BACHELOR THESIS

COMPUTER SCIENCE AND WEB ENGINEERING

Sharing Styles between Web Components

Author:
Luis WILLNAT

Supervisors:
Prof. Dr. Thomas KRETSCHMER
Dipl. Inf. Ralf HAFNER

I, Luis WILLNAT, *confirm that this bachelor's thesis is my own work and I have documented all sources and material used.*

Saarbrücken, 25. September 2018

_____

Contents

List of Figures

Nomenclature

API	Application Programming Interface, page 4
BEM	Block Element Modifier, page 10
CSS	Cascading Style Sheets, page 2
DOM	Document Object Model, page 2
ECMA	European Computer Manufacturers Association, page 2
ES6	ECMAScript Edition 6, page 5
HTML	HyperText Markup Language, page 2
IE	Internet Explorer, page 2
MDN	Mozilla Developer Network, page 4
NPM	Node Package Manager, page 4
Sass	Syntactically Awesome StyleSheets, page 25
VI	Virtual Identity, page 1
W3C	World Wide Web Consortium, page 5
XBL	XML Binding Language, page 2
XML	Extensible Markup Language, page 2
iFrame	inline Frame, page 7

Chapter 1

Introduction

1.1 Company

1.1.1 Presentation

This work takes place at the company Virtual Identity AG (VI). It is a web agency which is designing and developing websites for clients like Siemens, Union Investment and Roche. The origin of the company's name is not really known, it might be found during the foundation year in 1995 when there were not so many websites existing on the world wide web. Hence, VI did not have a real identity but a virtual one. The company is spread over two countries (Germany and Austria) with three locations. The first, which is in Freiburg, is currently the biggest office with seventy-two people. Another, with a larger office size, located in Munich has about sixty employees. The smallest office in Vienna (Austria) has approximately twenty people. The company has been created in Freiburg.

1.1.2 Organization

Several sections are split up through the office surface. One place for human resources, one for finance, one for design and a section with all company's developers. To build websites, the company created in 2014 its own front-end framework, which is called **Biotope**. It has been born four years ago and was initially planned just for the company. The framework's source code is available on Github[1] [1]. Thus other companies and developers could contribute and use it for their own internal projects. Some senior and intermediate developers from VI worked on the latest version 5.0 which should not only make the framework faster during development but also during production on the web server.

[1] Github is a service which hosts source code through the version control git.

Chapter 2

Web components

2.1 Background story

Web components are not a young innovation.

Microsoft started in 1998. They proposed HyperText Markup Language (HTML) components for Internet Explorer 5.5 (IE). Requirement was to write them in JScript [1] or in Visual Basic[2] [2]. IE 10 was the first browser version where HTML components are deprecated.

Mozilla was the next challenger with XML Binding Language (XBL) in 2001. Compared to Microsoft's components written in HTML, XBL components were written in XML. The company abandoned the project in 2012 due to lack of browser support [2].

Google mentioned the first time version 0 of web components in 2013. At that time, it was only supported by Chrome and Opera with enabled flag[3]. One year later, they both added default support. Same year, Polymer the web components library from Google itself, was born with aim to facilitate and promote web components [2].

The version 0 of web components did not get much support from browsers. Later in 2016, version 1 has been born and Chrome has shipped shadow Document Object Model and custom elements.

Unlike back end[4], on front end[5] we used to stick to three same programming languages for the past twenty years: HTML, Cascading Style Sheets (CSS) and JavaScript. Building modern web applications has become more and more complex. Businesses compete against each other on the web to attract visitor's attention to convert them into potential buyers. In fact, user experience and page load time are two main factors that determine whether a user stays or not on the website. On the other hand, developers are searching a way to develop faster to save money.

As shown on figure 2.1, showing a timeline of existing JavaScript technologies between 1999 and 2014, there was not only one way to develop on the web. This would have been perfect to keep a standard and simplify work for developers. However,

[1] JScript is Microsoft's dialect of the ECMAScript standard that is used in Microsoft's Internet Explorer.

[2] Visual Basic is an event-driven programming language to mainly build Microsoft graphical user interfaces.

[3] Enabling a flag means setting manually a support for a feature in browser's settings.

[4] Back end is a software's background with data manipulation and storage, everything a user does not see.

[5] Front end is the presentation layer of a software, what the user sees.

FIGURE 2.1: A timeline of JavaScript technologies [3]

having so many different technologies offers developers to have a choice and decide for the best one for each use case. The downside is developers have to learn them, sometimes rebuild entire applications to keep up to date with recent technologies. The problem, time is precious and expensive for companies.

This is the reason why organizations like Google, Mozilla and web component community members created a set of web standards in 2013. Unlike libraries and frameworks, web components are at the nearest of browsers, they embrace the power that plain JavaScript is offering.

Web components have three main goals:

1. HTML extension
2. Encapsulation
3. Reusability

HTML extension. Whenever browsers (e.g., Google Chrome or Mozilla Firefox) wish to add a new type of HTML element (e.g., input calendar), it takes a long time to implement it. They first need to discuss, then verify, validate everything and then finally implement it. That is exactly one problem which web components are solving. Browsers have to support web components and developers create the content. This way browsers will have less work and will give developers the freedom to develop their own HTML elements and extend existing ones.

Encapsulation. Encapsulated means that the element is able to work on his own without any additional dependencies. Web component's styles will not affect other components or the main document. Also, document styles will not change component's styles.

Reusability. Reusability simplifies the process of building complex websites. They promote the principles of reuse. Build once and use again. Web components have a great encapsulation that allows it to reuse a same element somewhere else inside the same project or even in another project. Building web components is plain HTML and JavaScript. There is no framework or library necessary to build or display a website, only a browser supporting it. As an illustration, jQuery[6][4] has been created to bring unanimity between browsers in order to facilitate developers' work.

[6]jQuery is a JavaScript library created to simplify client-side scripting.

Now browsers have to create browser's unanimity themselves to support web components.

The Application Programming Interface (API) simplicity from jQuery is similar to the ES6 [7] specification which web components can be written with.

Currently, all web components standards are officially supported by Chrome and Opera. According to Mozilla Developer Network (MDN), Firefox's shadow DOM implementation has been enabled by default in version 63 [5]. On the other hand, considering HTML imports, Safari and Firefox did not planned to support it [6][7].

Due to the lack of browser support, there are polyfills that virtually support web components.

To display a custom element in a browser, it is necessary to start a local server and have a browser supporting web components standards. For example, static-server which is available under the node package manager (npm) command *npm install static-server*. It can be started by typing *static-server* in the terminal folder where the project is located. The website should now run on the local server address *localhost:9080*.

2.2 Definition

Web components are a set of standardized rules for creating **encapsulated** and **reusable** user interface elements for the web.

Web components can be divided in four features:

- Custom Elements
- Shadow DOM
- HTML Imports
- HTML Templates

These features can be used alone or together. A very common combination seen today is custom elements attached with a shadow DOM. A custom element is not a web component. This is often misunderstood in the community. It is the name given to the four features.

2.3 Features

2.3.1 Custom Elements

Thanks to custom elements, a web developer can extend and create new HTML elements. Building a custom element is similar to components in Angular[8][8] or React [9] [9]. The primary difference is that they are native to the browser and based on a

[7] Shortcut for ECMAScript, a JavaScript language specification, created to have a standardized language.

[8] Angular is an open-source front-end web application platform from Google.

[9] React is a JavaScript library for building user interfaces. Maintained by Facebook.

World Wide Web Consortium (W3C) specification [10]. Because of this, custom elements should be able to work in any context where HTML works. The result is less code which can be reused on every website. Many examples combine custom elements with a shadow DOM to explain web components, but there is no requirement to use them both together. Each feature of the web components specifications has been built to work separately.

Listing 2.1 shows a custom element:

LISTING 2.1: custom-element.html

```
1  <template id="template">
2      <style>
3          h3 {
4              color: purple;
5          }
6      </style>
7
8      <h3>I am a custom element.</h3>
9  </template>
10
11 <script>
12     const doc = (document._currentScript || document.currentScript)
           .ownerDocument;
13     const template = doc.querySelector('#template');
14     class NativeComponent extends HTMLElement {
15         constructor() {
16             super();
17             this.appendChild(template.content);
18         }
19     }
20     customElements.define('custom-element', NativeComponent);
21 </script>
```

A custom element is separated in three parts:

- The element's template written in HTML inside `<template>` tags on lines 1-9.
- The element's style encapsulated inside the element's template and the `<style>` tags on lines 2-6.
- The element's script written in JavaScript inside `<script>` tags on lines 11-21.

On the last line in the script, it is necessary to `define()` the custom element just created in order to use it later. It is also possible to write a custom element in a JavaScript file by using an ECMAScript Edition 6 (ES6) class.

2.3.2 Shadow DOM

Shadow DOM allows developers to create DOM elements that do not directly belong to the document, but rather in an encapsulated sub-DOM tree called *shadow tree*. Like shown on figure 2.2, each shadow tree has a *shadow root* as its root which in turn has *shadow nodes* or may have another shadow tree.

An element with a shadow DOM attached will own, render and style a chunk of DOM which will be separated from the rest of the page. Listing 2.2 shows it is possible to hide an entire application just with one tag.

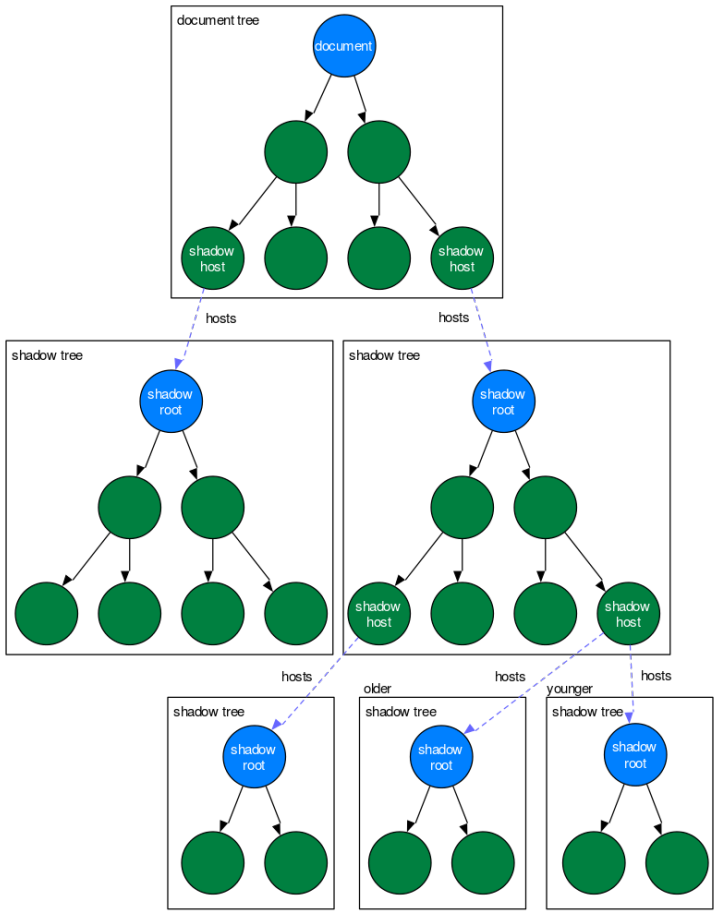

FIGURE 2.2: Shadow DOM representation [11]

LISTING 2.2: index.html

```
1 <!-- messenger-app's implementation details are hidden in the
     Shadow DOM. -->
2 <messenger-app></messenger-app>
```

Listing 2.3 is a vanilla custom element using a shadow DOM:

LISTING 2.3: custom-element-shadow-dom.html

```
1  class MyCustomElement extends HTMLElement {
2    constructor() {
3        super();
4      this.attachShadow({ mode: "open" });
5      }
6      connectedCallback() {
7          this.shadowRoot.innerHTML = `
8              <p>I am in the Shadow Root</p>
9          `;
10     }
11 }
12
13 window.customElements.define("my-element", MyCustomElement);
```

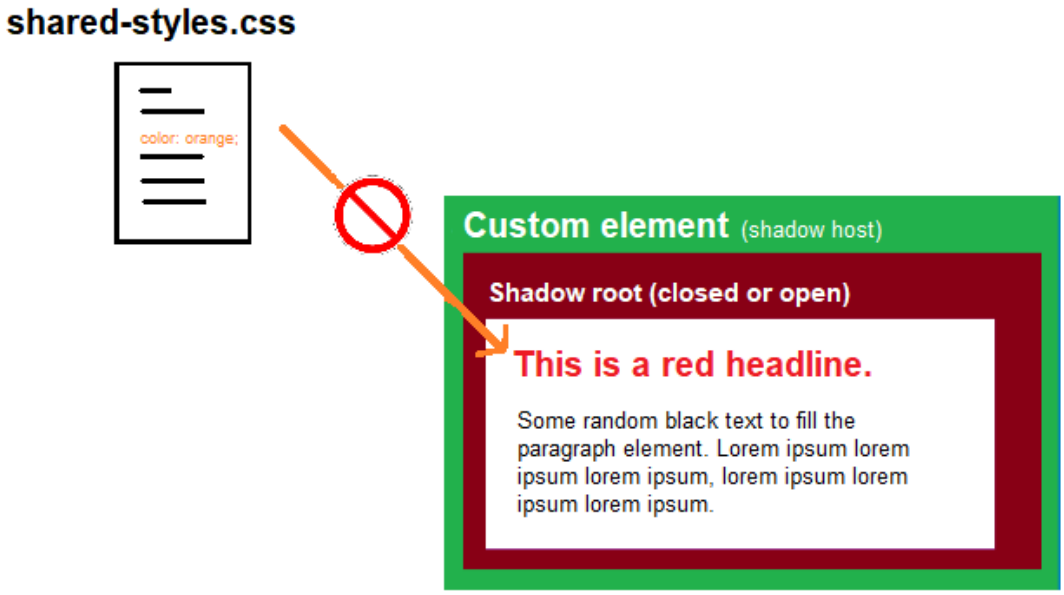

FIGURE 2.3: Shadow DOM encapsulation

The function `attachShadow()` on source code line 6 (listing 2.3) creates and attaches the shadow DOM to the component. The content is added by setting directly `innerHTML` in the `connectedCallback()` on the shadow root which turns out to be inefficient. This technique has high costs because it will be called each time when creating an instance of the component, whereas using a template element outside of the component will only be called once. Listing 2.4 demonstrates an example on how to parse the template's HTML only once. This parsed HTML content will later be added to the shadow root.

LISTING 2.4: custom-element-shadow-dom.html

```
1 let tmpl = document.createElement('template');
2 tmpl.innerHTML = `
3   <style>
4     h3 {
5       color: red;
6     }
7   </style>
8   <h3>I am a custom element</h3>
9 `;
```

Thanks to the shadow DOM feature a web component developer is able to create an encapsulated environment and be sure that he has control over the styling and that external CSS will not accidentally break their component. Figure 2.3 demonstrates that the color orange is not able to enter because of the shadow DOM that blocks styling from outside.

Shadow DOM is similar to iFrames[10], they both encapsulate styles. However, as Jake Harding from Twitter said: "Much lower memory utilization in the browser, and much faster render times. Tweets will appear faster and pages will scroll more smoothly, even when displaying multiple Tweets on the same page." [12]. In fact

[10] An iFrame (Inline Frame) is an HTML document embedded inside another HTML document on a website.

Twitter, Coca-Cola or Youtube are all using web components [13]. A shadow DOM is native, needs less maintenance and has a much better API to communicate from the parent element than iFrames.

When styling the shadow DOM, the keyword `:host` selects the shadow root host. It has no effect when it is used outside a shadow DOM.

- `:host` selects all instances of a custom element.
- `:host(.className)` selects all instances of a custom element having the class *className*.
- `:host-context(h1)` selects all instances of a custom element which are inside an h1 element.

2.3.3 HTML Imports

Thanks to HTML import feature, web components are modular. Importing a custom element from its HTML page is done by using:

LISTING 2.5: index.html

```
1  <link rel="import" href="custom-element.html">
```

The imported HTML file `custom-element.html` will be merged into `index.html` at build time. There it is reachable inside the HTML document with the template tags `<my-element></my-element>` (if the element has been defined like that) [14].

2.3.4 HTML Templates

The HTML template tag defines the view of a custom element. It is unused at page load until it is instantiated later by some JavaScript at runtime.

Everything between the `<template></template>`:

- Will not be displayed until it is activated by JavaScript.
- Has no effects on other elements. Scripts will not run and images will not load until activated [15].

To declare a template, a `<template>` tag is needed:

LISTING 2.6: custom-element.html

```
1  <template id="myTemplate">
2    <img src="" alt="Placeholder">
3    <div class="comment"></div>
4  </template>
```

The `appendChild()` JavaScript function will add the cloned template (saved in variable `t`) to the custom element's DOM:

LISTING 2.7: custom-element.html

```
1   var t = document.querySelector('#myTemplate');
2
3   var clone = document.importNode(t.content, true);
4   document.body.appendChild(clone);
```

2.4 Styling example

LISTING 2.8: custom-element.html

```
1  <template id="teaser">
2      <style>
3        h1 {
4            color: red;
5        }
6      </style>
7      <h1>Title</h1>
8  </template>
9
10 <script>
11     const doc = (document._currentScript || document.currentScript)
           .ownerDocument;
12     const template = doc.querySelector('#teaser');
13
14     customElements.define('custom-element', class extends
           HTMLElement {
15       constructor() {
16         super();
17
18       this.innerHTML = template.innerHTML;
19       }
20     });
21 </script>
```

The example shown on listing 2.8 is a native custom element without shadow DOM where styling, scripting and template are in one and only file. It is shareable because everything is encapsulated and can be imported (via HTML import) in every project without having to worry about dependencies.

2.5 Problem statement

CSS is a language for describing the rendering of structured documents (e.g. HTML). Companies want to save time and parsing[11] costs, so they are using methods to share styles between project files.

In general, they do it by two ways:

- A `normalize.css` project file is setting default project styles before component's styles. They can be overridden by component's styles.
- A `shared-styles.css` project file is sharing style values (e.g., text color) for all elements. These global styles should override component's styles (e.g., to apply a theme color).

[11] A browser reads the CSS code and translates it into machine code.

These project files are really helpful and time efficient. In fact, instead of setting or changing every component's style manually it is possible to change one file value to affect the whole website.

However, with the coming of shadow DOM encapsulation, shared styles cannot style through the shadow DOM.

Three possible solutions to apply shared styles:

1. Not to use the shadow DOM. Same problem like when not using web components, so not really an improvement. Without a shadow DOM, a custom element acts like a normal HTML object in the DOM and will be subject to page's styles and vice-versa. Developers would have to worry again about CSS specificity wars and use alternatives to simulate encapsulation (e.g., iFrames which encapsulates a document inside another). In the case of listing 2.8 the color of the component headline **and** all main document's headlines will be colored red. This is because `<custom-element>` has been added later in the DOM. Hence, every other HTML element in the main document is affected by that red styling.
2. Not to use a shadow DOM and limit styling leaks. Either by adding a naming methodology like Block Element Modifier BEM[12] or specify the component itself as a target selector like on listing 2.9 (here *custom-element*).
3. Use the shadow DOM and find a method that is able to break through it. This way, a custom element holding a shadow DOM still gets styles from global style files while preventing unwanted styling going outside and inside of the component. Adding a shadow DOM is more generic and more powerful than previous solutions.

LISTING 2.9: custom-element-no-shadow-dom.html

```
1 <style>
2   custom-element h3 {
3     color: purple;
4   }
5 </style>
```

2.6 Motivation

Initially, during the first version of web components (known as version 0) developers could penetrate the shadow DOM by using `::shadow` or `/deep/` CSS selectors [16]. This shadow piercing method has been removed in version 1, there is no brute force alternative to penetrate the shadow DOM. It is necessary to find out which new method is the most appropriate that allows both a precise custom element styling and component encapsulation like explained on figure 2.4.

Also, it should be possible for a custom element to present a stable styling *API*. That is, a custom element should not require knowledge of the element's internal details. A custom element's user should be able to style it while leaving component's styling source code untouched.

[12] BEM is explained in chapter 5.2.1.

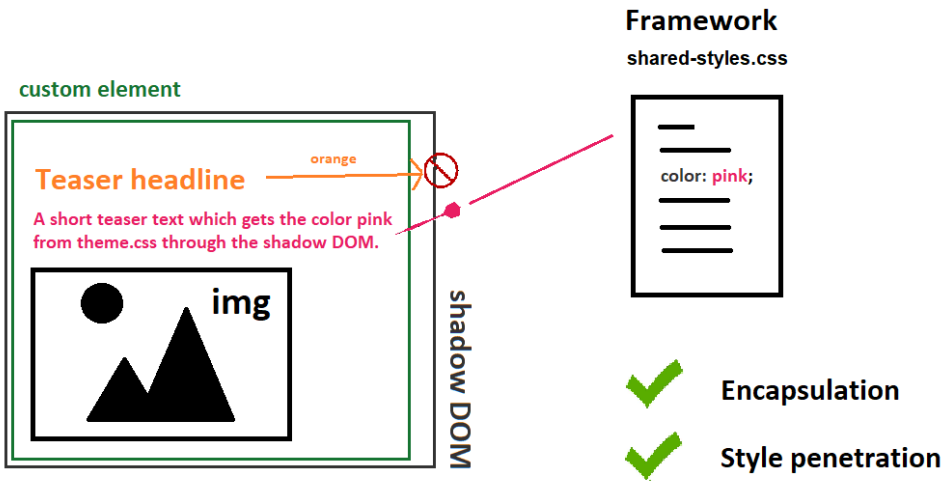

FIGURE 2.4: Combine style consistency and encapsulation

Figure 2.5 demonstrates the theoretical solution possible that only one `shared-styles.css` file is necessary to affect some styling properties of all custom elements in a project. Developers would save a lot of time instead of changing every custom element's manually.

2.7 Work structure

To determine which method a company should use to share styles between web components, it is essential to analyze each possibility with a view on:

- Ease of use: simplicity for a company to implement a sharing style method
- Performance: no issues and no styling problems. A shared style sheet is efficient when all elements targeted have been styled like expected

Each styling method needs to be evaluated through examples in order to find possible disadvantages or advantages by using it.

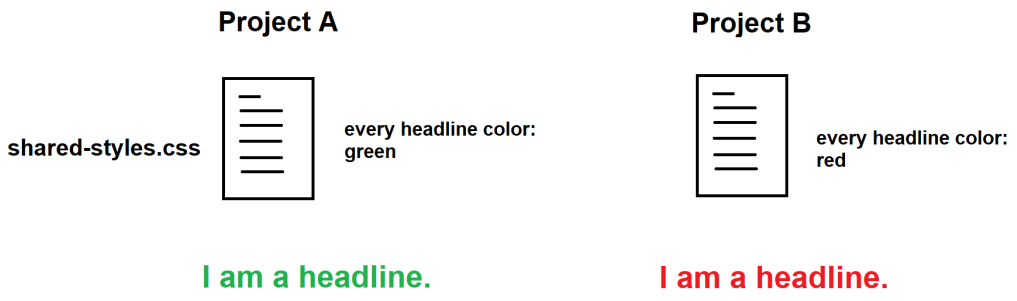

FIGURE 2.5: Model of a shared style sheet for a project

Chapter 3

Shadow DOM styling

Before version 1 of web components standard, developers used brute force methods to penetrate the shadow DOM. Today there are three methods that allow to break into it while keeping the shadow DOM encapsulation.

1. Inheritance
2. Custom properties
3. Import style sheets

To illustrate examples, `shared-styles.css` will be the project's style sheet file which will contain all styling that need to be shared across components.

3.1 Inheritance

When used in an HTML document, a custom element with a shadow DOM will inherit any styling information that is applied to its parent element, hence they are able to reach through the shadow DOM.

On listing 3.1, the `<custom-element></custom-element>` on line 10 is inheriting the `<h3>` color red of line 3.

LISTING 3.1: index.html

```
1  <style>
2    h3 {
3        color: red;
4     }
5  </style>
6  <h3>Index headline</h3>
7
8  <h3><custom-headline></custom-headline></h3>
```

The visual result is shown on figure 3.1. The property *red* passes through the shadow DOM.

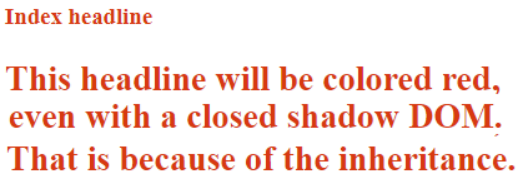

FIGURE 3.1: Inheriting styles from parents even with a shadow DOM

However, component's styles override document styles:

LISTING 3.2: custom-headline.html

```
1  <style>
2          h3 {
3              color: green;
4          }
5  </style>
```

The visual representation is shown on figure 3.2.

FIGURE 3.2: Inheriting styles from parents through a shadow DOM

3.2 Custom properties

Custom properties are a way to set and use variables in style sheets and mainly provide a way to add a styling API to an element which has a shadow DOM attached. This gives custom element's authors the possibility to **decide** which styling rules can be changed.

Declaring new variables have the notation of `--myVariable: value;` (note the **two** dashes at the beginning of the variable). They must be declared inside a selector and cannot be declared alone. A custom property cascades like other CSS properties. This means its value can be inherited from its children and as a result break through the shadow DOM.

A simple custom element structure:

LISTING 3.3: custom-element.html

```
1  <h3>Headline oustide of <i>.content</i> div.</h3>
2  <div class="content">
3      <h3>
4          Some random text with a link inside pointing to <a href="">
               nothing</a>.
5      </h3>
```

```
6  </div>
```

The corresponding style source code for listing 3.4 is shown on listing 3.5. There two custom properties are defined inside the element `.content`. Hence, every child (and their child) and the element `.content` itself have access to the custom property `--primary-color`. However, the headline of line 1 in listing 3.4 does not have any access to it because it is not in the `.content`'s scope[1]:

LISTING 3.4: custom-element.html

```
1  <style>
2     .content
3          {
4               --primary-color: green;
5               --link-color: orange;
6          }
7
8          h3 {
9               color: var(--primary-color, red);
10         }
11
12         a {
13              color: var(--link-color, red);
14         }
15
16 </style>
```

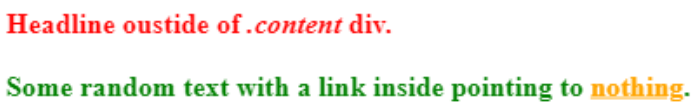

FIGURE 3.3: Scope of custom properties

Figure 3.3 is a visual representation of listing 3.3 and 3.4.

The orange link shown on figure 3.3 gets the color from its parent `.content`.

3.3 Import style sheets

Some years ago, the web component community proposed a solution which was not possible in version 0: supporting the `<link>` tag [17]. It is simpler, more familiar and avoids possible negative characteristics of `@import`. Importing a style sheet through the `<link>` tag will break through the shadow DOM. Imported style sheets will be applied because it is linked after the component's styles, therefore they have styling priority:

LISTING 3.5: custom-element.html

```
1  <style>
2          h3 {
3               color: red;
4          }
5  </style>
```

[1] A scope is the context in which an element is in effect.

```
6  <h3>Green headline. Affected by shared styles because link tag is
       after.</h3>
7  <link rel="stylesheet" href="shared-styles.css">
```

To link the style sheet before component's styles, a workaround solution is to use custom properties inside the imported style sheet:

LISTING 3.6: custom-element.html

```
1  <link rel="stylesheet" href="shared-styles.css">
2  <style>
3          :host {
4              --color-headline: red;
5          }
6
7          h3 {
8              color: var(--color-headline);
9          }
10     </style>
11     <h3>Green headline. Uses custom properties and shared styles
           are applied.</h3>
```

The result is visible on figure 3.4.

Index headline

Red headline. Not affected by shared styles because link tag is before.

Green headline. Uses custom properties and shared styles are applied.

Green headline. Affected by shared styles because link tag is after.

FIGURE 3.4: Import an external style sheet to get shared styles

Chapter 4

Sharing methods analysis

4.1 Inheritance

In order to share styles between components with inheritable properties, one way is to create a division (`<div></div>`) element with a specific class `.sharedStyles`. This element needs to be the parent of every element that needs shared styles. Then, `.sharedStyles` element's styles from listing 4.1 will cascade into the children elements `<h3></h3>` and `<custom-headline></custom-headline>` in listing 4.2.

LISTING 4.1: shared-styles.css

```
1  .sharedStyles {
2    color: red;
3  }
```

LISTING 4.2: index.html

```
1  <link rel="stylesheet" href="shared-styles.css">
2
3  <div class="sharedStyles"><h3>Index headline</h3></div>
4  <div class="sharedStyles"><custom-headline></custom-headline></div>
```

As displayed on figure 4.1, the property red from the headline will get inherited from the custom element `<custom-headline>` which in turn inherited from `.sharedStyles` element.

FIGURE 4.1: Inherit styles visualization

Regarding shared styles performance:

- `normalize.css`: efficient because component's styles override shared styles.
- `shared-styles.css`: This is not practical. As soon as the developer is setting a default style in the component, the shared value from `shared-styles.css` will not get applied, even with an `!important` keyword.

Sharing styles between web components thanks to inheritance is easy to implement. But with the fact that component's styles override shared styles, this sharing method is not suited for big companies or even individual working developers.

The source code for the shared style sheet.

LISTING 4.3: shared-styles.css

```
.sharedStyles {
   color: green !important;
}
```

LISTING 4.4: custom-element.html

```
<style>
        h3 {
            color: red;
        }
</style>
<h3>This headline will not get green, <br/>
  But will stay red, because component's styles override <br/>
  parent's styles.
</h3>
```

The visual representation of listing 4.3 and 4.4 is on figure 4.2.

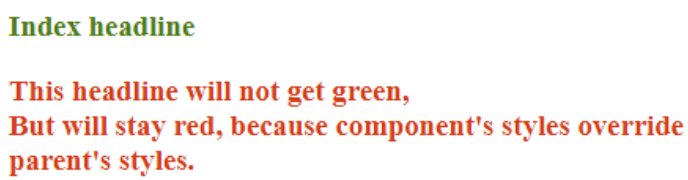

FIGURE 4.2: Inherit properties from parents to share styles

4.2 Custom properties

To prepare the environment, two shared files (`shared-styles.css` and `normalize.css`) are imported in the main document `index.html` in the `<head>` tags:

LISTING 4.5: custom-element.html

```
<head>
        <title>Sharing styles with custom properties</title>
        <link rel="stylesheet" type="text/css" href="
            shared-styles.css">
        <link rel="stylesheet" type="text/css" href="normalize.css"
            >
    </head>
    <body>
        <custom-element></custom-element>
        <link rel="import" href="custom-element.html">
    </body>
```

When breaking through the shadow DOM, having a `shared-styles.css` using custom properties is flexible for the reason that a component author is able to decide

which component's elements can be overridden by `shared-styles.css`. Although two conditions must be satisfied:

1. Component's custom properties must be set:

LISTING 4.6: custom-element.html

```
<style>
  h3 {
      var(--primary-color, red);
    }
</style>
```

2. `shared-styles.css` needs to target the custom element:

LISTING 4.7: shared-styles.css

```
      custom-element {
          --primary-color: green;
      }
```

Another way is to use the universal selector `*` to select all custom elements.

LISTING 4.8: shared-styles.css

```
      * {
        --primary-color: green;
      }
```

Then and only then, shared styles from outside will break through the shadow DOM and override component's custom properties.

In the case of `normalize.css`, it is too much work setting default styles using custom properties. Every value inside `normalize.css` has to be declared again in the component's file to allow a passage through the shadow DOM. For example, in the `normalize.css` file:

LISTING 4.9: normalize.css

```
/* Default styles for every component */
* {
  --link-color: green;
  --link-size: 18px;
  --border-color: orange;
  --border-size: 3px;
}
```

The source code below shows that a developer would have to include every value to get `normalize.css` styles. It is redundant and a component author would have to know every variable name, which sometimes may not be possible (e.g., if the component is from another company). Not to use custom properties in `normalize.css` does not help as styles will not get through the shadow DOM from the main document (`index.html`).

LISTING 4.10: custom-element.html

```
<style>
  h2 {
    color: red;
    border: var(--border-size, 5px) solid var(--border-color, red);
```

```
5    }
6    a {
7      color: var(--link-color, red);
8      font-size: var(--link-size, 13px);
9    }
10 </style>
```

Regarding shared styles performance:

- `normalize.css`: not efficient. Every custom property from the shared file has to be declared again in the component's styles.
- `shared-styles.css`: efficient. A component author can decide which selectors can be changed from outside.

Custom properties have been born with web components, hence it may be difficult to understand for newcomers. Also, the `normalize.css` performance is not high. Despite that, it is easy to implement and efficient when targeting specific elements of a component. Custom properties are not available on Internet Explorer 11[18]. This needs to be taken into account in case a web agency is building websites for clients having a broad spectrum of supported browsers.

4.3 Import style sheet with custom properties

The idea is to combine the strength of two methods: use custom properties from the shared styles to override component's styles **and** set default component's styles from outside with the link style sheet styling method.

Thanks to the `<link>` tag, the `normalize.css` file (listing 4.12) will be accessible inside the component's shadow DOM. Same for `shared-styles.css` (listing 4.13) thanks to custom properties.

LISTING 4.11: custom-element.html

```
1  <template id="template">
2      <link rel="stylesheet" href="normalize.css">
3      <style>
4          h3 {
5              color: var(--primary-color, red);
6          }
7      </style>
8      <h3>Headline gets green thanks to a custom property received
           from shared-styles.css</h3>
9      <a href="#">This link gets the color green from normalize.css.<
           /a>
10 </template>
```

Besides, if another element also needs shared styles, these will not be downloaded again (figure 4.3).

LISTING 4.12: normalize.css

```
1  a {
2      color: green;
3  }
4
5  h5 {
```

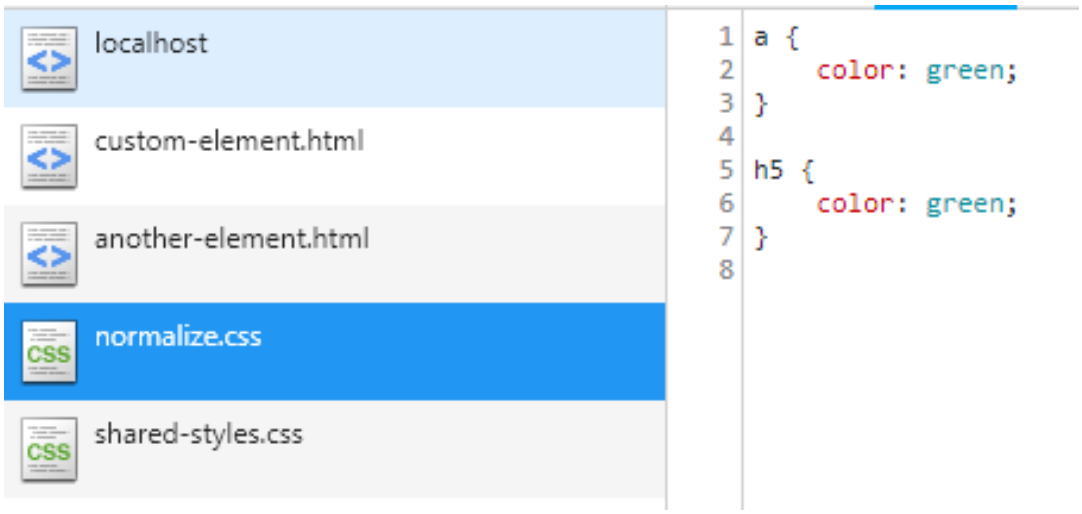

FIGURE 4.3: Network screenshot showing downloaded files

```
6       color: green;
7   }
```

As seen before, custom properties from the `shared-styles.css` will override component's styles if the component author allows it:

LISTING 4.13: shared-styles.css

```
1   h3 {
2       --primary-color: green;
3   }
```

To conclude, this sharing style method is efficient for both `normalize.css` and `shared-styles.css` by using custom properties and the link tag. It has the same difficulty than using custom properties alone but offers more sharing styles possibilities. Figure 4.4 is a visual representation of this sharing style method synchronized with previous source codes.

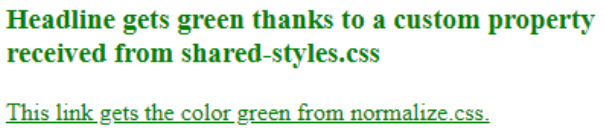

FIGURE 4.4: Representation of shared values between components using custom properties and a link tag

4.4 Polymer

4.4.1 Presentation

Creating web components from scratch can be hard to write, especially for newcomers. Thankfully, some libraries already exist even if web components features are not yet supported on every modern browser. Actually, these libraries have polyfills

so that web components are supported everywhere. They save a lot of time when having a lot of custom elements.

A short list of web components libraries:

- Polymer development began in 2013 with Google. It has a built-in solution to share styles between web components and has over 20 000 stars on Github, hence it is the most famous library to build web components [19]. Polymer's mission is to facilitate web components development of modern and complex applications [20].
- Skate.js is a library with a very small footprint to build custom elements for view libraries like React (2700 stars on Github) [21].
- X-Tag was initially developed by Mozilla but is now maintained by Microsoft [22]. It is similar to Polymer and uses the same polyfill. It has almost 400 stars on Github [23].
- Slim.js is similar to the other ones (+500 stars) [24][25].

Libraries are just build on top of web components by adding some syntactical sugar. It is actually the polyfill they use that makes it available in all browsers.

4.4.2 Polymer element

Since version 3 of Polymer, a Polymer element is an ES6 class with the extension `PolymerElement` imported from the JavaScript file `polymer-element.js`. Also, Polymer moved from HTML Imports to ES6 modules and from bower[1] to npm. This move towards the JavaScript development environment makes it easier to use Polymer with other popular libraries, frameworks and tools. Polymer elements have per default a shadow DOM attached.

When building Polymer elements, templates are defined by providing a `template()` getter (listing 3.8, lines 4-13) that returns a string and not a `<template>` element.

LISTING 4.14: polymer-element.js

```
1  import {html, PolymerElement} from '@polymer/polymer/
       polymer-element.js';
2
3  class CustomElement extends PolymerElement {
4    static get template() {
5      return html`
6        <style>
7          h3 {
8            color: var(--headline-color, red);
9          }
10       </style>
11       <h3>I am a headline from Polymer.</h3>
12     `;
13   }
14 }
15
```

[1]bower is a package manager for the web, mainly used for front-end dependencies in contrast to npm for back-end dependencies.

```
16  window.customElements.define('custom-element', CustomElement);
```

The returned template literal is converted into an instance of `HTMLTemplateElement`. This is for a performance and for a flexibility reason. By using the `html‘‘` template tag from *lit-html*[26], it is possible to use ES6 template strings and fast rendering. Whenever a variable inside the tag changes, only the value will change in the DOM, not the entire template. This template tag is similar to the virtual DOM from React[2].

For example, only the `message` part will be rendered again in case it changed.

LISTING 4.15: custom-element.html

```
1  var message = "world";
2  console.log(`Hello ${message}!`);
3
4  // => "Hello World!"
```

4.4.3 Styling module

The function `document.createElement('dom-module')` creates a Polymer styling module (in fact, it is a simple polymer element):

LISTING 4.16: shared-styles.js

```
1  const styleElement = document.createElement('dom-module');
```

The styling element has to be filled with styles that can be shared across web components (custom properties can be used to exactly target component's styles):

LISTING 4.17: shared-styles.js

```
1  styleElement.innerHTML =
2  `
3      <template>
4          <style>
5              /* Shared styles */
6              :host {
7                  --headline-color: green;
8              }
9              h1 {
10               color: purple;
11             }
12         </style>
13     </template>
14 `;
```

Finally, it is necessary to register the styling module (on listing 3.12) by using the same function `register()` as normal Polymer elements:

LISTING 4.18: shared-styles.js

```
1  styleElement.register('shared-styles');
```

[2] A JavaScript library built by Facebook to develop single page applications using a virtual DOM to render user interfaces.

After being registered, it is now possible to include the shared module in components that need shared styles. Listing 3.13 is displaying a Polymer Element with a headline. To add the shared styles module, it is necessary to use the keyword `include=""` with the name of the element registered before in listing 3.12, in that case *shared-styles*.

The headline (line 8, listing 4.19) will be green due to the custom property `--headline-color` of listing 4.17. It is important to have the same variable naming to correctly match the custom property value.

LISTING 4.19: custom-element.js

```
1  static get template() {
2    return html`
3      <style include="shared-styles">
4        h3 {
5          color: var(--headline-color, red);
6        }
7      </style>
8      <h3>I am a green headline from Polymer.</h3>
9    `;
10  }
```

4.4.4 Sharing styles

Sharing styles between web components using Polymer is probably the best way. Not only it provides an intuitive way to include styles with just one keyword `include=""`, but also has non-styling advantages like project structure, community support, backing from Google, HTML template literals, build settings...

A community of over 10 000 developers are registered on the official Slack[3] to answer questions and get help[27]. A website called WebComponents.org containing over 900 elements making it easy to share and import web components. The technology company Google is the initiator behind the Polymer project who is using it itself for their own products[13].

Developing web components with Polymer is easier and faster for newcomers because it adds some syntactical sugar on top of it (e.g., jQuery did for JavaScript).

A library like Polymer offers a project predefined template shown on figure 4.1.

- /**node modules**: Polymer uses since version 3 npm modules that allows developers to quickly install necessary dependencies and have the freedom to add external plugins.
- /**src**: location with all custom elements.
- /**styling**: custom folder created for the style module that will share styles between web components. Every other Polymer element will need to `include` the `shared-styles.js` module to get styles.
- /**test**: folder with custom elements tests.
- **.gitignore**: file with list of files that need to be ignored for the source control provider git.

[3]Slack is a team collaboration tools and services.

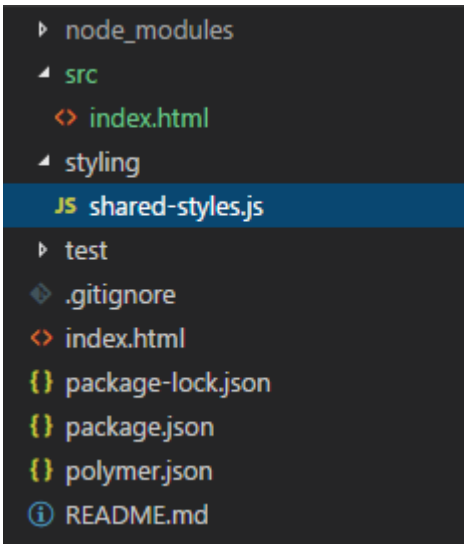

FIGURE 4.5: Polymer project structure

- **package-lock**: stores an exact dependency tree with each version of each npm module.
- **package.json**: information about requirements in the package.
- **polymer.json**: general information about the Polymer project.

The author of a Polymer element can provide custom CSS properties that the user can take advantage of to style the appearance of the element. This way, it is not necessary to know how the element's code works.

For example on listing 4.3, a developer has authored two elements, `<flex-container>` and `<flex-item>`, which can be used together to create layouts in columns or rows.

LISTING 4.20: index.html

```
1 <flex-container>
2   <flex-item>flex item 1</flex-item>
3   <flex-item>flex item 2</flex-item>
4   <flex-item>flex item 3</flex-item>
5 </flex-container>
```

In the documentation for flex-container the author has provided a custom CSS property, `-flex-direction`. It controls whether the flex-items are displayed in a column or a row. You can assign your own value to `-flex-direction` in your app:

LISTING 4.21: index.html

```
1 html {
2           /* Set the value of the custom CSS property
                --flex-direction */
3           --flex-direction: column
4         }
```

Besides, when the time comes to publish the app, Polymer offers a range of options when building the project for production [28]. There is no need for the developer to handle browser support as Polymer takes care of it.

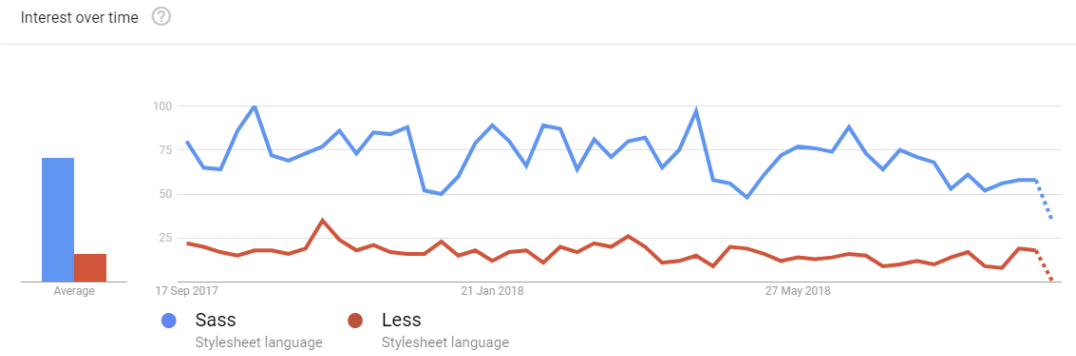

FIGURE 4.6: Google Trends comparing Sass and Less (preprocessor languages)

Due to all previous reasons, Polymer is the most appropriate tool for web components beginners wishing to start quickly, without much effort and with a built-in solution to share styles. Likewise, it is also interesting for companies that have a small budget but need to build a lot of components.

4.5 Preprocessor language

Normally, classic CSS consists of a selector that will apply styles to an element with given rules:

LISTING 4.22: styles.css

```
1 nav a {
2   color: green;
3 }
4 nav li {
5   color: purple;
6 }
```

On listing 4.22, line 1, `nav a` represents the selector (in this case the a navigation's link). On line 2, the keyword `color` represents the rule given to the element with the corresponding value `green`.

A preprocessor language is a scripting language that is interpreted into CSS. According a survey from Ashley Nolan asking about the preprocessor language usage, 63.49% of developers use Syntactically Awesome StyleSheets (Sass) against 10.22% for the second most famous Less [4] [29]. Moreover, regarding number of interest over time on Google Trends [30] (figure 4.6) the most famous scripting language is Sass [5].

Sass extends CSS by providing the methodology of object-oriented languages. It makes it possible to use variables, have a nested code (e.g., like HTML) and import Sass code from other `.scss` files [31].

[4] Extension of CSS with more functionality.

[5] Sass: Syntactically Awesome StyleSheets is an extension of CSS that adds power and elegance to the basic language.

On listing 4.2, the Sass code has the same effect as on listing 6.1. The difference is the fact that the encapsulation of CSS selectors provides a more logical structure for the developer, which will be more productive.

LISTING 4.23: styles.scss

```
1 nav {
2   a {
3     color: green;
4   }
5   li {
6   color: purple;
7   }
8 }
```

Using Sass in a front-end ecosystem is particularly interesting for the reason that Sass variables combined with link tags will achieve the same job as custom properties with link tags do.

A `shared-styles.scss` owning necessary variables that need to be shared across web components:

LISTING 4.24: shared-styles.scss

```
1 /* COLORS */
2 $primary-color: green;
3 $secondary-color: purple;
4
5 /* SIZES */
6 $primary-font-size: 15px;
```

When importing `shared-styles.scss` with the **@import** keyword, all variables will be accessible. This way, it is possible to get through the shadow DOM and style specific parts of a component:

LISTING 4.25: component-styles.scss

```
1 @import '../../styles/shared-styles';
2
3 /* COMPONENT DEFAULT STYLES */
4 $primary-color: black !default;
5 $secondary-color: purple,
6
7 h1 {
8   color: $primary-color;
9 }
```

Later, during CSS compilation, variables will be replaced by their respective values. This way, in the production build[6], everything will be optimized and each component will just have what it needs from the `shared-styles.scss`:

LISTING 4.26: component-styles.css

```
1 h1 {
2   color: green;
3 }
```

[6]The production build is the final code running on the web server.

With that solved, custom properties are actually not necessary when a project is using Sass as a preprocessor language. Moreover, Sass adds more functionality for styling which reinforces the decision to use Sass over custom properties.

Chapter 5

Design

Polymer's primary mission is to facilitate the development of web components. Its project structure, community support, backing from Google and styling possibilities makes it the best platform to start creating web components. However web agencies asking for more flexibility may need another solution. They tend to take the web component development process even further with external tools allowing companies to develop faster.

Regarding styling, the solution here with Biotope is to use the preprocessor language Sass combined with link tags. They are using Sass not only to share styles between web components but also to use mixins[1] or nesting. This means custom properties are not needed.

Biotope is a front-end framework that uses following most important technologies and tools on a Node.js[2] run-time environment:

- **Yarn** a node package manager to download and install required components for Node.js with ease [32].
- **HyperHTML** is a fast and light virtual DOM[3] alternative. Zero dependencies, no polyfills needed, and it fits in about 4.6 Kilobytes (minified[4]) [33].
- **Biotope element.** The framework Biotope is based on the Biotope element repository [34]. It is an extension of the vanilla HTML element which adds some additional features. Because it is a ES6 class, it can be used alone in any project just by importing it via a npm command.
- **Handlebars** provides the power necessary to let you build semantic templates effectively with no frustration [35].
- **Gulp** is a task runner tool that enables to automate build processes [36].
- **Typescript** is a programming language that when compiled[5] becomes JavaScript. It adds type checking and syntactical sugar[37].
- **Sass** to facilitate project's styling that allows CSS variables and imports. Sass code is compiled into CSS.

[1] With a mixin it is possible to make CSS declarations to reuse throughout the project.
[2] Node.js is an environment to run a webserver to browse components of the front end.
[3] A virtual DOM is the representation of the actual DOM that will check faster which parts of the DOM has been changed.
[4] Minification is the process of removing all unnecessary characters from source code without changing its functionality.
[5] Compiling is the process of translating one source code into another one.

5.1 Framework architecture

Every Biotope project has a resources folder containing general styling sheets organized in different folders. The file `styles.scss` shown on figure 5.1 and listing 5.1 imports every project-wide styling sheet. Hence, when the HTML page will be parsed by the browser, only one style sheet (`styles.css`) will be requested from the server.

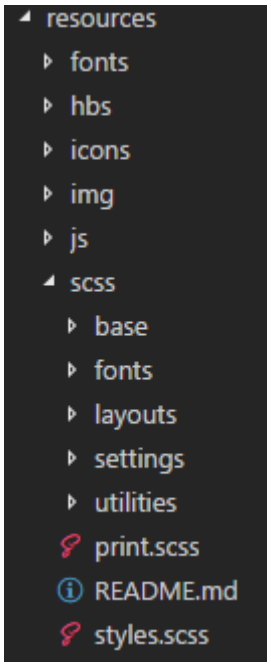

FIGURE 5.1: Resources folder inside a Biotope project

LISTING 5.1: styles.scss

```
1  @charset "UTF-8";
2
3  /* global settings */
4  @import "settings/settings";
5
6  /* functions */
7  @import "utilities/functions";
8
9  /* fonts */
10 @import "fonts/fonts";
11
12 /* breakpoint */
13 @import "../../../node_modules/breakpoint-sass/stylesheets/
       breakpoint";
14
15 /* layout */
16 @import "layouts/default";
17 @import "layouts/section";
18 @import "layouts/grid";
19
20 /* base styles */
21 @import "base/margins";
22 @import "base/types";
```

5.2 Styling tools

A company's main goal is to save money. In other words, time. Time can be saved by giving developers the chance to work faster while keeping the same code quality.

Regarding styling, there is one more way to improve code quality and developers productivity, a CSS name scheme.

5.2.1 Naming scheme

Having a naming scheme will help to get a clear CSS code that is scalable and reusable. If alone or in a team, this will increase productivity and help developers to understand even better the link between HTML and CSS. The most famous one is BEM which stands for Block Element Modifier.

- **Block**: an instance which is meaningful for itself alone and is defined by the class name in a division element. For example `<div class=''nameOfBlock''></div>`
- **Element**: a part of the block which alone does not mean anything. Each element belongs to a block.
  For example the title in a slideshow `<h1 class=''slideshow__title''></h1>` is an element belonging to the block slideshow,
  this means `nameOfBlock__nameOfElement`.
- **Modifier**: a variant of the element to primarly modify the appearance.
  To change the color of the slideshow headline
  a developer would do `class=''slideshow__title-greenFont''`
  (`nameOfBlock__nameOfElement-nameOfModifier`).

The color of the slideshow headline you would do `class=''slideshow__title-greenFont''`. (`nameOfBlock__nameOfElement-nameOfModifier`). brings clarity and structure at the same time :

LISTING 5.2: styles.scss

```
1 .slideshow {
2       background: grey;
3       &__title {
4           color: red;
5           &--greenFont {
6               color: green;
7           }
8       }
9 }
```

Chapter 6

Deployment

To illustrate a concept to share styles between web components, there are two biotope components.

1. Navigation
2. Teaser

Each of them will import via Sass `@import` two shared styles files called `settings.scss` and `normalize.scss`.

6.1 Global style sheets

6.1.1 shared-styles.scss

Listing 6.1 contains all styles that need to be shared across web components. This `shared-styles.scss` file needs to be imported in every component that will get shared styles. This means only necessary Sass variables will get through the shadow DOM and override Sass variables the component may need. This depends if the component author agreed to let the shared guide affect his elements.

LISTING 6.1: shared-styles.scss

```
1  // GLOBAL SASS SETTINGS
2
3  // FONTS
4  $font-family-sans-serif: Arial, sans-serif;
5  $font-weight-normal:     400;
6  $font-weight-bold:       600;
7
8  // Z-INDEXES
9  //$z-index-component-xyz: 1;
10
11 // COLORS
12 $white: #fff;
13 $black: #000;
14 $primary-color: green; // the color we will test later on
15
16 // COMPONENT STYLES
17 $content-max-width: 1280px;
18 $component-max-width: 1200px;
19
20 // breakpoint definitions
21 $xsmall:  300px;
22 $small:   600px;
```

```
23 $medium:   768px;
24 $large:    992px;
25 $xlarge:   1200px;
```

6.1.2 normalize.scss

Listing 6.2 contains all styles that component needs as default styles. These styles are able to be overridden from component's styles because they will be linked **before** them through a `<link>` tag. It will not be imported via Sass, because otherwise the whole `normalize.scss` source code would be in each custom element. This is not efficient. The file is located at `'/resources/scss/normalize/normalize';`.

LISTING 6.2: normalize.scss

```
1 h1 {
2     margin-left: 15px;
3     border-left: 3px solid green;
4     padding-left: 15px;
5 }
```

6.2 Navigation

The navigation's styles source code. On line 1, shared styles get imported from `settings.scss` file:

LISTING 6.3: Navigation.styles.scss

```
1  @import '../../resources/scss/settings/shared-styles';
2
3  .navigation {
4      height: 50px;
5      background-color: white;
6      display: inline-block;
7
8      nav {
9          display: inline-block;
10     }
11
12     ul {
13         list-style-type: none;
14         margin-left: 100px;
15     }
16
17     &__logo {
18         color: $primary-color;
19         text-decoration: none;
20     }
21 }
```

The variable `primary-color` retrieves its value from shared styles. It is important to verify that the variable is available in the shared file, otherwise the build task will throw an error. Once compiled through the gulp task, the Sass gets compiled to CSS code:

LISTING 6.4: Navigation.styles.css

```
1  .navigation {
2    height: 50 px;
3    background-color: white;
4    display: inline-block;
5  }
6  .navigation nav {
7    display: inline-block;
8  }
9  .navigation ul {
10   list-style-type: none;
11   margin-left: 100 px;
12 }
13 .navigation__logo {
14   color: red;
15   text-decoration: none;
16 }
```

The compiled `Navigation.styles.css` that contains all styles needed is linked to the navigation component's template:

LISTING 6.5: Navigation.ts

```
1  <link rel="stylesheet" type="text/css" href="resources/css/
       normalize/normalize.css">
2  <link rel="stylesheet" type="text/css" href="resources/components/
       Navigation/Navigation.styles.css">
3              <div class="navigation">
4                  <a class="navigation__logo" href="#">Website logo</
                       a>
5                  <nav>
6                      <ul>
7                          <slot></slot>
8                      </ul>
9                  </nav>
10             </div>
```

6.2.1 Navigation element

Very similar to the navigation component above, the navigation element is a sub-component from navigation. Actually, it is a link in the navigation encapsulated in it own shadow DOM. Line 2 of listing 6.6 is setting an alternative value in case `$primary-color` is not available. The keyword `!default` is really useful for setting component's default styles that can be overridden by shared styles.

LISTING 6.6: NavigationElement.styles.scss

```
1  @import '../../resources/scss/settings/shared-styles';
2  $primary-color: #ff0000 !default;
3
4  :host {
5      display: inline-block;
6
7      a {
8          text-decoration: none;
9          color: $primary-color;
10     }
11 }
```

Once compiled, `NavigationElement.styles.css` gets downloaded from the server when the component is available on the page.

LISTING 6.7: NavigationElement.styles.css

```
1 : host {
2   display: inline - block;
3 }: host a {
4   text-decoration: none;
5   color: red;
6 }
```

6.3 Teaser

In general, a teaser is a short text combined with a headline and sometimes a picture. This teaser has `<slot>` tags that are available to insert custom text from outside without knowing the component's structure:

LISTING 6.8: Teaser.ts

```
1 <link rel="stylesheet" type="text/css" href="resources/css/
    normalize/normalize.css">
2 <link rel="stylesheet" type="text/css" href="resources/components/
    Teaser/Teaser.styles.css">
3             <h1>Teaser</h1>
4             <x-text><slot></slot></x-text>
```

And the corresponding styles that will be linked to the template through a `<link>` tag which breaks into the shadow DOM:

LISTING 6.9: NavigationElement.styles.scss

```
1 @import '../../resources/scss/settings/shared-styles';
2 $primary-color: red !default;
3
4 h1 {
5     color: $primary-color;
6 }
```

6.4 Testing

It is necessary to pass two tests.

- Did `normalize.css` correctly passed through the shadow DOM ?
- Did `shared-styles.css` correctly override component's styles ?

Normalize.css. First `normalize.css` will be empty then it will be filled with styles from listing 6.2. This way, it is possible to see if styles have correctly affected the component. The result is visible on figure 6.1. There the border, padding and margin styles have been correctly changed. Differences are circled in blue.

Shared-styles.css In order to verify that shared styles are able to style web components having a shadow DOM and break through it, the `primary-color` (line 14,

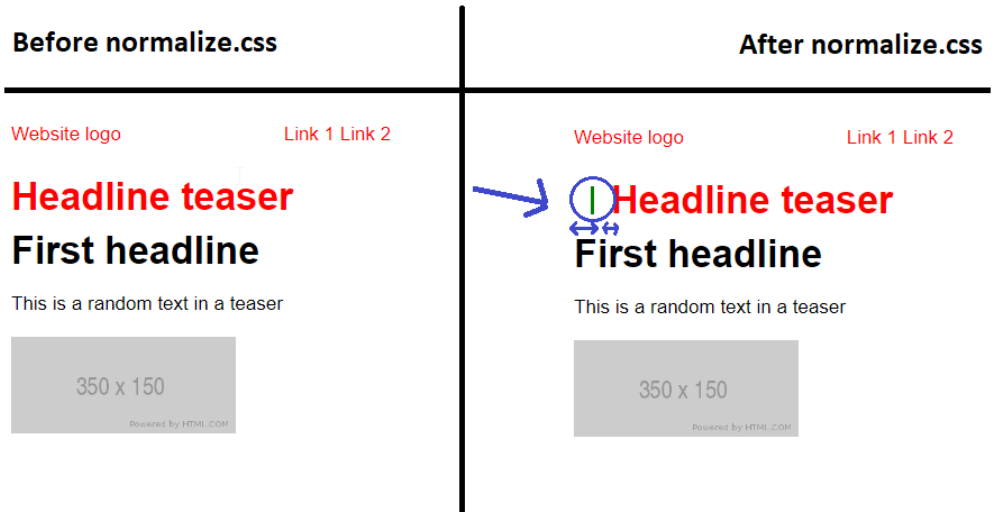

FIGURE 6.1: Screenshot of website before and after normalize.css styling

FIGURE 6.2: Screenshot of website before and after shared-styles.css styling

listing 6.1) in `settings.scss` will change from red to green. The primary color successfully changed from red to green. Differences are circled in blue. The result is visible on figure 6.2.

Chapter 7

Conclusion

7.1 Outlook

7.1.1 Web components

According to a survey from Ashley Nolan about knowledge of JavaScript libraries and frameworks, almost 71 percent of 5 461 developers heard of or read about Polymer, the primary library to build web components (figure 7.1) [29]. But only 90 developers feel comfortable using it. A low number showing that web components in general, not only Polymer, is not yet a mainstream usage. In comparison, approximately 80 percent of respondents feel comfortable with using jQuery. The front-end community has not yet seen any advantage by using web components over classic libraries.

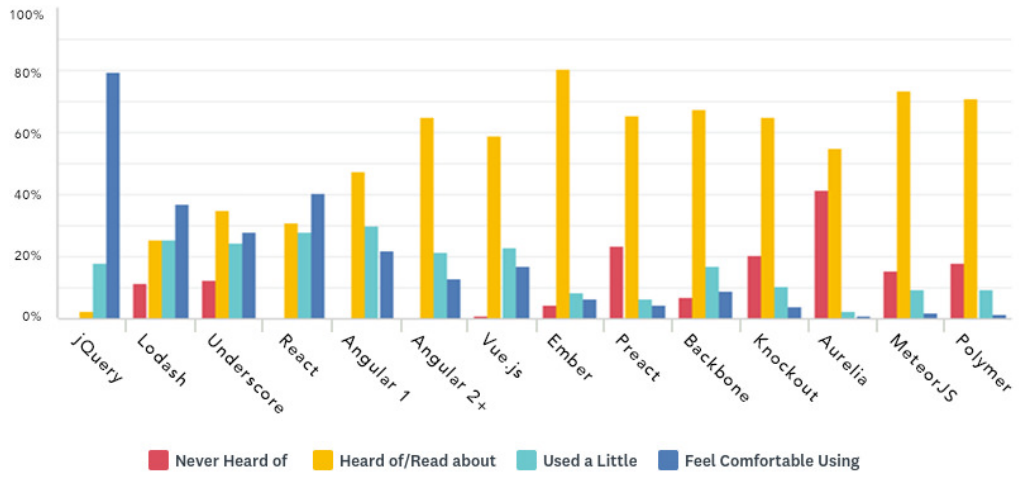

FIGURE 7.1: Knowledge of JavaScript libraries and frameworks [29]

Before the coming of web components, front-end developers built websites using all sort of tools, libraries (e.g., jQuery) or frameworks (e.g., Angular). They helped to develop faster and more structured. Companies needed to decide which technology is the most suited for their project depending their requirements. It should be easy to install, maintained, have a big community and a low learning curve. Sometimes, companies notice that there is a new framework better than the one they are using. They have to rebuild everything again and spend too much time on it. Now with web components, front end is coming back to pure JavaScript without any sugar on

top. The idea is not to use any framework or library again, hence project technologies may no more be outdated. It is also much easier to share components between projects and even companies.

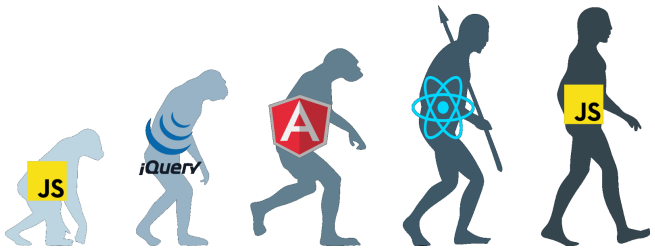

FIGURE 7.2: Front-end evolution seen by Mikhail Vazhenin [38]

However, with the coming of the library Polymer, is the front-end community starting all over again ?

7.1.2 Styling

As a quite recent and a yet not mainly used technology (figure 7.1), there is no standard way or definitive solution to style web components. Due to the usage of shadow DOM, it is hard to use the classic way to share styles. In fact, shadow DOM encapsulates everything and prevents styles to leave and to enter the custom element. Over the years, the web component community proposed different ways to allow at the same time encapsulation and shadow DOM styling (e.g. `<link>` tags). Today several methods are possible to share styles between web components. Either use single one or combine them with other depending on project's needs.

Some methods are still under consideration but not yet implemented like constructable style sheets which might be a long-term solution to share styles [39]. Constructable style sheets allow the creation of `StyleSheet` objects in JavaScript through a constructor function. The constructed style sheet could then be added to the shadow DOM through an API, which would allow the shadow DOM to use a set of shared styles.

Besides, `::part` and `::theme`, two pseudo-elements named **shadow parts**, are very likely to gain traction and receive more attention in the next years [40]. In combination with custom properties, which are able to break through the shadow DOM, these pseudo-elements allow shared styles to interact in safe, powerful ways, maintaining encapsulation without surrending all control. Not implemented yet and still in development [41], it is too early to say if shadow parts might be the ultimate way to share styles between web components.

Chapter 8

Source code

All code samples and images are available on Github: https://github.com/luiswill/bachelorWebComponentStyling

The project source code based on the Biotope framework is available on Github: https://github.com/luiswill/project-thesis.

A sample of the project is online: https://luiswill.github.io/02Personal.page.html.

Bibliography

[1] URL: https://github.com/biotope/biotope-build.

[2] Yevgeniy Valeyev. *Brief history of web components*. Feb. 2017. URL: https://fr.slideshare.net/YevgeniyValeyev/brief-history-of-web-components-72452483.

[3] Justin O'Neill. *A history of Javascript Frameworks*. test. May 2017. URL: https://medium.com/@oneeezy/frameworks-vs-web-components-9a7bd89da9d4.

[4] URL: https://jquery.com/.

[5] Mozilla Developer Network. *Using custom elements*. Sept. 2018. URL: https://mzl.la/2x54teG.

[6] *WebKit Feature Status*. URL: https://webkit.org/status/#feature-html-imports.

[7] Anne van Kesteren. *Mozilla and Web Components: Update*. Dec. 2014. URL: https://hacks.mozilla.org/2014/12/mozilla-and-web-components/.

[8] URL: https://angular.io/.

[9] URL: https://reactjs.org/.

[10] W3C. *Web Components specifications*. Mar. 2018. URL: https://github.com/w3c/webcomponents.

[11] 3dman-eu. *Custom element's representation*. Mar. 2017. URL: https://pixabay.com/fr/blocs-de-construction-insert-2065238/.

[12] Jake Harding. *Upcoming Change to Embedded Tweet Display on Web*. May 2016. URL: https://twittercommunity.com/t/upcoming-change-to-embedded-tweet-display-on-web/66215/2.

[13] URL: https://github.com/Polymer/polymer/wiki/Who%27s-using-Polymer%3F.

[14] Mozilla Developer Network. *HTML Imports*. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components/HTML_Imports.

[15] Eiji Kitamura. *Introduction to the template elements*. Oct. 2014. URL: www.webcomponents.org/community/articles/introduction-to-template-element.

[16] Eric Bidelman. *Shadow DOM 201*. Oct. 2016. URL: https://www.html5rocks.com/en/tutorials/webcomponents/shadowdom-201/.

[17] *Add support for external stylesheets*. May 2015. URL: https://www.w3.org/Bugs/Public/show_bug.cgi?id=22539.

[18] *Custom Properties support*. URL: https://caniuse.com/#search=custom%20properties.

[19] *Polymer Github repositoy*. URL: https://github.com/Polymer/polymer.

[20] *Polymer homepage*. URL: https://www.polymer-project.org/.

[21] *SkateJS Github repository*. URL: https://github.com/skatejs/skatejs.

[22] *X-Tag homepage*. URL: https://x-tag.github.io/.

[23] *X-Tag Github repository*. URL: https://github.com/x-tag/x-tag.

[24] *Slim.js Github repository*. URL: https://github.com/slimjs/slim.js/blob/master/README.md.

[25] *Slim.js homepage*. URL: http://slimjs.com/#/getting-started.

[26] *lit-html homepage*. URL: https://polymer.github.io/lit-html/.

[27] *Polymer Slack*. URL: https://polymer-slack.herokuapp.com/.

[28] *Polymer build settings*. URL: https://www.polymer-project.org/3.0/docs/tools/polymer-cli-commands#build.

[29] *Survey of The State of Front-End Tooling 2018 to 5461 developers*. 5461 responses. July 2018. URL: https://ashleynolan.co.uk/blog/frontend-tooling-survey-2018-results.

[30] *Google Trends*. Retrieved September 11, 2018 by searching Sass Stylesheet language and Less Stylesheet language. URL: https://trends.google.fr/trends/explore.

[31] *Sass Basics*. URL: https://sass-lang.com/guide..

[32] *Yarn homepage*. URL: https://yarnpkg.com/en/.

[33] *HyperHTML Github repository*. URL: https://github.com/WebReflection/hyperHTML.

[34] *Biotope element Github repository*. URL: https://github.com/biotope/biotope-element.

[35] *Handlebars homepage*. URL: http://handlebarsjs.com/.

[36] *Gulp homepage*. URL: https://gulpjs.com/.

[37] *Typescript homepage*. URL: https://www.typescriptlang.org/.

[38] Mikhail Vazhenin. *Survey of The State of Front-End Tooling 2018 to 5461developers*. July 2017. URL: https://codeburst.io/building-efficient-components-6ee2bdaea542.

[39] Tab Atkins-Bittner. *Constructable Stylesheet Objects*. Nov. 2016. URL: http://tabatkins.github.io/specs/construct-stylesheets/.

[40] Tab Atkins-Bittner. *CSS Shadow Parts*. Aug. 2018. URL: https://drafts.csswg.org/css-shadow-parts-1/.

[41] *Github shadow parts issues*. URL: https://github.com/w3c/csswg-drafts/labels/css-shadow-parts-1.