✎ Title

Sander Ronde

s.j.p.m.ronde@student.vu.nl

Student number: 2639938

Supervisor: Ivano Malavolta

April 14, 2021

Abstract—

Index Terms—✎ **keywords**

✎

I. 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 that was solved by the case study.

A. 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. For example, temperature, humidity, air pressure, etc. They also provide their customers with a dashboard that allows them to view the collected data. 30MHz has several so-called partners. These partners are consultancies in the field of agriculture (among others). They supply 30MHz with additional customers by suggesting their clients become customers of 30MHz as well.

30MHz supplies their customers with the so-called dashboard. An example of it can be seen in Figure 1. It is the central location where any customer can view their data. 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. An example of a widget is a chart that displays the value of the data over time.

Fig. 1: Widgets on the 30MHz dashboard

1) Apps: The large amounts of data collected by 30MHz can be utilized in many ways. Companies with domain knowledge and expertise in certain areas (such as the partners) can use it to 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), they decided to allow 3rd party developers to develop them instead. In practice, these will be built by the partners and will be provided to customers of 30MHz for a fee through a marketplace. There are currently two implementations:

- *Widgets:* Widgets in the dashboard that take data from one or more sensors and display it. These are made to provide information at a glance and are fairly small, as can be seen in Figure 1. An example would be a new way to display sun data by showing a sun if the plants are able to grow (it's daytime) and a moon if they're not (it's nighttime).
- *Apps:* Full pages in the dashboard web app. These fill the entire screen (bar some 30MHz branding) and can provide more rich and interactive experiences. An example would be a page where clients of a partner can tune some 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 style and principles as the rest of the dashboard. This ensures that users are familiar with the apps and that visual consistency across the platform is not broken. This concept has been applied many times. For example on Google's Android through Material Design [1], Apple's design on iOS [2], Zendesk Garden [3] and many more. Importantly, these companies all provide their app developers with a set of components in order to help them maintain their design language. Such a set of components is generally referred
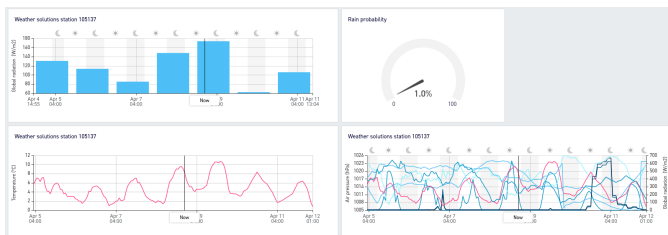
---

[1] https://material.io/

[2] https://developer.apple.com/design/

[3] https://garden.zendesk.com/

to as a UI library. Similarly, 30MHz wants to provide their 3rd party app developers with a UI library as well. In this paper, we'll be referring to the UI library 30MHz will provide to 3rd 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 set of components that cover the basic set of UI components (buttons, an input, a date picker, etc.), but since these are largely interwoven with other internal code, their source code can not just be provided to 3rd party developers. They have also been written in Angular [4], meaning that any developers who wish to develop their app in a different JavaScript (JS) framework are unable to do so. Looking at the most popular web frameworks in the latest Stack Overflow Developer Survey [5] (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 conversion from the set of internal components to a framework-agnostic format.

The obvious problem with the first option is that you're maintaining two separate copies of very similar code. This causes a number of issues. Firstly the time spent maintaining a component is doubled. Additionally, feature differences between the Angular framework and the framework-agnostic format we choose are going to lead to problems. Some things that work in Angular might need workarounds in the other format and the other way around. Another issue with this option is that the components have to be written entirely from scratch. While this would be manageable for simple components such as buttons, this is unfeasible for more complex components. One of which is the chart component. This component is important to have in the 30MHz design library, seeing as it is able to display the sensor data. However, the source code for the chart alone is already 500 lines long, most of which are tightly integrated with the rest of the platform. Through all of its references, the chart component references about half of the source files in the dashboard. Rewriting all of this in another framework is completely unfeasible and not worth the effort, leading us to explore the second option.

While the second option is not an easy one and it will likely be a very complex process to set up, it is one that will scale a lot better. Once it is set up, any new components will be automatically converted, and any changes will be propagated automatically. In the long run, this should save a lot of time. This is the option 30MHz eventually

decided on. However, a framework-agnostic format needs to be chosen to facilitate this process.

B. 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, a DSL for generating components in various frameworks [6]. It currently supports Web Components [7], Stencil [8], Angular and Polymer [9]. While this is fairly impressive, 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 converted to, this is not a great fit for us.

This brings us to the other option, namely Web Components. Web Components (also known as Custom Elements) are a technology proposed in 2013 [10] and implemented in major browsers in 2018 [11]. 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 [12], we can cover most JS frameworks by using Web Components as our target format.

C. Angular Elements

To perform the conversion of Angular components to Web Components, we will be using Angular Elements [13]. Angular Elements is a JS package published by the developers of Angular that allows the conversion of Angular components to Web Components. Angular apps are normally mounted to the DOM by the user through a call to the `bootstrapModule` function. After this call, the bootstrap component is mounted to the DOM, after which child components are 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. When this happens, a new root is created in place of this new HTML element. Instead of there being a single root in which everything is rendered (as is the case in a normal Angular app), components are all rendered in their own local root. We will be using Angular Elements for the conversion to Web Components in this case study.

[4]https://angular.io/
[5]https://insights.stackoverflow.com/survey/2020

[6]https://www.w3.org/TR/2013/WD-custom-elements-20130514/#about
[7]https://developer.mozilla.org/en-US/docs/Web/Web_Components
[8]https://stenciljs.com/
[9]https://www.polymer-project.org/
[10]https://www.w3.org/TR/2013/WD-custom-elements-20130514/#about
[11]https://caniuse.com/?search=webcomponents
[12]https://custom-elements-everywhere.com/
[13]https://angular.io/guide/elements

Fig. 2: An example of a component that provides type hints


```
class App extends React.Component<{}> {
  render() {
    return <MyButton />;
  }
}
          children?        (JSX attribute) children?: Reac…
          disabled?
          key?
          loading?
          ref?
          shape?
          size?
          type?
```
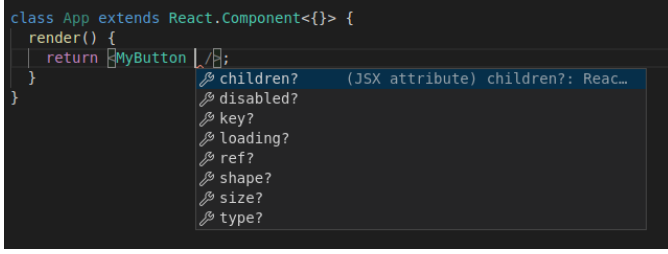
Fig. 3: An example of a component without type hints


```
class App extends React.Component<{}> {
  render() {
    return <my-custom-element />;
  }
}
          key?            (JSX attribute) React.Attribute…
```
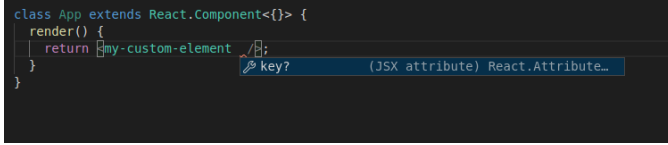
D. Javascript frameworks

While converting components to Web Components makes them usable in most JS frameworks, they do not provide a perfect experience. The first problem is them not being perfectly usable in every JS framework. As of the writing of this paper, there are still some issues preventing them from working completely in React JS [14]. 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 as such, do not integrate very well with the tooling provided by the framework. An example of type hinting provided by the framework and editor can be seen in Figure 2. Compared to Figure 3, which shows a component with no type hinting, Figure 2 provides the developer with much more information and shows them what options are available to them. Instead of having to search the web for the available properties, they are provided by the element's source code and displayed by the framework tooling and editor. In order to provide developers with proper tooling and type hinting for components, we will provide what we'll call a wrapper for each framework that either provides this tooling or does not support Web Components fully. This wrapper is written in the language the framework which it is targeting provides, and as such, allows the framework to infer information from its source code. Under the hood, this wrapper still makes use of the Web Components to render the components in the CC UI library, but this wrapper serves as glue code between the framework and the Web Components. By combining the two steps of converting the original Angular components to Web Components, and the Web Components to wrappers

for frameworks, we are able to provide developers with an experience that is native to their framework, regardless of the fact that the original source code was written in Angular.

II. 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 converting to Web Components), related work on Web Components themselves, and related work on the creation of wrappers around Web Components to target JS frameworks.

A. UI Libraries

We found a few studies that cover the area of UI libraries. Ky Nam *et al.* [1], Annala *et al.* [2], and Mráz *et al.* [3] all build UI libraries in their studies. They focus mostly on the technologies used, how they work, and how they contribute to the building of the UI library. Looking at blog posts, we find numerous posts on Web Components. In these blog posts [15] [16] [17] [18] [19], the authors provide guidance in setting up and creating a UI library. These blog posts mainly concern the basics, explaining how to get started with the process. We also find numerous examples of UI libraries. Some examples are Svelte Material UI [20] (written in Svelte), React Bootstrap [21] (React), Angular Material [22] (Angular), Wired Elements [23] (Web Components), Onsen [24] (multi-framework), and SyncFusion [25]. For all but the 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.

B. Angular Elements

Research on the area of Angular Elements is very sparse. In our search, we only found a single paper on this subject. Armengol Barahona *et al.* [4] uses Angular Elements to convert form components to Web Components so that they can be rendered dynamically. On the other hand, blog posts on this area are numerous. In various blog

---

[14]https://reactjs.org/

[15]https://www.toptal.com/designers/ui/design-framework

[16]https://dev.to/giteden/building-a-ui-component-library-for-your-startup-4cek

[17]https://www.emergeinteractive.com/insights/detail/how-to-ux-ui-design-system-component-library/

[18]https://codeburst.io/building-an-awesome-ui-component-library-in-2020-a85cb8bec20

[19]https://itnext.io/building-a-scalable-ui-component-library-4607de91955a

[20]https://sveltematerialui.com/

[21]https://react-bootstrap.github.io/

[22]https://material.angular.io/

[23]https://wiredjs.com/

[24]https://onsen.io/

[25]https://www.syncfusion.com/

posts [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37], the authors explain how to set up Angular Elements and how to use it to create a new component library. These blog posts mostly focus on creating new components or converting simple components through Angular Elements, not so much on converting larger and more complex components. They all use new and empty projects, contrary to two other blog posts [38] [39]. The authors use Angular Elements to convert existing AngularJS (an older version of Angular) components to the newer Angular. They do this by converting 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 convert the Angular code into Web Components, the Web Components are able to 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 are able to convert components one by one, converting the root component once all of its children have been converted as well.

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

C. Web Components

✎ Web Components section

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D. JS Framework Wrappers

We were unable to find any research on JS framework wrappers. This does not seem to be a problem that has been tackled very often, at least in literature. On the website *custom-elements-everywhere.com* [40], 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 [41]. In this blog post, the author explores various options to tackle this problem of passing non-primitive data.

III. Study Design

A. Metric definitions

In order to evaluate the effectiveness of converting an existing set of Angular components to a Web Component UI library, we need to define some metrics. We will be measuring two aspects of this process. The first is measuring the effectiveness of the resulting CC UI library. In order to do this, we will compare the components in the CC UI library to the Angular components they originate from, as well as to various different UI libraries. We will make this comparison using various metrics at both component granularity and at UI library granularity. We have chosen the following metrics: structural complexity (SC), cyclomatic complexity (CC), lines of code (LOC), maintainability (MA), size (SI), load time (LT), number of components in the UI library (NOC), and render time (RT). A brief description of these metrics can be seen in Table I. A detailed explanation of these metrics follows.

ID	Metric	Granularity	Description
SC	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	Cyclomatic complexity	Component	A quantitative measure of the number of linearly independent paths through a program's [5]
LOC	Lines of code	Component	The number of lines of code in a given component's source file
MA	Maintainability	Component	A derivative based on complexity, lines of code and Halstead volume [6]
RT	Render Time	Component	The render time of a given component
SI	Size	UI Library	The file size of the bundled up library
LT	Load Time	UI Library	Parsing and running time of the bundled up library in the browser (without download time)
NOC	Number of Components	UI Library	The number of components in a UI library

TABLE I: Metrics used in this study

[40]https://custom-elements-everywhere.com/

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

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.* [7]. In this paper, the authors evaluate the ability of various metrics to indicate the quality of Web Components. As such, we will be using these metrics to compare the quality of our Web Components to other Web Components. We'll be following almost all recommendations of 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 will not be using this to compare the quality of Web Components, but will instead be using 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 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) metrics. 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 will be comparing 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 will also not be using the consistency metric. The reason for this is fairly simple, namely that we don't have any components with the ability to update across different replicas. The same goes for the UI libraries we will be comparing it with.

2) Size: The size metric metric aims to measure the theoretical impact of loading the UI library over the network. We will be measuring 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 actually introducing the variable of network speed. In order to differentiate between a relatively large library and a library that has a lot of components, we also keep track of the number of components metric.

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. 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.

4) 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 being done with the rendering process This will likely be the most important metric for this study. 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 converting Angular components to Web Components will be too big. If it is slightly higher, the same or lower, we can conclude that the performance impact is minimal.

B. Metric targets

In order to get a sense of the state of the CC UI library, we will need to compare it to other UI libraries. To do so, we have gathered a list of various UI libraries targeting the JS frameworks that the CC UI library wrappers target. This way we are able to 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 were gathered by searching for the terms "Design Library", "UI Library", "javascript UI Library", "Svelte UI library", "React UI Library", "Angular UI Library", and "Web Component UI Library" on Google. We then added 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 II. While this is not a full list of all UI libraries, we feel like it is an accurate representation of the most popular UI libraries because of the fact that Google Search tends to return the most popular results in a given search. In order to get a fairly accurate representation of libraries using each JS framework, we selected the three UI libraries with the most github stars per JS framework. The list of included UI libraries can also be seen in Table II. In addition to comparing the CC UI library against other UI libraries, we will also be comparing it against the Angular components they originate from. 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 do a proper comparison between individual components of the UI library. For example a button component can not be compared with a date picker component, since date pickers tend to be a lot more complex. As such, 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, and Switch (or Checkbox). Since every UI library contains all of these, we can compare the metrics for a single component across all UI libraries. We will only be applying the various metrics to these three components in each UI library. We will also be including a stripped down version of the CC UI library in the set of UI libraries we gather metrics of. This version will only contain the three components mentioned above, and will allow for a fair comparison with other UI libraries. The reason for this will be further explained in Section IV-E.

IV. Experimental Setup

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

A. Gathering components

The SC, CC, LOC, and MA metrics are captured over 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 biggest source file for components as the entrypoint. Some UI libraries use a simple index file that re-exports the actual source file as the entrypoint. If this file were to be used as the entrypoint, it would result in an unrealistic depiction of the component source. Since the UI libraries we in this study always contain a single big source file, this did not result in any situations where the entrypoint was ambiguous.

B. 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 [7]. To gather these imports, we use the `typescript`[42] JS package. This package is able to generate an AST, or abstract syntax tree of the file. 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 [7], we only apply this process to the JS source code of a component, not the HTML source code. In the case of Svelte components, we separate the file into its JS code and HTML code and apply the process to the JS code only.

C. Cyclomatic Complexity, Maintainability, Lines of Code

In order to capture the cyclomatic complexity, maintainability metrics, and lines of code metrics, we feed the file into the `ts-complex`[43] JS package. This package is able to calculate the cyclomatic complexity, maintainability metrics, and lines of code of 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.

D. Machine specifications

All experiments were performed on a machine with an AMD Ryzen 5 4600H Six-core processor and 16GB of RAM. Since these experiments are partially timing-specific, the timing-specific experiments were ran sequentially and with minimal background tasks. This should eliminate the effect of experiments on each other, and should ensure the CPU is always able to dedicate a single core to the experiment. Since this machine has six cores, it should easily be able to dedicate one of them to the experiments at all times.

E. Size

In capturing the size metric we need to pay attention to a number of influential factors. The first factor is the fact that the source code of files is split up into multiple files, some of which are actually part of the build and some of which are not. Measuring 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 this issue, we use a JavaScript bundler. This 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 such as files that contain metadata. We will be using the `esbuild`[44] bundler for this process.

Another factor is the way in which the source code is written. A file containing a lot of comments will be larger than a file containing no comments. An 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 factor is the number 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 exactly the same with the exception that one of them contains a complex chart component, the library that contains the chart will still be bigger on average. This is the case regardless of the fact that any given component in the other library is the exact same size, the chart library just has more components. To get around this issue, we make use of *tree shaking*. 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 marking the same three components as components that should be included into the build for every UI library, we are able to create bundles that contain the exact same functionality and nothing more.

The tree shaking process is applicable to the UI libraries we are comparing the CC UI library against, however it is not applicable to the CC UI library itself. This is the case because every component is registered as a Web

---

[42]https://www.npmjs.com/package/typescript
[43]https://www.npmjs.com/package/ts-complex
[44]https://esbuild.github.io/

Component simply by loading the library, and as such every component is used. This would lead to the CC UI library having a much larger size than other UI libraries. To provide a fair comparison, we will be adding a stripped down version of the CC UI library to the set of UI libraries we are comparing against. This stripped down version only contains the three basic components, and as such allows for a 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 size of the resulting bundle.

F. Load Time

For the load time metric we will be capturing the parse- and runtime of the JS bundle described in Section IV-E. We explicitly exclude the network load time of the bundle. In order to collect this metric, we will be using the `puppeteer`[45] package. This package allows the running of and programmatic control over a headless browser (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 [46] for the page. 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 4.
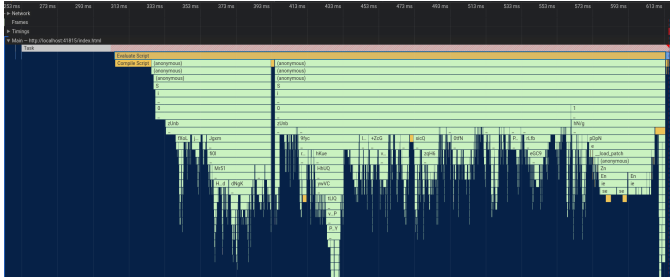

Fig. 4: An example of a Chrome profiler trace performed during a bundle load. The orange bar labeled "Evaluate Script" indicates the total load time and spans 308.54ms.

In order to eliminate variations as much as possible, we artificially slow down the speed of the processor by a factor of ten by using the `Emulation.setCPUThrottlingRate` command [47]. We also perform this operation five times and average the results.

G. 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

---

[45]https://github.com/puppeteer/puppeteer
[46]https://developer.chrome.com/docs/devtools/evaluate-performance/
[47]https://chromedevtools.github.io/devtools-protocol/tot/Emulation/#method-setCPUThrottlingRate

---

allows them to 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 1,2, 3, 4, 5. 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. 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.

```
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 1: The render-on-demand function in ReactJS

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

Listing 2: 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 3: 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 4: 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 5: The render-on-demand function in Web Components

We chose this specific event for the following reason. The chrome browser updates the view through a pipeline process. The full pipeline can be see in Figure 5. This process always starts with a JS, CSS, or HTML change. Then it performs a different set of pipeline events depending on what changed.
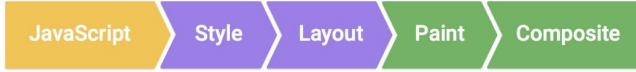

Fig. 5: The render pipeline in chrome

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 ran.
- *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 ran when an animation is active.

In capturing the render time, we want to capture the time until a component reaches its final state. While this state is fairly simple and 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 lasts 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 perfectly capture the time a component takes to render. An visual representation of this rendering process can be seen in Figure 6 and Figure 7.

While it has been shown that load time and perceived performance of a web page are not necessarily the same, as shown by Nathan *et al.* [8] and Gao *et al.* [9], we are able to use this metric on single components as a way to compare the CC UI library to the various UI libraries in performance.
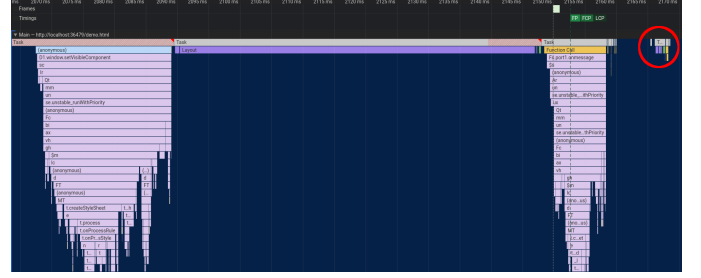

Fig. 6: An example of a Chrome profiler trace performed during the render of a component. The bar labeled "window.setVisibleComponent" indicates our start time. The end time falls within the red circle, a zoomed in version of which can be seen in Figure 7.
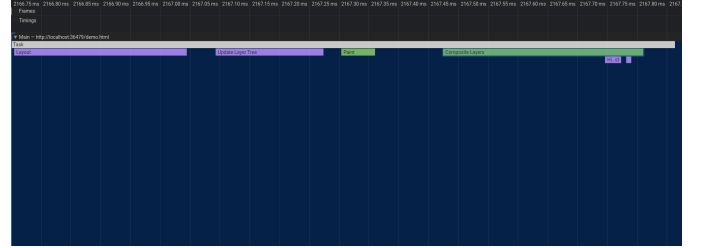

Fig. 7: An example of a Chrome profiler trace performed during the render of a component. The bar labeled "composite layers" signals our end time. Note that this falls just after a bar labeled "paint", signaling the paint event before the last composite event.

H. Number of Components

The number of components is captured separately for the UI library as a whole, and for the bundle described in Section IV-E. Since the bundles described in Section IV-E 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 will be gathering the number of components by applying the process described in Section IV-A to gather components, after which we count the number of them.

V. CASE STUDY

In this section we'll first be laying out the steps taken and the issues faced during the conversion from Angular components to Web Components. We'll be splitting this section up into two subsections. The first subsection will consist of issues faced that were not specific to Angular or Angular elements, while the second subsection will contain just Angular-specific issues. Note that because of this split the issues are no longer listed out in chronological order.

For a chronological overview of the various issues and their relative complexities see Table III. After the issue sections we'll discuss the various JS framework wrappers and how they were created. Lastly we'll be listing optimizations performed along with their effectiveness.

A. Web Component Issues

1) WC1: Global CSS:

Problem:
Angular components have a property called `encapsulation` [48]. 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 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 and Listing 7.

When converting the Angular components to Web Components, we ensure the components' contents are rendered within a ShadowRoot [49]. This effectively separates the component from the rest of the DOM, thereby also removing the ability of the global 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 [50] if it hasn't already been copied. We then use the `adoptedStylesheets` [51] property of a ShadowRoot to apply the global stylesheet in the component's own ShadowRoot as well. Because this Constructable Stylesheet is just a reference to a stylesheet that the browser re-uses, the performance impact of this process is minimal compared to copying the stylesheet's source code.

```
1  // my-component.html
2  <my-component></my-component>
3
4  // my-component.css
5  :host {
6    color: red;
7  }
```

Listing 6: The source code for a component

```
1  // my-component.html
2  <my-component _ngcontent-uix-c290></my-component>
3
4  // my-component.css
5  [_ngcontent-uix-c290] {
6    color: red;
7  }
```

Listing 7: An example of compiled code for the component in Listing 6.

2) WC2: Compatibility:

Problem: While browser support for Web Components is fairly wide as of this case study [52], it is not yet universal. Additionally, Safari has chosen not to implement support for so-called *Customized Built-In Elements*[53]. This feature allows for the extending of 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[54]), we need this feature to make the CC UI library work.

Solution: We add so-called polyfills to the final JS bundle. These are files that add support for unsupported features by implementing them in different ways, making use of the builtin feature if there already is support for it. In particular we use the `custom-elements`[55] and `custom-elements-builtin`[56] polyfills.

3) WC3: Tagname renaming:

Problem: As per the Web Components specification, all Web Components are required to have a hyphen in their tag name [57]. 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>`). However, this renaming leads to an issue with components that are being used inside other components. For example say the `DoubleCheckbox` component renders two checkboxes. Source code for

---

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

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

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

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

[52]https://caniuse.com/?search=components

[53]https://www.chromestatus.com/feature/4670146924773376

[54]https://angular.io/guide/attribute-directives

[55]https://www.npmjs.com/package/@ungap/custom-elements

[56]https://www.npmjs.com/package/@ungap/custom-elements-builtin

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

such a component can be seen in Listing 8. 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 are going to be used in the UI library with their prefixed variant. We make sure to not 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>
```

Listing 8: The source code for a `DoubleCheckbox` component.

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*[58]. 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 9. By changing the styles of the underlying Angular components to use CSS Custom Properties when available, we are able to provide theming. An example of this can be seen in Listing 10.

```
1 function setPrimaryColorTheme(color: string) {
2   document.documentElement.style.setProperty('color-
      primary', color);
3 }
4
5 setPrimaryColorTheme('red');
```

Listing 9: Applying CSS Custom Properties to the document

```
1 my-component {
2   /**
3    * Tries to use the --color-primary variable if
         available,
4    * falls back to blue when it is not defined.
5    */
6   background-color: var(--color-primary, blue);
7 }
```

Listing 10: An example of a component making use of CSS Custom Properties

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 of which 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 V-A6, the CC UI library should be at least usable by itself as well.

Solution: Because in this scenario there is no access to JavaScript, we can not make use of any solutions to this problem that utilize the setting of attributes or properties through JavaScript. Instead, we need to use just HTML. 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` [59], which, as mentioned on the documentation page, is not secure. We also feel like this problem is not prevalent enough in our case to warrant such a solution. Currently, all occurrences of classes as Angular component inputs have been taken care of.

6) WC6: Complex Attributes:

Problem: Continuing on with the same problem, we now need to come up with a way to solve the problem for even more situations, this time being able to make use of JavaScript since 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 the fact 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, assuming that they will only receive attributes once and that they'll never change.

Approaches: There are a few approaches to solving this issue, these can be grouped into three categories.

---

[58]https://developer.mozilla.org/en-US/docs/Web/CSS/Using_
CSS_custom_properties

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

- *Parent → child:* The parent node gets a reference to the child during the rendering process. Then the parent sets the properties on the child.
- *Child → parent:* 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:* 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, however, this approach is not always possible. 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've 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 React and Vue [60] 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 a number of steps. We start off by creating a class which we'll call *Intermediary*. This class has an instance manager attached to it which we'll 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're 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'll be storing it as a property of the defined Web Components. This means that we're able to access the `IntermediaryManager` property on the `customElements.get('cow-checkbox')` object and get a reference to the IntermediaryManager from both the parent side and the child side.

Now that we've taken care of this issue, we're 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 value is already known by the Intermediary, 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.

B. Angular Issues

1) A1: ng-deep:

Problem: Angular provides the `ng-deep` CSS selectors [61]. 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. It is an emulated and deprecated selector 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 [62], with both `::shadow` and `/deep/` making it into Chrome, they have since both been removed [63]. 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 11 and Listing 12.

```
1 // parent-component.html
2 <child-component></child-component>
3
4 // parent-component.css
5 ::ng-deep div {
6   color: red;
7 }
8
9 // child-component.ts
10 @Component({
11   ...
12 })
13 class ChildComponent {
```

[61] https://angular.io/guide/component-styles#deprecated-deep-and-ng-deep

[62] https://drafts.csswg.org/css-scoping/

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

[60] https://vuejs.org/

```
14    ...
15 }
```

Listing 11: A component before the ng-deep change

```
1  // parent-component.html
2  <child-component red></child-component>
3
4  // parent-component.css
5  /**
6   * ::ng-deep div {
7   *   color: red;
8   * }
9  */
10
11 // child-component.ts
12 @Component({
13   ...
14 })
15 class ChildComponent {
16   @Input() red: boolean;
17
18   constructor(private _elementRef: ElementRef) {
19     if (this.red) {
20       _elementRef.nativeElement.classList.add('red')
           ;
21     }
22   }
23 }
24
25 // child-component.css
26 :host[red] {
27   color: red;
28 }
```

Listing 12: A component after the ng-deep change

2) A2: createCustomElement:

Problem: The main export of the Angular Elements library is the `createCustomElement` function [64]. 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 builtin accessibility features provided by the browser. When converting this Angular component to a Web Component, we also wish to preserve these same features. This can be done by extending builtin HTML elements [65]. However, 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 [66].

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,

---

[64] https://angular.io/api/elements/createCustomElement
[65] https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements
[66] https://github.com/angular/angular/issues/19108

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.

3) A3: EventEmitters:

Problem: Angular implements so-called `EventEmitters` [67]. 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`, 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 converted 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 13.

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

Listing 13: A change made to an event listener. The definition of `unwrapEvent` can be seen in Listing 14

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

Listing 14: The `unwrapEvent` function

4) A4: Hierarchical Injectors:

Problem: Angular makes use of a feature called *Dependency Injection* [68]. This allows a parent module to provide its children with an instance of a certain 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. However,

---

[67] https://angular.io/api/core/EventEmitter
[68] https://angular.io/guide/dependency-injection

this dependency injection feature can also be leveraged to have a given component provide its own instance of a class only to its direct 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*[69] and it is a feature that is utilized by 30MHz in some areas.

Angular Elements does not support this feature intentionally [70]. Instead, it only supports the use case where modules provide their components with dependencies. The reason for this is that every component converted with Angular Elements is mounted to the DOM as its own root. It does not have a concept of a parent component, and as such is unable to look up the injector of its parent. While the pattern of Hierarchical Dependency Injection is not recommended [71], it is still a pattern used by 30MHz, and as such one we need to support 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 [72].

We first need to find the parent. This process is fairly straightforward. When the child is being rendered, we simply travel up the DOM tree until we find a node that has certain 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 very simple 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 so-called `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 craft this new

injector class instance we need a reference to that very 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 injector, one of which is the previously mentioned `Injector` and the other is the so-called `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 fairly 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 full code example of this process can be found in Listing 17.

5) A5: ngOnInit:

Problem: Angular Elements does intentionally not guarantee the order in which attributes are set on an element (even initial attributes) [73]. 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 15. While this isn't a problem if attributes are only used to handle visual state, this can cause great problems when they are used for 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 is set before the component's 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 full 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

---

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

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

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

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

[73]https://github.com/angular/angular/issues/29050

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 ran. 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.

```
1  // HTML source file
2  <my-element foo="bar" bar="baz" />
3
4  // can be transformed into any of the following:
5  // 1
6  const element = document.createElement('my-element')
      ;
7  element.setAttribute('foo', 'bar');
8  element.setAttribute('bar', 'baz');
9  parent.appendChild(element);
10
11 // 2
12 const element = document.createElement('my-element')
      ;
13 parent.appendChild(element);
14 element.setAttribute('foo', 'bar');
15 element.setAttribute('bar', 'baz');
16
17 // 3
18 const element = document.createElement('my-element')
      ;
19 element.setAttribute('foo', 'bar');
20 parent.appendChild(element);
21 element.setAttribute('bar', 'baz');
```

Listing 15: HTML source code and its Angular Elements equivalent

6) A6: Casing in attribute names:

Problem: In the process of converting 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`. While this is a good change to make, and one that ensures that name collisions are less likely to happen, it does present some issues. 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.

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 look 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 tough 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 conversion 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 actually works and the directive works perfectly.

8) A8: <ng-content>:

Problem: Angular uses the `<ng-content>` tag for so-called 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 [74]. An example of content projection can be seen in Listing 16. Content projection works fine in most scenarios but sometimes it does not work. The 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

---

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

source code. This ensures that when the `<ng-content>` tag does not work, the `<slot>` tag takes over instead.

```
1  // parent-component.html
2  <child-component>
3    <span id="my-span"></span>
4  </child-component>
5
6  // child-component.html
7  <div id="my-root">
8    <ng-content></ng-content>
9  </div>
10
11 // Effective DOM tree
12 <parent-component>
13   <child-component>
14     <div id="my-root">
15       <span id="my-span"></span>
16     </div>
17   </child-component>
18 </parent-component>
```

Listing 16: HTML source code and its Angular Elements equivalent

C. JS Framework Wrappers

We wrote JS framework wrappers for a total of three JS frameworks. The first one is React. Since complex attributes for Web Components do not work natively in React, this wrapper was required in order to get the CC UI library to work in the first place. Since React 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 React wrapper. The second JS framework is Svelte. While Svelte works perfectly with Web Components out of the box, we created a wrapper simply to provide typings for the developers similar to React. 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 when it came to HTML elements. The UI libraries we looked at were Vue (v2 and v3) [75], Polymer [76], lit-element [77], and wc-lib [78].

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 [79]. This package allows for the parsing of Type-Script 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 fairly 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 React and Svelte wrapper. In the process of creating an Angular wrapper however, we faced a few challenges. These challenges are described in detail below.

1) A9: Angular Attribute Order:

Problem: As previously described in Section V-B5, 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 obvious issue where the wrong contents are rendered.

Another problem stems from our approach in Section V-A6. 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 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.

2) 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

---

[75]https://vuejs.org/

[76]https://www.polymer-project.org/

[77]https://lit-element.polymer-project.org/

[78]https://www.npmjs.com/package/wc-lib

[79]https://www.npmjs.com/package/typescript

everything into a combination of browser-specific bundles. These bundles can then easily 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 [80] 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 however refuses to do this. They recommend using `peerDependencies` [81] instead. While their reasoning for this is valid, it does not apply in our case. Since 3rd 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` [82], and a little while later it was deprecated [83]. 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 however, 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 entrypoint 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 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 don't 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.

3) A11: Angular Ivy:
Problem: Angular has released a new version of its compiler called Ivy [84]. This Ivy compiler is set to replace the previous View Engine compiler. However, Angular does not allow the use of Ivy for projects built as libraries for now [85] 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 a number of bugs, one of which is causing the build to fail entirely in our case [86]. 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 [87]. Unfortunately the disabling of AOT introduces significant overhead during the loading of the built library, an issue which 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.

---

[80] https://fontawesome.com/pro
[81] https://github.com/ng-packagr/ng-packagr/blob/v10.1.0/docs/dependencies.md
[82] https://github.com/ng-packagr/ng-packagr/issues/1106
[83] https://github.com/ng-packagr/ng-packagr/commit/0c52486

[84] https://angular.io/guide/ivy
[85] https://angular.io/guide/ivy#maintaining-library-compatibility
[86] https://github.com/angular/angular/issues/25424
[87] https://github.com/angular/angular/issues/25424#issuecomment-465643237

D. Optimizations

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

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'll be referring to this CSS file as the CC CSS file from now on.

As mentioned in Section V-A1, 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's 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 file 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 IV-D. 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.

We improve this process in two ways. The first step is to simply stop scanning after we find the marker. 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 improvements 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 be able to add this attribute. At runtime we check whether there are any stylesheet tags that 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.

2) O2: Move CSS searching to initial load: While the above fixes provide great 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 quite large, and the fact that it is ran 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 huge.

We remove this performance impact from the first render by moving it to the moment the CC UI library is initialized. To make sure we don't 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.

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	React	Yes	5.0.0-alpha.28	https://material-ui.com/
React Bootstrap	19.2k	React	Yes	1.5.2	https://react-bootstrap.github.io/
React Semantic UI	12.2k	React	Yes	2.0.3	https://react.semantic-ui.com/
Evergreen	10.6k	React	No	-	https://evergreen.segment.com/
Rebass	7.2k	React	No	-	https://rebassjs.org/
Grommet	7.1k	React	No	-	https://v2.grommet.io/
Baseweb	6.2k	React	No	-	https://baseweb.design/
Ant Design	5.3k	React	No	-	https://ant.design/
Elemental UI	4.3k	React	No	-	http://elemental-ui.com/home
Zendesk Garden	858	React	No	-	https://garden.zendesk.com/
Shards React	649	React	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
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/
Syncfusion	unknown (not on github)	Multi-Framework	No (paid)	1.0.0	https://www.syncfusion.com/

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

Section	Section Name	Relative Complexity
V-A1	Global CSS	simple
V-A2	Compatibility	simple
V-A3	Tagname renaming	simple
V-A4	Theming	simple
V-A5	Non-string Attributes	medium
V-A6	Complex Attributes	complex
V-B1	ng-deep	simple
V-B2	createCustomElement	simple
V-B3	EventEmitters	simple
V-B4	Hierarchical Injectors	hard
V-B5	ngOnInit	medium
V-B6	Casing in attribute names	simple
V-B7	Angular directives	simple
V-B8	<ng-content>	simple
V-C1	Angular Attribute Order	medium
V-C2	Bundling Angular Imports	hard
V-C3	Angular Ivy	hard
V-D1	Reduce time searching for CSS	simple
V-D2	Move CSS searching to initial load	simple

TABLE III: Sections in chronological order along with their relative complexities

REFERENCES

[1] T. Ky Nam, "Ui library project setup for vaimo group with modern web technology," 2019.

[2] L. Annala, "Documentation of a ui-library used in web development," 2017.

[3] M. Mráz, "Component-based ui web development," 2019.

[4] J. Armengol Barahona, "Development of an angular library for dynamic loading of web components," B.S. thesis, Universitat Politècnica de Catalunya, 2020.

[5] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[6] M. H. Halstead, "Elements of software science," 1977.

[7] A.-L. Martinez-Ortiz, D. Lizcano, M. Ortega, L. Ruiz, and G. López, "A quality model for web components," 11 2016, pp. 430–432.

[8] V. Nathan, "Measuring time to interactivity for modern web pages," Ph.D. dissertation, Massachusetts Institute of Technology, 2018.

[9] Q. Gao, P. Dey, and P. 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*, 2017, pp. 13–18.

```
1  // This injector is provided by Angular to the
       root module
2  var rootInjector = ...;
3
4  // Create a fake injector
5  const _MOCK_INJECTOR = {
6    get() {
7      return {
8        run() {},
9        resolveComponentFactory() {
10          return {
11            inputs: [],
12          };
13        },
14      };
15    },
16  };
17
18  // Extract defualt NgElementStrategy
19  function getDefaultNgElementStrategy() {
20    const customElement = createCustomElement(
         EmptyAngularComponent, {
21      injector: _MOCK_INJECTOR,
22    });
23    const proto = customElement.prototype;
24    const strategyInstance = proto.ngElementStrategy
         ;
25    return strategyInstance.constructor;
26
27  }
28
29  // Get the NodeInjector class
30  function getNgInjectorClass(rootInjector) {
31    const componentFactory = rootInjector.get(
         ComponentFactoryResolver).
         resolveComponentFactory(
         EmptyAngularComponent);
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 17: The code for creating a Hierarchical Injector in an Angular Elements component