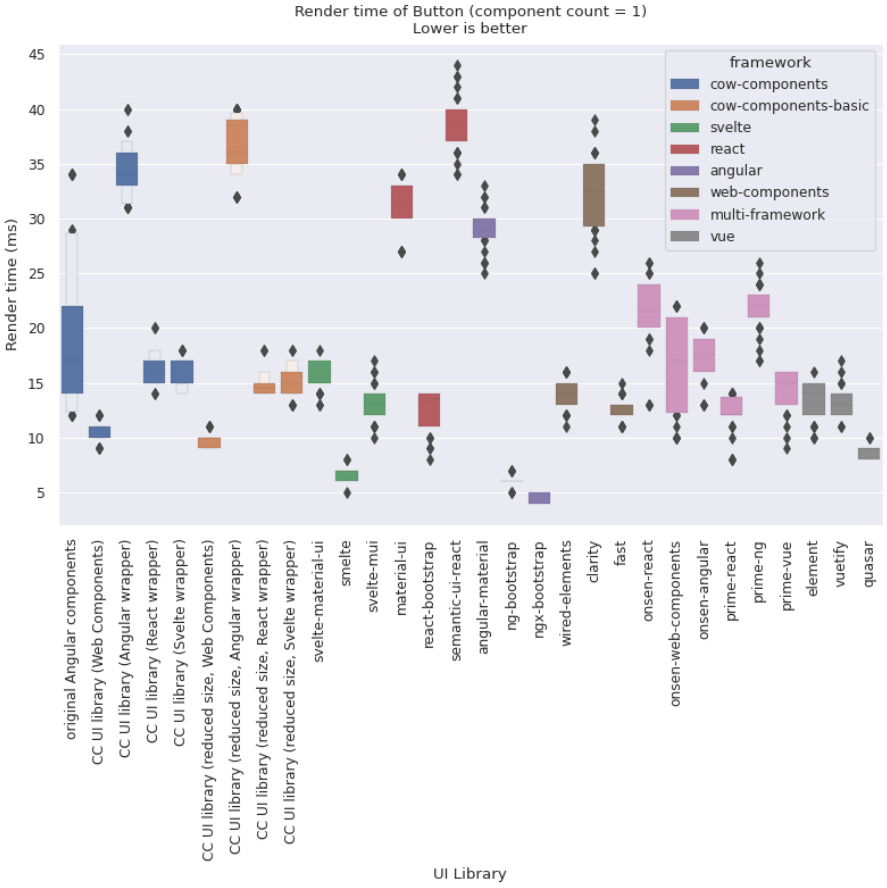


Figure 7.4: Render times of a single Button. The reduced size CC UI library is the build of the library with less components, as described in Section 5.6.

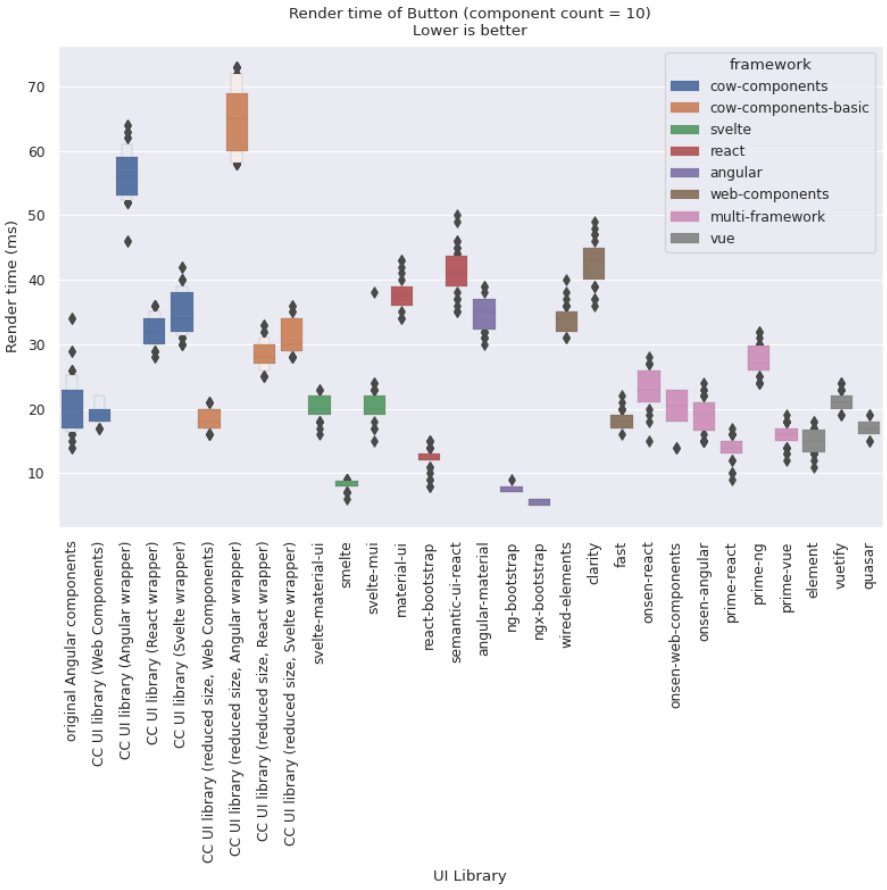


Figure 7.5: Render times of 10 Buttons. The reduced size CC UI library is the build of the library with less components, as described in Section 5.6.

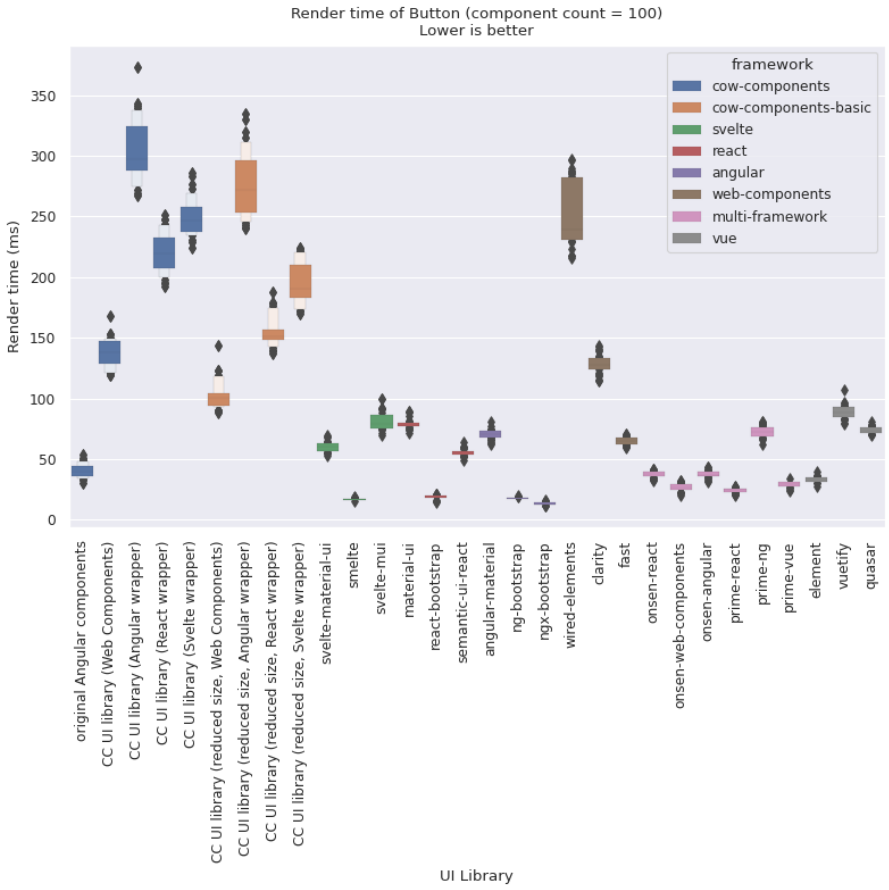


Figure 7.6: Render times of 100 Buttons. The reduced size CC UI library is the build of the library with less components, as described in Section 5.6.

slower than other frameworks. The previously mentioned performance benchmarks again support this.

We now apply our findings to the CC UI library JS framework wrappers. In our case, the Web Components version is the fastest simply because every other wrapper builds on top of this version. This means that it is impossible for another framework to be faster than it. We see this as well in the results, with the Web Components version coming in at 10ms. After that, the Svelte wrapper is the fastest, with a render time of 15ms. Interestingly, however, the ReactJS wrapper (16ms) is only slightly slower than the Svelte wrapper, while the Angular wrapper is significantly slower than both of them, coming in at 34ms. This is in contrast to what we just found, where both Angular and ReactJS were slow. It could be that the various internals of ReactJS that keep track of state and properties are slow. These are likely to be used a lot by regular ReactJS UI libraries, which need to handle their state entirely in ReactJS, while our ReactJS wrapper renders a component and passes it its properties once, making minimal use of methods exposed by ReactJS. In general, the CC UI library seems to be able to compete with the render times of other UI libraries, being faster than the vast majority of them.

7.2 Load Time

The load time metric allows us to evaluate the initial performance impact of the CC UI library. Again, we compare the various wrappers to each other as well as the original Angular components. As we elaborate on later, the Angular wrapper is significantly slower than any other UI library. For this reason, we split every figure into both a figure with and without the Angular wrapper. This should help show the scale of both this significant outlier while not reducing the scale's precision for other UI libraries.

7.2.1 Cow Components

The load time of the CC UI libraries can be seen in Figure 7.7 (without the Angular wrapper) and Figure 7.8 (with the Angular wrapper). When we compare the load time of the CC UI library to the load time of the original 30MHz dashboard, we find that the CC UI library is significantly slower, coming in at 385ms compared to the 30MHz dashboard's 199ms. This is likely because the 30MHz dashboard has been optimized specifically for the initial load time. It loads the minimum amount of JavaScript needed to render the page. After this, other files are only loaded on an as-needed basis. The CC UI library, on the other hand, has to be contained in a single file. Splitting it up into multiple files and

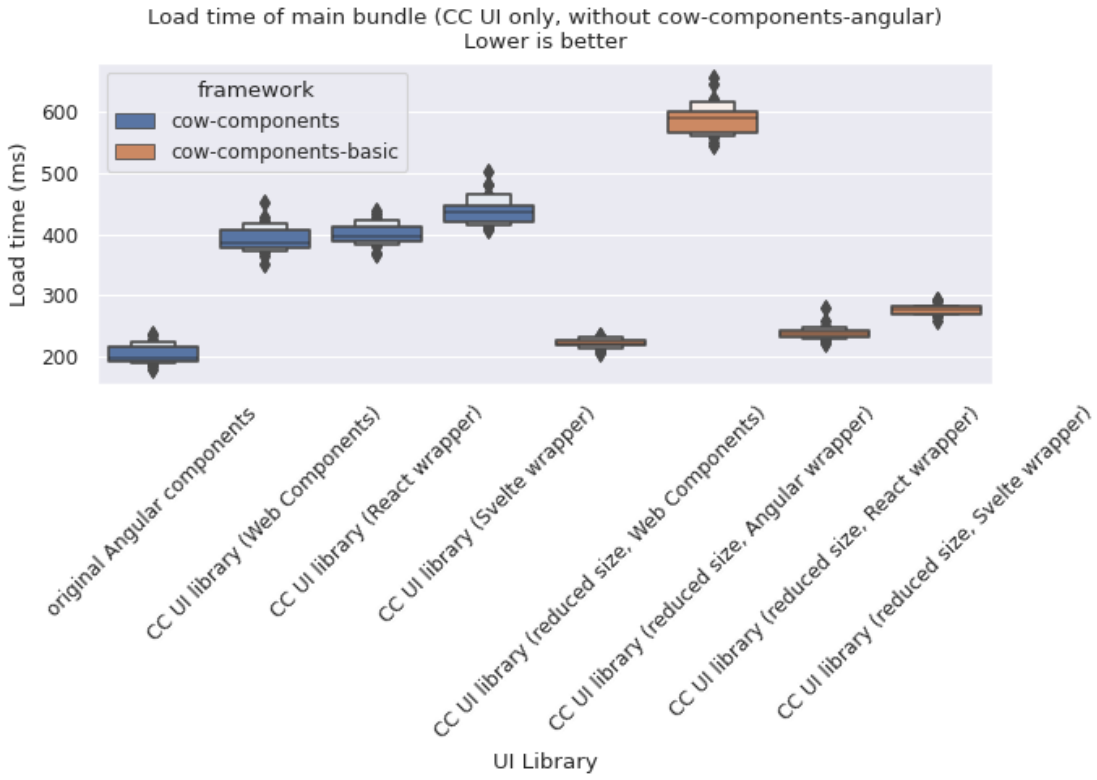


Figure 7.7: Load time of the main JS bundle (CC UI only, without Angular wrapper).

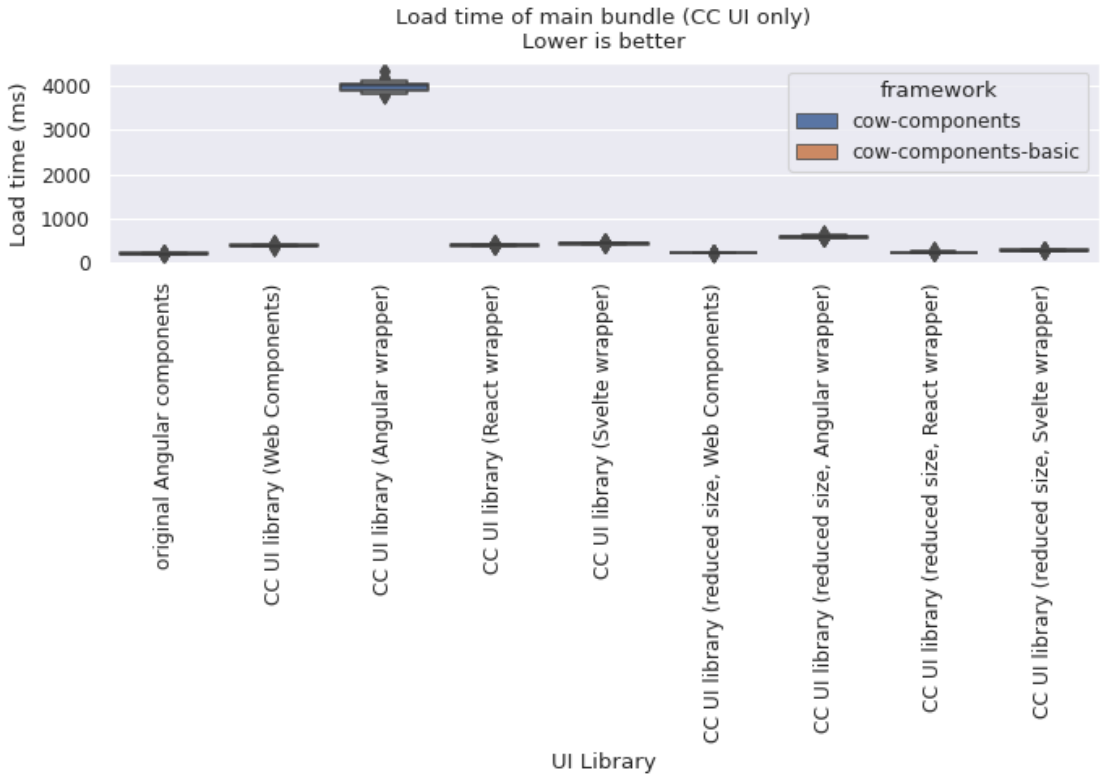


Figure 7.8: Load time of the main JS bundle (CC UI only).

instructing third-party developers to have multiple JS bundles to make the CC UI library work would be a terrible developer experience. Concatenating the files into a single big bundle means all of the code has to be parsed and executed, slowing down execution by quite a lot. Comparing the various wrappers to each other, we find the ReactJS and Svelte wrappers to have load times of 395ms and 434ms, respectively. This is only slightly slower than the CC UI library. The added load time is likely to be added by the JS frameworks themselves. Finally, we can see that the Angular wrapper is by far the slowest, with a load time of 4000ms. This is not entirely unexpected. As mentioned in Section 6.3.11, we had to disable AOT compilation for the Angular wrapper. This means all Angular compilation happens in the browser instead of during the compilation of the JS bundle. This is likely to be the reason why the Angular wrapper is so slow.

Taking a look at the reduced-size CC UI library, we find a loading time of 223ms for the CC UI library. This is only 12.6% higher than the 30MHz dashboard. It appears that a significant portion of the loading was spent on these removed components. Further, the JS framework wrappers loading times are 588ms, 238m, and 277ms for the Angular, ReactJS, and Svelte wrappers, respectively. Again, the difference between the ReactJS and Svelte wrappers is minimal, with the Angular wrapper being significantly slower.

7.2.2 UI Libraries

The load times of other UI libraries can be seen in Figure 7.9 (without Angular wrapper) and Figure 7.10 (with Angular wrapper). Other UI libraries largely differ in load time as well. Svelte UI libraries are by far the fastest, having an average load time of 2.64ms, followed closely by Web Components UI libraries at 11ms and ReactJS UI libraries with 22ms. After this, Vue UI libraries are the fastest, with a load time of 57ms. Finally, we have Angular, which with an average loading time of 106ms, is by far the slowest. Interestingly, we can see that the different distributions of multi-framework UI libraries follow this same pattern. For example the `prime-ng` UI library is 329% slower than the `prime-react` UI library. Similarly, `onsen-angular` is 167% slower than `onsen-react` and 308% slower `onsen-web-components`. This could also be one of the factors that are causing our Angular wrapper to be slower, although the lack of AOT compilation is still by far the most influential factor.

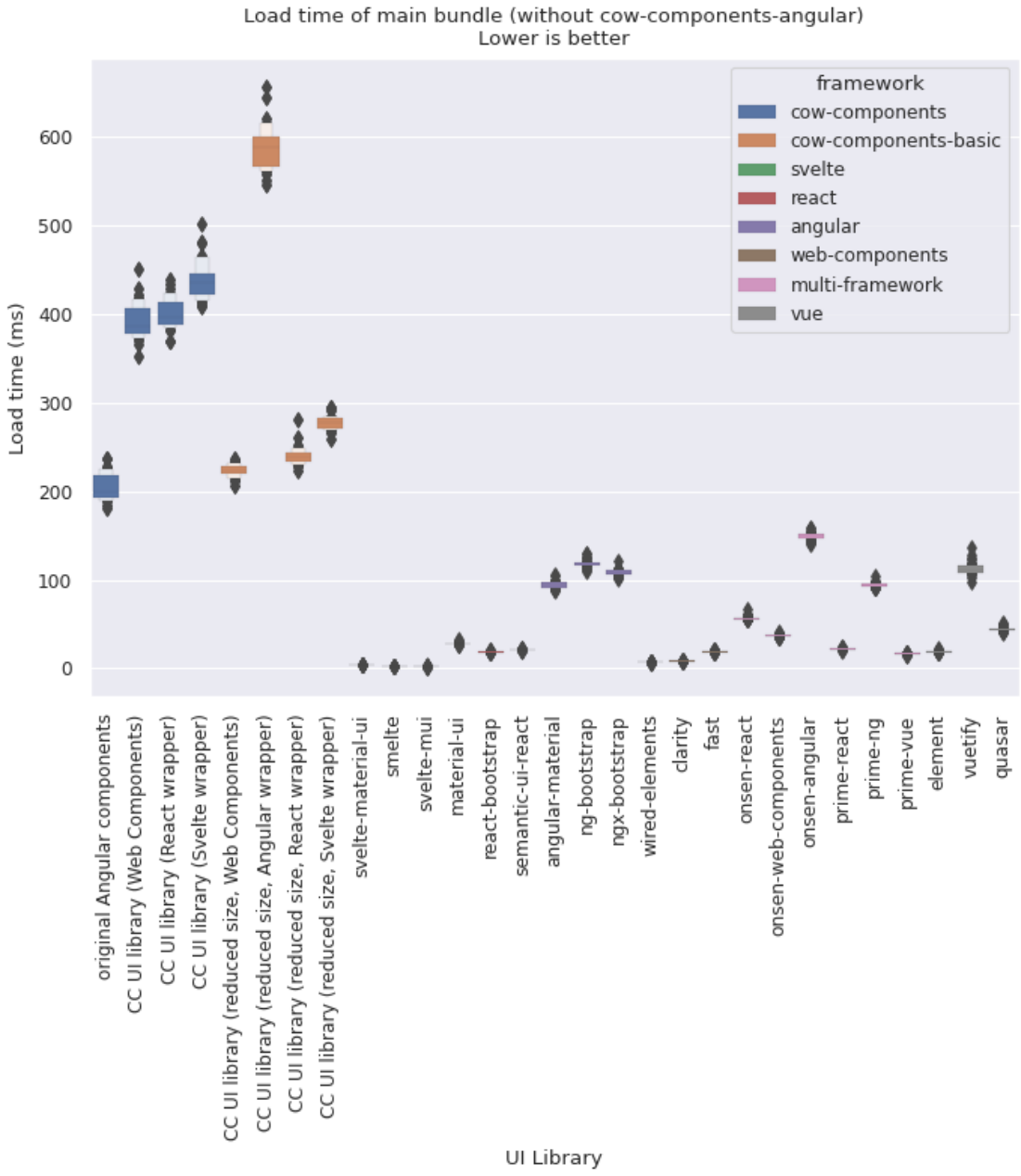


Figure 7.9: Load time of the main JS bundle (without Angular wrapper).

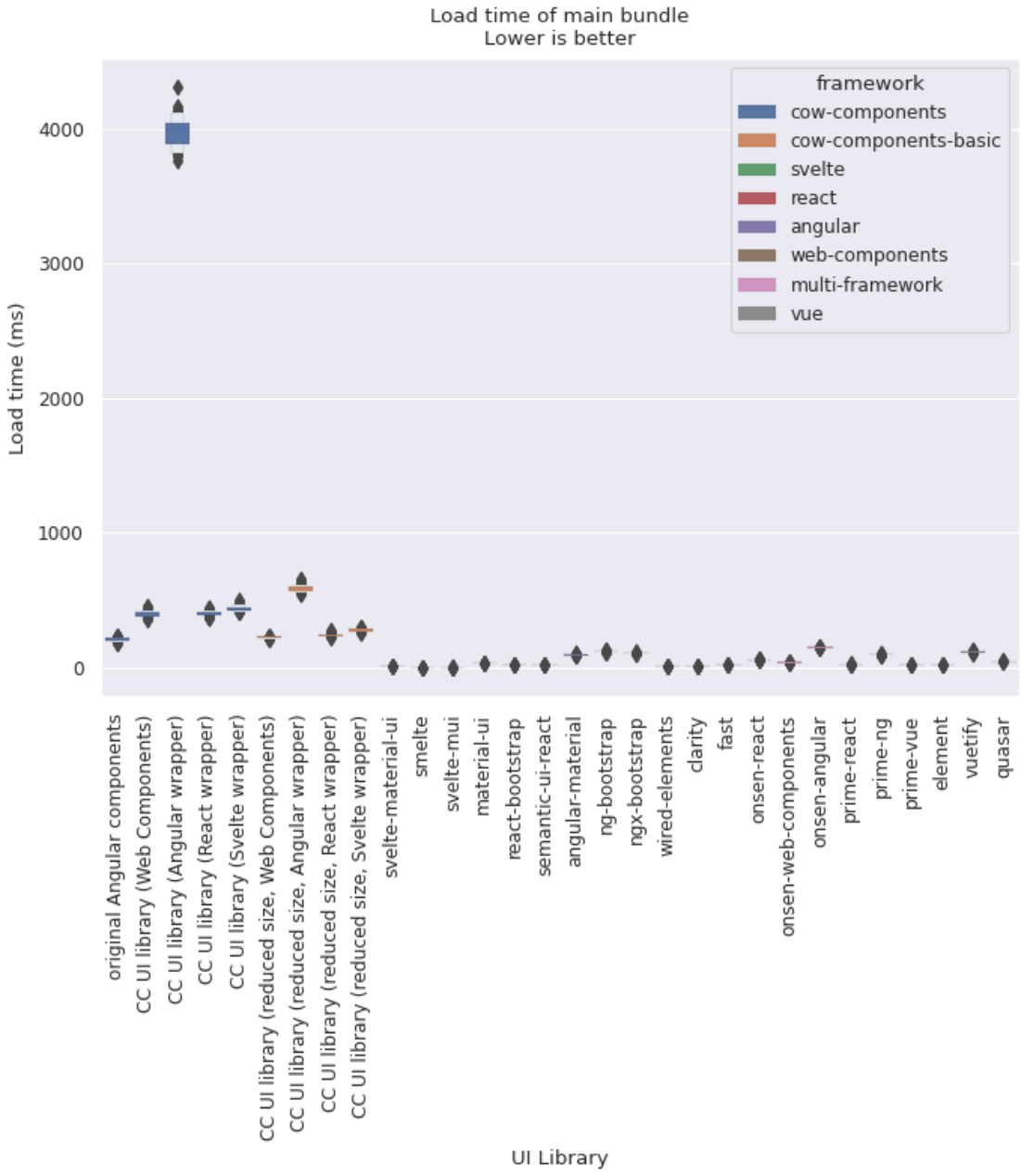


Figure 7.10: Load time of the main JS bundle.

7.3 Bundle Size

Bundle size is a more abstract representation of the previous metric, allowing us to take a look at the impact of just the bundle size itself. This excludes any performance impact that can be attributed to poorly optimized code. This also allows us to look at what the performance impact of the Angular wrapper would be if there was no issue with the AOT compilation.

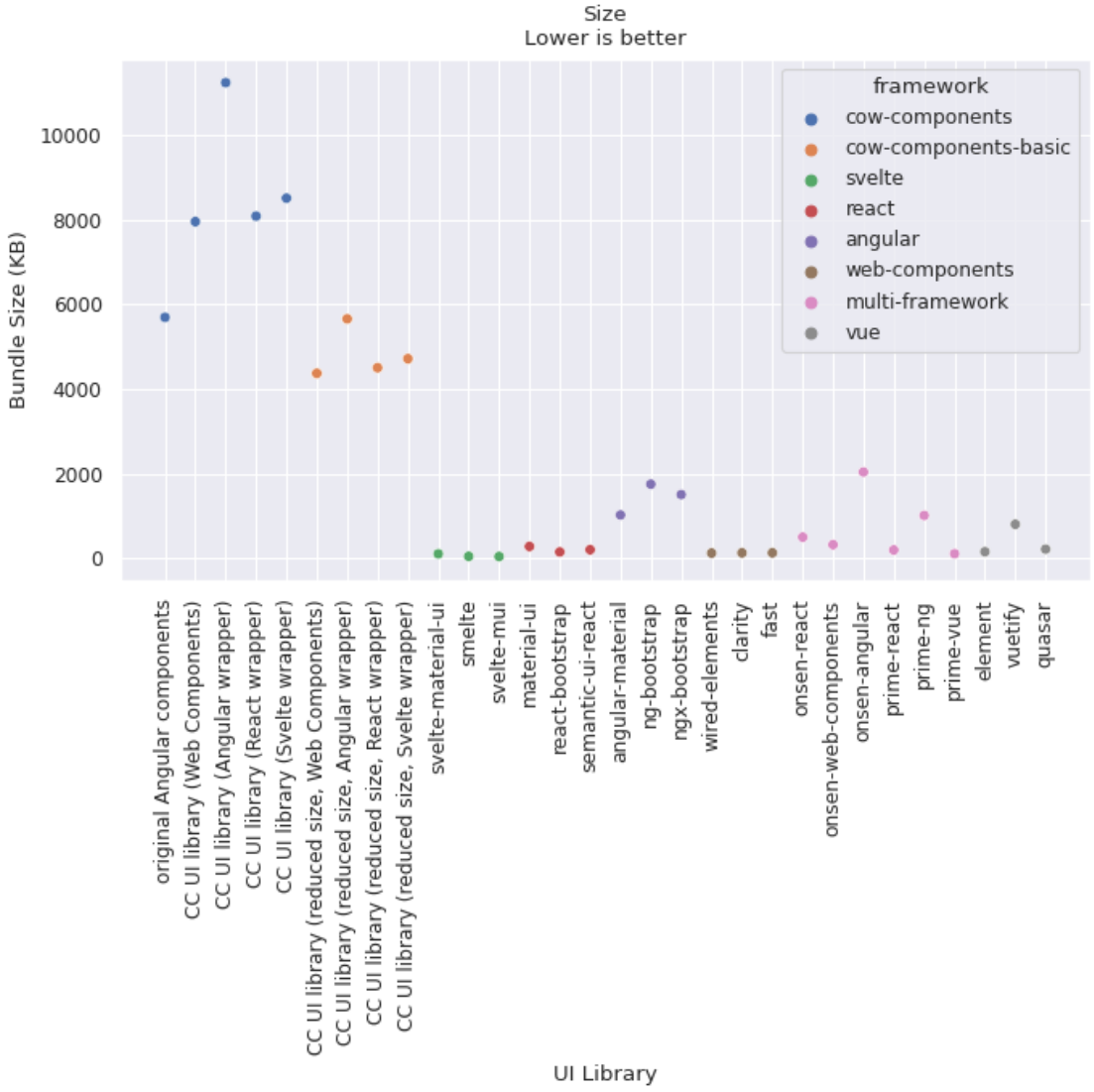


Figure 7.11: Size of the main JS bundle.

The various bundle sizes can be found in Figure 7.11. We can first of all see that the bundle sizes correlate strongly with the load times. From this, we can conclude that they

are an excellent representation of the load time metric. We find sizes average sizes of 57KB, 120KB, 204KB, and 385KB for Svelte, Web Component, ReactJS, and Vue UI libraries, respectively. As expected, Angular is by far the biggest, with a size of 1422KB. This same trend is also visible in our various wrappers. The Angular wrapper is by far the biggest, with a size of 11,251KB. With the strong correlation between load time and bundle size, we can conclude that a large part of the Angular wrapper's slow load time can be attributed to the large bundle size.

7.4 Paint time

The paint time time metric should give us an idea of the real-world loading time of the CC UI library. As described in Chapter 5, we replicated a page containing all components in the various distributions of the CC UI library. This means that all versions are rendering essentially the same page but in their own framework.

The resulting paint times can be found in Figure 7.12. We have included both the `First Paint` and `First Contentful Paint` metrics, which are entirely the same for all the wrappers, only differing for the 30MHz dashboard. We find a first paint of 194ms and a first contentful paint of 233ms for the 30MHz dashboard. The Web Components version of the CC UI library has a first paint (and first contentful paint) of 25ms. This is 87% faster than the 30MHz dashboard. This is likely because the dashboard also needs to run background tasks. These are tasks such as checking whether a user has logged in and fetching data. The CC UI library, on the other hand, has been stripped of this unneeded functionality. Apart from this, we can see a familiar trend of ReactJS and Svelte being slightly slower than the original, with first paint times of 618ms and 616ms, respectively. Finally, we find the Angular wrapper to have a first paint time of 2029ms.

7.5 Quality of Web Components

In this section, we take a look at the quality of the Web Components in the CC UI library. Note that we essentially measure the quality of the original Angular components. This means that the conclusions drawn in this section only apply to the 30MHz codebase and will not be the same for other source codebases.

Cyclomatic complexity: The cyclomatic complexities of the various UI libraries can be seen in Figure 7.13. The cyclomatic complexity of the CC UI library is has a median value of 2 and a mean value of 10.4. The average cyclomatic complexity of the medians

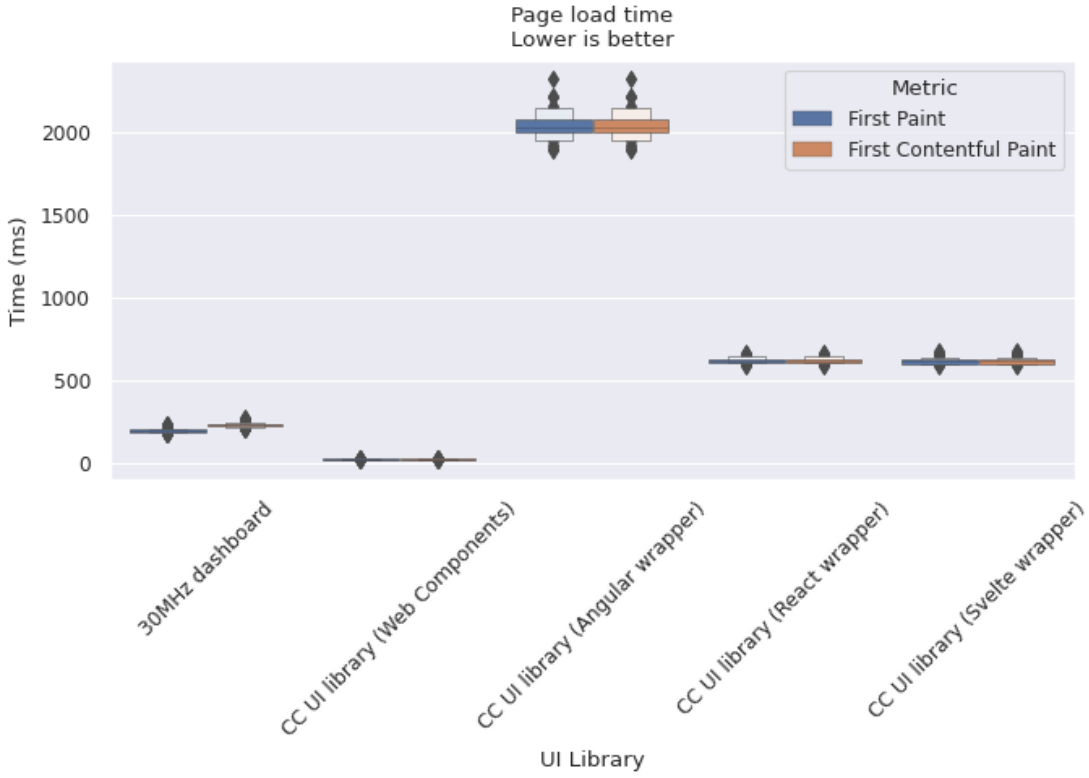


Figure 7.12: First paint metrics for the various demo pages.

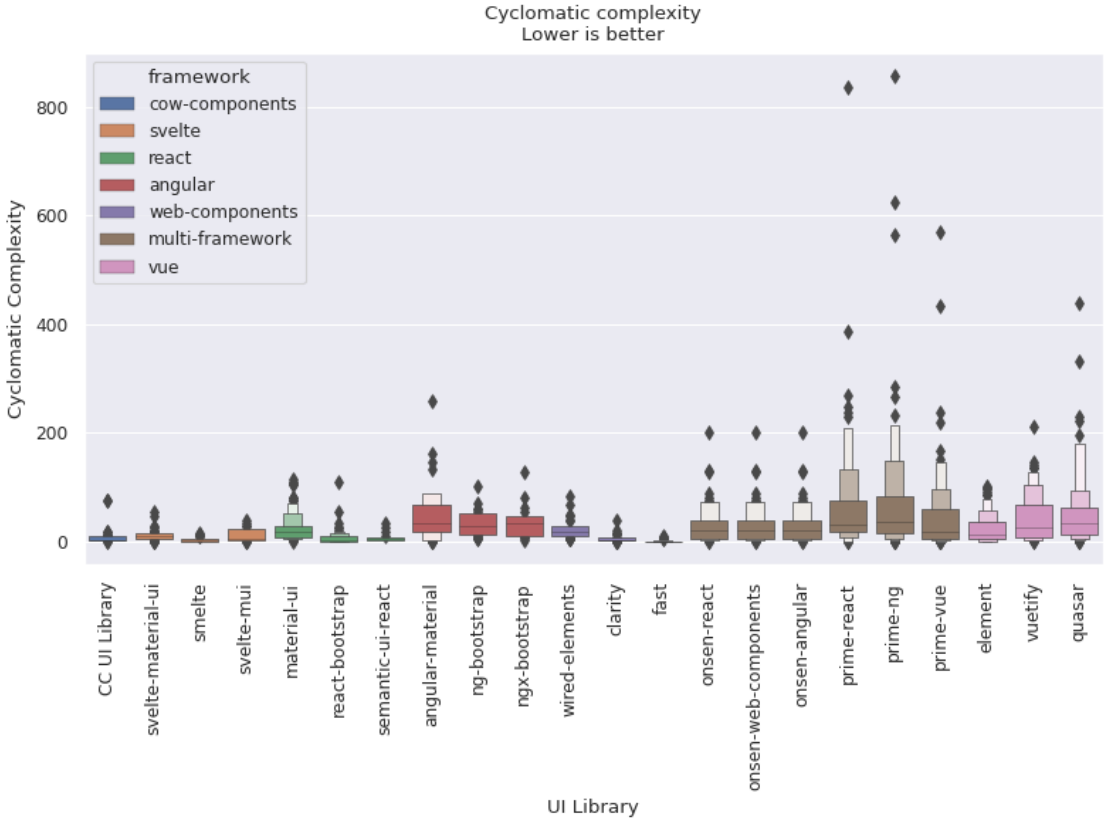


Figure 7.13: Cyclomatic complexity of the various UI libraries.

of all UI libraries is 17.5. Multi-framework UI libraries, in particular, have a very high cyclomatic complexity. The median cyclomatic complexity for all `onsen`-based UI libraries is 22, while the cyclomatic complexities for `prime-react`, `prime-ng`, and `prime-vue` are 31.5, 36, and 18 respectively. This makes sense since these libraries often try to share the source code between the various frameworks as much as possible, leading to many imports.

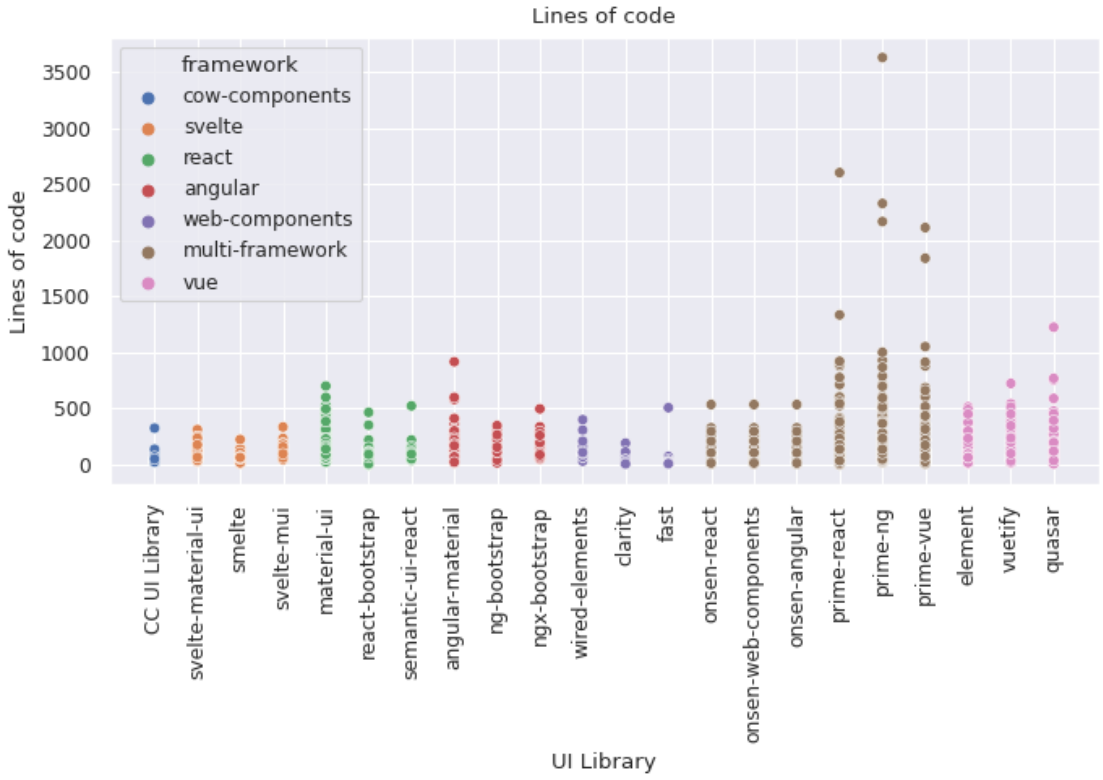


Figure 7.14: Lines of code of the various UI libraries.

Lines of code: The amounts of lines of code can be seen in figure 7.14. Again we see the same trend of the CC UI library being relatively low in complexity (and as such lines of code), with the median lines of code being 26. The average of the medians of all UI libraries is 108.75.

Structural complexity: The structural complexities can be seen in figure 7.15. This time there is a large variation in the structural complexity of CC UI library components, with the median being 2 and the average being 12.6. This outlier is likely to be the Chart component, which is by far the biggest component. The average of the other UI libraries' median structural complexity is 4.2. This suggests that the median structural complexity of the CC UI library is relatively low, which is good.

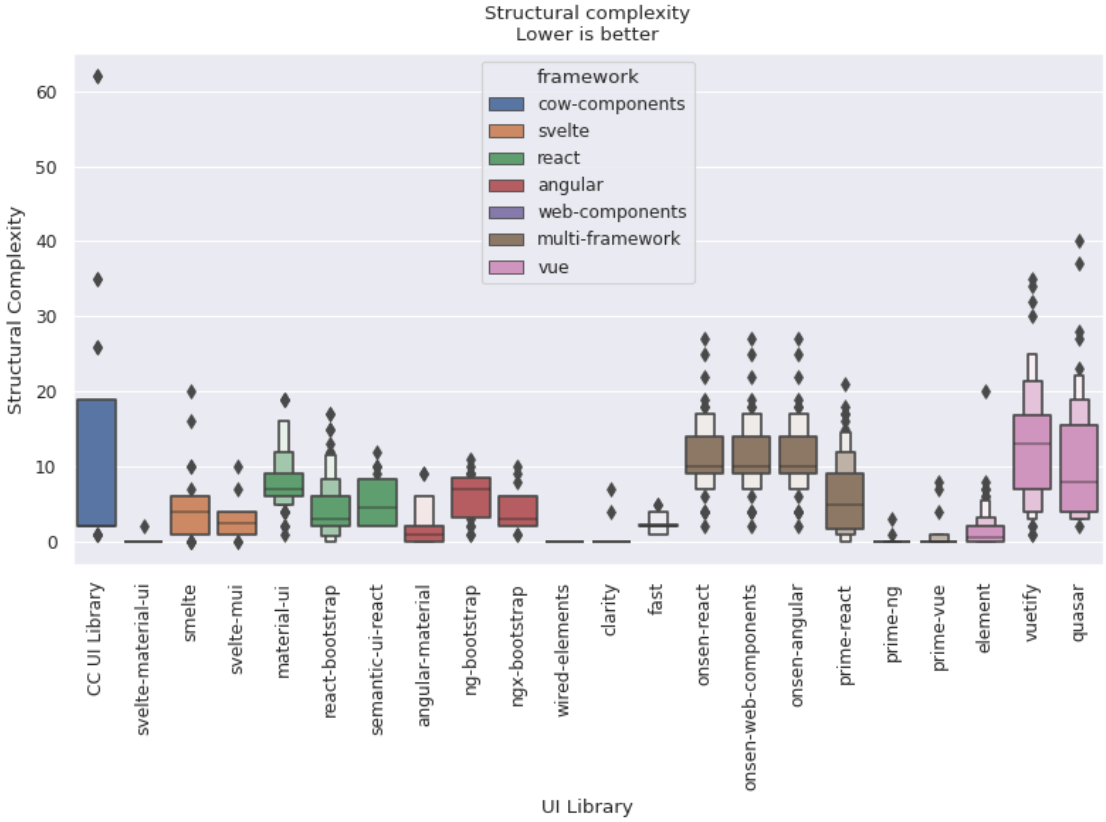


Figure 7.15: Structural complexity of the various UI libraries.

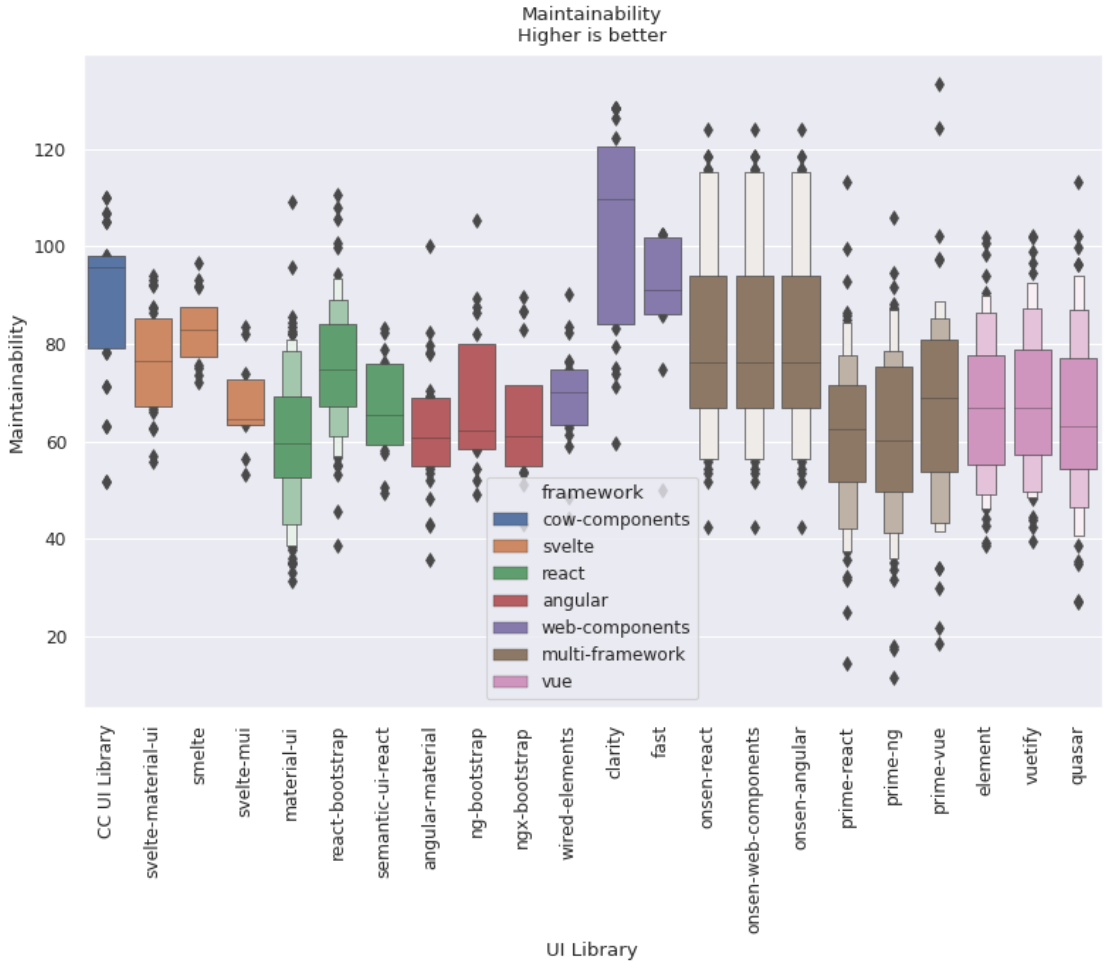


Figure 7.16: Maintainability of the various UI libraries.

Maintainability: The maintainabilities can be seen in figure 7.16. The median maintainability of the CC UI library is 96, with the average median maintainability of the UI libraries being 71. Higher maintainability is better, meaning the CC UI library scores quite well in this metric. All together, we can conclude that the quality of the CC UI library components (and as such, the Angular components they are based on) is quite high.

7.6 Time spent on the project

While the technical results of this project are important, we also decided to take a look at the business side of this project. An important factor here would be the amount of effort required to complete this project. In total, this project took five months of fulltime-equivalent (FTE) to complete. An estimation would be that about one month was spent on Web Component related issues, three months on Angular related issues and one month on creating JS framework wrappers, and one month on other tasks such as creating a build pipeline and package distributions. Note that the time taken is entirely separate from the number of components in the resulting UI library, meaning an added component would not increase the time taken at all. Depending on the time required to build the UI library from scratch combined with the time taken to maintain the UI library and adding new components, this project could very well be worth it.

8

Threats to Validity

In this chapter, we cover threats to the validity of this study. We visit various categories of threats to validity, as laid out in the work performed by Wohling *et al.* (12).

8.1 Conclusion Validity

A possible threat to conclusion validity would be the user study presented in Chapter 9. Drawing a conclusion from such a low sample size would lead to a high risk of an invalid conclusion. Instead, we opted to mark the resulting findings as mere indications to be proven in future work.

8.2 Internal Validity

An internal threat to validity could be the measurement of our metrics being influenced by external factors. As described in Section 5.7 we explicitly remove the factor of network speed from our benchmarks. This leaves only the factor of available system resources as a possible variable. In order to eliminate this factor, we took several steps. We first ensured a clean testing environment by shutting down all unneeded background processes on the test machine. This should vastly reduce the amount of fluctuation in available system resources. Secondly, we ensured that only a single test is running at a time. This means every test has the entire computer to itself (in practice, it likely only uses a single core) and does not compete with other tests for system resources. Lastly, we apply all the steps described in Section 5.5 which includes randomizing the order in which the tests are run and increasing the number of tests to thirty measurements per test. This should ensure that any possible fluctuations are smoothed out and shared across all tests.

8.3 Construct Validity

In order to ensure that the results of the experiments can be generalized to the created Web Component library, metrics that allow for the comparison of individual components have to be found. In order to validate the quality of components, we use a set of metrics that was validated through a user study by Martinez-Ortiz *et al.* (6), as described in Section 3.4. In addition, we use objective performance metrics that are independent of user perception. Gao *et al.* (9) describes such a problem for full web pages, where pages could be perceived as loaded while parts are still loading. To avoid such a situation, we have chosen three basic components that are either not loaded at all or fully loaded, namely the Button, Input, and Switch. This allows us to measure the loading time of individual components with a metric that is independent of user perception.

8.4 External Validity

A large number of the problems we face in this case study applied to Web Components in general, as shown in Section 6.2. Still, a significant share of the problems is related to Angular, more specifically Angular 10. Likewise, a significant part of them is specific to the 30MHz codebase. Given a different codebase or a different Angular version, different problems might be faced in the process of converting Angular components to Web Components. While the issues might differ from framework to framework or even from Angular version to Angular version, the results should be generalizable to other frameworks and Angular versions. Bugs we encountered are likely to be fixed, and performance is likely to only improve in future Angular versions, as has been the case with the Angular Ivy compiler [1]. Additionally, a significant amount of the faced problems apply to Web Components in general, as shown in Section 6.2. For this reason, we believe that the feasibility of the applied process will at worst stay the same and at best improve for other Angular versions or JS frameworks. Further, while the specific UI library we created is dependent mainly on the 30MHz codebase and its specific architecture and contents, we make sure to compare the created CC UI libraries with the original 30MHz codebase itself, ensuring all results are relative to the original. This should ensure we answer the research question for a generalized case. If we were to compare the CC UI library solely to other UI libraries, the answer to the research question would only apply to the case of 30MHz.

[1] `https://angular.io/guide/ivy`

9

Discussion

9.1 Discussion of Results

The results described in Chapter 7 show that the creation of a UI library from an existing codebase is very well possible in an Angular application. Render times are only slightly higher, remaining competitive with various other UI libraries. One negative aspect seems to be that the render times increase pretty quickly with a higher number of components. Further, load times are not significantly higher in all cases except the Angular wrapper. This should result in a good user experience across the board, being slightly slower than the original components but providing access to them in the most popular JS frameworks. We can say that the answer to RQ1 is that it is definitely technically feasible to migrate Angular components to Web Components.

While the technical results of this project are important, we also evaluated the business side of this project. We find the time spent to be five months of FTE. We also took a look at the degree to which this project interferes with the original codebase and its developers' workflows through a questionnaire answered by the three front-end developers at 30MHz. It should be noted that of the three front-end developers, two indicate they only work on the front-end once a week or less. Additionally, the third front-end developer had recently joined the company and has not witnessed the complete development process described in this paper. As such, we are unable to draw definitive conclusions from the results of the questionnaire, and we instead treat the results as mere indications.

In questioning these three front-end developers at 30MHz, we found that, on average, they rated the impact of changes to the main codebase as a 2.6 on a scale from 0 (no impact at all) to 10 (significant impact). For a process that interlocks with the main codebase so heavily, this is a very low number, leading us to believe that the impact was

slim. Additionally, there are some new factors that developers have to keep in mind while developing new components. An example of this is the need for better documentation for components in order to ensure the automatically generated documentation is correct. Another example would be the need to add a new UI component to the array containing all components that are to be included in the UI library. On average, the developers rated the impact of these changes to be a 2, signaling that the everyday impact is not very large. Lastly, we asked developers how often the existence of this project blocked their workflow. They all indicated they had not been blocked once, meaning this project was executed entirely without blocking other developers' workflow. These results suggest that the business viability of the migration of Angular components to a UI library is relatively high as well, leading to minimal impact on current developers and their workflow while requiring relatively little time. Especially in a situation where there are many UI components, the time spent on this project is significantly smaller than the time spent recreating them.

All in all, we can conclude that the answer to RQ1 is that the process of migrating Angular components to a Web Component UI library is feasible. We hope this case study convinces businesses who are considering this process to take the steps we have taken over creating an entirely new UI library. In addition to being used in the manner we described, that is, the creation of a UI library for third parties, this process could also be applied to components that are internal to a business. With the ever-increasing amount of platforms with which users are able to interact (desktops, phones, tablets, televisions, smart fridges), the number of platforms for which businesses need to develop an application also increases. Since most of these platforms require different software stacks, Web Components could provide a basis for generating components for other platforms. For example, the main large web app can be built in Angular, with another small internal web app being built in ReactJS (using the ReactJS wrapper), another internal web app built in Vue, and the mobile apps built using React Native [1] or Apache Cordova [2].

What the results of this study do not tell us is the viability of migrating components from any other JS framework to Web Components. In this case study, we specifically targeted Angular, which provides the simple Angular Elements tool [3]. Other JS frameworks might not have such tools available, which might make this process less straightforward. However, we believe that the process of migrating components from any other popular JS framework

[1] `https://reactnative.dev/`

[2] `https://cordova.apache.org/`

[3] `https://angular.io/guide/elements`

to Web Components may very well be significantly easier than from Angular components. A large number of the issues we faced were Angular related, as described in Section 6.3.1. Those issues were also by far the hardest to solve. Most of these issues would not appear when using other JS frameworks.

9.2 Checklist for Migration to Web Components

Based on the work performed in Chapter 6 we present a checklist going over the various steps required to perform a migration to Web Components. Note that this checklist contains the steps likely to be required for conversion to Web Components in a general sense and not the conversion specifically from Angular. In order to aid in the use of the checklist, we include hints about how the described steps have been tackled in the presented case study.

9.2.1 Checklist

1. **Rendering components**

   Description: Wrap the individual components in Web Components that render them to the DOM. When completed successfully, appending a given HTML tag to the DOM such as `<x-button>` should render the corresponding component from the original set of components. This process will differ from JS framework to JS framework but will generally come down to the following steps:

   (a) Iterate over every component in the set of original components

   (b) Register a new Web Component for every component with the task of rendering that component. Choose a tag name under which to register this Web Component as a Custom Element.

   (c) When rendering this Web Component, append a new ShadowRoot to it in the DOM.

   (d) To this ShadowRoot append a rendering root in whichever JS framework is being used.

   (e) Append the to-be-rendered component to the rendering root of the target JS framework.

   Reference Solution: In the case of Angular, this step is almost entirely taken care of by Angular Elements. This is also the approach we used to solve this problem.

2. **Compatibility (optional)**

   Description: While browser support for Web Components is relatively widespread, it is not yet supported by every browser and every browser version. Depending on the browsers that need to be supported, polyfills might be needed to add support for unsupported features. An overview of browser support for Web Components can be found on *caniuse.com* [1].

   Reference Solution: As described in Section 6.2.2, we make use of polyfills to add support for Web Components for those browsers that do not already support it. In particular we make use of the `custom-elements`[2] and `custom-elements-builtin`[3] polyfills.

3. **Global CSS application**

   Description: Generally styles are applied to components through a global stylesheet. Wrapping components in ShadowRoots will cause this global stylesheet to no longer apply to them, effectively removing any styling from them.

   Reference Solution: We solve this issue by injecting this global stylesheet into every ShadowRoot, making use of `adoptedStylesheets` to reduce performance impact of this approach. A more detailed description of our approach can be found in Section 6.2.1.

4. **Application of Complex Attributes (optional)**

   Description: Most JS frameworks allow the passing of complex JS values to components through attributes. This is in contrast to Web Components, which only allow the passing of string values to components through HTML attributes. This problem is described in-depth in Section 6.2.6. Note that this problem may no longer be required when support for Web Components is completed in all major JS frameworks. The state of support for Web Components can be found on the website *custom-elements-everywhere.com* [4]. If the migrated components do not need complex attributes and work with string attributes as well, this step can be skipped.

   Solution: Our solution to this problem is described in Section 6.2.6 as well. Essentially, we create wrappers for three of the largest JS frameworks that wrap around the created Web Components. The wrappers for these components then facilitate the

[1] `https://caniuse.com/?search=components`
[2] `https://www.npmjs.com/package/@ungap/custom-elements`
[3] `https://www.npmjs.com/package/@ungap/custom-elements-builtin`
[4] `https://custom-elements-everywhere.com/`

use of complex attributes by applying the complex attributes they receive directly to the original component instance that is rendered inside of the Web Component.

10

Conclusion

In this case study, a number of aspects of migrating a set of Angular components to Web Components are evaluated. These aspects include the question whether the migration process is possible at all, what a possible performance impact is, and how the resulting migrated components relate to other component libraries in the field. Chapter 6 describes the various issues we faced during this project, eventually showing that it is possible to migrated a set of Angular components to Web Components. In Chapter 7 we evaluate the end resulting Web Components, comparing them to both the original Angular components they were created from and various other UI libraries. We find that the newly created Web Components are only slightly slower in rendering and take only about twice as much time to load. The resulting components hold up very well compared to other UI libraries, initially being faster than quite a few of them but becoming relatively slower as the number of rendered components increases. This means that although the Web Components library was migrateded from Angular components, it can compete quite well with other UI libraries that were written from scratch. This confirms the technical feasibility of migrating Angular components to Web Components.

Further, when looking at the business side in Section 7.6 and Chapter 9, we find that the impact on the existing codebase and other developers is minimal. We also find the time spent on this migration to be definitely worth it depending on the time required to build the UI library from scratch combined with the time taken to maintain the UI library and adding new components. This indicates that this migration is a worthwhile investment, leading to freedom from maintaining two sets of components and the ability to easily add a new component to the Web Components library without issues. It should be noted that these results are not based on sufficient data to draw definitive conclusions. Instead, we indicate that these were our results and leave definitive proof for future work.

Further future work could be done on improving the performance of the created Web Components. Minimizing the performance impact of this migration process would aid in making these components just as viable as components written from scratch. Additionally, an interesting area to focus on is the list of issues faced during the migration process. Future work could revolve around performing another such case study and comparing the issues faced, extracting a list of issues that are always faced during this migration process. Such a list would allow for the creation of structural solutions to these problems, for example in the form of a freely downloadable JS package that aids in the migration process.

Finally, one shortcoming of this thesis is that we were only able to evaluate the effectiveness of migrating **Angular** components to Web Components. Not the migration of components from any JS framework to Web Components. We conjecture that this process should be just as feasible, with other frameworks likely taking significantly less time to migrated than Angular. As such, we believe further research into the migration of components from other JS frameworks to Web Components would be very beneficial, eventually leading to a situation where we can conclude that components from all JS frameworks can be migrated to Web Components, eventually making them re-usable across all JS frameworks. Furthermore, future work could be done on improving the performance of the created Web Components. Lastly,

Appendix

.1 Code for creating a Hierarchical Injector in an Angular Elements component

```
// This injector is provided by Angular to the root module
var rootInjector = ...;

// Create a fake injector
const _MOCK_INJECTOR = {
  get() {
    return {
      run() {},
      resolveComponentFactory() {
        return {
          inputs: [],
        };
      },
    };
  },
};

// Extract defualt NgElementStrategy
function getDefaultNgElementStrategy() {
  const customElement = createCustomElement(EmptyAngularComponent, {
    injector: _MOCK_INJECTOR,
  });
  const proto = customElement.prototype;
  const strategyInstance = proto.ngElementStrategy;
  return strategyInstance.constructor;

}

// Get the NodeInjector class
function getNgInjectorClass(rootInjector) {
  const componentFactory = rootInjector.get(ComponentFactoryResolver).
      resolveComponentFactory(EmptyAngularComponent);
```

.1 Code for creating a Hierarchical Injector in an Angular Elements component

```
32    const componentInstance = componentFactory.create(rootInjector, []);
33    return componentInstance.injector.constructor
34  }
35
36  // Get given HTML element's nodeinjector
37  function getNodeInjector(rootInjector, host) {
38    const hostContext = host.__ngContext__;
39    const lView = hostContext.lView;
40    const tNode = lView[1].data[hostContext.nodeIndex];
41    const NodeInjectorClass = getNgInjectorClass(rootInjector);
42    return new NodeInjectorClass(tNode, lView);
43  }
44
45  // Create a custom NgElementStrategy
46  class CustomNgElementStrategy extends getDefaultNgElementStrategy() {
47    originalInjector = this.injector;
48
49    connect(element): void {
50      if (this.injector === this.originalInjector) {
51        const localRoot = element.getRootNode();
52        const host = localRoot.host;
53        const nodeInjector = getNodeInjector(rootInjector, host);
54
55        this.injector = Injector.create({
56          providers: [],
57          parent: nodeInjector,
58        });
59
60        return this._connectSuperWithDelayedInit(element);
61      }
62
63      return super.connect(element);
64    }
65  }
66
67  // Create a custom NgElementStrategyFactory that creates
68  // instances of our custom NgElementStrategy
69  class CustomNgElementStrategyFactory {
70    constructor(
71      private _StrategyConstructor,
72      component,
73      injector,
74    ) {
75      this.componentFactory = injector
76        .get(ComponentFactoryResolver)
77        .resolveComponentFactory(component);
78    }
```

```
79
80    create(injector) {
81      return new this._StrategyConstructor(this.componentFactory, injector);
82    }
83  }
84
85  // Provide the CustomNgElementStrategyFactory to the createCustomElement
86  // function and create a new Web Component
87  const WebComponent = createCustomElement(AngularComponent, {
88    injector: rootInjector,
89    strategyFactory: new CustomNgElementStrategyFactory(
90      CustomNgElementStrategy,
91      AngularComponent,
92      rootInjector
93    )
94  })
```

Listing 1: The code for creating a Hierarchical Injector in an Angular Elements component

.2 Code used for render-on-demand functions for various JS frameworks

```
1    const App = () => {
2      const [ visibleComponent, setVisibleComponent ] = React.useState(null);
3
4      window.setVisibleComponent = (componentName) => {
5        setVisibleComponent(componentName);
6      }
7
8      return (
9        { visibleComponent === 'Button' && <Button />}
10     )
11   };
```

Listing 2: The render-on-demand function in ReactJS

```
1    <button *ngIf="visibleComponent == 'Button'" />
```

Listing 3: The render-on-demand function in Angular (HTML file)

```
1    @Component({
2      ...
3    })
4    export class AppComponent {
5      constructor(private _cd: ChangeDetectorRef) {
6        window.setVisibleComponent = (componentName) => {
```

```
7           this.visibleComponent = componentName;
8           this._cd.detectChanges();
9         }
10      }
11
12      public visibleComponent = null;
13    }
```

Listing 4: The render-on-demand function in Angular (JavaScript file)

```
1     <script>
2       window.setVisibleComponent = (componentName) => {
3         visibleComponent = componentName;
4       }
5
6       let visibleComponent = null;
7     </script>
8
9     {#if visibleComponent === 'Button'}
10      <Button />
11    {/if}
```

Listing 5: The render-on-demand function in Svelte

```
1     window.setVisibleComponent = (componentName) => {
2       if (componentName === 'Button') {
3         document.body.appendChild(document.createElement('x-button'));
4       }
5     }
```

Listing 6: The render-on-demand function in Web Components

.3 Render times for all components

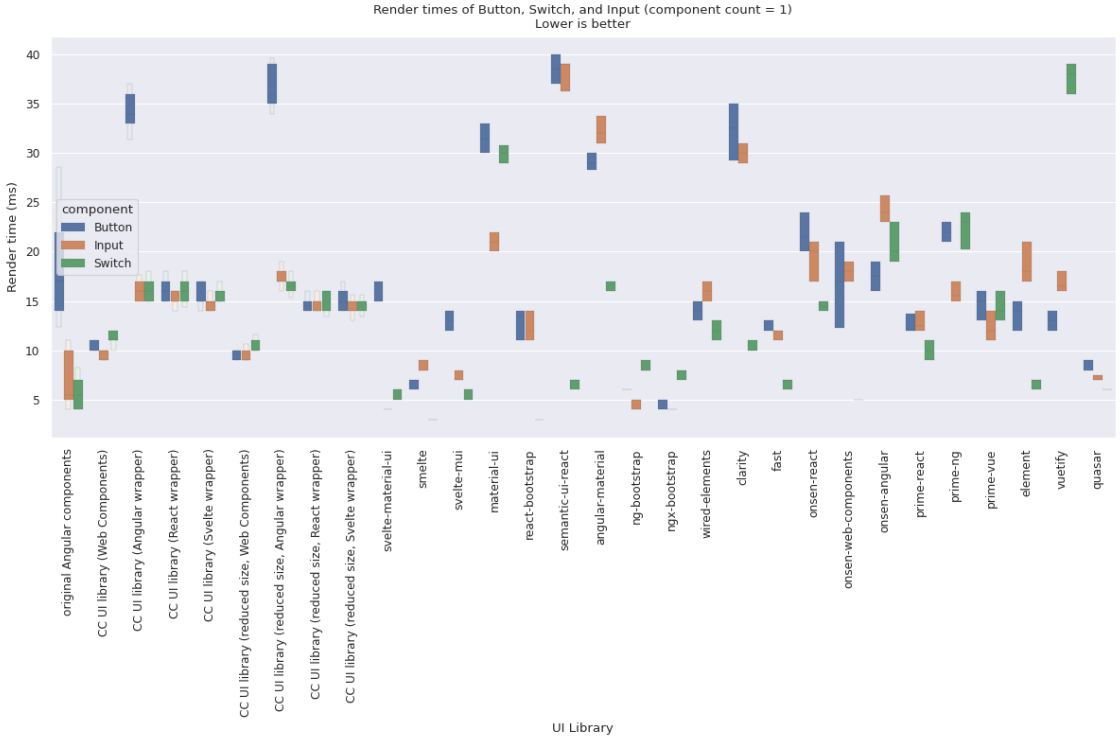


Figure 1: Render times of a single Button, Switch, or Input component

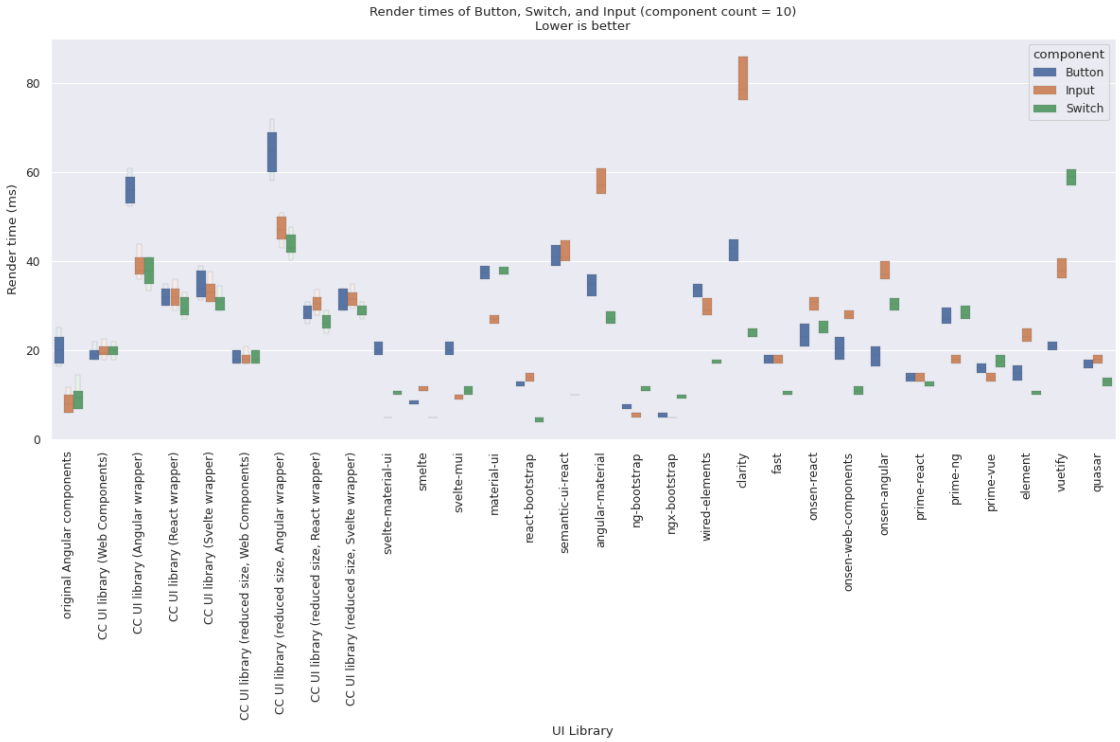


Figure 2: Render times of ten Button, Switch, or Input components

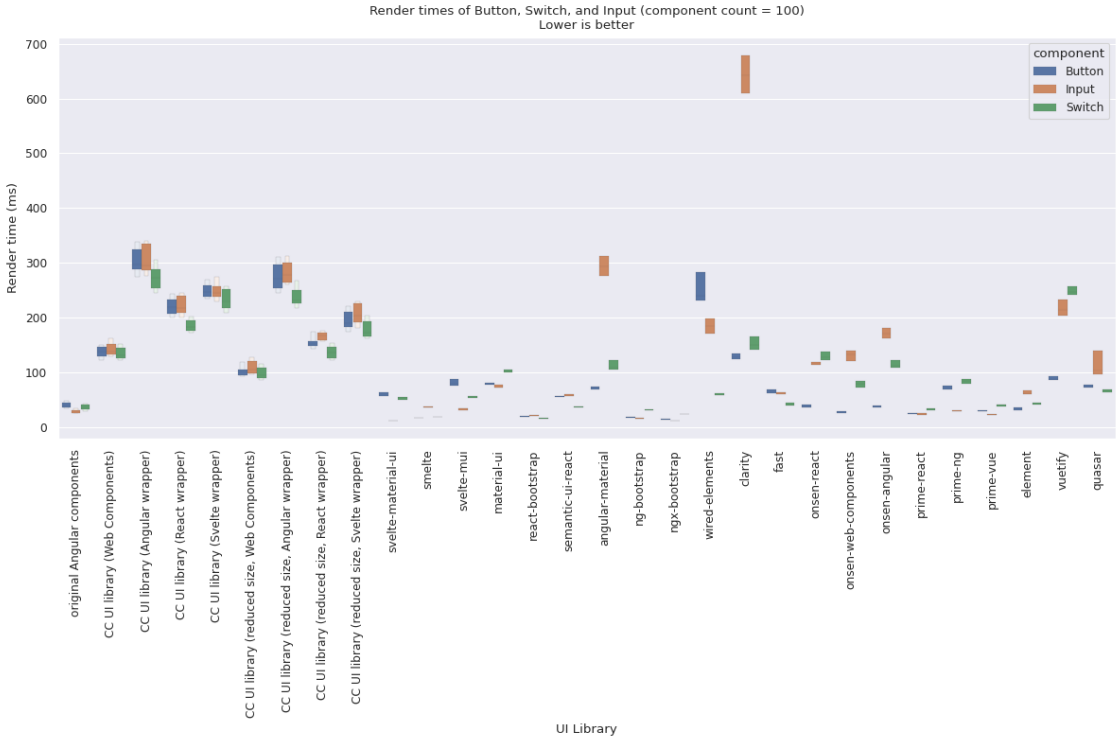


Figure 3: Render times of one hundred single Button, Switch, or Input components

References

[1] Pedro J Molina. **Quid: prototyping web components on the web**. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–5, 2019. 9

[2] Marcel Mráz. **Component-based UI Web Development**. 2019. 13

[3] Ralf Lämmel and Simon Peyton Jones. **Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming**. **38**, pages 26–37, 03 2003. 13

[4] Trinh Ky Nam. **UI library project setup for Vaimo group with modern web technology**. 2019. 13

[5] Jaume Armengol Barahona. *Development of an Angular library for dynamic loading of web components*. B.S. thesis, Universitat Politècnica de Catalunya, 2020. 14

[6] Andres-Leonardo Martinez-Ortiz, David Lizcano, M. Ortega, L. Ruiz, and G. López. **A quality model for web components**. pages 430–432, 11 2016. 16, 19, 24, 25, 79

[7] T. J. McCabe. **A Complexity Measure**. *IEEE Transactions on Software Engineering*, **SE-2**(4):308–320, 1976. 16, 19

[8] Maurice H Halstead. **Elements of software science**. 1977. 16, 19

[9] Qingzhu Gao, Prasenjit Dey, and Parvez Ahammad. **Perceived performance of top retail webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold qoe**. In *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 13–18, 2017. 17, 79

[10] Vikram Nathan. *Measuring time to interactivity for modern Web pages*. PhD thesis, Massachusetts Institute of Technology, 2018. 17

[11] Jasper van Riet, Flavia Paganelli, and Ivano Malavolta. **From 6.2 to 0.15 seconds–an Industrial Case Study on Mobile Web Performance**. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 746–755. IEEE, 2020. 17

[12] Claes. Wohlin. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1st ed. 2012. edition, 2012. 78