Web Components for Design Systems

Elke Perner, Michael Pollak, Oliver Reichmann, Martin Sackl

Institute of Interactive Systems and Data Science (ISDS),
Graz University of Technology
A-8010 Graz, Austria

07 Dec 2020

Abstract

The aim of this paper is to introduce to a state of the art technology to build websites, Web Components. This survey should give an overview of what Web Components are, which features it has, the advantages and possible disadvantages, what libraries are there to help with developments and how they can be used in bigger design systems.

Furthermore, this paper will show examples on how to build such components with different helper libraries, and compare a few of these in matter of how big, fast and popular they are. All things considered, this paper wants to give an overview of a fairly new technique in web development that is supported by all modern browsers.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

To develop simple content for the web, we use the Hypertext Markup Language (HTML). Since HTML alone looks plain and simple we additionally use Cascading Stylesheets (CSS) to make it look pretty and Javascript to add some functionality to the front-end, often with the help of big frameworks, like Vue, Angular or React just to name a few popular ones. But most of them also bring a lot of overhead with them, one has to install additional components, learn how to use it and find out if they work on the variety of browsers. Additionally CSS can get very confusing when working on large projects. That is exactly why web components, small, independent code blocks are the perfect solution. They are versatile, reusable and supported by all modern browsers and therefore run on all devices.

Chapter 2

What are Web Components

As the name already tells, web components are specific components or elements created to be used in websites or web applications. Basically, a component consists of one or multiple HTML elements, including some content and style, and can be easily inserted in HTML code. Goal is to create individual, reusable and completely independent HTML tags which can be, if required, integrated multiple times on a site without conflicting with other elements. Therefore, web components mainly consist of three web-APIs, which are **Custom Elements**, **Shadow DOM** and **HTML Templates**. With the help of these specifications, the content of web components is not visible in the actual HTML code, as they are represented as single, fully encapsulated elements. Web components thus show several advantages like reusability, readability, scoping (using shadow DOMs), declarations and consistency.

2.1 Custom Elements

There are a bunch of different HTML element types. Some of the most common tags are div, p, a, span, table and so on. In some cases, these tags are completely sufficient but often it can be an advantage to define specific custom elements. By default, such custom elements would be rendered and displayed neutrally, so they have to be explicitly defined and the browser has to be informed how to handle these elements. This can be done by using JavaScript, which is shown in the following listings.

```
class MyCustomElement extends HTMLElement {
  constructor() {
  super();
  this._myVariable = null;
}

attributeChangedCallback(newValue) {
  this._myVariable = newValue;
}

connectedCallback()  {
  this.update();
}

update() {
  ...
}
}
```

Listing 2.1: Custom Tag Class

2.1.1 Defining a Custom Element

Defining a custom tag is simply done by `window.customElements.define(<my-custom-element>, MyCustomElement); .`

The second parameter points to the new class, which also has to be defined. This class extends HTMLElement and besides a constructor can include functions for changing attributes or a connected callback. See listing 2.1 for an example.

After defining the new element, it can be used in the HTML code like `<my-custom-element></my-custom-element>`.

2.2 Self-closing Custom Elements

Even if a custom element does not contain content it is not possible to create self closing tags. It is not supported to abbreviate <ok-cancel></ok-cancel> to a simplified <ok-cancel/> because the standard lists all self closing - or void - elements. The handling of self-closing elements is browser specific, in our testing Mozilla Firefox as well as Chromium did present the non-valid code correctly.

These tags are listed as valid void elements

- **area**
- **base**
- **base**
- **col**
- **embed**
- **hr**
- **img**

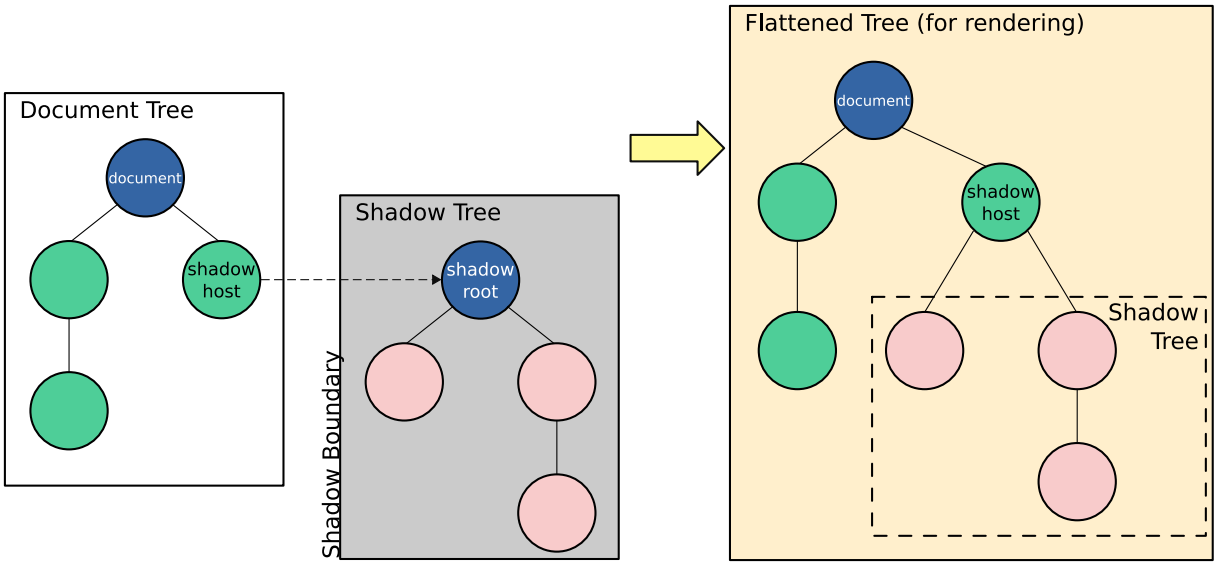

Figure 2.1: Illustration of the Shadow DOM. [Image extracted from MDN [2020] and used under the terms of CC-BY-SA 2.5.]

- **input**
- **link**
- **meta**
- **param**
- **source**
- **track**
- **wbr**

For more information see the syntax specifications: `https://html.spec.whatwg.org/multipage/syntax.html#void-element`.

2.3 Shadow DOM

A DOM (short for Document Object Model) is an interface between the HTML and JavaScript used on a website or web application and can be seen as a tree of elements of the HTML structure. In this tree, all elements are objects, which can be changed or deleted, as well as new objects added with the help of JavaScript. It is also possible to edit attributes and style of these objects.
Now a shadow DOM is also a DOM, but represents a single, completely independent node with it's own range of validity within the origin HTML DOM of a website. It encapsulates HTML and CSS from the HTML DOM and it's content is therefore not visible to debuggers or scripts. Figure 2.1 shows a normal document tree with a shadow DOM (marked as shadow host) node. The advantage of such shadow DOMS is that different parts of elements (styles, names, IDs, etc.) do not clash with other elements in the root DOM. Almost all available web component libraries are using shadow DOMs and hence a shadow DOM is a very important aspect.

2.4 HTML Templates

Basically, HTML templates are mark-up templates, which are not displayed on the actual site, but are there to build the base of user-defined custom elements, which can then be used on the website several times. The containing elements of such a template stay inactive and are not getting rendered, unless this is explicitly called by a script. Their advantage is that they do not have any impact on the page loading time and are therefore great alternatives to other JavaScript approaches. Two important elements used for such HTML templates are <template> and <slot>.

2.4.1 <template>

With this tag, HTML templates can be defined. As already mentioned, these elements are not displayed at the initial page load. It's main purpose is to be a fundamental structure for custom elements.

2.4.2 <slot>

The slot-tag is a placeholder and can be filled with custom content. With these tags, different DOM-fragments can be created with a single HTML template.

2.5 TypeScript Support

To develop and use web components, a good understanding of JavaScript is needed and advised. Since many web developers prefer the programming language TypeScript over Javascript, a big question is, if different web component libraries support TypeScript. TypeScript is an open source programming language based on JavaScript (TypeScript code is also JavaScript code, but not the other way around) and extends it by using types. These types describe the shade of objects. In general TypeScript provides better documentation of objects, catches errors and makes the code easier and clearer. TypeScript, for example, checks if variable types like string or number are always assigned, which JavaScript does not. This can prevent errors and bugs in the code. According to Barot [2019], users who have used TypeScript and would use it again increased from around 20% to over 45%. Also current statistics on npm (over 15 million weekly downloads) and GitHub (more than 2000 watches and over 65.000 stars) show that this programming language's user base is increasing permanently. Therefore, TypeScript supporting web component libraries have an advantage against non-supporting libraries. We will have a more detailed look about which libraries are supporting TypeScript in a later chapter, including LitElement, Stencil, FAST and Gallop where on a special note libraries like Svelte and Riot are supporting it, but need a preprocessor therefore.

Chapter 3

Classification

Since there are quite a lot of different web components libraries available online (webcomponents.dev states over 40 different libraries on their website, shown in figure 3.1) it is a good idea to classify these libraries. One approach therefore comes also from webcomponents.dev. They classify them into HTMLElement-based, class-based, hook-based, object-based, compiler-based and wrapped into a custom widget. This approach of classification mostly relies on the style of coding of the individual library and does not really relate to their functionality. Since web components are always HTML-based and also using classes by default, but some libraries may require writing an object or a hook, we decided to make this classification more general and are now dividing web components libraries into compiler-based and class-based libraries. As the classification names already tell, the code of compiler-based libraries needs to be compiled into native custom elements, without any dependency to a framework.

3.1 Comparison

As already mentioned, there exist several libraries, offering a simple and promising way to create own, custom web components, rather than write all code by yourself. Using these libraries can be more efficient and can lead to cleaner code. But which web components library is best? Which offers the best functionalities and which library offers the easiest use? This, of course, is a very important question and partially subjective. Rather than asking which is the best library, developers should choose the library, which fits their use cases best. These could more relate to bundle size or performance, but also on given technical conditions and also individual coding preferences. To give an overview, state some important facts, which should be considered in choosing the right library, and find the most promising frameworks a ranking was performed on several libraries of the webcomponents.dev website (see figure 3.1 for the whole list of libraries). We therefore selected 15 different libraries, each with a slightly different, promising aspect (good and bad aspects), which are shown in table 3.1 . The table does not include the column *Use of Shadow DOM*, since all of the libraries are using it.

3.2 Parameters for Ranking

- **Category**
  if the library is compiler-based or class-based (see previous section for more details)
- **TypeScript Support**
  does the library support TypeScript, does it need a preprocessor
- **Provides a Component Library**
  does the library provide an online component library, in which users can select predefined components

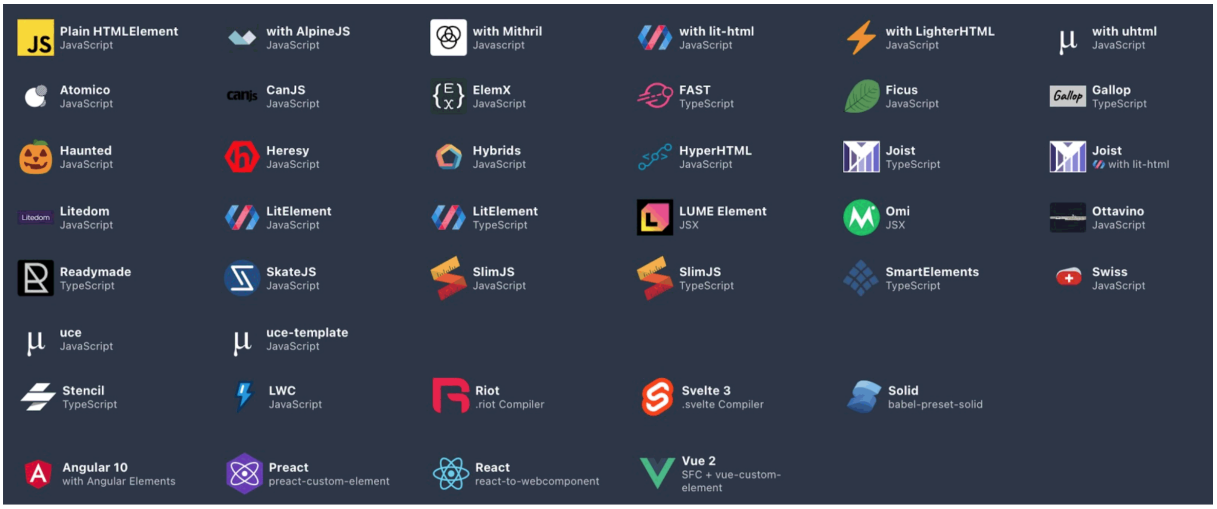

Figure 3.1: List of web component libraries. [Image extracted from DIV-RIOTS [2020]. ©by DIV-RIOTS.]

- **Use of Shadow DOM**
  is a shadow DOM used in this library
- **Minified Bundle Size**
  the minified bundle size in KB for one component, taken from DIV-RIOTS [2020], Figure 3.2 shows the comparison of all libraries
- **Learning Effort**
  a subjective ranking from 1-10 (where 10 is no learning effort and 1 is a high learning effort), ranked by us
- **Performance in Browser**
  the performance of parsing JavaScript and creating the DOM tree in ms, benchmarked with Marks [2020] in Google Chrome, taken from DIV-RIOTS [2020], Figure 3.3 shows the comparison of all libraries
- **User Base**
  ranking from 1-10 (where 10 is a high user base and 1 is no/small user base) based on weekly downloads (from npm) and rated stars (from github)
- **Good Documentation**
  a subjective ranking from 1-10 (where 10 is a very good documentation and 1 is no/bad documentation provided on the official library website)

From the table above, we selected the five most interesting web component libraries and wanted to find out the individual differences in programming with these libraries.

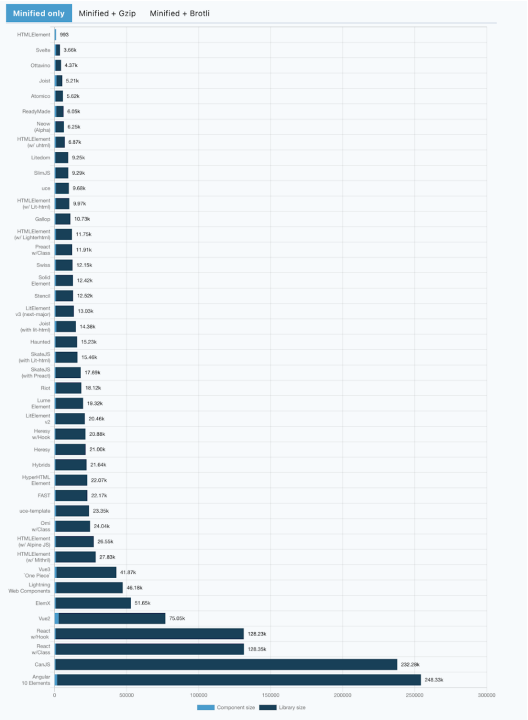

Figure 3.2: Minified bundle sizes in KB. [Image extracted from DIV-RIOTS [2020]. ©by DIV-RIOTS.]

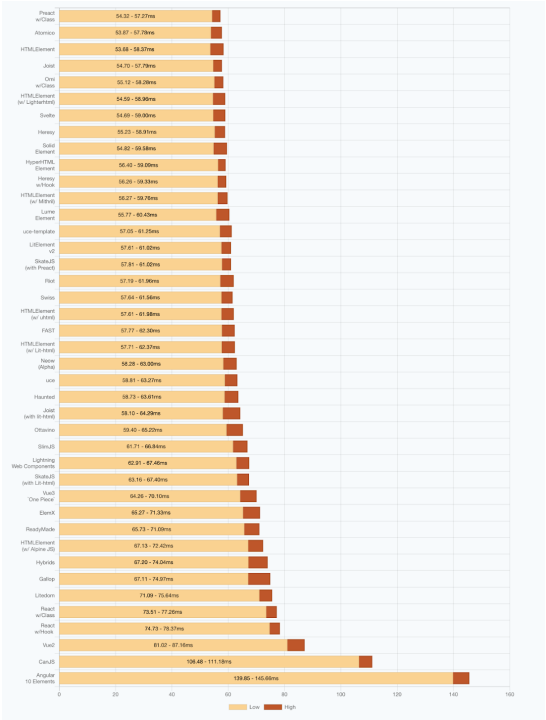

Figure 3.3: Performance of parsing JavaScript and creating the DOM in milliseconds. [Image extracted from DIV-RIOTS [2020]. ©by DIV-RIOTS.]

Name	Category	TypeScript	Size	Performance	User Base	Documentation
HTMLElements	Base	Yes	10	10	10	10
FAST	class-based	Yes	5	7	6	5
Joist	class-based	Yes	10	9	2	3
LitElement	class-based	Yes	6	7	9	10
Neow (Alpha)	class-based	Yes	8	5	1	3
ReadyMade	class-based	Yes	8	2	3	7
Riot	Wrapped	preprocessor	5	9	1	8
Svelte	Wrapped	preprocessor	10	9	6	7
Lightning Web Components	Compiler	Yes	3	4	6	10
Stencil	Compiler	Yes	5	-	2	9
Ottavino	Object based	No	10	4	4	5
Atomico	Hook based	Yes	8	10	2	6
Gallop	Hook based	Yes	6	2	2	1
Haunted	Hook based	Yes	5	4	5	7

Table 3.1: Web Component Libraries, sorted by classification

Chapter 4

Example of Selected Libraries

From the table above, we selected the following three libraries for the development of a simple example:

- HTML Elements
- LitElement
- Stencil

The example we created is a simple OK-Cancel Button. See Figure 4.1 for the preview of the final result.

4.1 HTML Elements

See the example code in Listing 4.1. The full code can be found on GitHub [Pollak 2020a].

4.2 LitElement

This example in 4.2 is a bit shortened as the css details are not crucial. The full code can be found on GitHub [Pollak 2020c]. For a narrated version of the source code we recommend our video [Pollak 2020b].

Figure 4.1: Result of the OK-Cancel Button example. [Image created by the authors.]

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8"/>
    <title>Web Components Test - HTMLElements</title>
</head>
<body>
    <ok-cancel></ok-cancel>

<script>
const template = document.createElement('template');
template.innerHTML = `
  <style>
    * {
      font-size: 200%;
    }
    button {
      width: 8rem;
      height: 4rem;
      border: none;
      border-radius: 0.7rem;
      background-color: green;
      color: white;
    }
    button#cancel {
      background-color: red;
    }
  </style>
  <button id="ok">ok</button>
  <button id="cancel">cancel</button>
  <div id="feedback"></div>`;

class okcanceldialogue extends HTMLElement {
  constructor() {
    super();
    this.feedback = "";
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.shadowRoot.appendChild(template.content.cloneNode(true));
    this.shadowRoot.getElementById('ok').onclick = () => this.ok();
    this.shadowRoot.getElementById('cancel').onclick = () => this.cancel();
  }

  ok() {
    this.shadowRoot.getElementById('feedback').innerHTML = "thanks for clicking ok."
        ;
  }

  cancel() {
    this.shadowRoot.getElementById('feedback').innerHTML = "thanks for nothing.";
  }

}

customElements.define('ok-cancel', okcanceldialogue);

</script>
</body>
</html>
```

Listing 4.1: HTML Elements

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8"/>
    <title>Web Components Test - LitElement</title>

</head>
<body>
    <ok-cancel></ok-cancel>

<script type="module">
import { LitElement, html, css } from 'lit-element';

export class OkCancel extends LitElement {
  static properties = {
    feedback: { type: String },
  };

  static styles = css`
    * {
      font-size: 200%;
    }
    button {
      width: 8rem;
      height: 4rem;
      border: none;
      border-radius: 0.7rem;
      background-color: green;
      color: white;
    }
    button#cancel {
      background-color: red;
    }
  `;

  constructor() {
    super();
    this.feedback = '';
  }

  ok() {
    this.feedback = "thanks for clicking ok.";
  }

  cancel() {
    this.feedback = "thanks for nothing.";
  }

  render() {
    return html`
      <button @click="${this.ok}">ok</button>
      <button id="cancel" @click="${this.cancel}">cancel</button>
      <div>${this.feedback}</div>
    `;
  }
}

customElements.define('ok-cancel', OkCancel);
</script>
</body>
</html>
```

Listing 4.2: LitElement

```
 import { Component, Prop, h } from '@stencil/core';

@Component({
  tag: 'ok-cancel',
  styleUrl: 'ok-cancel.css',
  shadow: true,
})
export class OkCancel {
  @Prop() title: string = "The ok-cancel dialogue with Stencil";
  @Prop() feedback: string;

  ok(){
    this.feedback = 'Thanks for clicking OK'
  }

  cancel(){
    this.feedback = 'Thanks for nothing'
  }

  render() {
    return (
  <div>
    <div class="title"> {this.title} </div>
      <div class="actions">
          <button class="buttonOk" onClick={() => this.ok()}>OK</button>
       
          <button class="buttonCancel" onClick={() => this.cancel()}>Cancel</button>

        </div>
      <div class="feedback"> {this.feedback} </div>
    </div>

    );
  }

}
```

Listing 4.3: Stencil

4.3 Stencil

See Listing 4.3 for an example code of the Stencil library.

The @Component in the header section defines the name of the web component which is later the name of the tag to call the web component. Also the corresponding css file and the shadow parameters are set there. The shadow default value is true, because it's a big advantage of web components to use the shadow DOM and this parameter defines it.

Below the component definition tag, the class is defined and contains the javascript or typescript code and the following return content holds html code. The upper code snipped is just a short example but shows the relevant structure of a running stencil web component.

Chapter 5

Recommendations

To develop a small number of custom elements we highly recommend to stick to HTMLElements as it has no overhead to speak of while offering solid documentation to the user.

In case a project utilises custom web components extensively, for example within a design system, things change. Starting at 20 - 30 custom tags within the class based web component libraries LitElements offers a solid performance with little overhead.

For more app based infrastructure we recommend Stencil because of its big user base and solid starter infrastructure. You can fastly create a compile based web component and you can easily test it on your own webpage without big obstacles. With knowledge in html, css and javascript you have the bricks to create stencil web components and that's why we recommend it. You can also try Svelte and Riot, because they are similar developed then Stencil but the first package installation is more complex and could lead to inexplicable errors.

The class based web component library FAST, developed and backed by Microsoft, was very hard to install and use. It deserves a spot on the non-recommendations list because of its hard to understand documentation, very complex and intransparent structure and overall challenging nature. We would not recommend using FAST unless one has a really good reason to go down that path.

Lightning Web Component, is developed by Salesforce and is similar then FAST. You have a big community and documentation but the content is overloaded and unstructured and you can hardly find helpful content if you are not familiar with developing lighting web components. You need to install Visual Studio because only this is well documented and it?s quite complex to publish self created web components at your own webpage. It is necessary to publish all web components at npm and for this you need a github account with all of the source code. Furthermore your github account must be connected to your npm account, which you have to create at first and only if you have those, you are able to publish your developed web components. Because of these circumstances and about the complex installation to get started it?s definitely not recommended.

Chapter 6

Web Components for Design Systems

One of the big advantages of Web Components are Shadow DOMs, meaning we have one CSS per component. That is, because we want to protect the style from the rest of the sides. The big problem we have here is that in some cases we want some style from outside, for example in bigger projects we want some sort of consistency and have our components look like the rest of the page. One solution would be to use CSS Shadow PartsTthemes here but this is not widely supported yet so we will not look into them here. What is more interesting are Constructible Style Sheets [Miller 2020] which are available in Chrome and with Polyfill also in other Browsers. It is an extension of the JavaScript CSSStyleSheet object API. With this it is possible to load one or more CSS files that then can be adapted by one more multiple Shadow DOMs on the page. The sheets get shared and do not have to be reloaded by every component, each one can pick which one to adopt. They are not duplicates, each stylesheet only exists once. That makes it easy to maintain and change the appearance of the whole page with just touching a one file, no matter how many components are using it. With this Technology in connection with Web Components we can create Design Systems that run everywhere and do not have to be framework specific.

One good Tool to create UI components is Storybook [Storybook 2020]. This sandbox tool is open source and can be used to develop and test whole design systems in isolation. If one does not want to invest time to create their own system of components, there are also many libraries available. One of them would be the Salesforce Component Library [Salesforce 2020] which focuses on Lightning components (since Salesforce is the company behind Lightning). A big and well-know Web Component-based-UI library is Ionic (which is also behind the Stencil library)

There are of course also libraries from the big companies, like the PolymerProject [Polymer 2020] from Google, which also created LitElement and lit-thml or UI5 Web Components from SAP [SAP 2020]. But there are also other smaller ones, namely Wired Elements [Shihn 2020]. An easy way to find a lot of components is to look up webcomponents.dev [DIV-RIOTS 2020], they feature more than 2000 different components and collections from Github.

As of right now, Web components can also be found in popular and bigger frameworks. They can be used and integrated seamlessly for example with Angular, Vue (2 and 3) and React and are therefore the perfect addition to every project.

Chapter 7

Concluding Remarks

Web components have a great potential with their modularity and compatibility with modern browsers. They are small and fast and can be added to every project without much installation. The basics, so with plain HTMLElements, are very easy to understand and to learn, and for bigger projects there are many Web Component libraries that can make it easy to build fast.

Of course not all of them are good for the same thing, users have to pick the right tool for the right sort of application. For using them we do not have to depend on the development of other frameworks (Vue, Angular or similar) but we can still use them without problems.

Bibliography

Barot, Saurabh [2019]. *Why Developers Love to Use TypeScript in 2021?* Aglowid Blog. 19 Jul 2019. https://aglowiditsolutions.com/blog/why-use-typescript/ (cited on page 6).

DIV-RIOTS [2020]. *All the Ways to Make a Web Component*. WebComponents.dev Blog. 07 Oct 2020. https://webcomponents.dev/blog/all-the-ways-to-make-a-web-component/ (cited on pages 8–9, 17).

Marks, Alexander [2020]. *tachometer*. GitHub. 07 Oct 2020. https://github.com/Polymer/tachometer (cited on page 8).

MDN [2020]. *Using Shadow DOM*. MDN. 12 Dec 2020. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM (cited on page 5).

Miller, Jason [2020]. *Constructable Stylesheets: Seamless Reusable Styles*. Google Developers. 07 Oct 2020. https://developers.google.com/web/updates/2019/02/constructable-stylesheets (cited on page 17).

Pollak, Michael [2020a]. *HTML Elements*. GitHub. 27 Nov 2020. https://github.com/michaelpollak/iaweb2020/blob/main/htmlelements.html (cited on page 11).

Pollak, Michael [2020b]. *htmlelements and litelement example*. Showcase Video on YouTube. 20 Nov 2020. https://youtu.be/DwhcgEnRJJo (cited on page 11).

Pollak, Michael [2020c]. *LitElements*. GitHub. 27 Nov 2020. https://github.com/michaelpollak/iaweb2020/blob/main/litelement.html (cited on page 11).

Polymer [2020]. *Polymer Project*. 07 Oct 2020. https://www.polymer-project.org/ (cited on page 17).

Salesforce [2020]. *Salesforce Component Library*. 07 Oct 2020. https://developer.salesforce.com/docs/component-library/overview/components (cited on page 17).

SAP [2020]. *UI5 Web Components*. 07 Oct 2020. https://sap.github.io/ui5-webcomponents/ (cited on page 17).

Shihn, Preet [2020]. *Wired Elements*. 07 Oct 2020. https://wiredjs.com/ (cited on page 17).

Storybook [2020]. *Storybook*. 07 Oct 2020. https://storybook.js.org/ (cited on page 17).