Application of a Declarative Shadow DOM for the Definition of Web Components

Andreas Hahn

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2020

Advisor:

FH-Prof. DI Rimbert Rudisch-Sommer

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 30, 2020

Andreas Hahn

Contents

Acknowledgments

It was intense working on this thesis and investigating Web Components during the times of a pandemic. While it was demanding, it also allowed me to work more on my interests and backgrounds in research and on how a web standard emerges. The idea of this thesis was born after an argument that Web Components are not there yet. The statement during a discussion was "Server-side rendering is a must-have, Web Components can't be server-side rendered so leave them behind." Then I started thinking and looking for solutions to that particular problem.

At this point, I would like to thank my supervisor FH-Prof. DI Rimbert Rudisch-Sommer for providing me with ideas, feedback and the necessary flexibility to write this thesis during the COVID-19 pandemic. I also want to thank bet-at-home for supporting me and my vision and for giving me the possibility to work on this thesis. Thanks to my colleagues, for their feedback, as well as to everyone I had the pleasure of telling about my research.

Finally, my respect and gratefulness goes out to all developers driving the web community and its standards forward. Despite mostly not scientifically publishing their results, this community has come up with brilliant ideas about how to make web development better. Without the efforts of these never-resting people, this research and project would have never happened in the current form. To contribute towards this active community, all the work involved in this thesis was also open-sourced and published on GitHub[1].

Most of all, thanks to my friends, family, and especially my sister, who was always there to encourage and advise me when I needed a push to continue.

[1] https://github.com/drdreo/declarative-shadow

Abstract

Web Components complement client-side web development with a native component model. However, as of today, the interest in adopting the web standard is still limited because of disadvantages such as server-side rendering. The idea of it is nothing new, but it has been said that Web Components can not be rendered on the server due to the lack of a declarative representation of a shadow DOM. The purpose of a declarative approach is to empower non-scripting environments with the features Web Components provide, especially the encapsulation aspect of the shadow DOM. This would also enhance server-side pre-rendering of Web Components and additionally push the adoption of this feature further among the web community. A declarative shadow root can potentially establish a new way of attaching a shadow DOM additionally to the imperative way. The goal is not to replace the current mechanism, but to expand the use cases of the shadow DOM.

The original proposal of a declarative shadow DOM was declined in 2019 by browser vendors and web specification developers due to implementation costs. The main aim of this thesis is to create a reasonable prototype, point out the advantages and disadvantages of a declarative approach as well as provide reasons to reconsider developing it.

Kurzfassung

Webkomponenten ergänzen die clientseitige Webentwicklung um ein natives Komponentenmodell. Jedoch fehlt, vom heutigen Stand aus, das Interesse um den Webstandard anzunehmen, da Nachteile, wie das serverseitige Rendering, existieren. Die Idee ist nicht neu, aber es wird angenommen, dass Webkomponenten nicht auf einem Server gerendert werden können, da es keine deklarative Darstellung eines „shadow DOM“ gibt. Der Zweck eines deklarativen Ansatzes besteht darin, nicht-skriptbasierte Umgebungen mit der Funktionalität auszustatten, die Webkomponenten bieten, insbesondere der Abkapselung durch das „shadow DOM“. Dies würde ebenso das serverseitige rendern von Webkomponenten, respektive die Zugänglichkeit deren Inhalte für Maschienen verbessern. Ein deklarativer „shadow-root“ kann potenziell eine neue Art der Instanziierung eines „shadow DOM“ zusätzlich zum imperativen Weg etablieren, obwohl der derzeitige Mechanismus dadurch nicht ersetzt wird, sondern die Anwendungsbereiche des „shadow DOM“ erweitert werden sollen.

Der ursprüngliche Vorschlag eines deklarativen „shadow DOM“ wurde 2019 von Browser-Herstellern und Entwicklern von Web-Spezifikationen wegen den Implementierungskosten abgelehnt. Das Hauptziel dieser Arbeit ist es, einen vernünftigen Prototyp zu erstellen, die Vor- und Nachteile eines deklarativen Ansatzes aufzuzeigen sowie Gründe zu liefern, um das Weiterentwickeln in Erwägung zu ziehen.

Chapter 1

Introduction

The motivation for the work presented in this thesis comes from the rising interest in component-based web development. Components have become an essential concept in almost any modern web framework. To cope with the trend, and also to be more future-proof, using Web Components has to become more feature-rich and productive.

Over the years, web engineers tried to introduce a semantic web to empower machines with the ability to understand and analyse websites. Berners-Lee and Fischetti's vision is to make it possible for machines to handle the web, more specifically, to recognise the meaning of the information contained in a website [3]. However, since the proposal of Web Components by Alex Russell in 2011 [8], there exists a new possibility to structure websites with actual components and isolate sections inside a web page. Due to this Web Component standard, the gained semantic web faces a threat. Through the shadow DOM, Web Components are able to isolate markup of web pages which crawlers and bots as of today are unable to access and interpret. This makes Web Components a less viable choice because search engine optimisation (SEO) is an essential goal for making a website competitive. Web Components render information on the client-side and to optimise such a website, the page can be server-side rendered beforehand to provide the initial HTML with sufficient information. The idea of it is nothing new, but it has been said that Web Components can not be rendered on the server due to the lack of a declarative representation of a shadow DOM.

Moreover, since 2019 all *evergreen*[1] web browsers support the required parts of Web Components. The key features like encapsulation, reusability and framework agnostic are the strengths that should not be underestimated. Although the courage to adopt the web standard, as of today, is still limited. This might be because major JavaScript frameworks handle the component approach already fairly well with reliable tooling available. Web Components, on the other hand, are somewhat fresh and do not have a pre-defined way of handling common problems yet. Some of those problems include the usage of SCSS, which is a convenient way to write CSS but is not supported natively. Other more complicated problems like the representation of a shadow DOM during server-side rendering or a declarative way to create Web Components also exist and have no standard solution yet.

Web Components combined with the shadow DOM provide a way to encapsulate

[1] Browsers that are automatically upgraded to future versions, i.e. Chrome, Firefox, Opera, Edge.

HTML, logic and also introduce a native way of CSS style scoping. However, Web Components have to be defined via JavaScript. Hence, the functionality a shadow DOM provides is not available in non-scripting environments like for SEO cralwers, bots or for users who disable JavaScript on web pages. A declarative, HTML-only way for encapsulation would vanish the requirement of JavaScript completely for such use cases. This is why a declarative shadow DOM caught some interest over the past and became an often requested feature from developers. The intention is not to replace the current imperative way of attaching a shadow root but to expand the use cases for a shadow DOM. Nevertheless, browser vendors have rejected the previous proposals of standardising a declarative shadow DOM syntax for a variety of reasons. This thesis attempts to resolve prior objections and provides a solution that all parties might be able to approve.

The main objective of this thesis consequently is the exploration of a relevant declarative shadow DOM approach. This involves both the conception of such a mechanism, as well as the implementation. The approach will be implemented as a custom element because changing a browser's parser is not feasible for the scope of this thesis. The proposed solution includes a new HTML element for a declarative shadow DOM which allows HTML authors to create style scopes as they please and makes it possible to server-side render Web Components. As far as the implementation is concerned, a prototype was developed that takes the existing proposal for declarative custom elements [16], as well as the declined proposal [25] into consideration to comply and create a feasible solution. The goal also includes to create a potential basis for declarative Web Components. The idea behind it is not only to make server-side pre-rendering of Web Components more efficient and straight-forward, but also to push the adoption of this idea further among the community. In total, the contribution of this thesis can be summarised as follows:

- Analysis of existing proposals and discussions which recommend certain behaviour and the choice of instantiation.
- Implementation of a prototype following these recommendations.
- Evaluation of the implemented solution.
- Creation of an open-source project which can be used and further enhanced by other developers.

1.1 Outline

This thesis is organised in seven chapters. In the beginning, the context around Web Components, as well as the presently available mechanisms are established by introducing the reader to the web standard, current use cases of the technology as well as background information on how the specification has evolved over the years in contrast to framework solutions in Chapter 2.

Chapter 3 takes a look at existing proposals and how those were trying to achieve a declarative solution. The core discussion between developers, browser vendors and specification developers, which inspired the whole idea of the thesis, is introduced, and the main advantages of such an approach are further explained. Current workarounds for a declarative shadow DOM are then discussed, analysed and presented with a concrete

server-side rendering example.

In Chapter 4, requirements for the planned solution are established by analysing the aim of a declarative shadow DOM, stating the current lack of a proper solution and also by demonstrating a concrete use case. Based on these requirements, a solution is proposed to define the expected behaviour and usage.

Chapter 5 lays out the entire technical implementation details. It defines the scope of the project to then cover the testing setup, before offering more insight into the underlying algorithm. Afterwards, the application of the solution is presented, and finally, possible emerging problems and polyfill strategies are mentioned.

In Chapter 6, the results of this work are evaluated. First, the specification is discussed, after which a look into performance is made by comparing website load times and bytes transferred. The reasonableness is then reviewed, followed by a demonstration of how SEO crawlers are able to interpret the solution and finally, the chapter concludes by mentioning possible improvements.

Chapter 2

Web Components

Web pages consist of building blocks, so-called HTML elements. There are standardised or "native" elements that have a predefined look and an expected functionality. The behaviour and styling are provided without any involvement of the developer. Every HTML element works according to this principle, which makes HTML code easier to write and to predict. By combining such elements with specially defined stylesheets, complex structures of HTML elements with complex CSS and JavaScript can be built.

To avoid developers having to keep the entire structure in mind, it has proven itself to break code down into small, manageable parts. In units that make the whole web page maintainable. These parts can be simple functions, modules, components or many other approaches that divide systems into atomic parts like explained in [19].

As the name Web Component suggests, this technology allows web development to be split into smaller, reusable, modular components through encapsulation which a shadow DOM (see Section 2.3) provides. The clear advantage is that Web Components can be created entirely with native HTML, CSS and JavaScript, independent of a framework, library or tool. Native APIs allow developers to define new HTML elements that are compatible with any framework, not just with Angular[1] or React[2], but any web application, including legacy applications like Java Server Pages[3]. A Web Component is based on the following specifications:

- Custom elements: APIs to define new HTML elements.
- HTML template: An HTML fragment that is not rendered, but stored until instantiated via JavaScript.
- Shadow DOM: Encapsulates DOM and styling from the rest of the web page.

All parts of the Web Component specification (except the HTML import API, which was not mentioned yet because the concept was never adopted as a standard and was superseded in favour of ES modules) have already been merged into the DOM and HTML specification [21, 22]. As of writing this thesis, all necessary features of Web Components are supported in every major browser (Chrome, chromium-based Edge, Firefox, Safari)[4].

[1] https://angular.io
[2] https://reactjs.org
[3] https://www.oracle.com/technetwork/java/index-jsp-138231.html
[4] https://caniuse.com/#search=web%20components

2.1 Custom Elements

The custom elements interface gives developers the ability to create their own customized DOM elements, extend existing HTML tags and use elements created by other developers. Although this was achievable before by using non-standard elements and adding functionality through scripting, such elements have been non-conforming and not very suitable. By defining a custom element, authors tell the parser how to construct an encountered element accordingly and how elements of that type should behave. The following snippet shows the definition of a new custom element called "my-element":

```
class MyElement extends HTMLElement {...};
customElements.define("my-element", MyElement);
```

2.2 HTML Templates

The purpose of the `<template>` element is to declare HTML fragments that can be cloned and inserted into the document via JavaScript during runtime. This element serves as a storage for HTML markup, which only renders its content when cloned and inserted. Inside a template, scripts are not executed, and assets do not load. Additionally, a template node itself is not considered to be in the document until it is activated. Instances of Web Components that make use of a `<template>` element only reuse a single template reference. This specification adds client-side templating to the feature-set of a web developer. It is used as shown in the following snippet:

```
<template>
    <span>Hello there!</span>
</template>
```

2.3 Shadow DOM

The shadow DOM API provides a way to create and include a new DOM tree within the rendering document, not in the main DOM tree, but as a special kind of sub-tree with restricted access, called shadow DOM tree or short shadow tree. The regular DOM node, which has the shadow tree attached, is called the shadow host. The root node of the shadow tree is called the shadow root, underneath which any regular DOM element can be inserted. The border where the shadow DOM ends, and the regular DOM begins is called the shadow boundary.

A shadow DOM can be attached to any valid custom element, `article`, `aside`, `blockquote`, `body`, `div`, `footer`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `header`, `main`, `nav`, `p`, `section`, or `span` elements. The API is used as follows:

```
const div = document.createElement('div');
const shadowRoot = div.attachShadow({mode: 'open'});
```

The key intention of a shadow DOM is the separation of the regular DOM tree and shadow DOM trees (as seen in Figure 2.1), which enables the encapsulation of DOM elements and their associated styles. Scripts, as well as styles from the main document, can not leak in. Neither can scripts or styles defined inside a shadow DOM affect the rest

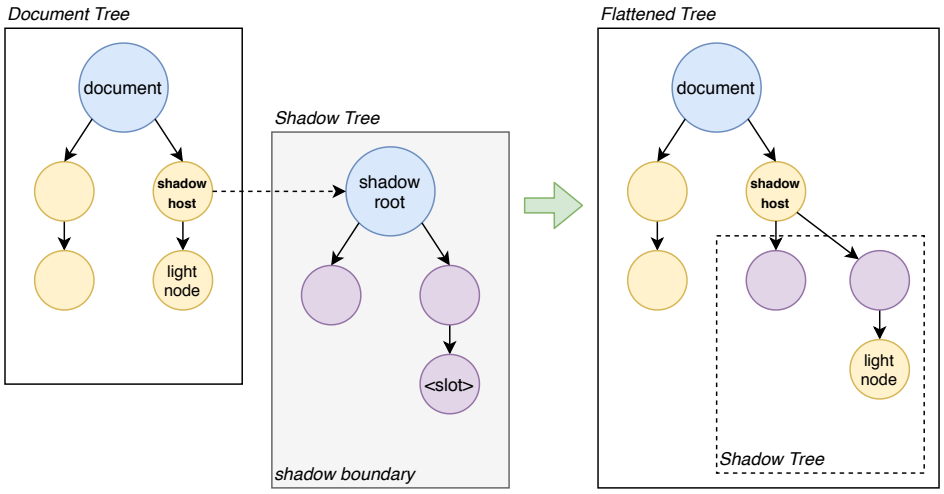


Figure 2.1: Separation of the main DOM tree and the shadow DOM tree. Resulting in a flattened tree when being rendered. (a) The document tree containing a shadow host that contains a light DOM node. (b) The shadow tree containing a `<slot>` element. (c) The rendered flattened tree with distributed light DOM. Image source [11].

of a web page, which prevents unintended behaviour or styling. This makes it possible to reuse generalized or common style selectors and even reduce selector complexity, for example, by referring to element tags directly, without concerns about style interference. Although this shadow boundary exists, so-called slots provide a mechanism to use DOM elements from the main document and render them inside the shadow tree. Slots make up for the lack of composition between components, which is discussed in Section 2.3.1.

2.3.1 Slotting

By default, if an element has a shadow DOM, the shadow tree is rendered instead of the element's children. This means any children in the light DOM (elements nested underneath the shadow host) are not rendered. Nevertheless, a `<slot>` element can be added to the shadow tree to allow children to render in its place. A slot is kind of a placeholder defining where child nodes render. A unique behaviour of shadow trees is that nested elements are not necessarily being rendered in the same order they appear in the light DOM. Any element in the light DOM that does not specify a named slot is rendered in place of the default slot (a slot without a name). Elements that have a slot attribute and specify a name are placed instead of that named slot. The process of turning the light DOM and shadow DOM tree into a single tree for rendering is called flattening the tree. However, the flattened tree exists only for rendering and event-handling purposes. The nodes in the document are not repositioned.

The author of a component can control where a child element should be distributed into the flattened tree (as seen in Fig. 2.1 (c)) using a default slot and named slots. This allows for powerful composition of Web Components and makes them more dynamic since the rendered output changes depending on the user of the component. The user of such an element has to keep in mind that the slot attribute is only recognized for

direct children of the shadow host. This native behaviour to distribute nodes is unique and only available when having a shadow DOM.

2.3.2 Style Encapsulation

Styles inside a shadow DOM are scoped to the shadow tree and do not affect elements outside of the shadow boundary. Style selectors outside the shadow tree do not match elements inside the tree. However, inheritable style properties like color or font still inherit down from host to shadow tree. Other than that, it is not possible to directly style anything inside a shadow tree using a CSS rule from the outside. To allow users to customize a component's visual representation, it is possible to expose specific styling properties using CSS custom properties and mixins. Custom properties allow authors to describe the style values they would like to reuse in a component's stylesheet. They inherit through the document's sub-tree, allowing selectors to overwrite the value of a custom property for a given sub-tree without affecting other sub-trees. Custom property names are also able to inherit across component boundaries providing a sort of styling API for components. The following snippet shows how such a styling interface can be set up. It defines a custom CSS property `--component-bg-color` with a default background color "blue", which is used by the `my-element` stylesheet and when it is being hovered, only that one specific element has its background color changed to purple. Other instances of the component are unaffected by the selector:

```
:root {
    --component-bg-color: blue;
}
:host {
    background-color: var(--component-bg-color);
}
my-element:hover {
    --component-bg-color: purple;
}
```

Nonetheless, fully isolating every web component would inevitably lead to a considerable amount of duplicated CSS, especially when it comes to setting up common styling for native elements. When developers want to use an anchor tag, an input field or any other native element in their Web Component, they most likely should be styled consistently like the rest of the website. If a Web Component encapsulates all of the styles it needs, the common CSS of native elements would have to be duplicated across all instances. This would not only hit on performance but also increase maintenance costs. Instead of encapsulating all of the styles, Web Components could only encapsulate their unique styles and import a set of shared styles to handle common styles, as suggested in Section 2.4, using constructable stylesheets.

2.3.3 Data Encapsulation

Web Components can be used to encapsulate data like Beeswax [6], a client-side browser extension for private web applications, is trying to achieve. It utilizes shadow trees to render private data in plain text, but present it as ciphertext to the hosting application. Security concerns are something that our fast-growing digital society has to think of more than ever. Regarding sensitive data leakage of big companies, securing user data

in the browser is a crucial goal. Web Components can help with that but are not the final answer. The analysis of ShadowCrypt [5] is showing that a user interface can easily be mocked to trick users into believing they are entering data securely while they are not. Although their work does not rely on any weakness of the shadow DOM boundary but shows once again that shadow DOM is designed to support web developers and was never meant as a security enhancement. The Web Component specification never defined any safety guarantees, therefore Brucker and Herzberg are suggesting a well-defined interface in [4] through which Web Components can interact with each other. It specifies more regulated accessor functions to reduce unexpected effects outside of the shadow boundary. This is one possible solution to enhance component safety and can potentially lead to more secure applications through Web Components.

2.4 Reusability

Component-based development, as described in [1], emphasizes modular composition of independent and specialized components, which together form a more extensive system. It focuses on a modular approach, which considers the interoperability of components that were specifically designed to be reused within an architecture. Reusability is an essential characteristic of a high-quality software. It is good practice to design and implement software components in such a way that many different programs can reuse them. The capability to reuse relies principally on the ability to build larger constructs from smaller parts and being able to recognize commonalities among those parts. This applies also to Web Components, which are reusable by nature. Essential for this technology is that the author identifies a common use case and abstracts it away from the rest of the web site through a component. This makes it easier to grasp the intention of specific parts, provides a single place for modification and therefore makes the site more maintainable. Additionally, Web Components can be used unlimited and anywhere in the document besides some exceptions like inside the `<head>` element. They even can be shared and reused across multiple websites and projects.

Nevertheless, there is a downside to careless reuse. In complex applications, the element count of a web page can easily exceed tens of thousands. The styles for these components are presumably specified in a small number of stylesheets, perhaps one for each component. Most Web Components make use of the shadow DOM. For a stylesheet to take effect within a shadow DOM, it currently must be specified using a `<style>` element within the shadow tree. Using Web Components excessively can potentially have a noticeable time and memory cost if web browsers force the stylesheet rules to be parsed and stored once for every `<style>` element. Although browsers do some internal optimizations considering the reuse of styles, one can not guarantee that these optimizations always happen. However, the duplications are probably not needed as Web Components will most likely use the same style rules and hardly change them dynamically. A new web specification called “Constructable Stylesheets” has already been proposed at [23], which would solve this exact issue. Constructable stylesheets provide the functionality to define and arrange shared CSS styles, and then apply those styles without duplication to multiple shadow roots or the document. Yet only Chrome has support for this feature, and other browser vendors besides Firefox do not plan to add it in the near future.

2.5 Application Example: Design Systems

A design system is the single source of truth that captures UI patterns which enables teams to design, realize and develop a product consistently. Not only does it contain a collection of concrete components, but it also defines workflows, guidelines and best practices. As the number of platforms and clients increases, the need for a universal UI knowledge-base is becoming more important to reduce inconsistent styles. Such a system should be platform-independent, future proof and easy to customize. That is where Web Components benefit through encapsulating a single UI element's styles, making it the single source of truth. Such components are almost platform-independent as browser-support is sufficient by now, and most client devices have access to a web browser. Using a consistent set of web standards makes them a future-proof choice as well.

Utilizing Web Components to implement a design system is the optimal application considering their reusability and encapsulation. It removes the brittleness that arose over the years with the introduction of naming conventions like Block Element Modifier (BEM) and other methodologies for CSS selectors to reduce their specificity and avoid descendants and location-specific selectors. Those methods were introduced due to selectors from different authors, projects or even applications defining style properties on the same element, which can lead to unintended consequences for the visual appearance of the document [20]. The unfavourable effects occur due to the browsers using a cascade algorithm to calculate which style declarations take effect. This becomes more or less negligible when writing component tailored stylesheets each with its specificity scope. As mentioned in Section 2.3.2, CSS variables can pierce through the shadow boundary, but these are just atomic style rules when the actual need is an entire set of rules. Until recently, Web Components had to duplicate numerous styles in every component. Chrome now supports the use of "Shadow Parts" [5], but there are several configurations to do both in the design system and in the component to mark and allow parts to flow through components. Therefore, the more efficient option would be the broad adoption of constructable stylesheets.

2.6 The History of the Specification

It is worth noting that several proposals in the past have been introduced in an attempt to improve the HTML iframe and its related encapsulation features. None of these have survived in any meaningful way on the web today, like the HTML components proposal in 1998 [24]. Only after a relaunch of the proposal in 2011 at [8] has the technology created broader interest. Some members of the web development community see the Web Component's promise as a broken one [12]. Already deprecating one of the original APIs—HTML imports—did not help to grow confidence either. There have been many attempts to componentize web development, but none has settled so far. The concept was good—extending the DOM with custom tags that enhance readability and reusability. The implementations were not. With the release of Node.js in 2009 and, therefore, the possibility to run JavaScript on a server, a rising trend developed in the direction of universal JavaScript frameworks that have own ways to create components

[5] https://www.w3.org/TR/css-shadow-parts-1

that encapsulate functionality and appearance. The interest in having a native and standardised approach was not sufficient enough to develop it further. Especially when considering that the specification again depends on JavaScript as frameworks do.

Browser support is excellent by now, but that has taken some time and discussion to convince browser vendors to implement all required standards. It took until 2018 for a browser besides Chrome to adopt the shadow DOM. Browser support has always been a deal-breaker for companies since they may want to serve their content without excluding customers and polyfilling[6] shadow DOM is performance intensive.

Additionally, another technology that tackles the same problem means new ways of achieving the same thing again. Web Component developers had and still have to find solutions for problems which frameworks have already figured out. The framework maintainers provide common ways that are tested and approved. The lack of best practices and common solutions for Web Components are deterrent for newcomers. Modern frameworks already provide features like style encapsulation, code scoping or performance-boosting through server-side rendering and Web Components yet have to find ways to achieve these.

2.7 The Future of Web Components

The future heavily depends on the past. Something called the "jQuery crisis" had changed the web community's point of view. Back in the days, jQuery solved an elemental task: connecting the user interface with JavaScript values. Nothing is as essential to any interactive application as reflecting data on the screen. In the past, JavaScript offered a very clumsy API to address this, which is one of the reasons for the incredible success of jQuery[7]. The other killer feature was that jQuery standardised web APIs in a convenient way, which created a common ground for developers. But it seems that in 2013 and 2014 (as seen in Figure 2.2), the web community recognized that the jQuery way that had seen universal adoption had become obsolete due to improvements of the JavaScript API and that it was time to move on to a new model. The single-page application (SPA) revolution happened, which showed that the jQuery approach to manipulate user interfaces can be replaced with the Angular/React/Vue component model. If everyone agrees on the same framework, things can be great for a while. Even then, two or three years down the line, a framework can become dated. Using outdated technology can create a feeling of being blocked or hindered to be innovative, especially to developers who want to keep their skills up-to-date with the rest of the web community. At this point, an organization is faced with the choice of refactoring their entire technology stack to use a new framework or keeping the old one and facing the perception of not being innovative.

The web community moves forward pretty fast as new tools, libraries and frameworks emerge on a daily basis, which created a wave of "framework fatigue" as Farrell titled it [2]. The massive outpouring of new frameworks, techniques, and ideas makes it hardly possible to keep up with all the information and choices available to start a web project. The only alternative is to go completely native, which reduces available options to a

[6] A Polyfill is code that adds a feature for web browsers that do not support it natively.

[7] http://libscore.com/#libs Libscore shows that jQuery is still the most-used tool in production.

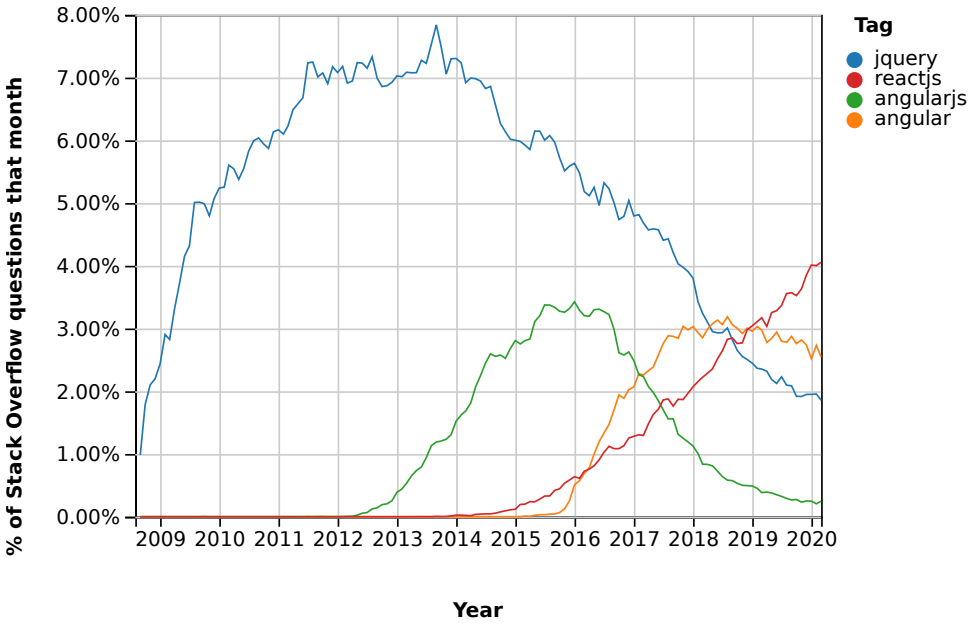


Figure 2.2: The graph shows how technologies have trended on Stack Overflow over time based on use of their tags since 2008. Image source [17].

sole one. Nowadays, it is again justified to create web sites with vanilla HTML, CSS and JavaScript. Having a framework can be an overkill for small websites or applications. Using native APIs for such projects is more convenient than ever. One of the most prominent benefits of not betting on a framework is being able to focus on core web development concepts rather than learning framework-specific skills that may or may not carry over to the next popular framework. Angular, React, or Vue are therefore working on a solution to compile their framework dependent components to native, standalone Web Components. This idea was further explained by Strazzullo in [7]:

> A very interesting side effect of the emergence of web components is the birth of a bunch of tools that are called disappearing frameworks (or invisible frameworks). The basic idea is to write code like with any other UI framework, like React. When you create the production bundle, the output will be standard web components. In other words, during compile time, the framework will simply dissolve.

Such disappearing frameworks would pave the way for an open web platform where developers can choose their favoured framework or none at all. It would be possible to create React components and use them in Angular and vice versa. The artificial walls of frameworks may come down in the future as Web Components bridge the gap between frameworks and enable independent universal components generated by disappearing frameworks like Stencil[8] or Svelte[9].

[8] https://stenciljs.com
[9] https://svelte.dev

2.8 Pre-Rendering Web Components

Pre-rendering is a process to create all elements contained in a web page on the server, rendering the first view before serving the page. Additional optimizations can be applied, like pre-fetching valuable resources, to help further boost the performance as well. Server-side rendering a webpage ensures that users have a visual response as soon as possible instead of a blank page, even if some parts might not be interactable from the beginning since assets still need to load. Although, deferring loading of crucial resources can lead to confusion for users when custom UI elements are not interactive yet. A standard solution is to use a loading indicator to bridge the time until the website is fully ready. This is often done by single-page applications that serve an empty HTML shell which gets populated via JavaScript on the client-side.

The trend of pre-rendering emerged since people realized that most JavaScript-powered websites just consist of an empty body tag, hiding away the initial content unnecessarily. To improve the competitiveness of SPAs, developers recognized that server-side rendering the application before sending it to the client would be beneficial. The most important advantages are the positive influences on performance, user experience and search engine optimization (SEO). Serving content upfront improves the perceived performance and reduces the workload on the client, which enables weaker devices to keep up with load times. Server-side rendered pages finish loading faster because the content is already available to the browser on the first request. This also increases user experience because the user is able to interact with the page faster and does not need to wait for it to build up. Additionally, content needs to be accessible for SEO purposes, but having an empty body tag hides all information until the app has been initialized. Having the content available in plain HTML makes it crawlable and indexable for search engines and bots. Although most search engines already execute JavaScript during crawling, not all support shadow DOM yet.

Due to universal JavaScript, which makes it possible to use client-side code on a Node.js server, almost all modern JavaScript frameworks provide a way to server-side render an application by running the framework code on the server to produce an initial set of native HTML that can be sent to the browser. Unfortunately, there is no standardised server-side rendering technique for Web Components established yet. The fundamental limitations regarding this are the dependency on browser-specific DOM APIs that are not available natively on the server, and the lack of a declarative way to represent the shadow DOM. Some possible approaches to overcome those limitations are described and evaluated in Section 3.5.

Chapter 3

State Of The Art

This chapter lists current and past proposals regarding the topic. A declarative custom elements proposal is ongoing, which can potentially take care of a declarative shadow DOM as well. The core discussion encountered on the internet is summarized, and the main points are stated in the following sections. Then the declined and the brand new proposal are introduced, and the differences to this solution are pointed out. Finally, the server-side rendering of Web Components and its connection to a declarative shadow DOM are highlighted.

The idea of a declarative shadow DOM has been around for a while. Glazkov already suggested to introduce a way for a declarative shadow DOM back in 2015[1]. The main goal was for performance reasons to be able to serialise and cache a composed tree, but it was never formalized into a concrete proposal. Several others, like the Web Hypertext Application Technology Working Group[2] (WHATWG) and the Web Platform Incubator Community Group[3] (WICG), also discussed the idea of a declarative shadow DOM. They cross-referenced and exchanged information from each debate, but the proposal was neglected at the Tokyo Web Components F2F in 2018[4], where it was resolved not to proceed due to implementation complexity and lack of developer need. As of writing this thesis, the WHATWG has recently rebooted the idea[5] and already began implementing and testing the new proposal for a declarative shadow DOM in Chromium. This affirms the currentness as well as the public interest in this thesis' topic.

3.1 Declarative Custom Elements

The declarative custom elements proposal [16] is one approach for a HTML declarative syntax to create custom elements. The suggested syntax introduces a new `<definition>` element that accepts the name of a custom element as well as a constructor class. A `<template>` element is used to store the HTML markup, and a class has to be provided via a `<script>` element. The proposal suggests the definition of a declarative custom

[1] https://www.w3.org/Bugs/Public/show_bug.cgi?id=28441
[2] https://github.com/whatwg/dom/issues/510
[3] https://discourse.wicg.io/t/declarative-shadow-dom/1904
[4] https://github.com/whatwg/dom/issues/510#issuecomment-370980398
[5] https://github.com/whatwg/dom/issues/831

element like the following:

```
<definition name="my-element" constructor="MyElement">
    <template shadowmode="closed">...</template>
    <script>
    class MyElement extends HTMLElement { ... }
    </script>
</definition>
```

The proposal also covers the automatic attachment of a shadow root with and without providing a custom element class. The downside of this approach is that the author of a custom element usually wants to add behaviour to it, which requires scripting anyways. However, numerous use cases cannot support or do not need any scripting and custom elements at all. Defining and instantiating a custom element every time an author wants to use a shadow DOM for an element, without any need for custom behaviours or scripting, is hard to justify. When the only desired goal is to use shadow DOM features, HTML encapsulation in HTML, or supporting non-JS environments, waiting for the declarative custom elements proposal is not worth it. Right now, there is simply no way to achieve the mentioned goals, and declarative custom elements do not change that either.

3.2 Discussion

The fundamental discussion between the community, browser vendors, and specification developers at issue #510 “Declarative Shadow DOM” for the GitHub project of the DOM specification[6] which has been going on for years now initiated the starting idea of this thesis. Several arguments for and against the implementation of a declarative shadow DOM as well as adding it to the specification were exchanged and gathered as comments spread across public mailing lists and various other GitHub issues.

It is currently possible to achieve a kind of declarative shadow DOM result using JavaScript. A short script can be appended to the desired node, which can take care of the shadow attachment, as demonstrated in the following code snippet:

```
<shadow-host>
</shadow-host>
<script>
    const shadowRoot = document.currentScript.previousElementSibling.attachShadow({
    mode: "open" });
    shadowRoot.innerHTML = `<child-element></child-element>`;
</script>
```

Although this does not work well with tooling, neither is it easily maintainable nor enhancing the usefulness of a declarative syntax. A `<script>` element containing the shadow DOM attachment logic does add additional complexity in terms of the correct handling of its string representation. This also reduces the benefits of server-side rendering, since the browser has to do the processing work on the client again. The core problem with a client-side workaround is that it needs JavaScript, which is the primary purpose a declarative HTML solution is trying to defeat.

[6] https://github.com/whatwg/dom/issues/510

A native solution can enhance frameworks that have declarative templating syntax, like GlimmerJS[7], to enable developers to define a shadow DOM declaratively within the template which allows for creating structured and more modular HTML documents. Additionally, authors can utilize the `<shadowroot>` element to introduce a new style scope, similar to how it is possible in JavaScript to create a new block in order to introduce a new scope for variables. If the only desire is to create a new style scope for CSS isolation, using the shadow root element annihilates the requirement of JavaScript. For example, a need for encapsulated styles can be satisfied by simply wrapping the defined styles with the `<shadowroot>` element, as demonstrated in the following:

```
<div>
    <shadowroot>
        <styles>
            encapsulated style rules
        </styles>
        <div>styled content</div>
    </shadowroot>
</div>
```

It adds convenience as far as writing HTML is concerned, which is a much different use case compared to using the shadow DOM to define component internals. The scoping mechanism also opens up the capability to create styled and modular DOM fragments in a single HTML file without the need for JavaScript, which currently is not possible.

Furthermore, having shadow DOM as an exclusive JavaScript API means that some features like CSS encapsulation are unavailable if JavaScript is disabled. Therefore, having CSS isolation in HTML without the necessity of JavaScript is a solid argument. CSS scoping is programmatically encapsulated, unlike just following conventions, for example, the BEM schema. Native CSS scopes enable authors to write a CSS class without having to follow naming conventions to avoid a collision, which also removes the need for complex CSS selectors to achieve the demanded specificity.

Moreover, a native shadow root element makes it possible to write a server-side renderer that renders an output of a computed shadow root and insert it as a child of an element with a clearly defined shadow boundary identifier. Said identifier can then inform the browser how it should render the particular element at that moment, including where a shadow DOM needs to be attached. This feature also improves load times when the server has done all the templating work already during prerendering to make the render happen faster on the client-side. Tools that read the DOM and its contents, like SEO crawlers or bots, would have to update to know about declarative shadow roots, but that is certainly easier than supporting JavaScript in the first place.

Contrarily, since the main argument is style encapsulation, having a preprocessor attaching an additional class automatically to create a new scope can achieve the same result. Although it is not entirely foolproof because it can not be guaranteed that someone also reuses that specifically added class or some other colliding class name by themselves, taking big projects and their potentially vast amount of maintainers into consideration. It also adds effort to write collision-free class names.

Considering existing frameworks like React, the shadow DOM primarily adds overhead and unnecessary complexity, when the only desired feature out of a shadow DOM

[7] https://glimmerjs.com

is scoped CSS which is already baked into the tool.

Nonetheless, a core doubt on adding this behaviour to browsers is the implementation cost. Adding a new `<shadowroot>` element to the standard would require non-trivial parser work and thus adds complexity. On the same order of magnitude as when the `<template>` element was implemented, and browser vendors still are recovering from that.

3.3 Declarative Shadow DOM Proposal (Version 1)

The declarative Shadow DOM proposal [25] by Wytrębowicz illustrates one approach for a declarative syntax of a shadow root as well as the expected behaviour of such. He aggregates the thoughts of the core discussions[8] but does not define a concrete implementation of a shadow root element nor an application for it. It was discontinued due to browser vendors argueing that the implementation complexity is too high and developer benefits are lacking. It served as a starting point, leading to the own approach of this thesis. The proposal suggests that a `<shadowroot>` element should be used to create a shadow root declaratively, as seen in the following snippet:

```
<host-element>
    <shadowroot mode="open">
        <h2>Shadow Content</h2>
        <slot></slot>
    </shadowroot>
    <h2>Light content</h2>
</host-element>
```

3.4 Declarative Shadow DOM Proposal (Version 2)

As of writing this thesis, the proposal mentioned before in Section 3.3 was recently revived by Freed [14]. However, he adopted the idea which led him to another solution using the existing `<template>` element with new attributes. Although, he hinted that it would be more ergonomic to use a new defined `<shadowroot>` element rather than overloading the template tag with attributes. Freed still chose to implement the `<template>` element solution because of the following reasons:

1. Compatibility issues could come up until a new `<shadowroot>` tag is supported by all rendering engines.
2. Even though a new `<shadowroot>` element would have similar parser semantics as the `<template>` element, it would generate a non-trivial effort to implement.

Furthermore, he pointed out that a DOM tree that contains a shadow root should also be serializable using `element.innerHTML`. Nonetheless, the current behaviour of `innerHTML` is to exclude shadow roots or any of their contents. For this reason he suggested adding a new method to `Element` called `getInnerHTML()` to avoid compatibility issues. When called with the option `includeShadowRoots:true` on nodes that contain shadow hosts, the returned HTML will include a `<template shadowroot>` tag containing the shadow root contents for each node.

[8] https://github.com/whatwg/dom/issues/510

His proposal to create a shadow root declaratively suggests to utilize the `<template>` element with an attribute named `shadowroot` like:

```
<host-element>
    <template shadowroot="open">
        <style>shadow styles</style>
        <h2>Shadow Content</h2>
        <slot></slot>
    </template>
    <h2>Light content</h2>
</host-element>
```

3.5 Server-Side Rendering of Web Components

The key intention of server-side rendering, that functionality should be usable without JavaScript, might be obsolete these days. As of today, [18] states that about 95% of all websites use JavaScript, which makes the fallback for non-JavaScript websites less critical than in the early ages of the internet. Nevertheless, it is a desirable goal that Web Components work to a certain extent without JavaScript. Although, the definition of "work" here should be narrowed down to the content representation only, which is concealed when a Web Component is used, while JavaScript is disabled or any part of the standard is unsupported.

Another aspect is to minimize the time until the First Contentful Paint (FCP)[9] happens in the browser. This means reducing the workload a browser has to do upfront, which includes interpreting and displaying a component. The time needed to show a custom component will always be longer compared to a vanilla HTML element, due to their nature and how they are being parsed and rendered by the browser. Therefore, the aim is to provide a vanilla HTML alternative until the actual component is ready for display and upgrade it afterwards.

As mentioned in Section 2.8, Web Components are certainly at a disadvantage when it comes to SSR compared to framework components. The tricky part is that Web Components encapsulate their markup inside the shadow DOM and can adapt it depending on light DOM and attached attributes. Hence, it is only possible to capture a snapshot of a component's current state during prerendering if the server has access to its shadow DOM. Thanks to universal JavaScript, JavaScript code from the client can be executed in the same way on the server. Unfortunately, Node.js, which is a JavaScript runtime, does not have access to the necessary APIs—shadow DOM, custom elements, etc.—out of the box. There are only a few known ways yet that make them accessible. First, a tool like a headless browser (a browser without a GUI) can be utilized, or secondly, the global Node.js scope can be patched to make the shadow DOM API available and then use the capability of universal JavaScript.

3.5.1 Headless Browser

A headless browser allows running a browser engine on a server to render and access a web page programmatically. Common use cases are, for example, crawling web pages,

[9] The First Contentful Paint is when the first bit of DOM content is rendered on the page. This metric states when the user receives consumable information.

extracting metadata (e.g., the DOM), or generating images from page contents. Some popular headless browsers are

- Pupeteer[10],
- PhantomJS[11].

The idea behind this approach is to render a page that contains the Web Component, utilize the DOM API to access its shadow root and extract the current DOM structure. This adds some overhead, like launching a browser or creating a new page on every render, which might not justify for sever-side rendering dynamically where render time is crucial. It might be sufficient for static server-side rendering like prerendering static components that do not change often.

3.5.2 Universal JavaScript

Universal or isomorphic JavaScript is code that runs both on the client and the server. JavaScript implementations of various web standards exist for use with Node.js. Those provide so-called shims that patch the global scope of Node, for example, to make it possible to interpret HTML or access the DOM API like in a regular browser. The performance advantage over headless browsers is given because it only adds relevant features, and a full-fledged browser does not take care of that. Some implementations available are

- JSDOM[12],
- Undom[13], or
- Domino[14].

As of writing this thesis, only JSDOM supports the necessary feature, shadow DOM, to server-side render Web Components.

3.5.3 Prerendering

To effectively prerender Web Components, the actual HTML representation has to be extracted, and the reconstruction on the client-side has to be possible. Those steps are called *serialisation* and *rehydration*. The serialisation step reduces the component's DOM tree to a single string of HTML elements that can be served to the client. The rehydration step reconstructs the actual Web Component from the serialised string.

Shadow DOM Serialisation

Serialisation means to extract the DOM tree, including the shadow DOM, and convert it to a composed HTML string equivalent without losing information of the original structure. Since there is no standardised way of doing this, Justin Fagnani and Trey Shugart discussed an approach in [13]. The key problem that lacks standardisation is the representation of a shadow root in a string. Fagnani came up with a new custom

[10] https://github.com/puppeteer/puppeteer
[11] https://phantomjs.org
[12] https://github.com/jsdom/jsdom
[13] https://github.com/developit/undom
[14] https://github.com/fgnass/domino

element solution—<shadow-root>–which also handles the rehydration automatically when stamped and the suggested application looks like:

```
<my-element>
    Light DOM
    <shadow-root>
        Shadow DOM
    </shadow-root>
<my-element>
```

Having the <shadow-root> element as a native, browser-supported element will make the whole serialisation and rehydration process of Web Components redundant since it can be converted to a declarative Web Component. Although the approach shown by Fagnani is a proof-of-concept and leaves room for improvement, the idea behind it is the possibility to create a shadow root without JavaScript, declaratively using plain HTML. Spinning this idea further, declarative shadow DOM combined with declarative custom elements can create a new way for declarative Web Components, which the HTML standard is currently lacking.

Rehydration

Rehydration ensures that the original state is reapplied and the Web Component can take over with the correct data. This includes the initial attributes and the correct order of DOM elements in the light DOM. The serialisation step has to consider how to store the initial state which can be done using a <template> element that acts as an invisible storage.

Example Algorithm

To demonstrate a possible prerendering scenario, a made-up Web Component is used as a starting point. The HTML markup and the usage of the component are defined in the following snippet:

```
<!--HTML content of the component-->
<div>
    <h1>Hello</h1>
    <slot/>
</div>

<!--Usage of the component-->
<web-component>
    <p>World</p>
</web-component>
```

A basic prerendering implementation that produces a sufficient outcome can generate a representation of the demonstrated Web Component like shown in Program 3.1. With a headless browser, it is possible to let a component set up its markup and after it has finished doing so, the shadow DOM tree can be extracted via querying the contained nodes programmatically, which requires the shadow mode to be set to open. The headless browser ensures that the component has already processed slotted elements from the light DOM (the <p> element in this example will be placed instead of the <slot/> element) and applied optional conditional changes. Extracting said construct results in

Program 3.1: Possible outcome of prerendering the example Web Component. Contains the composed shadow DOM with the light DOM. The original light DOM is stored inside a `<template>` element.

```
1 <web-component>
2     <div>
3         <h1>Hello</h1>
4         <p>World</p>
5     </div>
6     <template>
7        <p>World</p>
8     </template>
9 </web-component>
```

a merged DOM. To re-create the original structure from it, the initial light DOM is stored inside a `<template>` element which acts as an invisible placeholder.

This serialised representative is shown until the Web Component is ready to take over in the browser. Afterwards, a rehydration script completely removes the merged light DOM of the Web Component (lines 2 to 5 in Program 3.1). Then set attributes are reapplied. Finally, the current light DOM of the component is overwritten with the content of the `<template>` element. These steps ensure that the registered Web Component can take over with the exact state during prerendering.

This solution is one custom-made way to serialise a component and rehydrate it on the client. There are many different workarounds to expose the content of a Web Component, but the ultimate goal would be to be able to represent a shadow root completely declaratively, natively with an HTML element in a string. This would eventually lead to a more convenient prerendering approach, since the shadow DOM does not need to be serialised and rehydrated manually any longer.

Chapter 4

Own Approach

In a first step of identifying possible requirements for a declarative shadow root solution, different discussions of browser vendors, specification developers and the community were analysed to find common ground, gather thoughts and determine a possible approach. In this chapter, the reader is introduced to the considered factors. The goals and an example application are presented and discussed. After defining the requirements and behaviour, the general solution and usage of the declarative shadow root will be addressed.

While most relevant research has concentrated on the why, not the how, no system known so far has combined the technical requirements with a practical application, except for Freed, as mentioned in Section 3.4 who uses a different strategy, that sets the present approach apart.

4.1 Goal

The essential goal of the declarative shadow root is to be able to completely describe a DOM tree containing a shadow DOM using only HTML without losing information of the original structure. It is worth mentioning that the purpose is not to replace the current way of creating Web Components but to add a new, declarative way to create a shadow DOM. Due to reasons mentioned in Section 3.5, the current imperative-exclusive shadow DOM API is not compatible with server-side rendering. A declarative shadow DOM is a requirement to support shadow DOM in server-side rendering environments. Therefore, the idea of developing a declarative shadow DOM solution was born.

Another related characteristic for server-side rendering solutions is that they allow for isomorphic or universal code, as specified in Section 3.5.2. The code building the content should run on the server as well as on the client, producing the same result. That expects a DOM tree to be serialised on the server and deserialised on the client back into the same tree, including shadow root nodes. This is currently not possible without some custom scripting effort.

A shadow root is a special kind of node. It is no actual DOM node yet, neither is it part of the DOM representation. Although for a more natural orientation for developers, developer tools inside browsers do show a shadow root indicator named `#shadow-root` (which is also used by the upcoming code snippets), as shown in the following example:

```
<div class="profile-card">
    #shadow-root (open)
        <link rel="stylesheet" type="text/css" href="card.css">
        <div class="header">...</div>
        <div class="container">...</div>
    <div slot="header">John D. Eveloper</div>
</div>
```

This shadow root indication is not serialisable yet and therefore is excluded in accessors that return the HTML contents, like `innerHTML`. The actual DOM structure of the profile card example in the document is:

```
<div class="profile-card">
    <div slot="header">John D. Eveloper</div>
</div>
```

Apparently, it is not possible to copy a DOM tree (via accessing the `innerHTML` or using the developer tools) containing a shadow root, and then reuse that tree in a different place with the same result. This functionality needs some sort of new method, which also includes the shadow root's serialised HTML representation as a declarative shadow root, as mentioned by Freed in Section 3.4.

In addition to the aspects stated above, another commonly-cited motivation, as mentioned in Section 3.2, is a CSS scoping mechanism. One of the central features of shadow DOM is style encapsulation. CSS developers are typically more comfortable with HTML and CSS than they are with JavaScript and most likely prefer to be able to achieve their styling objectives without resorting to it. Some design systems even restrict or ban the use of JavaScript to style elements. Providing CSS developers with a practical and declarative way to utilise shadow DOM without requiring any JavaScript would allow them to benefit from the style scoping feature as well. It would make scenarios possible, where developers are not able to easily add new styles into an existing application without interfering with existing ones. Using a declarative shadow DOM voids these concerns due to the creation of a new CSS scope. To demonstrate such a scenario, a concrete webpage is presented in the next section where possible components are identified, and the advantages of a declarative shadow DOM are outlined.

4.2 Application

First of all, in the planning phase of such a website, common sections that might be reused on other web pages should be identified and declared as components. Sections that are not reused are identified as well. In the example seen in Figure 4.1, a header or a footer component (Figure 4.1(a)) can be used on several pages of the website, presumably with the same look and behaviour and thus, are identified as encapsulated components. For that purpose, implementing those components as a Web Component fits best due to reusability. For lesser common sections, it may not be necessary to create a Web Component. For those sections, the declarative shadow DOM can be utilised to benefit from the encapsulation features as well. The profile section is most likely only displayed on the profile page and therefore will not be reused. Nonetheless, it makes sense to group said section together inside a shadow DOM in order to make it easier to maintain and independent of the rest. The section (Figure 4.1(b)) contains

Figure 4.1: Prototype of the general user interface of a profile webpage. The page consists of sections that are implemented as Web Components (outlined with a red dashed border) and sections that are using the declarative shadow root (outlined with a green dashed border). The following sections are defined: (a) a header and footer section, (b) a profile section which contains a profile card section as well as a profile statistics section, (c) a comment section and (d) an ad section. All sections are encapsulated with their own shadow DOM.

a profile card, which includes some user-specific data, and the profile statistics, which show several numbers about the user's activity. Both parts are encapsulated using the declarative shadow root. A comment section (Figure 4.1(c)), which shows the most recent comments of the user, is also realised as a Web Component since it might be reused on other pages.

Advertisements as ad banners are very common nowadays, such a section (Figure 4.1(d)) might have a single use case or multiple ads require a completely different

markup, and hence they do not benefit from being a Web Component as much. Nonetheless, ads can be encapsulated with a declarative shadow DOM to prevent errors in the third-party integration to affect the website and vice-versa. Although, such integrations can be implemented with an iframe, not all are, which can affect the outcome of the website negatively. This will most likely not replace the iframe, however, brings up more use cases where encapsulation is needed, but the overhead and restrictive access that comes with iframes is gratuitous compared to simply wrapping the desired section into a declarative shadow DOM.

4.3 Requirements for a Declarative Shadow Root

In this section, the obtained requirements are stated and explained. The goal of this thesis is to be as close as possible to a solution which would be acceptable for browser vendors and specification developers. Some were mentioned or defined in the discussion groups and thus are reflected here. While many factors could be taken into consideration, the two main aspects are:

- Choice of Identifier: How to identify the shadow host?
- Behaviour: How should it behave?

The goal is to find and formalise the advantages and main issues with each of the possibilities and then define requirements for the shadow root mechanism introduced in the next chapter.

4.3.1 Choice of Identifier

Since a declarative shadow root does not exist yet, there are mainly three options to instantiate a shadow root declaratively and tell the parser how to identify the shadow boundary. The first option is to add an attribute to supported HTML elements, which informs the parser to add a shadow root to it. The second option is to overload an existing element with new behaviour, and the third is to introduce a new HTML element to the web standard.

Global Attribute

The first possibility is to introduce a new global attribute (attributes common to all HTML elements) for elements that support attaching a shadow root. Applying such an attribute informs the parser to pull apart this element's DOM into shadow and light DOM. As an example, a `<div>` element can have an attribute called "shadowroot" applied like:

```
<div shadowroot>
    Hello
    <slot>
        <p>World</p>
    </slot>
</div>
```

The parser detects the attribute and knows to handle this element differently. Any child nodes are put into the shadow DOM except children of slots. Those get placed into

the light DOM. After parsing the DOM, the attribute is removed. The outcome of the shown example above is the following:

```
<div>
    #shadow-root
        Hello <slot></slot>
    <p>World</p>
</div>
```

Although, this would not be the declarative equivalent of the imperative shadow DOM API, but instead a way to tell the parser to create the shadow and light trees of an element in a declarative manner. Therefore, this approach is sufficient for generators or renderers that want to serialise a shadow DOM but does not fully meet the need for a declarative shadow root solution.

Another possible approach is that such an attribute informs the parser to attach a shadow root to the encountered element and move existing content into it. However, the method brings up more concerns about the possible outcome when manipulating that attribute, which may lead to authors' uncertainty. Since any element would be able to behave differently depending on a single attribute, this seems to be a non-starter for specification developers.

Repurpose an existing Element

A different approach that requires an additional element is to repurpose an existing one. Reusing the `<template>` element is the closest choice here because of its inertness. The parser processes the contents of a template while loading a page. However, it does so only to ensure that the contents are valid. The contained elements do not render, and the included resources are not fetched. Therefore, it is a viable choice to define HTML markup inside a `<template>` element and inform the parser through a shadow boundary identifier, like a new attribute, to put the contents of it into a shadow DOM, which is then attached to the template's parent node.

Furthermore, the JavaScript shadow root API (`Element.attachShadow()`) requires a `ShadowRootInit` parameter. This parameter is a dictionary containing two fields so far. Therefore, the declarative identifier has to have the possibility to add parameters as well as being extensible if more emerge in the future. The first field of the dictionary is the mode, which is a string that specifies the encapsulation mode for the shadow DOM tree. It is required and can either be `open` or `closed`. This decides whether or not the shadow root is accessible from the outside via JavaScript. The second field is the `delegatesFocus` option, which is a boolean variable that defines behaviour that mitigates custom element focusability issues.

Since the attachment requires settings like the shadow mode, there has to be a way to add such options. The most straight forward approach is to add those settings via an attribute-value combination. Because of conventions, built-in attribute names do not use hyphens and are lower case. Hence, a valid attribute name to identify the declarative functionality is "shadowmode" or short "shadow" instead of "shadow-mode" and the attribute "delegatesfocus" is also possible. The following can be an option to attach an open shadow root declaratively:

```
<div>
    <template shadow="open">
```

```
            shadow content
            <img src='...'>
        </template>
    </div>
```

After parsing the resulting DOM looks as follows:

```
    <div>
        #shadow-root (open)
            shadow content
            <img src='...'>
    </div>
```

Although overloading an existing element with new behaviour means less changes in the parser logic than creating an entirely new element, it can confuse authors because of the different semantics and resulting behaviour depending on some attributes. Changing the semantic of an existing element is never a good idea. The intended use of the `<template>` element is that it is inert as long as it is not inserted into the document. However, descendant nodes of a declarative shadow DOM are effectively not inert from the user's perspective. Resources like images appear inside of the template's markup, but they will remain in the shadow tree as the final result, after parsing is finished. The necessary resources have to be fetched, but putting those inside of a `<template>` means that they are not participating in the document and therefore crawlers may not recognise them, thus negating the benefits of SSR altogether. The needed behaviour differs significantly from the usual `<template>` element's usage pattern. Furthermore, a single attribute as a visual indication of such a significant different semantics is too weak and can make it harder for developers to expect certain behaviour.

Creating a new Element

Creating a new element for the declarative shadow root behaviour is probably the best choice ergonomic-wise. Nonetheless, the web platform has to consider conflicts when introducing new features, especially when it comes to introducing new elements, as the goal is to maintain compatibility with existing content[1]. Therefore, a query against the httparchive[2] was executed by Hayato Ito, which had the result that no element with the specific name—shadowroot—was in use[3]. Therefore, it is safe to introduce a new special element named `<shadowroot>` that, if found in an HTML markup, creates a shadow tree and attaches it to its immediate parent. Any elements within it end up in that shadow tree.

The advantage of an explicit element as an shadow boundary identifier is the possibility to apply parameters via additional attributes in a clear manner. Therefore, required options like the shadow mode or optional settings like `delegatesfocus` can be set by adding them as attributes.

```
    <div>
        <shadowroot mode="open|closed" delegatesfocus>
            shadow content
            <img src='...'>
```

[1] https://www.w3.org/TR/html-design-principles/#support-existing-content
[2] https://httparchive.org
[3] https://github.com/whatwg/dom/issues/510#issuecomment-330486945

```
      </shadowroot>
  </div>
```

4.3.2 Behaviour

Besides the syntax, the expected behaviour of a declarative shadow root has to be defined.

1. The shadow boundary identifier, be it an attribute or an element, should inform the parser automatically to attach a shadow root to the hosting element. No further interactions are required via JavaScript, compared to a template's content insertion. This guarantees the interoperability with server-side rendered shadow roots and boosts the instantiation performance when being parsed natively by the browser instead of JavaScript.
2. Siblings of the shadow root element are not rendered unless it contains a `<slot>` element. This behaviour occurs because of how the shadow DOM works. The flattened tree does not consider light DOM content without a `<slot>` element in the shadow tree. That is something authors have to consider when using the shadow root element.
3. What happens to the shadow boundary identifier after parsing is not defined concretely. One possibility is that it remains in the DOM like the `<template>` element currently does. The identifier is ignored after parsing and hence, can stay in the DOM and is treated like a dead element without interference.

   Although keeping unused or dead elements in the DOM is unnecessary and can potentially increase the memory usage. Therefore, the identifier should disappear after being parsed. Removing it from the DOM tree reduces the confusion of authors through unnecessary, dead elements or attributes, avoids dumping the DOM and lessens memory consumption.

   Moreover, it makes feature detection easier because unsupported user agents still contain the identifiers, and a polyfill can take that into consideration. If the identifier is removed from the light tree, a new element, `<shadowroot>` for example, fits the purpose better than repurposing the `<template>` because a template element's behaviour currently is to remain in the DOM tree.
4. The declarative shadow root should be a parser-only feature, and adding the identifying attribute to an existing `<template>` element, for example, has no effect. A user can still imperatively create a `<template>` element, add the shadow mode attribute to it, and place it into the DOM, but the user is unable to observe its brief existence. The parser recognises the `<template shadow="open">` and attaches a shadow root and removes the element as it is parsed. The attachment can either happen when the opening tag or the closing tag is encountered. Which version fits better was already elaborated by Freed [14]. As a convention, declarative shadow roots should not be created via JavaScript using the `document.createElement()` API.
5. Since the `attachShadow()` method requires the mode parameter, the declarative shadow root solution should be coherent and require the mode as well. Hence, it can not default to a shadow root mode, like `open`, but has to be able to define

parameter values. If the value of the mode can not be validated, the parser should not treat it as a request for a shadow root insertion and instead leave the content inert.

6. The general functionality of markup elements is the ability to nest and compose bigger structures. Thus, the declarative solution should be nestable as well and structures like the following have to be possible:

```
<custom-a>
    <shadowroot>
        <slot></slot>
        content
    </shadowroot>
    <div>
        <shadowroot>
            <custom-b>
                <shadowroot>
                    <slot></slot>
                    content
                </shadowroot>
                content
            </custom-c>
        </shadowroot>
    </div>
</custom-b>
```

Although, developers should keep in mind that nesting `<shadowroot>` elements in each other will throw an error.

4.3.3 Specification

During development, tests[4] were created to cover most expected behaviour and to demonstrate its functionality. These tests can be seen as the specification or a detailed description of the desired behaviour additionally to the mentioned aspects in Section 4.3.2. The tests include more real-world use cases and rare scenarios, which have to be covered nonetheless, and are defined as the following:

1. Once parsed, it should attach a shadow root to the host, and append its content.
2. Combined declarative shadow and light DOM should be parsed and processed as such.
3. A parsed `<shadowroot>` element and its children must not appear in accessor properties like `host.childNodes`, `host.innerHTML` or `host.children`.
4. Imperative access to a declarative attached shadow root should use already settled APIs.
5. Imperatively created `<shadowroot>` elements should not attach a shadow root and be inert.
6. It should have a `mode` attribute equal to `open` or `closed`, otherwise it is inert.
7. Scripts inside a `<shadowroot>` element are processed after stamping.
8. It should throw an error when calling `attachShadow` on an element that already has a shadow root attached by a declarative shadow root element.

[4] https://github.com/drdreo/declarative-shadow/blob/master/test/shadow-root.test.js

9. It should throw a `NotSupportedError` DOMException when the host element does not support shadow DOM.
10. Nesting two `<shadowroot>` nodes under the same host should result in the same behaviour when attaching a shadow to an element that already has one attached. Since multiple shadow roots are prohibited, an error is thrown. The best practice is to only nest a single declarative shadow root element under each host.
11. The `<shadowroot>` element can be used inside the content of a `<template>` element. It is also processed during parsing of such a template. Although, in general the `<template>` element is inert. Hence, it will not attach a shadow root until the template's content is connected to the document. Before that, it shows up in `template.childNodes` as a `HTMLShadowRootElement`.

```
<template>
    <shadowroot mode="open">
        Shadow content
    </shadowroot>
    <span>Light content</span>
</template>
```

Before this template is stamped, the template's children amount equals two (the `<shadowroot>` and the `<span>` node). The `<shadowroot>` element does not attach a shadow DOM, as there is no host yet. When the above template is stamped to the document, the `<shadowroot>` element is being processed and adds its contents to the parent element's shadow root when applicable.

4.4 Solution

First of all, a specific element nicely correlates to the way the developer tools currently show attached shadow roots. It is an intuitive and self-explanatory approach for developers who encounter a declarative shadow root. Technically and implementation-wise, overloading the existing `<template>` element or creating a new `<shadowroot>` element would be pretty similar. Although having a separate implementation would let them evolve in different directions if needed. An overloaded tag with various attributes that changes its behaviour and semantics is counter-intuitive in terms of expecting a certain functionality. As mentioned in the section before, the shadow root identifier is removed from the DOM tree after parsing. It makes sense that the identifier is a macro syntax for the parser, which hoists its children into a shadow root, and then the identifier itself disappears. Reusing the `<template>` element for this feature is unsuitable, due to the different semantics.

Optional parameters that may become more in the future have to be able to be applied as well. For this reason, using global attributes can become error-prone when developers want to use a declarative shadow root but have to add a combination of multiple attributes beforehand in order to do so. The visual distinction of a shadow root is also more difficult when applying attributes to ordinary elements instead of using an element as an explicit identifier that defines the functionality and its purpose clearly.

A `<shadowroot>` element does not only represent the equivalent of the imperative API; moreover, it is the best approach for developers concerning ergonomics, extensibil-

ity, and readability. Therefore, this thesis's approach focuses on a new HTML element combined with attributes to enable a declarative shadow root. The allowed attributes are:

- **mode** of type `ShadowRootMode`—the shadow root mode.
- **delegatesfocus** of type `boolean`—fixes the custom element issues around focusability.

The `HTMLShadowRootElement` is derived from the `HTMLElement` interface, but without implementing any additional methods. Using Web IDL, the interface is defined as follows:

```
interface HTMLShadowRootElement : HTMLElement {
    attribute ShadowRootMode    mode;
    attribute boolean           delegatesfocus;
};
enum ShadowRootMode { "open", "closed" };
```

To conclude, as an example, the proposed declarative `<shadowroot>` element can be used in the following way:

```
<div>
    <shadowroot mode="open" delegatesfocus>
        shadow content
    </shadowroot>
</div>
```

The resulting DOM is then rendered as:

```
<div>
    #shadow-root (open)
        shadow content
</div>
```

Chapter 5

Declarative Shadow Root

This chapter dives into the technical implementation details, gives an overview of the used technologies and explains why they were chosen over others. In summary, a declarative shadow DOM solution was developed and tested in the course of this project. The project is realised using custom elements. Choosing custom elements has many advantages, from modern web technologies' quick prototyping capabilities to the interoperability with a native solution when available. The project has profited from the increasing interest in Web Components and the resulting features over the past years.

A few words should also mention the distribution of this project. Without the thriving open-source community, some of the technologies and tools used by this project may not exist. For this reason, and to give back to the community, the solution and test environment developed in the course of this project have been published as open-source on GitHub[1] and are freely available for anyone to use and further investigate.

At first, the project scope is defined. After discussing the general setup and testing environment, the shadow root algorithm is investigated. At the end, an insight into a real-life usage is demonstrated.

5.1 Project Scope

Before jumping into technical details, the scope of the project should be discussed. As the aim of the present work is to explore ways of adding the declarative shadow DOM feature to the web platform, the goal of the project was to build and test the mechanisms described in the previous chapters. The resulting solution is mainly aimed towards developers and browser vendors, both to showcase the new feature as well as to test and explore the opportunities with such a declarative shadow DOM further.

A custom element solution is realised to avoid changing browser internals and the overhead it brings with it. Nevertheless, it demonstrates the same goals as a native implementation: behaviour, outcome, and usage. The approach employs a custom element that can represent a native HTML element the closest, except for the exact naming of the tag. Compared to a native `<shadowroot>` element implemented directly in the browser's code, the `<shadow-root>` element implementation will most likely lack performance, but this can be neglected because the core goals are nonetheless apparent

[1] https://github.com/drdreo/declarative-shadow

and evaluable.

In the following, the general architecture and technologies used in the project will be analysed and described to then discuss implementation details, problems and solutions for the main feature.

5.2 Testing Environment

The application basically consists of two parts: the `<shadow-root>` element as a custom element implementation and the testing environment. The environment's only requirements are on the one hand to serve static HTML files to validate the custom element application, and on the other hand, to provide tools for creating tests. These tests have to be able to instantiate HTML, including custom elements, and to support shadow DOM. All of which is covered by open-wc[2]. They provide an opinionated package[3] that combines and configures many of the recommended testing libraries (mocha[4], chai[5], sinon[6]) to minimize the amount of configuration required to set up tests.

The environment was set up via a CLI tool, also provided by the testing package, after following the *Testing Workflow for Web Components* [9]. With this kind of setup, various tests like code coverage or snapshot tests are possible, and even BrowserStack[7] is configured out of the box. The main advantage is that HTML can be mocked and tested easily using the library. Additionally, essential assertions, to compare light DOM and shadow DOM, are straightforward through accessor properties which made development way more comfortable (see Program 5.1).

Program 5.1: Using the test environment to create an HTML mock which processes custom elements and enables light and shadow DOM assertions in a convenient way.

```
const el = await fixture(html`
            <div>
                <shadow-root mode="open">
                    <h2>Shadow DOM</h2>
                </shadow-root>
                <div>Light DOM</div>
            </div>
            `);
expect(el).lightDom.to.equal(`<div>Light DOM</div>`);
expect(el).shadowDom.to.equal(`<h2>Shadow DOM</h2>`);
```

[2] https://open-wc.org
[3] https://npmjs.com/package/@open-wc/testing
[4] https://mochajs.org
[5] https://chaijs.com
[6] https://sinonjs.org
[7] https://browserstack.com

5.3 The Shadow Root Element

The project generally follows best-practices in web development[8] and utilises modern CSS3 and JavaScript features and libraries. The software is written in the latest *ECMAScript2020* (ES2020), makes use of the *node package manager*[9] (npm) for managing dependencies and *es-dev-server*[10] as a web server for development without bundling.

To dive right into the algorithm and attachment logic of a declarative shadow root element, the entire code responsible for that is discussed in the following section. Afterwards, the possible compatibility issues and polyfilling strategies are mentioned.

5.3.1 Algorithm

The whole algorithm is realised in a single file containing the custom element class `ShadowRoot`. The shadow root attachment can either happen inside the `constructor` of the class or the custom elements lifecycle callback `connectedCallback`[11]. Both are viable options, although, starting the algorithm in the constructor is a best practise due to [15]:

> The constructor is when you have exclusive knowledge of your element. It's a great time to setup implementation details that you don't want other elements messing around with. Doing this work in a later callback, like the connectedCallback, means you will need to guard against situations where your element is detached and then reattached to the document.

As seen in Program 5.2 line 5, the `attach()` method is called inside the constructor, which starts the shadow root attachment process. If doing any computational work inside the constructor, the `super()` call is mandatory because every custom element has to extend at least from the base `HTMLElement` class and therefore has to call the superclass constructor beforehand. If the constructor is omitted, the default constructor is used, which calls the parent constructor and passes along any arguments that are provided. This is important for polyfilling due to some of them changing the extended base class, which is covered in Section 5.4.3.

If there is no parent element, the attachment process is aborted, and the element stays inert. This can be checked by accessing the `parentElement` property (see line 10), which returns the DOM node's parent element or null. It equals to null if the parent is not a DOM node—e.g. for the `<html>`—or the custom element is registered too early, for example in the `<head>`, when the DOM tree is not parsed yet, which causes the properties to be not set. Additionally, creating the element imperatively by calling `document.createElement()` will abort the process, as intended by the requirements mentioned in Section 4.3.2, since it has no parent element.

To full-fill the requirement of a valid mode attribute value of *open* or *closed*, line 15 checks for such values. If neither is set to the exact string, the attachment process is

[8] https://developers.google.com/web/fundamentals/web-components/best-practices

[9] https://www.npmjs.com

[10] https://github.com/open-wc/open-wc/tree/master/packages/es-dev-server

[11] https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-api:concept-custom-element-definition-lifecycle-callbacks

cancelled, and the element stays inert. If the mode is valid, the parent's shadow root is checked for existence. If it has already a shadow root attached, an error is thrown in line 24 to inform the developer, as well as mirroring the behaviour of imperatively attaching a shadow root to an element that has one attached. This is due to nesting multiple declarative shadow root elements under the same parent container is prohibited, as specified in Section 4.3.3 Item 10. Although, if the parent's shadow root is still *null*, the shadow DOM is attached via the imperative `attachShadow()` API, which consumes the parameters `mode` and `delegatesFocus`. The `delegatesfocus` attribute is a boolean, hence it is just checked for existence on the `<shadow-root delegatesfocus>`.

Line 27 to 29 traverses all children and appends them to the shadow root. As a side note, `appendChild` moves a node to the end of the list of children of a specified parent node. The method basically removes the reference to the current parent and sets it to the new one. This is why the `firstChild` property always contains the correct reference to the first child node until there are no children left. To complete the process, the `<shadow-root>` element itself is removed in line 31.

5.3.2 Application

Since one of the declared goals was the creation of an open-source prototype that most closely reflects a possible native implementation, a few words should also be spent on the usage of the element by other developers. This section will, therefore, go into details of how to set up the desired website seen in Figure 4.1, use the new element and lastly the expected outcome. The code-examples of this section are simple and made up, yet show sufficient code to grasp the concept.

First of all, the shadow root class file needs to be included in the website. After that, the custom element needs to be registered at the `CustomElementRegistry`, which is the controller of custom elements on a web document. Both steps should be done at the end of the `<body>` element to ensure that the DOM is already parsed as intended. The element is named *shadow-root* and its class object is the `ShadowRoot` class. The registration step is shown in the following snippet:

```
customElements.define('shadow-root', ShadowRoot);
```

Developers can then freely utilise the element as seen in the profile page example Program 5.3, which demonstrates a way to use encapsulated styles with a declarative shadow DOM.

Using a link element inside a shadow DOM (Program 5.3 line 12) is one way to use structured CSS files combined with CSS encapsulation. This enables developers to decide if they want to write styles inline (line 5), outsource them to an external file or do both. Furthermore, all features of a shadow DOM are available for developers to use, which includes the slotting mechanism. Although, when writing HTML by hand, using `<slot>` elements inside a declarative shadow DOM, as demonstrated in line 14, would not make much sense, since the author can manually place the desired node there. The benefits of the feature present themselves when it comes to server-side rendering of a Web Component where it is serialized into a string, and the shadow DOM can then be represented with an explicit element. As mentioned in Section 3.5.3, the developed `<shadow-root>` element can be combined with a pre-rendering solution to deliver progressive enhancements for Web Components.

Program 5.2: The declarative shadow root algorithm. It is realized as a custom element to get as close to a native implementation as possible.

```
class ShadowRoot extends HTMLElement {

    constructor() {
        super();
        this.attach();
    }

    attach() {
        const parent = this.parentElement;
        if (!parent) {
            return;
        }

        const mode = this.getAttribute("mode");
        if (mode !== "open" && mode !== "closed") {
            return;
        }

        let shadowRoot = parent.shadowRoot;
        if (!shadowRoot) {
            const delegatesFocus = this.hasAttribute("delegatesFocus");
            shadowRoot = parent.attachShadow({mode, delegatesFocus});
        } else {
            throw new Error(`Shadow root already attached to <${parent.tagName}>`);
        }

        while (this.firstChild) {
            shadowRoot.appendChild(this.firstChild);
        }

        parent.removeChild(this);
    }
}
```

5.4 Compatibility and Polyfilling

In the following sections required enhancements and encountered compatibility issues are mentioned.

5.4.1 Declarative Shadow DOM Awareness

For custom elements or Web Components, developers have to consider to make them declarative shadow DOM aware. This means that the custom element may already have a shadow attached through a declarative shadow root during server-side rendering, and therefore the developer has to take care of it during initialization. This could not happen before, but due to a new way of attaching a shadow DOM, such a case has to be covered as well. A simple check if a shadow root already exists before attaching the shadow DOM can make a Web Component declarative shadow DOM aware:

Program 5.3: The profile page example HTML structure which interface was already shown in Figure 4.1. All styles contained are encapsulated through a shadow DOM, either by a Web Component or the declarative shadow root. Some parts of the HTML are left out to reduce unnecessary lines of code.

```
<div class="container">
    <my-header></my-header>
    <div class="profile">
        <shadow-root mode="open">
            <style>
                :host {
                    display: flex;
                }
            </style>
            <div class="profile__card">
                <shadow-root mode="open">
                    <link rel="stylesheet" type="text/css" href="card.css">
                    <div class="header">
                        <slot name="header"></slot>
                        <img src="..."/>
                    </div>
                    <div class="container">
                        <p>Web Developer</p>
                        <ul>
                            <li>HTML</li>
                            <li>CSS</li>
                            <li>Web Components</li>
                        </ul>
                    </div>
                </shadow-root>
                <div slot="header">John D. Eveloper</div>
            </div>
            <div class="profile__stats">...</div>
        </shadow-root>
    </div>
    <my-comments class="comments"></my-comments>
    <div class="ad">
        <shadow-root mode="open">...</shadow-root>
    </div>
    <my-footer></my-footer>
</div>
```

```
if(!this.shadowRoot){
    this.attachShadow({mode: 'open'});
}
```

It allows to progressively enhance server-side rendered Web Components. Considering the following element:

```
customElements.define('hello-world', class extends HTMLElement {
    constructor(){
        super();
        // Create new shadow root or overwrite existing (server-side rendered) one
        if(!this.shadowRoot){
```

```
            this.attachShadow({mode: 'open'});
        }
        this.shadowRoot.innerHTML = 'Hello <slot></slot>';
    }
});
```

Then after pre-rendering the component, which moves the shadow DOM content inside the `<shadow-root>` element, using it in the following way will render "Hello World" regardless whether the shadow DOM is attached via the declarative method, or by the custom element, which overwrites the shadow content after initializing. In both cases `helloWorld.children` contains only text nodes, and the outcome is the same.

```
<hello-world>
    <shadow-root mode="open">
        Hello <slot></slot>
    </shadow-root>
    World
</hello-world>
```

5.4.2 Compatibility Issues

User agents interpret an unsupported element with no valid custom element name as an `HTMLUnknownElement`, which basically is an element without any particular behaviour or styling. This also applies for browsers that do not support and implement the suggested native `<shadowroot>` element. Therefore, styles defined inside an unsupported `<shadowroot>` element, respectively an `HTMLUnknownElement` are not encapsulated as they are supposed to be and will cause unintended interference within a website, which also applies for the in this chapter implemented custom element. Polyfilling is required to prevent this from happening.

As soon as a native declarative shadow DOM is implemented in the browser, this shadow root class can be used to polyfill the missing functionality. Although, unknown elements can not have a shadow root attached and therefore, the polyfill has to consider a workaround for that.

5.4.3 Polyfilling

Declarative shadow DOM is not syntactical sugar or a wrapper element for an existing API, but a missing feature, hence any polyfill strategy comes with a cost. The most significant downside is that each solution would rely on JavaScript, which neglects the whole purpose of supporting non-JavaScript environments.

To detect if a polyfill is required, developers can check for the existing property in the element's prototype or verify if the `<shadow-root>` element gets removed after parsing. One possibility for a polyfill strategy is to wrap the declarative shadow root within a `<template>` element, so it stays inert when the page is loading, and no unintended styles are able to leak out, considering the `<template>` element is supported in the client browser. The content can then be moved to a new Web Component to guarantee style encapsulation.

Also worth to mention is that existing custom element and Web Component poly-

fills[12] may patch native classes like `HTMLElement`, and return a different instance of it. If a constructor and a mandatory `super()` call are needed, optional arguments must be passed along, and the return value of such a super call has to be returned by the constructor as mentioned by [10]:

```
class CustomElement extends HTMLElement {
  constructor(...args) {
    const self = super(...args);
    // self functionality written in here
    // return the right context
    return self;
  }
}
```

[12] https://github.com/webcomponents/polyfills/tree/master/packages/webcomponentsjs

Chapter 6

Evaluation of a Declarative Shadow Root

In this chapter, the results of the project are discussed. Such a system can be evaluated in several ways: The specification can be checked if it was met, the performance of the developed custom element can be analysed by measuring load times and size and then compare those to alternative approaches, and the reasonableness of the element for developers can be assessed. Additionally, this chapter offers insight into some semantic considerations regarding prerendering and search engine optimisations. A comparison is then made by analysing the outcome for SEO crawlers and the information they can index. Finally, the encountered issues and therefore, possible improvements are stated.

6.1 Specification

The specification mentioned in Section 4.3.3 consists of a mix of automated and manual tests. Some of the defined scenarios were unable to be mocked by the testing environment, and hence were manually tested by constructing them inside of a website and testing them manually. All continuous integration tests are successful[1], with a total of 18 tests ran, and six were checked manually after all.

In the end, the specified shadow boundary identifier is not as important as introducing this new feature to the web platform. Whether the boundary is identified by an attribute or a wrapper element makes not much of a difference from a users' perspective. Although, it can cause a non-trivial work when introducing new attributes to existing elements or creating new elements at all. Changing parser logic has to be kept to a minimum, and this leads to the assumption that even if a new `<shadowroot>` element would be intuitive and more ergonomic, the developers in charge will not introduce a new element but add the behaviour to the existing `<template>` element.

6.2 Performance

Since a native implementation was not feasible, an empirical analysis of the different render times of a native element compared to a custom element can not be done. However, a comparison of the average load time, as well as the total bytes delivered of

[1] https://travis-ci.com/github/drdreo/declarative-shadow

an HTML document caused by different techniques, which have a similar declarative shadow DOM mechanism, can be made.

As a side note, Mason Freed has created an open-source benchmark[2] for his native Chrome implementation on which this performance measurements build upon and has recently published his results. A third-party library called *tachometer*[3] was used for the measurements. It is a convenient benchmark tool, which through repeated sampling and measurements can accurately detect variations in timing.

The benchmark was executed on a MacBookPro 13-inch, 2018, 2.3GHz Quad-Core Intel i5, 8 GB 2133MHz LPDDR3 utilising a headless Chrome with version 83.0.4103.61. Every method was repeated and measured 50 times and the tested HTML file included 10000 elements per document. The elapsed time measurement is started just before the first shadow DOM is attached to their respective element and is stopped with the `load` event of the `window` object being triggered, which occurs when the whole page and all resources have been loaded. The benchmark can be found on the GitHub repository[4] and includes 3 different methods of attaching a shadow root:

1. Declarative shadow root element—the result of this thesis.
2. An inline script—attachment logic is added inside the host element.
3. A single-loop script—attachment happens at the end of the web page.

The `<shadow-root>` element was used as suggested in Section 5.3.2. The inline script method replicates a possible alternative. It uses an inline script that is placed immediately after each `<template>`, which is used as a shadow boundary identifier here. The script attaches the shadow root and moves the template contents into it. For completeness, the used nodes are also removed, so that the resulting DOM tree is identical to the declarative output. This approach is the most straightforward replica of a declarative shadow DOM solution because the shadow root is attached and populated immediately after being parsed.

The third method executes a single script at the end of the HTML document that loops over all encountered elements to attach the shadow roots and move the associated contents into them. This snippet also removes the used `<template>` elements and the final `<script>` node. This approach is optimised for speed because the entire page is parsed first, and then one script takes care of the shadow root attachment and population. The advantage here is that the parser is not blocked by a script for each shadow root. The major disadvantage is that the entire page consists of inert `<template>` elements that do not render until the final script loops over and processes them into shadow roots.

As seen in Table 6.1, out of all the examined approaches, the declarative solution transferred the least bytes with a total of 1034.65 KiB, as well as the shortest average time until the website is fully loaded with 219.23 ms. The differences of each method are shown in Table 6.2. In the end, the declarative approach was roughly 82% faster than the inline script and 4% to 6% faster compared to the single-loop method.

Nevertheless, it is important to note that this is just a tentative sneak-peek into performance. The code snippets are not optimised, hence comparing a single script per

[2] https://github.com/mfreed7/declarative-shadow-dom/tree/master/perf_tests/explainer_example
[3] https://npmjs.com/package/tachometer
[4] https://github.com/drdreo/declarative-shadow/tree/master/perf

Table 6.1: Benchmark results of each method as an overview. The total delivered bytes in kibibytes and average time in milliseconds needed for loading the website are shown.

	Bytes	*Avg. time*
Declarative	1034.65 KiB	219.23 - 221.29 ms
Inline	3720.14 KiB	1245.22 - 1250.77 ms
Single Loop	1044.70 KiB	230.76 - 232.79 ms

Table 6.2: Benchmark of each method against an other. The timing differences are shown in comparison with each method.

	vs. Declarative	*vs. Inline*	*vs. Single-loop*
Declarative	–	−1023.93 to −1031.54 ms	−9.47 to −13.56 ms
Inline	+1023.93 to +1031.54 ms	–	+1013.26 to +1019.17 ms
Single Loop	+9.47 to +13.56 ms	−1013.26 to −1019.17 ms	–

shadow root and a single script per page seems a bit extreme. Forced style renderings were eliminated, like done by Freed, by wrapping the test set inside an element with the style properties set to `display:none` and `contain:strict`, which reduces the processing time needed for redrawing the page by informing the browser that this section is independent from the rest of the document.

An interesting aspect is that the reported results of Freed are very similar to the ones shown in Table 6.1, even though he did not use a custom element, but like already mentioned in Section 3.4, he used the `<template>` element with the functionality implemented natively in Chrome. Comparing his figures with the same test setup shows that the total bytes of his implementation are nearly equivalent to the figures shown here. His implementation transferred a total of 1034.63 KiB and needed 248.72 ms to 254.60 ms to load on average. This may lead to the conclusion that the custom element approach is slightly faster than the native declarative shadow DOM solution.

6.3 Semantics

Tim Berners-Lee mentions the importance of making the web accessible for web agents and making web sites processable and understandable for machines. To quote his vision [3]:

> Machines become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A “Semantic Web”, which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines, leaving humans to provide the

> inspiration and intuition. The “intelligent agents” people have touted for ages will finally materialize.

Having encapsulated elements is a great feature for developers which enhances existing workflows and can contribute to making web development more structured and organised. However, it works against the hard-earned semantic web approaches which led to great inventions and improvements over the years. Currently, a shadow root removes the capability of computers to analyse its content. One may argue that in times of security and privacy concerns, such a layer of data encapsulation is a desirable mechanism, as mentioned in Section 2.3.3.

Nonetheless, a developer feature should not weaken the semantic web but persist the possibilities earned through it. A declarative shadow root element will restore lost semantics caused by the shadow DOM, respectively encapsulated Web Components, through presenting the shadow DOM's content openly before moving it to an encapsulated area.

6.3.1 Prerendering

As far as prerendering is concerned, the new element simplifies the serialization process, especially when a new accessor method is natively implemented, which serializes an element with a shadow DOM automatically. Additional reconstruction steps, like rehydration, are redundant if the browser can construct the according DOM on its own.

This feature finally makes the shadow DOM choice a viable option considering production-ready requirements for web applications like SEO compatibility. Because as of today, most companies that already use Web Components or custom elements in production choose not to use shadow DOM yet due to the lack of support for older browsers. This may lead to a reconsideration when a declarative shadow root is finally introduced as a web standard which makes it possible for incompatible environments to at least see the contents of such encapsulated elements.

6.3.2 SEO

With the declarative shadow DOM, bots and crawlers that only support HTML and no JavaScript have a chance to catch up with the latest specification, without the need for JavaScript workarounds and to support shadow DOM. This solution suggests a declarative, HTML-only approach, which is finally a way to declare a shadow DOM that reaches SEO crawlers immediately. Even if they do not support the latest specification, they can process a `<shadowroot>` as an unknown element and still interpret the content of it because it is available in plain HTML.

To further investigate what a crawler would be able to index and to demonstrate the SEO advantages of the project, a test website was created using a Web Component for the profile card section shown in Program 5.3 with the same shadow DOM outcome, additionally to the declarative shadow root version. The Web Component was used as follows, which shows that the list of skills is not included in the light DOM:

```
<profile-card>
    <div slot="header">John D. Eveloper</div>
</profile-card>
```

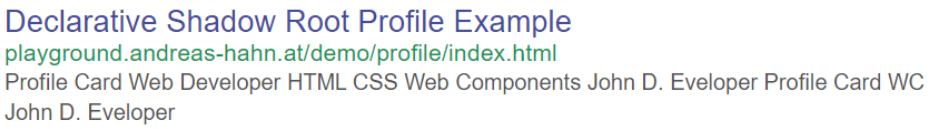


Figure 6.1: SEO search result preview of a test website displaying a profile card. Includes one version of the profile card using a declarative shadow root and another version using a Web Component.

The available content for crawlers was tested using an SEO test site called *seotesteronline*[5]. Figure 6.1 shows how the crawler indexed the test site. As a result, the Web Component hides information contained in the shadow DOM. This is because of crawlers and bots can only see and index content that is available in the source code and thus available in the light DOM, *John D. Eveloper* in this case. Further information like the rendered list—HTML, CSS, Web Components—is only available in the declarative solution. Although most crawlers nowadays execute JavaScript, shadow DOM content will still not be accessed, and the information is lost.

6.4 Reasonableness

Generally, it needs to be said that the project is still a prototype and does not provide stability necessary for a production-ready solution. One particular problem that this project should solve is a JavaScript-free solution for environments that do not support it, but since a custom-made workaround has to use JavaScript, this limitation can not be overcome. Another major problem is that it has to be added manually to the HTML document and does not benefit from the accessibility a native implementation offers.

Nonetheless, the outcome of the project shows that it adds new, unique and desirable ways for HTML and CSS authors to consider during the planning phase of a website. It enables them to create independent sections that do not affect others in terms of styling and the DOM, which cannot be messed with from the outside, without the need for JavaScript at all. Even third-party widgets or plugins that might interfere with existing styles can be integrated without a hassle by wrapping them into their own style scope as demonstrated in Program 5.3.

The web community has desired a mechanism for style isolation for quite some time now, and moving HTML into the shadow DOM is the easiest way to do so while using standards the web platform has to offer. Requiring developers to create a single-use Web Component to get the desired encapsulation seems gratuitous. Even if the mentioned proposal for declarative custom elements is adopted, that solution will always have additional and unnecessary steps when the only desire is to move elements into a shadow DOM, which is all it takes for style isolation.

Avoiding additional unnecessary steps, even if they are objectively relatively simple, is the whole point of this project. A rather complex script that can be copy-pasted by the developer is still too much additional complexity for what should be a simple task. Ergonomics is the most prominent aspect here. If this feature should be adopted by

[5] https://suite.seotesteronline.com/seo-checker

a large number of developers, it must be easy and straightforward to implement, and more importantly, practical, rather than feeling like an advanced hack for complicated use cases only. An appropriate shadow boundary identifier right at the position where isolation should occur feels simple and more or less like ordinary HTML as it is supposed to be. A client-side script that executes some APIs and moves elements does not feel intuitive.

As well as enhancing prerender techniques of Web Components with a declarative shadow root brings back semantics to those components which is analyzed in Section 6.3.

6.5 Further Improvements

Because the scope and complexity of changing browser internals is too much for the scope of this thesis and was not feasible, therefore, the most crucial improvement includes the native incorporation of this prototype into the parser logic instead of having it handled by a custom element. Adding the behaviour to the web standard means a big step into componentization and would boost the adoption rate and the confidence of bringing shadow DOM to the tech stack of developers.

Also, the decision to choose a new element may not be as practical as initially thought. Due to the implementation costs and the compatibility issues, repurposing the `<template>` element will most likely be more economical, which is the reason why Freed implemented a prototype doing so.

As far as the custom element algorithm implemented in this project is concerned, it has shown the potential for improvement as well. The explicit throwing of an error if the element has already a shadow root is not strictly necessary, since it can just call the `attachShadow()` method without the check and that will handle the correct error throwing already. Furthermore, the defined behaviour in Section 4.3.2 Item 5, that the content should be inert if the mode value is not valid can be improved to set elements actually inactive. Currently, it stops the attachment process and leaves the content as is which can cause unintended styling and resources are still fetched.

The declarative shadow DOM makes it possible to have a kind of static Web Component. It can have a markup, but that is unable to change dynamically, leaving the slotting mechanism aside. As a further improvement, and to bring the web one step closer to declarative Web Components that are fully functional as if created via JavaScript, the possibility to add functionality and logic can be integrated. Like the declarative custom elements proposal mentioned in Section 3.1, a technique to add a custom element class to the declarative created shadow DOM would evolve the declarative shadow DOM from being static to being dynamic. This might be done simply by declaring a `<script>` element which advances the shadow host with the custom element behaviour. In the end, this would enable a potential way for the definition of declarative Web Components in the future.

Chapter 7

Conclusion

The declarative shadow root is an element-based encapsulation method for structuring and scoping sections of HTML. It uses a new native element to inform the parser where it has to attach a new shadow root and what HTML elements need to be included. This declaratively defined sections can also be seen as components. This new way of attaching a shadow root will not replace the existing imperative way of doing so but enhances non-scripting environments with the shadow DOM functionality.

Component-based development is a popular trend among the web community today. Many web frameworks base their architecture on components and implement their own scoping mechanisms. The big difference between these frameworks and Web Components is that Web Components do the componentization on the native DOM level. Although, the goal is not to replace these frameworks since frameworks focus on a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. These systems glue components together and can provide much more functionality than encapsulating building blocks.

Nonetheless, Web Components can enhance those frameworks and have the potential to replace their component layer with a native and uniform solution when the time of disappearing frameworks has come. Recently frameworks showed much interest in adding a build feature which converts their framework-specific components into native Web Components, for example, Angular Elements[1]. However, the web standard lacks convenience for developers to push the adoption rate further. I regard this work as a contribution towards making Web Components more exciting and more powerful to use.

Even during the SPA revolution, developers came across the point where the need for pre-rendering of a JavaScript built website arose. Therefore, a server-side rendering solution for every popular modern framework was created to keep up with the requirements of the community. Someone could argue that this improvement is already an expectancy and Web Components should not be left out on the communities premise.

Like any emerging standard, Web Components do not offer all the answers just yet. Then again, no popular framework does. Even if "no-framework" Web Components are not the right answer, a modern framework will likely be built with them one day, although it may not be apparent. Enriching the standards would lead to broader and faster adoption among developers and open up a spectrum of use cases. The only thing

[1] https://angular.io/guide/elements

that is really lacking as of today are real-live examples of using the native component model in production and the best practices as a consequence of those. For instance, pre-rendering of a shadow root (to serve HTML before hiding the component's content inside the shadow DOM) which would enable search engines that do not support shadow DOM to index all the content again.

When I started this thesis, the perspective was that the feature is declined because of not being sufficiently useful. However, when I finished writing the thesis, the proposal was revived, implemented, tested and again presented to browser vendors and specification developers. That showed how versatile and fast living the web is and I am excited in what direction the declarative shadow DOM will go.

Only time will show how and if developers adapt to a declarative shadow DOM and find use cases where such a declarative encapsulation is more beneficial over current methodologies, especially regarding CSS and the current naming conventions to simulate a style scope. It probably will be less used manually by authors but programmatically by renderers or serialization algorithms that render an output of a computed shadow DOM and insert it as a child of an element with a clearly defined boundary that tells the browser how to construct the encountered element correctly including its shadow DOM.

Appendix A

Supplementary Materials

List of supplementary data submitted to the degree-granting institution for archival storage (in ZIP format).

A.1 PDF Files

Path: /

thesis.pdf	Master thesis (complete document)

A.2 Media Files

Path: /media

*.pdf, *.svg	vector graphics
*.jpg, *.png	raster images

A.3 Online Sources (PDF Captures)

Path: /online-sources

testing_wc.pdf	Testing Workflow for Web Components [9]
using-ce_MDN.pdf . .	Custom Element Polyfilling [10]
using-sdm_MDN.pdf .	Using Shadow DOM [11]
dsr_fagnani.pdf	Declarative Shadow Root Element[13]
dsd_freed.pdf	Declarative Shadow DOM Proposal (Version 2) [14]
ce-best-practices.pdf . .	Custom Element Best Practices [15]
dce_proposal.pdf	Declarative Custom Elements Proposal [16]
component-web-dev.pdf	The Benefits of Component Driven Web Development [19]
css_taggart.pdf	I'm super good at CSS and I don't recommend the cascade, don't @ me [20]
promise-wc.pdf	The Broken Promise of Web Components [12]

javascript-usage.pdf . .	Usage Statistics of JavaScript as client-side Programming Language on Websites [18]
dom-standard.pdf . . .	DOM. Living Standard [21]
html-standard.pdf . . .	HTML. Living Standard [22]
construct-stylesheets.pdf	Constructable Stylesheet Objects [23]
html-components.pdf .	HTML Components [24]
dsd_w3c.pdf	Declarative Shadow DOM Proposal (Version 1) [25]

References

Literature

[1] Mikio Aoyama. "New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development?" (July 1998), p. 5 (cit. on p. 8).

[2] Farrell Ben. *Web Components in Action*. Manning Publications, Aug. 2019, p. 432 (cit. on p. 10).

[3] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. 1st. Harper San Francisco, 1999, p. 226 (cit. on pp. 1, 41).

[4] Achim D. Brucker and Michael Herzberg. "A Formally Verified Model of Web Components". In: *Formal Aspects of Component Software (FACS)*. Ed. by Sung-Shik Jongmans and Farhad Arbab. Lecture Notes in Computer Science 12018. Amsterdam, The Netherlands: Springer-Verlag, 2019, pp. 51–71 (cit. on p. 8).

[5] Michael Freyberger et al. "Cracking ShadowCrypt: Exploring the Limitations of Secure I/O Systems in Internet Browsers". *Proceedings on Privacy Enhancing Technologies* (Apr. 2018), pp. 47–63 (cit. on p. 8).

[6] Jean-Sebastien Legare, Robert Sumi, and William Aiello. "Beeswax: a platform for private web apps". *Proceedings on Privacy Enhancing Technologies* (July 2016), pp. 24–40 (cit. on p. 7).

[7] Francesco Strazzullo. *Frameworkless Front-End Development*. Apress, 2019, p. 248 (cit. on p. 11).

Audio-visual media

[8] Alex Russell. *Web Components and Model Driven Views by Alex Russell*. 2011. URL: https://fronteers.nl/congres/2011/sessions/web-components-and-model-driven-views-alex-russell (visited on 01/03/2020) (cit. on pp. 1, 9).

Online sources

[9] Thomas Allmer. *Testing Workflow for Web Components*. Apr. 2019. URL: https://dev.to/open-wc/testing-workflow-for-web-components-g73 (visited on 05/15/2020) (cit. on p. 32).

[10] MDN Contributors. *Custom Element Polyfilling*. Apr. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements#Polyfills_vs._classes (visited on 05/21/2020) (cit. on p. 38).

[11] MDN Contributors. *Using Shadow DOM*. Oct. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM (visited on 01/09/2020) (cit. on p. 6).

[12] Dmitrii Dimandt. *The Broken Promise of Web Components*. Mar. 2017. URL: https://dmitriid.com/blog/2017/03/the-broken-promise-of-web-components/ (cit. on p. 9).

[13] Justin Fagnani. *Declarative Shadow Root Element*. Dec. 2016. URL: https://gist.github.com/justinfagnani/936791248120749ff1f8188f1f4064d9 (visited on 12/14/2019) (cit. on p. 18).

[14] Mason Freed. *Declarative Shadow DOM Proposal (Version 2)*. Apr. 2020. URL: https://github.com/mfreed7/declarative-shadow-dom/blob/master/README.md (visited on 04/28/2020) (cit. on pp. 16, 27).

[15] Google. *Custom Element Best Practices*. Sept. 2019. URL: https://developers.google.com/web/fundamentals/web-components/best-practices (visited on 05/16/2020) (cit. on p. 33).

[16] Ryosuke Niwa. *Declarative Custom Elements Proposal*. Nov. 2017. URL: https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Declarative-Custom-Elements-Strawman.md (visited on 01/22/2020) (cit. on pp. 2, 13).

[17] Stack Overflow. *Stack Overflow Trends*. Apr. 2020. URL: https://insights.stackoverflow.com/trends?tags=jquery%2Cangular%2Cangularjs%2Creactjs (visited on 04/08/2020) (cit. on p. 11).

[18] Q-Success. *Usage Statistics of JavaScript as client-side Programming Language on Websites*. URL: https://w3techs.com/technologies/details/cp-javascript (visited on 12/14/2019) (cit. on p. 17).

[19] Fredrik Söderquist. *The Benefits of Component Driven Web Development*. Mar. 2017. URL: https://dev.to/fregu/the-benefits-of-component-driven-web-development (cit. on p. 4).

[20] Simon Taggart. *I'm super good at CSS and I don't recommend the Cascade, don't @ me*. Jan. 2019. URL: https://www.simontaggart.com/2019-01-11-im-super-good-at-css-and-i-dont-recommend-the-cascade-dont-@-me/ (visited on 04/14/2020) (cit. on p. 9).

[21] WHATWG. *DOM*. Living Standard. Feb. 2020. URL: https://dom.spec.whatwg.org/ (visited on 03/09/2020) (cit. on p. 4).

[22] WHATWG. *HTML*. Living Standard. Mar. 2020. URL: https://html.spec.whatwg.org (visited on 03/09/2020) (cit. on p. 4).

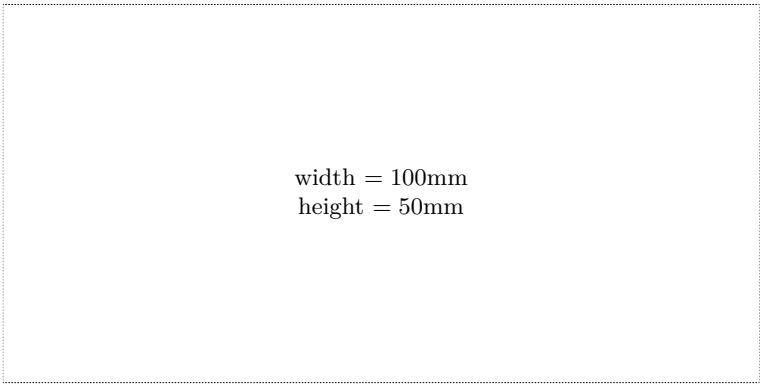
[23] WICG. *Constructable Stylesheet Objects*. A Collection of Interesting Ideas. Mar. 2020. URL: https://wicg.github.io/construct-stylesheets (visited on 03/10/2020) (cit. on p. 8).

[24] Chris Wilson. *HTML Components*. Componentizing Web Applications. Oct. 1998. URL: https://www.w3.org/TR/1998/NOTE-HTMLComponents-19981023 (visited on 04/04/2020) (cit. on p. 9).

[25] Tomek Wytrębowicz. *Declarative Shadow DOM Proposal*. Aug. 2018. URL: https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Declarative-Shadow-DOM.md (visited on 12/20/2019) (cit. on pp. 2, 16).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —