Reusability of Web Components

Klaus Fischer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2016

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, September 16, 2016

Klaus Fischer

Contents

Declaration **iii**

Kurzfassung **vii**

Abstract **viii**

1 Introduction **1**
1.1 Goal of the thesis 1
1.2 Motivation 2
1.3 Thesis contents 2

2 Technical background **3**
2.1 Modular programming concepts 4
2.1.1 Composition over inheritance 4
2.1.2 Interfaces 5
2.1.3 Mediator pattern 5
2.1.4 Observer pattern 5
2.2 Web component technologies 6
2.2.1 Custom elements 6
2.2.2 HTML templates 7
2.2.3 HTML imports 8
2.2.4 Shadow DOM 9

3 State of the Art **13**
3.1 Vanilla JavaScript 13
3.1.1 Custom elements and lifecycle callbacks 14
3.1.2 Shadow DOM 15
3.1.3 HTML template element 16
3.1.4 HTML imports 16
3.2 X-Tag 18
3.3 Bosonic 20
3.4 Angular 2 22
3.5 Polymer 23

4 Technical design **25**
4.1 Backend . . . 25
4.2 Frontend . . . 26

5 Implementation **29**
5.1 Toolchain . . . 29
5.2 Keep-API: backend . . . 31
5.3 Keep: frontend . . . 33
5.3.1 Look and feel . . . 37
5.3.2 Components . . . 39

6 Architectural design concepts **47**
6.1 A modular mindset . . . 47
6.2 Managing content in components . . . 48
6.3 Communication between components . . . 50
6.3.1 API design . . . 51
6.4 Semantics and accessibility . . . 52
6.4.1 Element naming . . . 52
6.4.2 Accessibility . . . 52
6.5 Visual design . . . 53
6.6 Testing . . . 55
6.7 Updates . . . 56
6.8 Deployment . . . 56

7 Benefits, consequences and limitations **58**
7.1 Components instead of pages . . . 58
7.2 Architectural decisions . . . 59
7.3 Choosing a framework . . . 60
7.4 High quality components . . . 60
7.5 Updating components . . . 61

8 Conclusion **63**
8.1 Outlook . . . 63

A CD Contents **65**
A.1 Thesis . . . 65
A.2 Resources . . . 65
A.3 Listings . . . 66
A.4 Thesis Project . . . 66

B Listings **67**
B.1 Web components with plain JavaScript . . . 67
B.1.1 my-tooltip.js . . . 67
B.1.2 index.html . . . 67
B.2 Shadow DOM . . . 68

B.2.1 my-tooltip.js 68
B.2.2 index.html 69
B.3 HTML template element 70
B.3.1 index.html 70
B.4 Complete tooltip example 71
B.4.1 my-tooltip.html 71
B.4.2 my-tooltip.js 71
B.4.3 index.html 73
B.5 X-Tag example 73
B.5.1 movie-quoter.html 73
B.5.2 index.html 76
B.6 Angular 2 example 77
B.6.1 app.component.ts 77
B.6.2 nice-greeter.component.ts 77
B.6.3 movie-quoter.component.ts 78

References **80**
Literature 80
Online sources 80

Kurzfassung

Im Bereich der Webentwicklung finden sich viele Aufgaben, die sich ständig wiederholen. Elemente wie Buttons, Formulare oder Navigationen werden oft für jedes Projekt neu entworfen und implementiert. Dies kostet immer wieder Zeit, Aufwand und Geld. Webentwickler müssen sich in die verschiedenen Konzepten und Implementierungen immer wieder einarbeiten, um Änderungen umzusetzen.

Web Components ermöglichen eine W3C-standardisierte Umsetzung von eigenen HTML Elementen mit gekapseltem Verhalten. Solche Komponenten werden einmal gebaut, bieten Anpassungsmöglichkeiten von außen und können in verschiedenen Webprojekten importiert werden. Ist die Architektur der Komponenten richtig ausgelegt, lösen sie Scoping-Konflikte, vermeiden Redundanz und verbessern die Code-Semantik.

Diese Arbeit untersucht, *wie* wiederverwendbare Web Components gebaut werden können. Architekturkonzepte, wie die Kommunikation zwischen Komponenten (mithilfe eine mehrschichtigen API), das Verschieben der Denkweise von einzelnen Webseiten zu wiederverwendbaren Komponenten, der Umgang mit Inhalten in Komponenten, der Update- und Deployment-Prozess ebenso wie das Reduzieren von Programmabhängigkeiten sind die wichtigsten Forschungsthemen. Vorteile, Konsequenzen und Limitierungen beim Einsatz von Web Components werden ebenfalls diskutiert.

Die Vorteile von wiederverwendbaren Komponenten sollen die Webcommunity ermutigen, mehr Web Components zu verwenden, Best Practices zu etablieren und weitere Verbesserungen zu entdecken.

Abstract

Many tasks in web development are recurring. Elements, like buttons, forms or navigations get engineered and built from scratch with each project. This takes time, effort and money. Concepts and implementation differ, so developers need more time to get familiar with different projects to implement changes.

Web components are a W3C-standardized way to build custom HTML elements with encapsulated behaviour. Such components are built once, allow customization and can be reused in different web projects. If built with the right architecture in mind, web components solve scoping conflicts, avoid repetition and improve the semantic meaning of the source code.

This thesis explores *how* to build reusable web components. Architectural design concepts like the importance of communication between components (involving a layered API), shifting the mindset from pages to reusable components, managing content in components, the updating and deployment process as well as minimizing dependencies are the most important research topics. Resulting benefits, consequences and limitations of using web components are also highlighted.

The benefits of resuable components should encourage the web community to employ web components, establish best practices and explore further improvements.

Chapter 1

Introduction

The popularity of websites is growing at a fast pace and so does the complexity when building them. Serving web content with a consistent quality on a wide range of devices challenges everyone who contributes in the process of bringing content to the web. One approach of tackling this problem is to focus on a modular architecture, where small components are built and assembled to complete sites. Such high quality components may take a longer to develop initially, but once built they can be imported and reused easily across different websites.

1.1 Goal of the thesis

Goal of this thesis is to explore the field of web components with a focus on three research questions:

- What are the important aspects when designing a web component regarding architecture and reusability?
- What are the consequences and limitations when employing an architecture based on web components?
- How can a component be validated for reusability?

Building components is not only a task for developers, it starts already in the concept phase of a project and heavily involves a visual design part to succeed with a system of components. The following chapters should inspire the community around web components and everyone who is approaching componentized architecture in the web landscape. Best practices, benefits and holes have to be explored as well as shared to improve the field of web components.

1.2 Motivation

Being involved in multiple web projects, I experienced that many tasks in the web development process are recurring. A common case are buttons. Nearly every website uses buttons (or styled hyperlinks) in a way and for every website those buttons have to be designed, implemented and tested. If we take 10 websites and consider them seperated, in sum 10 sets of buttons would have to be created. Now, instead of implementing 10 different button sets, one for each website, what about implementing one set of buttons which is reusable and customizable on each of them? This wouldn't just reduce the time of implementing, but also brain power in the design process and afterwards in the testing phase.

So the motivation behind this thesis is to find out how it's possible to save time and effort when building reusable components which can be used across different web projects. By using standardized HTML, JS and CSS features new possibilities for the web are coming up.

Furthermore, the human aspect of learning is motivating. If a developer knows how to handle components, other projects with the same components are easier to understand. What gets interesting here is the question what happens when a component changes? Which concepts can be applied to component design to make them reusable? Asking such questions should inspire developers to design components carefully and explore possible weaknesses early on.

1.3 Thesis contents

Chapter 2 starts off with modular programming concepts in general and describes the four essential standards for web components. Continuing with existing implementations, chapter 3 shows short examples with plain JavaScript and different frameworks. Furthermore, chapters 4 and 5 discuss the thesis project, a website built with web components and *Polymer*.[1] Derived from the practical usage, chapter 6 formalizes implementation-independent concepts around modular architectures with web components. Benefits, consequences and limitations when employing web components are discussed in chapter 7. Finally the thesis concludes in chapter 8 and provides a future outlook.

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

Chapter 2

Technical background

This chapter introduces the 4 specifications used for web components and general modular programming concepts which can be applied when using web components. The term *web components* itself covers multiple web specifications, standardized via the W3C Consortium [24]:

- Custom elements [11],
- HTML templates [23],
- Shadow DOM [12],
- HTML imports [13].

Once those specs are completely implemented by the browser vendors, web components are working natively without any additional library.[1] In the meantime *polyfills*[2] are available online[3] to support cross-platform web components as of today. The current implementation status can be checked on *Are We Componentized Yet?*.[4]

Bringing component-based development to the World Wide Web is one goal of web components. Encapsulating markup (HTML), style (CSS) and behaviour (JavaScript) without affecting other parts of a web application should be possible. There are mainly two preconditions required to reach this goal. At first the web component specs have to be implemented by the browser vendors so the cross-platform support increases. Second, web developers have to learn about web components and create them. Building such components involves a lot of decisions in terms of architecture and functionality (What *is* possible with the component and what is *not*?), which may have consequences on the further evolution of the component.

Modularization, which is discussed in section 2.1, may be a familiar concept in classic software engineering, but it's hard to execute in the front-end

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

[2] Chunks of JavaScript, which enable certain functionality in older browsers.

[3] webcomponents.org

[4] http://jonrimmer.github.io/are-we-componentized-yet/

web environment. The browser landscape is diverse (don't neglect all the mobile browsers) which may lead to unexpected component behaviour in different environments. In the upcoming section some concepts of modular programming are examined and leveraged to the web platform.

2.1 Modular programming concepts

In this section more high level concepts of modular programming will be discussed which help to create reusable web components. When thinking of simple components these concepts may seem overmuch, but in a ecosystem of multiple components those concepts unfold their advantages.

2.1.1 Composition over inheritance

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition [2, p. 30]. When developing components it's often the case that one component uses functionality from multiple other components. For example, if the goal is to create a `<newsletter-registration>` component which has an input and a button to send the input value. One way to solve this would be multiple inheritance because the newsletter-registration element has to have the functionality of a `HTMLInputElement` as well as an `HTMLButtonElement`. Since JavaScript is a prototyped-based language, each object has a reference to a single prototype and multiple inheritance is not supported. Of course the new element could inherit from `HTMLInputElement` and add the markup for a `<button>` element manually, but that would be confusing because the final behaviour of the newsletter-registration is pretty different from the inherited `HTMLInputElement`. However, it's possible to *concatenate* objects (e.g. with `Object.assign()` but this would lead to even more confusion if combined objects have functions with the same name.

Much easier, the newsletter-registration element consists of a `<input>` and a `<button>` element without overriding any of their behaviour or knowing of their internal implementation. The functionality of submitting the input value on button click will be implemented in the newsletter-registration component. Composition enhances the flexibility of components because it eliminates the limitation of single inheritance.

In ES2015 it is possible to use the `class` keyword, but this is only syntactical sugar over the existing prototype-based inheritance.[5] Composition can be seen as a main concept in component development, as trees evolve by combining components.

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

2.1.2 Interfaces

Functions inside a component can be publicly accessible or just private for internal computation. Speaking in design patterns the public methods would act like a *Facade*, a higher-level interface which makes the subsystem (the component) easier to use [2, p. 175]. When designing a components public API it has to be decided about the level of abstraction. Either the methods are high-level conceptualized or also low-level functions are available, which allow more customization. Changing the public API later in production leads to a major update and possibly causes unexpected behaviour if the component is used in multiple places or projects.

Components can be styled by using CSS Custom Properties [21] which would be the styling interface for a component. If a component declares a CSS variable, a parent component can assign a values to it and style its children that way. JavaScript properties of a component can also be public and changed directly via data-binding. In a small component this may be sufficient, in larger components setters and getters can be implemented.

2.1.3 Mediator pattern

The *Mediator* defines an object that encapsulates how a set of objects interact and promotes loose coupling by keeping objects from refering to each other explicitly and control their interaction independently [2, p. 255]. An illustrative example for a mediator would be a form with multiple input fields. To prevent that each input has to know about all other sibling inputs (e.g. because of validation or dynamic form options) the form can mediate between the inputs. As a mediator the form handles changes between the inputs and promotes loose coupling. In situations where multiple components are acting together, a wrapper component can manage the data flow without touching the implementation its child components.

2.1.4 Observer pattern

Although a component should be designed as stateless as possible, in case of forms or more complex web applications sometimes states are needed and this is where the *Observer* pattern listed in the book *Design Patterns* [2, p. 273] can be used:

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

In web applications this pattern can be realized by adding event listeners or mutation observer[6] to certain DOM elements. If a DOM element changes

[6] https://developer.mozilla.org/en/docs/Web/API/MutationObserver

or an event is fired, the listeners or observers are notified. Mapping this concept to web components, a component can fire an event and subscribed listeners will execute. In addition to manually fired events with the Custom Elements specification (section 2.2.1) additional methods will be executed when a component gets added, removed or changes. Libraries like *Polymer* or *Angular 2* provide one-way or two-way-databinding of component properties also with an *notify* option, so parent components get notified on change.

These concepts should help to find the right level of granularity when designing components, keep the components independent and embrace reusability. The next section narrows down to the specifications for bringing web components to the browser.

2.2 Web component technologies

In this section the four main building blocks of web components will be introduced. At the time of writing several *Google* engineers, mainly Dimitri Glazkov and Hayato Ito are working on the W3C specifications. In the upcoming sections the term *document* is used for describing a single webpage with a DOM.[7]

2.2.1 Custom elements

The motivation behind custom elements is the notion that developers should be able to create their own functional DOM elements and existing HTML elements (like `<video>`) can be understood easier. When bringing an element to life, the most important part in the beginning is the naming. Also referred to as *custom element type* the name has to match the NCName production[8] (generally short lowercase naming) and *needs* to contain a `U+002D HYPHEN-MINUS` (a dash) character. Already occupied names by SVG and MathML elements must not be used (see [11] under section *Concepts*).

To enrich an element with functionality a *custom element prototype* has to be created. Since JavaScript is a prototype-based language, properties and functions of the element are attached to the custom element prototype. When creating a default custom element, one would inherit the prototype of `HTMLElement`, but it's also possible to inherit functionality from other interfaces like `HTMLButtonElement`[9]. This would change the usage of the element from `<fancy-button>` to `<button is='fancybutton'>`.

After choosing a name and a prototype, the element has to be registered to the document. This has to happen because on document creation and other events (when adding, removing or manipulating a custom element in

[7] https://dom.spec.whatwg.org/#introduction-to-the-dom
[8] https://www.w3.org/TR/xml-names/#NT-NCName
[9] https://developer.mozilla.org/en-US/docs/Web/API under section *H*

the DOM) the document registry will be consulted to determine the elements behaviour.

After element registration the following *lifecycle callbacks* are attached to each custom element [8]:

- The `createdCallback` is invoked after a custom element instance is created and its definition is registered.
- The `attachedCallback` fires when the custom element is inserted into a document and this document has a browsing context.[10]
- The `detachedCallback` fires when a custom element is removed from the document and this document has a browsing context.
- The `attributeChangedCallback` fires when a custom elements attribute is added, changed or removed. Additionally, the old value will be passed too.

If an element gets imported asynchronously, the lifecycle callbacks are of great use to access DOM outside the element. If a custom element appears in the DOM, but its `createdCallback` wasn't invoked yet (because the import is asynchronously), the CSS pseudo class `:unresolved` is attached to the element, so a *flash of unstyled content*[11] can be prevented.

2.2.2 HTML templates

For creating reusable elements there has to be a way to create reusable markup and the `<template>` HTML element solves that matter. The template element is used to declare fragments of HTML that can be cloned and inserted in the document by script [23]. These fragments can be imagined as reusable pieces of DOM which can be customized and filled with data on rendering. A template has certain properties [9]:

- *The content is effectively inert until activated.* The markup is hidden DOM and does not render.
- Any content within a template won't have side effects. *Scripts won't run, images won't load, audio won't play* until the template is used.
- *Content can't be selected.* Using `document.getElementById()` or `querySelector()` won't return child nodes of a template.
- There are *certain places* [23] where the `<template>` may be placed.

So this is good news for performance because content inside a `<template>` element won't require resources until it is used. In listing B.3.1 a template for a table row that will be cloned multiple times and filled with data.

[10] https://html.spec.whatwg.org/multipage/browsers.html#browsing-context
[11] https://www.w3.org/TR/custom-elements/#bib-FOUC

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <title>Human Being</title>
        <link rel="import" href="/imports/heart.html">
    </head>
    <body>
        <p>What is a body without a heart?</p>
    </body>
</html>
```

Listing 2.1: Importing a HTML document.

2.2.3 HTML imports

With HTML imports it is possible to reuse existing HTML documents in other HTML documents. The `<link>` element is used in the `<head>` section of a HTML document to refer an import (listing 2.1).

If the document parser encounters an import, it is *blocking scripts*, so the parser will fetch the external file first before continuing to parse the rest of the document. This can be undesired because a slow network connection the fetching blocks the page rendering noticeably. Adding an `async` attribute to the `<link>` will unblock rendering, but if code relies on the imported element it has to be made sure that the element is available (with lifecycle callbacks or the `:unresolved` CSS class).

A file will be imported regardless of the `media` attribute of the `<link>` matching the environment. When requesting a document to import it is restricted to the Content Security Policy[12] of the master document (the import referrer) to prevent cross site scripting attacks and only importing resources from trusted locations.

The browser requests each import as external resource like stylesheets, images or scripts. This means additional files have to be transferred and the site may slow down in case of excessive imports. If a site imports a web component, all the dependencies of the web component will be requested too, so the list of requests can grow really fast. When parsing the HTML imports, the browser will internally create an import map, to know which web component imports which resources. In case multiple web components are importing the same resource, the browser will request the file only once and use the import map for referring. A way of minimizing the payload of imports is to bundle them on the server side and import just the bundled document. Currently this is possible with the *npm* package *vulcanize*[13], but as noted in the description of *vulcanize*, it may be obsolete in the future

[12] https://www.w3.org/TR/CSP3/
[13] https://github.com/Polymer/vulcanize

because of the *HTTP/2 Server Push* feature where the server can estimate which other resources have to be sent to the client for rendering a requested resource. This may save additional request roundtrips. Chapter 7 deals with more benefits and consequences.

For now simple elements can be created and imported. Scoping is still a matter, since an HTML Import alone doesn't avoid any scoping conflict. The power of *shadow DOM* makes it possible to create *encapsulated* web components.

2.2.4 Shadow DOM

Shadow DOM enhances custom elements with 2 main abilities:

- *Hide implementation details:* element-related markup will be added to the Shadow DOM tree.
- *Style encapsulation:* restricts CSS styles to the context of the custom element and won't bleed out [12].

To make use of the shadow DOM a so called *shadow tree* has to be attached to a document DOM element. In the document DOM any element can host zero or one shadow tree. If a DOM element hosts a shadow tree it's called a *shadow host*. The root node of a shadow tree itself is named *shadow root*. Figure 2.1, taken from the W3C specification, pictures the encapsulation of the shadow tree and the reference from the shadow host. The shadow root only serves as logical reference, it is not attached to the shadow host as a child node. Of course it's possible that components use other components. A tree of trees like in figure 2.2 will be built for logical reference. Afterwards a "flattened" version of this tree (without the shadow root nodes as noted above) will be composed for rendering and is called a *document composed tree* (figure 2.3). Therefore, most of the existing APIs for accessing nodes are *scoped* and won't affect other component trees. For example, `document.getElementById(elementId)` never returns an element in a shadow tree, even when the element has the given `elementId` because only the documents DOM would be searched and not the shadow trees [23].

Events

If an event is dispatched in a shadow DOM, it either crosses the shadow tree boundaries or is terminated at the shadow root. Special events (like `scroll` or `selectstart`) won't leak into ancestor trees to avoid unexpected user agent behaviour.[14] For example, the `selectstart` event fires when a user starts a new (text) selection. If the user selects text inside an `<input>` element, only text inside the `<input>` can be selected. It would be highly disturbing if text content *outside* the `<input>` would be added to a user

[14] https://www.w3.org/TR/shadow-dom/#events-that-are-not-leaked-into-ancestor-trees

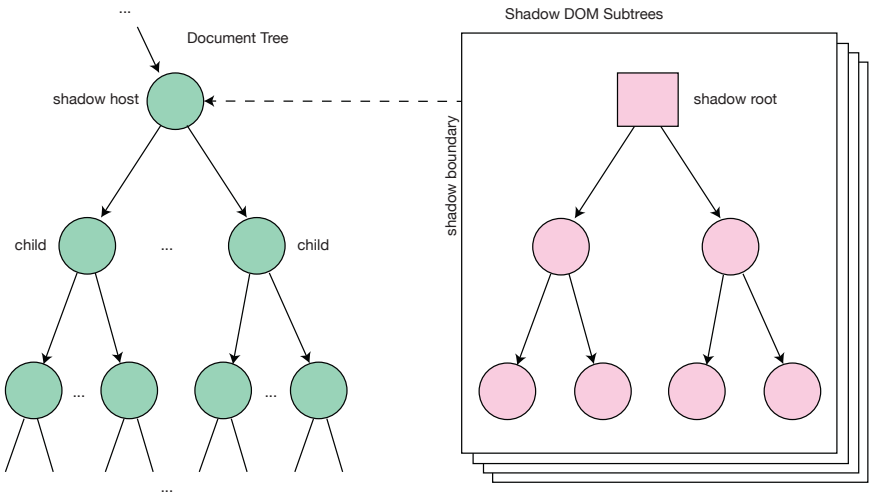


Figure 2.1: A document tree with a node containing a shadow dom [12].

selection that was started inside an `<input>`. Nevertheless, for most of the events an *event path* will be constructed which ranges across shadow trees. This happens on purpose to trace events happening in child components. This is good news for an author of the document DOM. If some node will get a shadow DOM attached to it, event listeners still would work. As an author of the shadow DOM (or when building a web component) this also means, one has to be aware of other events happening in the component because of nesting other components inside.

Styling

If a `<style>` tag is appended to a shadow DOM, all CSS rules inside this tag are applied only to the elements of the shadow DOM and won't bleed out to the document. Even if the selector matches elements outside the shadow DOM. Styles defined on document level won't be applied to shadow DOM elements either.

In listing B.2.1 all paragraphs inside the `<my-tooltip>` element will be in capital letters. Other paragraphs outside the element are not affected. Also just the paragraph in the document DOM will be colored and not the ones inside `<my-tooltip>`. Sometimes it is desired to style the host element itself, in this case `<my-tooltip>`. This can be done at the document level or from inside the custom element with the `:root` selector. CSS Custom Properties make it possible to use variables in CSS and style elements in a more flexible way [21].

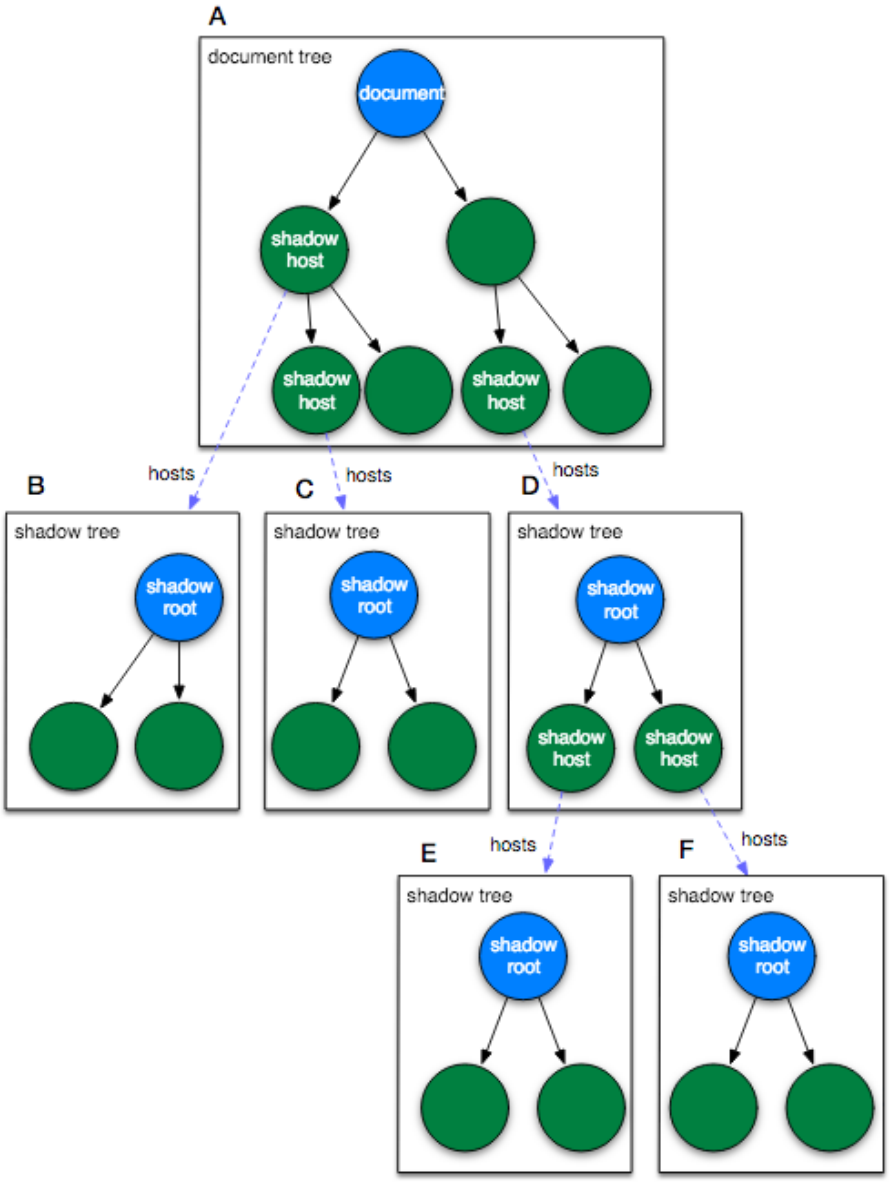


Figure 2.2: In the figure, there are six component trees named A, B, C, D, E and F. The shadow trees B, C and D are hosted by elements which participate in the document tree A. The shadow trees E and F are hosted by elements which participates in the shadow tree D [12].

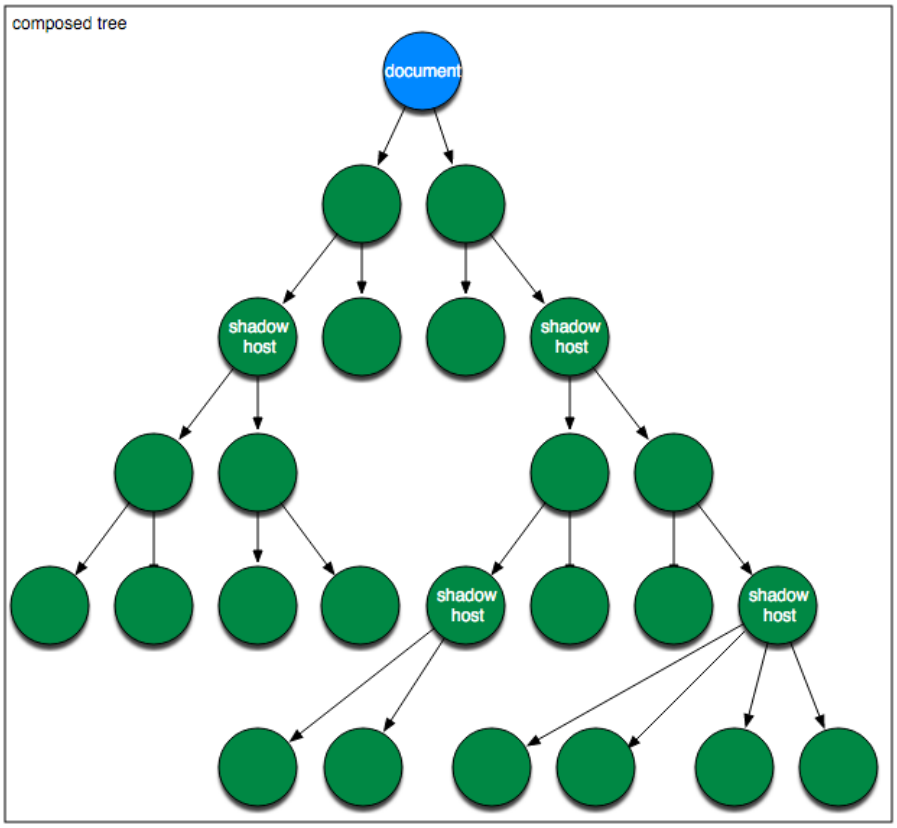


Figure 2.3: A composed version of the tree in figure 2.2.

Chapter 3

State of the Art

Previous sections have discussed the specifications of web components and listings were using just vanilla JavaScript (no additional libraries[1]) and the web components polyfill. Once the browser vendors have implemented the W3C specifications, the polyfills will be obsolete. Additional libraries and frameworks emerged during the evolution of web components to ease their usage. All of the following libraries utilize the four W3C standards as mentioned in section 2.2, so it is basically possible to interchange components with limited functionality, unless a component uses a framework-specific feature. Already existing open-source web components for all kinds of libraries are listed at `customelements.io`.[2]

3.1 Vanilla JavaScript

Custom HTML elements can be registered in JavaScript and use *lifecycle callback methods* (section 2.2.1) for handling the elements behavior when it is created, added, removed or its content gets updated. The markup of a component lives inside a HTML template tag (section 2.2.2), so on every occurrence of the component this markup will be inserted into the document DOM. Another important part of a web component is the *shadow DOM* (section 2.2.4). Once created, it encapsulates CSS styles and enables the developer to hide implementation details from the user. At last a component can be imported to a website via *HTML imports* (section 2.2.3).

Although a component is encapsulated it needs an interface to be customized in terms of behavior and styles. A component can expose properties which can be altered when the component is in use. CSS custom properties [21] are covering flexible styling and a combination of common HTML data attributes with JavaScript variables handle business logic. Focusing

[1] http://vanilla-js.com/

[2] https://customelements.io/

```
1 // 1. Create a prototype inheriting from a basic HTMLElement
2 var MyTooltipProto = Object.create(HTMLElement.prototype);
3
4 // 2. Give my-tooltip a foo() method.
5 MyTooltipProto.foo = function() {
6   alert('foo() called');
7 };
8
9 // 3. Define a property "bar".
10 Object.defineProperty(MyTooltipProto, "bar", {
11   value: 5,
12   writable: true
13 });
```

Listing 3.1: Create prototype for a custom element.

on reusability, an engineer has to draw the line how customizable the component should be. The following sections progressively enhance a tooltip component with the 4 web component technologies.

3.1.1 Custom elements and lifecycle callbacks

At first a HTML custom element `<my-tooltip>` with a property and a function is created. Complete listings for the following examples are available in appendix B.1. The web components polyfill[3] allows the usage of web component as of today and needs to be included to provide cross-browser support. Since the polyfill changes from time to time, it will be imported via *Bower*[4] (more on *Bower* in section 5.1).

The basis of a web component is a new prototype inheriting from the `HTMLElement` prototype. Properties and functions will be added, so all objects with a reference to this prototype gain access to them (listing 3.1). Now a new HTML element named `<my-tooltip>` is registered and associated with the newly created prototype object containing all functionality (listing 3.2). If the DOM parser now encounters a `<my-tooltip>` element, it will check the document registry and use the assigned prototype. To demonstrate the *lifecycle callbacks* (see 2.2.1) each time a `<my-tooltip>` element will be created in the DOM, a random value will be assigned to the components *bar* property (listing 3.3). Loading the JavaScript file via a `script` tag enables `<my-tooltip>` HTML elements in a document. After the `DOMContentLoaded` event, the property `bar` of each tooltip element will be logged to the console (listing 3.4).

[3] http://webcomponents.org/polyfills/
[4] http://bower.io

```
var MyTooltip = document.registerElement('my-tooltip', {prototype:
    MyTooltipProto});
```

Listing 3.2: Register a new custom element.

```
MyTooltipProto.createdCallback = function() {
  console.log("i was created");
  // assign a random number as id
  this.bar = window.crypto.getRandomValues(new Uint32Array(1))[0];
}
```

Listing 3.3: Assign a random number in the `createdCallback`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>01 - Native Web Components</title>
  <script src="bower_components/webcomponentsjs/webcomponents-lite.js">
    </script>
  <script src="elements/my-tooltip.js"></script>
  <script>
    document.addEventListener('DOMContentLoaded', function() {

      var tt = document.querySelectorAll('my-tooltip');
      tt.forEach(function(tooltip) {
        console.log(tooltip.bar);
      });
    });
  </script>
</head>
<body>
  Hello Tooltip!
  <my-tooltip>Hey, i am number 1.</my-tooltip>
  <my-tooltip>Hello there, i am number 2!</my-tooltip>

</body>
</html>
```

Listing 3.4: Using a custom element.

3.1.2 Shadow DOM

To avoid the pollution of the global scope and its conflicts (see [6, p. 112]), the previous example will be enhanced with shadow DOM (complete listings in the appendix B.2). The `createdCallback` function will be used for adding the shadow DOM (listing 3.5) since it is executed automatically

```
1 MyTooltipProto.createdCallback = function() {
2   console.log("i was created");
3   // assign a random number as id
4   this.bar = window.crypto.getRandomValues(new Uint32Array(1))[0];
5   this.addShadowDOM();
6 }
7
8 MyTooltipProto.addShadowDOM = function() {
9   // 6. add shadow root element
10  var shadow = this.createShadowRoot();
11  // style element is scoped inside the shadow dom
12  shadow.innerHTML = "<style>p { text-transform: uppercase; }</style>";
13  shadow.innerHTML += "<p>I'm the Shadow DOM!</p>";
14 }
```

Listing 3.5: Adding shadow DOM to a custom element.

everytime a new `<my-tooltip>` element is added to the DOM. Since the `<style>` node is added inside the shadow DOM, only the paragraphs inside the tooltip component will be uppercase. For further demonstration of the encapsulation the main HTML document colors all paragraphs, but the tooltip paragraphs won't be affected since document styles won't cross shadow DOM boundaries (see [6, p. 112]).

3.1.3 HTML template element

To avoid HTML markup in JavaScript, the HTML5 specification includes a `<template>` element. In listing B.3 a template for a table row is created, filled with data, cloned and appended to a table. The insertion points, where data will be inserted into the template, are coded imperatively in JavaScript. A proposal for the `<slot>` HTML element is existing, so a template can contain multiple slot elements. These slot elements can be named via data attributes and filled with content in a more declarative way [22].

3.1.4 HTML imports

Utilizing HTML imports is shown in listing B.4. The tooltip component can be imported as HTML file instead of a JavaScript file and also accepts the tooltip text as a data attribute (listing 3.6).

On top of the imported `my-tooltip.html` the whole component behaviour is outsourced in a separate file. The CSS selector `:host` targets the `<my-tooltip>` element itself and special states like `:hover` are added with brackets. Below the `<span class='tt'></span>` serves as a insertion point for the tooltip text. If text or other HTML content will be nested inside a `<my-tooltip>` tag, the `<content>` tag of the tooltip template will be replaced with that content. At last the component will be registered with a pro-

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>04 - Complete tooltip</title>
  <script src="bower_components/webcomponentsjs/webcomponents.js">
    </script>
  <link rel="import" href="elements/my-tooltip.html">
  <script>
    document.addEventListener('WebComponentsReady', function() {
      console.log("Imports loaded & elements registered!");
    });
  </script>
</head>
<body>
  <h1>Hello Tooltip!</h1>
  <p>I am sample text and a <my-tooltip data-text="This is super secret
     info">keyword</my-tooltip>has a tooltip.</p>
  <p><my-tooltip data-text="more secret info">Me too!</my-tooltip>
    Check it out via hovering the text.</p>


</body>
</html>
```

Listing 3.6: index.html with HTML Import.

totype object defined in `my-tooltip.js` (listing 3.7). To prevent pollution of the global namespace, the `myTooltip` prototype for the `<my-tooltip>` element) handles all tooltip behaviour. The `insertIntoDocument()` method handles the insertion and updating of a `<my-tooltip>` element. As a parameter it gets a reference to the new DOM element and id of the template where the content should be inserted. Now the `<my-tooltip>` template can be filled with data and cloned into the shadow DOM of the main document (listing 3.8).

Obviously the data-binding and bootstraping has room for optimization. Combining the *Custom Elements v1 spec*[5], the *Shadow DOM v1 spec*[6] and ES2015 syntax makes it already easier to create and register custom elements with shadow DOM. The next section deals with *X-Tag* which provides syntactic sugar and makes it easier to create new web components.

[5] https://html.spec.whatwg.org/multipage/scripting.html#custom-elements
[6] http://w3c.github.io/webcomponents/spec/shadow/

```
<!-- all behaviour -->
<script src="my-tooltip.js"></script>

<template id="tt">
  <style>
    :host {
      position: relative;
      text-decoration: underline solid gray;
    }
    .tt {
      display: none;
      position: absolute;
      top: -2.5em;
      left: 0;
      padding: 0.4em 0.5em;
      border-radius: 3px;

      font-family: sans-serif;
      font-size: 12px;
      white-space: nowrap;
      background-color: #444;
      color: #fff;
    }
    :host(:hover) {
      cursor: pointer;
    }
    :host(:hover) .tt {
      display: inline-block;
    }
  </style>
  <span class="tt"></span>
  <content></content>
</template>

<script>
  document.registerElement('my-tooltip', {prototype: myTooltip});
</script>
```

Listing 3.7: my-tooltip.html: includes the markup, styling and registration for the component.

3.2 X-Tag

X-Tag[7] is a lightweight library created by Daniel Buchner. It is also built on top of the web component polyfills and enables a more readable way to register elements and add events than vanilla JavaScript. A notable feature is the event delegation for managing custom events inside a component. *X-Tag* aims for a lightweight approach. Shadow DOM and HTML templates are

[7] x-tag.github.io/

```
1  var insertIntoDocument = (function() {
2    "use strict";
3    var importDoc;
4    // reference to the my-tooltip.html
5    importDoc = (document._currentScript || document.currentScript).
     ownerDocument;
6    // current my-tooltip DOM element, id of template tag in
     my-tooltip.html
7    return function (obj, idTemplate) {
8      var template = importDoc.getElementById(idTemplate),
9          clone = document.importNode(template.content, true);
10
11     // fill template
12     fetchInfo(obj, clone);
13     console.log(clone.querySelector(".tt"));
14     // attach full template to shadow root if text is set
15     if (obj.text) {
16       var sd = obj.shadowRoot;
17       // if the object has a shadowRoot, empty it
18       if(sd) {
19         while(sd.firstChild) {
20           sd.removeChild(sd.firstChild);
21         }
22
23       } else {
24         // if the object has no shadowRoot create one
25         sd = obj.createShadowRoot();
26       }
27       // append filled template to shadowRoot
28       sd.appendChild(clone);
29
30     }
31   };
32 }());
```

Listing 3.8: my-tooltip.js: Import the template tag from my-tooltip.html, fill it with content and append it to the shadow DOM in the main document.

optional compared to *Polymer*. At the time of writing, the open source elements on `customelements.io`[8] are roughly a third of the available *Polymer* components. Depending on the website requirements it has to be chosen between an easier way for registering custom elements or a full-featured library.

In the upcoming example a `<movie-quoter>` component will be created that shows famous movie quotes and provides a changeable style as well as a public method for showing a new quote. To work with *X-Tag*, the web

[8] https://customelements.io

```
bower i --save webcomponentsjs x-tag-core
```

Listing 3.9: Install *X-Tag* and the web components polyfills.

components polyfill and *X-Tag* itself can be installed via *Bower* (listing 3.9).

Starting out with the `movie-quoter.html`, the *X-Tag* library is loaded and the markup will be styled depending on the data attribute `theme`. To enable HTML imports, the snippet for referencing the importer (`index.html`) and importee (`movie-quoter.html`) is used as in the vanilla JavaScript examples.

An element can be registered with `xtag.register(('element-name', component))`. The `element-name` is simply the name of the HTML tag, and the component object is similar to the prototype of the previous examples. With the `lifecycle` and `methods` properties it's possible to attach lifecycle callbacks and other public methods for a component (listing 3.10).

What really differs from previous examples is the `accessors` property. It is responsible for defining component properties as well as corresponding setters and getters. *X-Tag* handles the syncing of DOM and properties in JavaScript without having to write additional code. The `theme` property of the `movie-quoter` component can be either *minimal* or *fancy* and its current value is reflected in the HTML data-attribute `theme`. No matter if the data-attribute is changed in the DOM or the JavaScript property, there are no conflicts, as those two are always in sync. Taking a look at events, *X-Tag* provides multiple ways to attach event handlers with the `events` property on element registration or `xtag.addEvents()` method. Summing up *X-Tag*, it enhances readability of code and may be suited for really small projects since it's not a fully featured library.

3.3 Bosonic

The Bosonic Framework[9] was created by Raphaël Rougeron and also aims for the lightweight approach but more in the direction of providing a rich element library than a framework for building new web components. Typically, the components are styled very minimalistic to provide a bare boilerplate for customization. Rougeron states in the Bosonic FAQ [7]:

> I personally believe Web Components are ideal for low-level, "standard", reusable UI elements like dialogs, tabs or dropdowns, and that they should be combined with a more powerful library or framework like React, Angular or Ember in order to develop a complete application.

[9] http://bosonic.github.io/

```
1    xtag.register('movie-quoter', {
2        lifecycle: {
3            created: function() {
4                this.bquote = template.querySelector('blockquote');
5                this.quote = template.querySelector('p');
6                this.who = template.querySelector('.who');
7                this.movie = template.querySelector('.movie');
8                // Creates the shadow root
9                this.shadowRoot = this.createShadowRoot();
10               this.setQuote();
11           },
12           attributeChanged: function() {
13               console.log("attribute changed");
14           }
15       },
16       accessors: {
17           theme: {
18               attribute: {},
19               get: function(){
20                   return this.getAttribute('theme') || "minimal"
21               },
22               set: function(value){
23                   this.xtag.data.theme = value;
24               }
25           }
26       },
27       methods: {
28           setQuote: function() {
29               var quoteNum = getRand(quotes.length);
30               var q = quotes[quoteNum];
31
32               this.quote.textContent = q.quote;
33               this.who.textContent = q.who;
34               this.movie.textContent = q.movie;
35               // Removes shadow root content
36               this.shadowRoot.innerHTML = '';
37               // Adds a template clone into shadow root
38               var clone = importer.importNode(template, true);
39               this.shadowRoot.appendChild(clone);
40           }
41       },
42       events: {
43           'click': function (event) {
44               console.log('a movie-quoter was clicked');
45           }
46       }
```

Listing 3.10: movie-quoter.html: x-tag.register() part.

At the time of writing the Bosonic Framework is going through bigger changes in terms of architecture and workflow, so further details are omitted because it seems like the Bosonic Team needs some time to settle on the contents for releases.

3.4 Angular 2

Also very focused on component driven development is Googles *Angular* Framework.[10] The application architecture changed fundamentally in *Angular 2* to a more component-driven strategy. *Angular* comes as fully-featured framework and the web component standards are used under the hood similar to *Polymer*. The only specification which is directly used by developers are custom elements. Per default shadow DOM isn't activated, e Angular adds HTML attributes to the elements to preserve scoping, but it is possible to opt-in shadow DOM with `encapsulation: ViewEncapsulation.Native`. Since e Angular is a JavaScript framework, HTML imports aren't used, but SystemJS handles module loading. HTML Templates are also not written when coding e Angular Components, but compiled internally [20]. As Victor Savkins article tells, the *Angular 2* team is embracing web standards and it is possible to use non-e Angular web components inside of *Angular 2* applications.

Since *Angular 2* is a complete framework for web applications this section just covers the creation of a new component and not all the concepts around *Angular 2*. This example was started with the *Angular 2 Quickstart* source, for prototyping a small component. The `<my-app>` component serves as an entry point for the whole application (listing 3.11). *Angular 2* code is written in TypeScript[11], a typed superset of JavaScript, so the source files have a `.ts` ending and compile to plain JavaScript. Components can be imported on top via a ES2015 import and added to the `directives` array, so the parser has the source available when it encounters a `<nice-greeter>` element. Once the parser encounters a `<nice-greeter>` element, the imported module takes over (listing 3.12).

Template and styles can also be coded in external files for structuring purposes. For the developer it's not necessary to handle template importing like in the previous examples with vanilla JavaScript or *X-Tag*. Properties and behaviours are declared in the Javascript class. Mechanisms for one-way as well as two-way data-binding for binding whole models (not just simple values) are available in *Angular 2* but won't be discussed in this thesis.

As a bit more advanced example the `<movie-quoter>` component from the *X-Tag* section was re-created in *Angular* and is available in appendix B.6.

[10] https://angular.io/
[11] https://www.typescriptlang.org/

```
import {Component} from 'angular2/core';
import {NiceGreeterComponent} from './nice-greeter.component';
import {MovieQuoterComponent} from './movie-quoter.component';

@Component({
    selector: 'my-app',
    template: `<h1>My First Angular 2 App App</h1>
          <nice-greeter>content</nice-greeter>
          <h3>Another H3</h3>
          <movie-quoter></movie-quoter>`,
    directives: [NiceGreeterComponent, MovieQuoterComponent]
})
export class AppComponent { }
```

Listing 3.11: my-app.ts: Entry point of the whole application.

```
import {Component} from 'angular2/core';
import {ViewEncapsulation} from 'angular2/core';

@Component({
    selector: 'nice-greeter',
    template: `<h3>Hi {{name}}</h3>
    <button (click)="logMyName()">Log my name</button>`,
    styles: [`
      h3 {
        color: red;
      }
    `],
  // activate native shadow dom.
  // inspect the result, comment this line and inspect again
  encapsulation: ViewEncapsulation.Native
})
export class NiceGreeterComponent {
  name: String;
  constructor() {
    this.name = 'Max';
  }
  logMyName() {
    console.log(this.name);
  }
}
```

Listing 3.12: nice-greeter.component.ts: Shows the `name` property.

3.5 Polymer

The *Polymer* project[12] is a library for web components developed by multiple *Google* engineers. Notable is the big catalog of elements[13], ranging from

[12] polymer-project.org
[13] elements.polymer-project.org

versatile *Core Elements* (reduced styling, built for customization) up to the *Paper Elements* which conform the *Material Design Guidelines*.[14] The components are maintained by the *Polymer* team and can be used as well as extended. This library comes in handy when building cross platform apps which are utilizing material design.

In addition to the four common lifecycle callback methods, *Polymer* introduces another handler called *ready*, which is fired when *Polymer* has finished creating and initializing the elementss local DOM (markup inside the element) [14].

There are also concepts available for handling data-binding in a more readable way. When composing components this reduces the complexity and makes it easier to understand already written code.

Furthermore, *Polymer* uses *shady DOM* per default which is a shadow DOM compatible API fast than the shadow DOM polyfill. If desired, it is possible to switch back to the shadow DOM polyfill per default [17]. Currently the library is production ready where developers benefit from out of the box complete components, but have to watch out for updates, if the components change in future versions.

Being backed by Google, like *Angular 2*, the *Polymer* project plays a serious role in the field of web components with many available components and constant updates.

Starting with just JavaScript, going over *X-Tag*, *Bosonic*, *Angular 2* and finally ending up with *Polymer*. There is a whole range of libraries available, from lightweight to fully-featured. Depending on the project requirements, each choice comes with consequences which should fit the project. Short examples for JavaScript-only, *X-Tag* and *Angular 2* are in the appendix to demonstrate the programming style of each library. The thesis project implementation uses *Polymer* and is discussed in chapter 5.

[14] https://material.google.com/

Chapter 4

Technical design

After exploring various web components implementations, the thesis project examines the reusability of web components on a medium-sized web project. Different requirements were set at the beginning and one of them was the usage of forms. Forms provide an opportunity to inspect data-binding between the UI and the data-model because a lot of events are happening on user input which the data-binding mechanism has to handle. It was also part of the requirements to examine library dependencies when building a website with web components. For example, if routing can be handled with web components or an additional library is necessary. Since a medium-sized web project would need more than one component, the composition of multiple components was examined. Mixing the project requirements together, the result is a website where media projects (movies, books, articles, and so on) could be added, edited and managed.

The whole thesis project is divided into two subprojects:

- *keep-api (backend):* contains the database and API for media projects (no web components, written in node.js).
- *keep (frontend):* is a website built with *Polymer* interacting with the backend API.

4.1 Backend

As shown in figure 4.1 the backend consists of two main parts. A database stores the media projects and an API serves as data interface.

MongoDB[1] is used on the database layer. To validate the database documents against a schema, the node module *mongoose*[2] lies on top of the database layer. *Swagger*[3] is used on the API layer as data interface. With

[1] https://mongodb.com
[2] http://mongoosejs.com
[3] http://swagger.io/

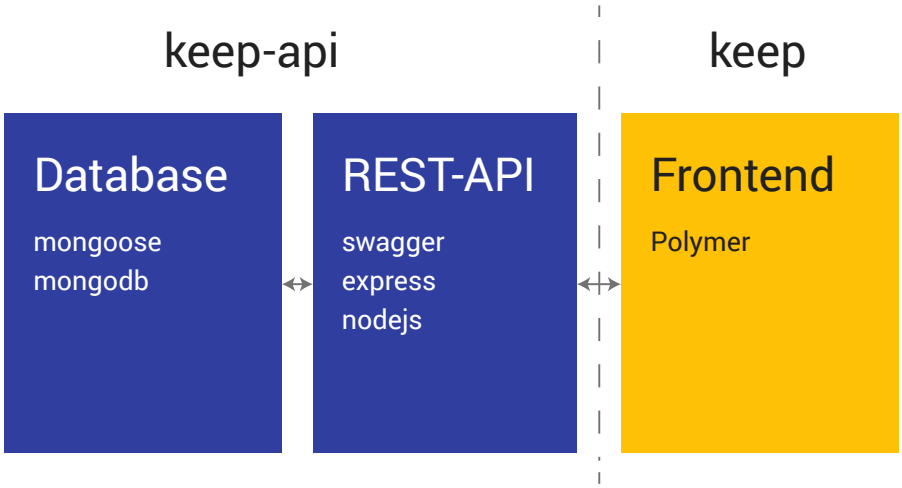


Figure 4.1: Both projects acting together. Keep is the website built with *Polymer*, communicating with the *Swagger* API.

swagger it is possible to create a REST API [1] which handles the communication between the database and incoming requests from the frontend. By utilizing the REST API, it is possible to operate on data via HTTP requests. For example, a `GET /projects/5` request returns all necessary information about the project with the id 5 as a JSON[4] object. This way the backend is completely independent from the frontend.

As the project focus should stay on the frontend, the data model was kept fairly simple. Speaking in *NoSQL* terms, the main collection is called *projects*, and every project can have multiple *tags* which are referenced in an own collection (see figure 4.2).

4.2 Frontend

The website for managing the media projects was built around three main use cases:

- *List projects:* Show all available projects in a list which is sortable (figure 4.3).
- *View single project:* A form for viewing, editing and saving an existing project (figure 4.4).
- *Create new project:* A form for creating a new project.

In the next step wireframes were created. Since reusable web components are the projects focus, it was important to be aware of recurring elements in the wireframes. The wireframe for the new project view is missing on

[4] http://www.json.org/

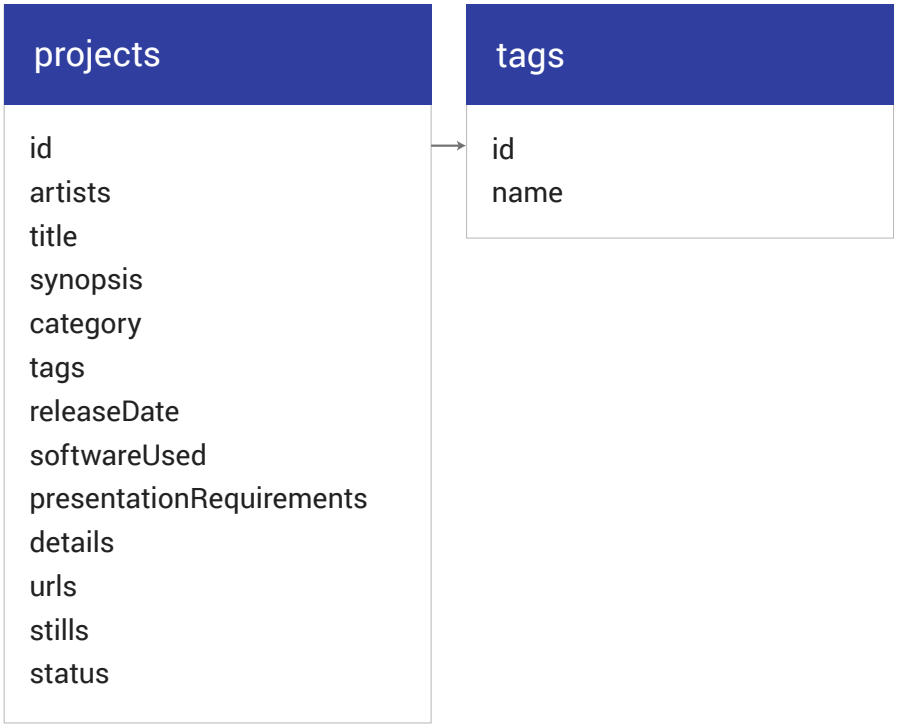


Figure 4.2: mongoDB database object model.

purpose because the single view can be reused just by disabling the input form fields. The result of iterating over sketches was a list of components which will be further discussed in chapter 5. This was a crucial outcome for further decisions concerning the implementation. What already came up in the technical design phase was the importance of communication between components. Multiple form inputs are needed to represent a whole media project, so dependencies between form sections had to be managed. There were no preconditions about server requirements since they depend on the implementation of the frontend.

In summary, the technical design phase highlighted the clear frontend backend separation. The use cases and wireframes were essential to extract (recurring) components.

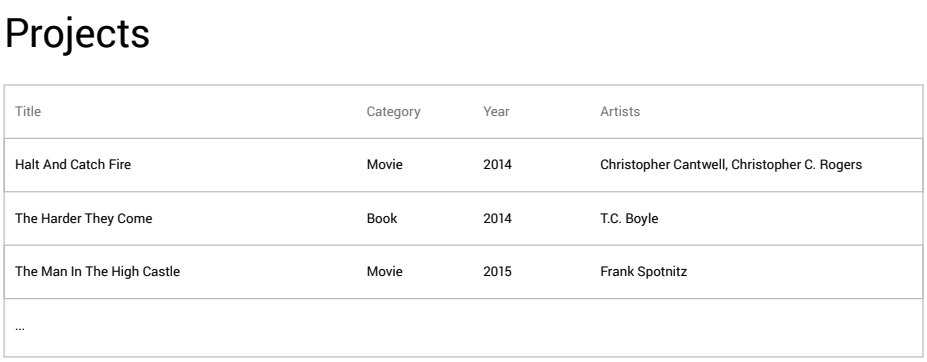


Figure 4.3: Wireframe of the project list view.

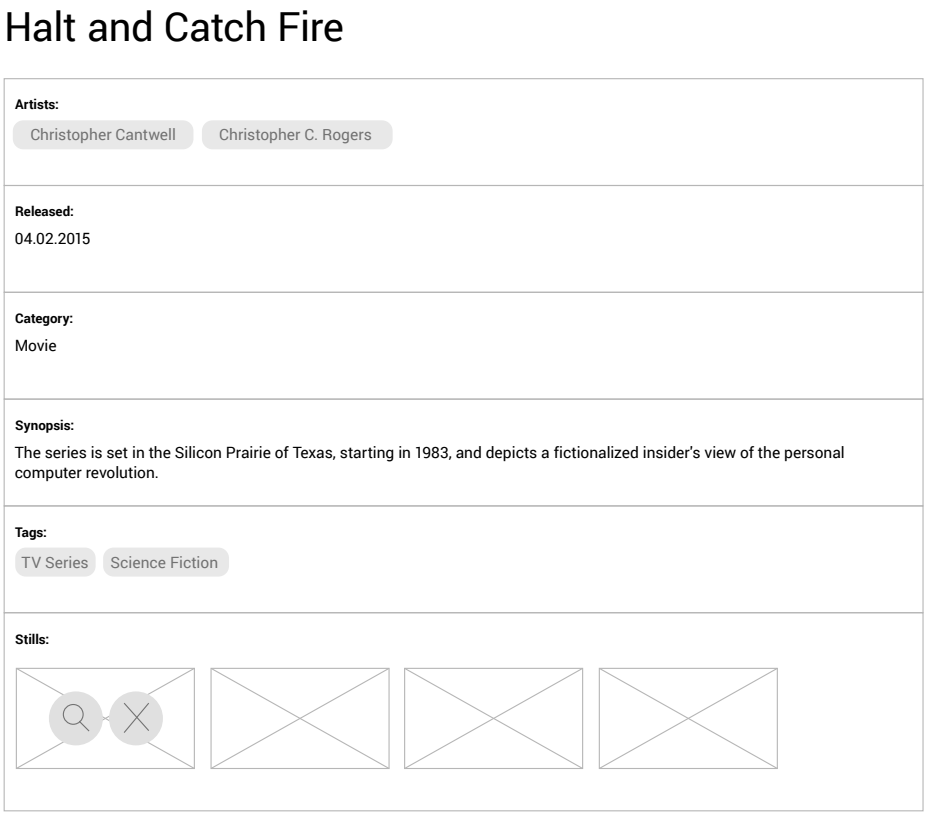


Figure 4.4: Wireframe of a single project view that is also used for creating a new project.

Chapter 5

Implementation

Starting off with the toolchain, the following chapter discusses technologies, frontend and backend implementation of the media archive.

5.1 Toolchain

This topic deserves an own chapter because repetitive tasks can be automated like components can be reused. With task runners like *Grunt*[1] or *Gulp*[2] it's possible to define tasks which are executed on every build of the web project. For example, one task may be minification of CSS styles. Tasks can depend on other tasks. This way they can be chained and one command executes several tasks in a row or concurrently. *Grunt* leans toward the configuration over convention and performs file operations directly on the file system. *Gulp* became popular shortly after *Grunt* was released and heads more in the convention over configuration direction. A *Gulp* task typically starts by reading a bunch of files from the file system into a *vinyl stream*, a virtual file system. Then plugins (like CSS minification) alter the stream and at last the files are written to the file system. This needs less I/O operations than *Grunt*, where files are written after each task and not streamed. Choosing a task runner is also a matter of taste. Both have their advantages and for this project *Gulp* was chosen. *Gulp* is available via *npm*[3], a package manager for JavaScript modules (shipped with every node.js installation) which also plays a central role in the project structure.

Npm is used in both projects, backend and frontend. In the backend it loads node modules necessary for running the API, in the frontend project the node modules are needed just for the build process, not for running the website. The `package.json` keeps basic project infos and lists npm packages on which the project depends. Once a `package.json` is existing in a project

[1] https://gruntjs.com
[2] https://gulpjs.org
[3] https://npmjs.com

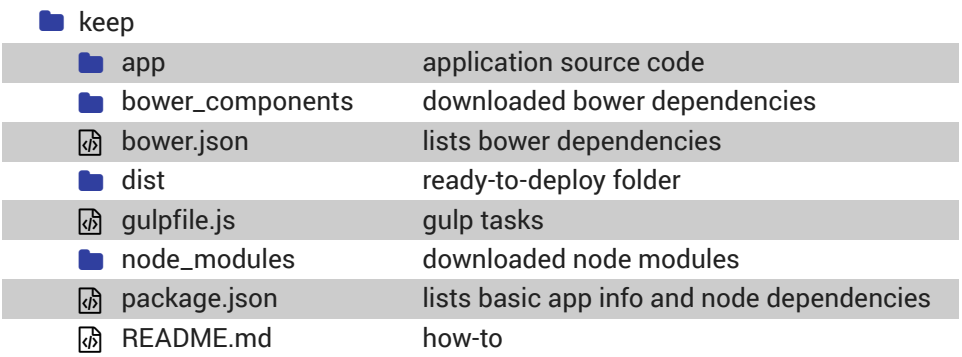


Figure 5.1: Folder structure of Keep, the frontend.

directory, the command `npm install` scans the listed dependencies, looks them up in the npm registry online (managed by npm, Inc.) and downloads them to a `node_modules` folder. On each `npm install` the `package.json` will be checked, new packages downloaded and existing packages updated (depending on the package version listed in the `package.json` file). Once *Gulp* is loaded as node module, a `gulpfile.js` in the project directory lists available *Gulp* tasks which can be executed with `gulp yourtaskname`. In case of the backend there are two main *Gulp* tasks.

- `gulp`: executes the default task which spawns a mongod daemon for running the database and fires up the *Swagger* API[4] for accessing the database.
- `gulp edit`: spawns a mongod daemon and starts *Swagger* in the edit mode, which enables customization of the API via a browser interface.

When deciding to work on the API, it would only take `gulp edit` to launch everything and dive directly into actual work without having to prepare and start everything up at first. Taking a look at the gulpfile in the frontend, there are many more tasks lifting the heavy weight when developing or deploying. Goal of the project structure is to develop everything in the `app` folder, loading dependencies from `bower_components` and `node_modules` and then build a ready-to-deploy version into the `dist` folder (see figure 5.1). The gulpfile is mostly based on the *Polymer Starter Kit*[5] and defines two main tasks.

- `gulp`: builds a ready-to-deploy version into the `dist` folder.
- `gulp serve`: starts a local webserver that serves the *Polymer* app in the browser and refreshes the site in case source files change.

[4] http://swagger.io/
[5] https://github.com/PolymerElements/polymer-starter-kit

All build steps are splitted in tasks which are invoked by those main tasks:

- `clean`: empties the `dist` folder to start with a clean slate before building a production-ready version.
- `copy`: simply copies certain files into the `dist` folder.
- `styles`: compiles Sass[6] files, concatenates, vendor-prefix and minifies CSS.
- `jshint`: lints JavaScript files and reports errors.
- `images`: minifies png, jpeg, gif and svg images.
- `html, polybuild, vulcanize`: *Vulcanize*[7] and *Polybuild*[8] inline and minify imported HTML docs to reduce the number of needed HTTP requests from the client. With HTTP 1.1 this brings a performance improvement. With HTTP/2 this may be obsolete since multiplexing and server push abilities of HTTP/2 may serve multiple small files faster than one big file containing all the imports. This was not evaluated in the thesis, but would be an interesting topic for future research.
- `browserSync`: serves the app locally. Starts a webserver and refreshes the page, if source files change.

Furthermore, some service worker[9] tasks are included for offline-capability of the web app which were not explored in the thesis. Reviewing the toolchain critically, one big dependency is *npm*. The project relies heavily on the *npm* ecosystem, both backend and frontend. Everyone can publish node modules on *npm* resulting in a wide range of packages. However, it's hard to predict the lifetime and maintainance of modules. *Gulp* is also incoporated tightly in the workflow as dependency. *Npm* scripts are emerging in the community to replace *Grunt* or *Gulp* as build tools. Starting from ground up, the next section discusses the backend where the media project data is stored and served via an API.

5.2 Keep-API: backend

Starting from bottom up, this section discusses the implementation of the projects backend.

A mongod daemon process makes sure the database is up and running. To gain access to the database, the whole backend is written in *node.js*.[10] More precisely the lightweight framework *express*[11] is running everything

[6] http://sass-lang.com
[7] https://www.npmjs.com/package/vulcanize
[8] https://github.com/PolymerLabs/polybuild
[9] https://developer.mozilla.org/en/docs/Web/API/Service_Worker_API
[10] http://nodejs.org
[11] http://expressjs.org

above the database layer.

For database transactions a node module called *mongoose*[12] is responsible for validating the schema of database objects. With mongoose a schema can be created like a new JavaScript object. In this case 2 schemes for the 2 collections (projects and tags) were created. When fetching from the database or inserting into the database, mongoose will validate the transactional data against the schema and create JavaScript objects, which are easy to handle in *node.js*. With mongoose it's easier to keep the database "clean" from unexpected data. With mongoose and express it would be already possible to write API methods, which can be used from other applications. For really small applications this would be perfectly fine, but during the research process the API framework *swagger* was explored. *Swagger* is open source software and generates documentation plus a testing interface for a REST API. There's a vast range of integrations for multiple programming languages.[13] In this case the *express* adapter was used. This is possible because the whole API is declarated in YAML or JSON and just references the actual implementation methods, which can be written in any language. For example, the GET method for a single project is declarated in YAML (listing 5.1) and implemented in the referenced controller.

Swagger will look for the method `getProjectById` in the referenced `projects` controller. As visible in the responses, schemas are available for referencing more complex response types. This avoids code repetition since the schemas are defined just once. A nice-to-have would be referencing moongoose models to avoid duplicate declaration of the database object model. The whole API was declarated in YAML and implemented in *node.js*. When starting the API with `gulp edit`, *Swagger* provides a web interface for live-editing the YAML file and generates documentation available for testing and looking up operations (figure 5.2). The `edit` task is watching for file changes on the YAML declaration as well as the referenced controllers and refreshs the API immediately.

For internationalization the npm module *i18n*[14] is used to translate the API. By using the `__("My message")` method, the translated string will be returned. Depending on the received HTTP Header *Accept-Language* the node module *i18n* will fetch the translation from the corresponding JSON file named after the language. Additionally, other methods are possible for determining the language to be used.[15] With the data stored in the mongoDB and *Swagger* running as REST API the backend is complete and ready to serve project data for the frontend.

[12] http://mongoosejs.com/
[13] http://swagger.io/open-source-integrations/
[14] http://npmjs.com/package/i18n
[15] https://www.npmjs.com/package/i18n

```
/projects/{projectId}:
  x-swagger-router-controller: projects

  get:
    tags:
      - "project"
    summary: Get project by id
    operationId: getProjectById
    parameters:
      - name: projectId
        in: path
        description: id of project
        type: string
        required: true
    responses:
      200:
        description: Success
        schema:
          $ref: "#/definitions/ProjectSchema"
      400:
        description: Bad request, please refer to Swagger Docs
      500:
        description: Internal server error
      501:
        description: Not implemented
        schema:
          $ref: "#/definitions/ErrorResponse"
```

Listing 5.1: Excerpt of *Swagger* API definition. HTTP GET method for fetching a single project (`http://api.url/projects/13`). The `operationId` references the function inside the `projects.js` controller listed as `x-swagger-router-controller`.

5.3 Keep: frontend

Since the backend hasn't used web components at all, the main part of the implementation was the frontend. It was implemented in HTML, CSS and JavaScript using the *Polymer*[16] library for web components. The communcation with the *Swagger* API is handled via Polymers `iron-ajax` web component.[17]

When looking at the wireframes, some recurring elements appear already (see figure 5.3), but not only these are needed. Also a layout grid, buttons, form elements and whole lots of other elements are needed for building the website. So a nice-to-have would be a pool of already available web components. After studying the available libraries for web components, the library of choice was *Polymer* because of several reasons:

[16] https://www.polymer-project.org/1.0/
[17] https://elements.polymer-project.org/elements/iron-ajax

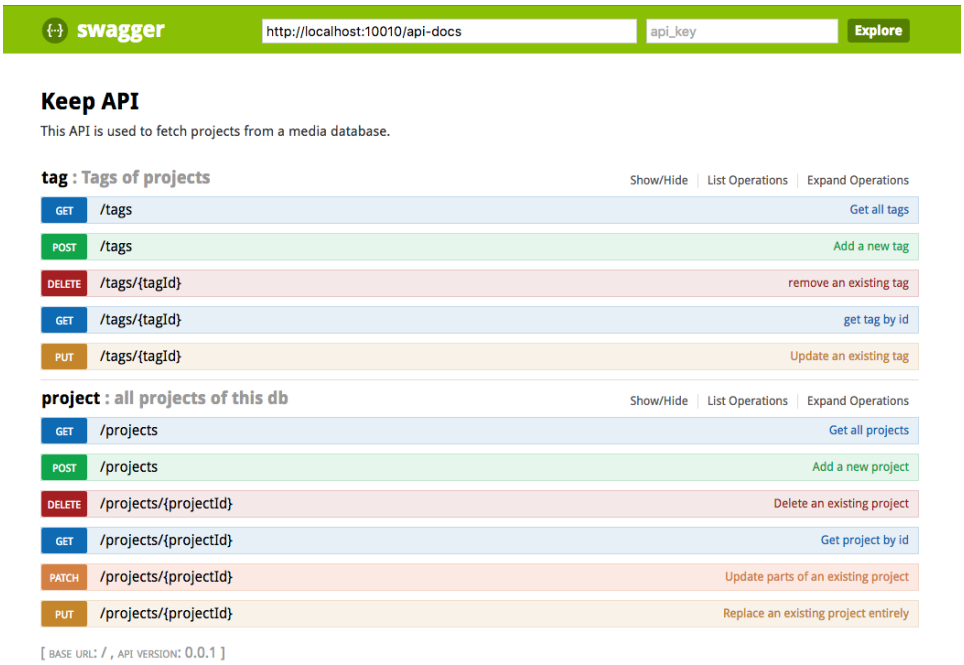


Figure 5.2: Generated *Swagger* documentation from the API definition in YAML. All operations are expandable and can be tested here.

- *More than syntactic sugar: X-Tag* eases the registering compared to native JavaScript but *Polymer* goes one step further and provides a flexible event system.
- *Built on standards:* Under the hood *Polymer* utilizes the web components standards, which I personally think is very important because standards are needed for a long term solution.
- *Element catalog:* By far the most advanced and rich catalog of available elements[18] compared to other smaller frameworks.
- *Docs and community:* Excellent documentation as well as a large community online.
- *Production ready:* At the time of writing this thesis *Polymer* has reached version 1.1 which is production ready.
- *Customizable toolchain:* All steps in the toolchain are located in a projects gulpfile. Read more about the toolchain in section 5.1.

For sure these reasons are debatable but *Polymer* fulfilled the requirements at the time of implementation and suited well for exploring the field of reusable web component concepts.

[18] https://elements.polymer-project.org/

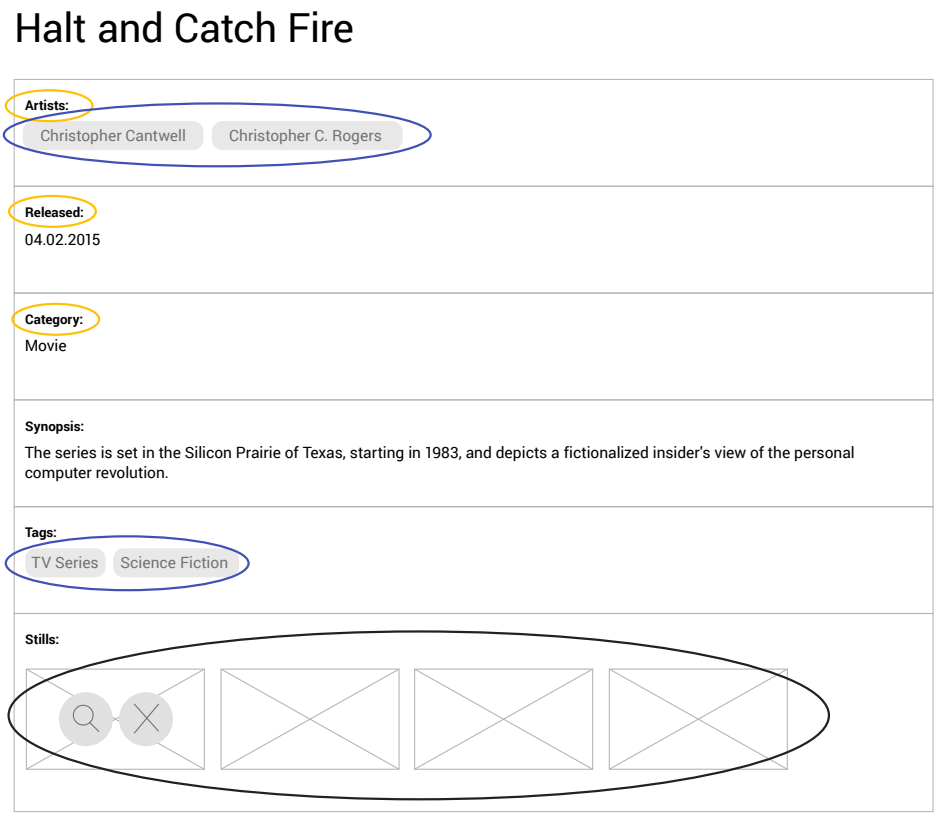


Figure 5.3: Recurring elements in the wireframe are highlighted with ellipses.

As a first boilerplate the *Polymer Starter Kit*[19] was used for an application structure. Once set up at the beginning of the project, the workflow for frontend development is the following:

- *Start gulp:* `gulp serve` takes the source files from the `app` folder and starts up a local HTTP server for development.
- *Develop components:* by creating and compositing components in the folder `app/elements`, the site structure will be built.
- *Deploy:* `gulp build` runs various tasks on the source files (more detail in section 5.1) and creates a ready-to-deploy version in the `dist` folder. The production version needs no server side frameworks. It's just HTML, CSS and JavaScript.

Figures 5.4 and 5.5 provide a structural overview, where to find which part

[19] https://github.com/PolymerElements/polymer-starter-kit

of the code. Taking the `index.html` as a starting point, several resources are loaded:

- The web components polyfill,
- `elements.html` (via HTML imports), which contains the list of all used components,
- `shared-styles.html` and `app-theme.html` for application-wide styles,
- web components used in the `<body>` and the
- `app.js` for bootstrap JavaScript.

Inspecting the `elements.html` more in detail, at first *Polymer* web components are imported. Those components are imported via *Bower* to provide a convenient way of updating components. With a single line (e.g. `bower update iron-ajax@1.2.0`) an element can be updated to a distinct or latest version. Application-specific components are not available via *Bower*, but located in the `app/elements` folder and imported also in the `elements.html`. Client-side routing happens in the `routing.html` with *page.js*.[20] The Polymer team is working on a more advanced routing component called *app-route*[21] which is not yet implemented in *Keep*. Application-wide styles inside the `shared-styles.html` are *not* available in all components per default to avoid cross-scope styling conflicts. A component has to import the `shared-styles.html` file, then all the styles are available inside. The color scheme and other theme-dependent styles are located in the `app-theme.html`.

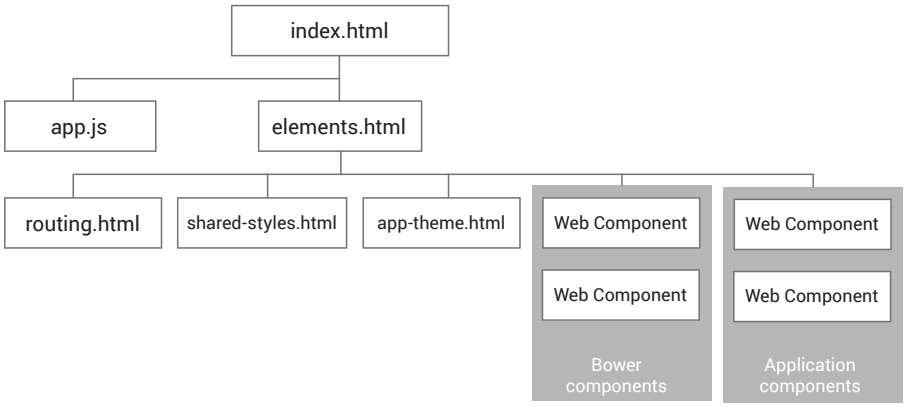


Figure 5.4: HTML Import structure of the frontend.

[20] https://visionmedia.github.io/page.js/
[21] https://elements.polymer-project.org/elements/app-route

keep	
app	application source code
elements	web components (self built, application specific)
images	application-wide images
scripts	application-wide scripts
styles	application-wide styles (css variables)
index.html	starting point
bower_components	bower dependencies
bower.json	list of bower dependencies
dist	ready-to-deploy folder
gulpfile.js	gulp tasks
node_modules	node modules (just for development, not in production)
package.json	lists basic app info and node dependencies
README.md	how-to

Figure 5.5: Folder structure of the frontend.

5.3.1 Look and feel

Keep uses a wide range of components from the *Polymer* Element Catalog.[22] There are bare-metal elements available with very little styling that allow high customization (*Polymer* core elements). A whole category of elements is dedicated to the *Material Design Guidelines*[23] for a consistent look and feel across various devices. Those elemens are importing *Polymer* core elements and enhance them with styling and behaviour. Studying the *Material Design Guidelines*, it's possible for everyone to create and contribute to a range of very well matching components. Following an already existing visual guide, there was no need to reinvent the user experience for the thesis project. This lead to a great benefit in implementation time and consistency because the screendesigns went from scribbles (where to put which components) directly into a functional prototype in the browser. The material design guidelines are proposing clear solutions for many user interactions in the projects frontend. Various tools around material design, like the material palette[24], allowed quick customization of *Keep* and made it possible to combine components in visual harmony (figures 5.8 and 5.7). Many resources for visually designing components are available online, but more on this in chapter 6.

[22] https://elements.polymer-project.org/
[23] https://material.google.com/
[24] materialpalette.com

Figure 5.6: Keep: Homepage, using three `keep-tile` components for navigation.

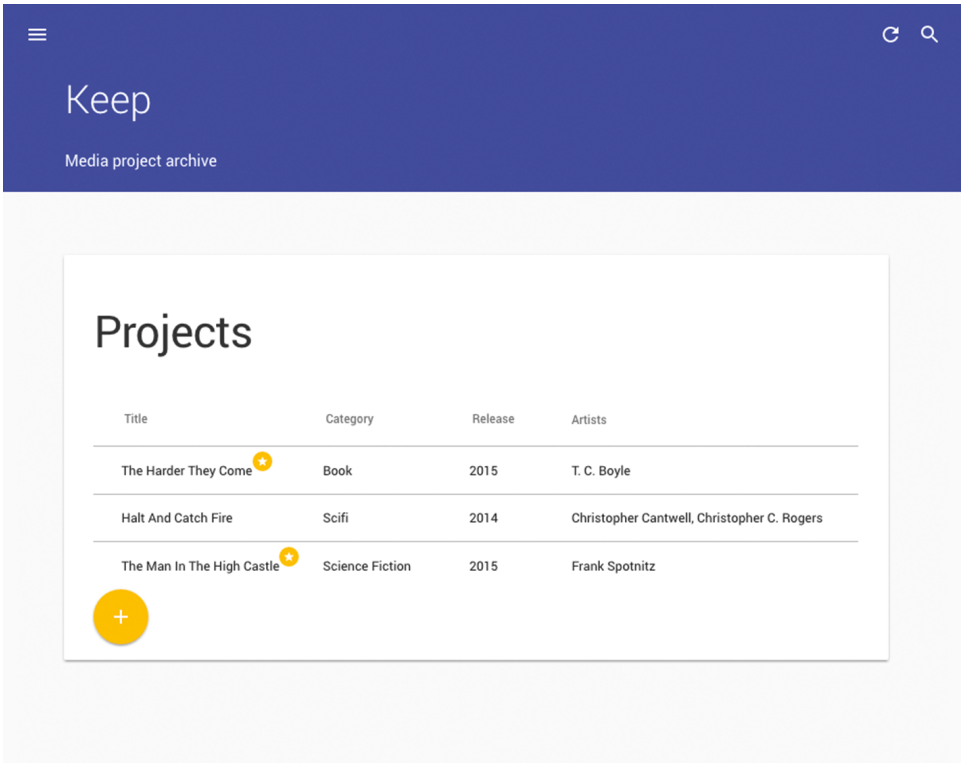


Figure 5.7: Keep: Project overview using the `paper-datatable` component for listing all projects.

```
yo polymer:el <component-name>
```

Listing 5.2: Command for creating a new component. Uses the *Yeoman* generator for *Polymer* projects.

```
:host {
  --keep-tile-background: #e0e0e0;
  --keep-tile-icon-color: #424242;

  --keep-tile-label-color: #ffffff;
  --keep-tile-label-background: #999999;

  --keep-tile-action-btn-color: #ffffff;

  --keep-tile-icon-width: 50%;
  --keep-tile-icon-height: auto;

  --keep-tile-margin: 2px;
}
```

Listing 5.3: CSS Custom Properties of `keep-tile`.

5.3.2 Components

Keep is essentially a single-page application where the content is swapped with *page.js*. To separate concerns, for each page a component was created. It also aids offline caching, if the page content is bundled in one component. The components used throughout the application are mostly material design components by *Google*. For convenience, the *Google Chrome* extension *Polysearch*[25] was used for searching the *Polymer* element catalog.

With the help of a *Yeoman* generator[26] the project was initially scaffolded and new components can be added with a single command, which creates a new HTML file for the component and it can be chosen, if the component should be imported in the projects `elements.html` at once.

As a best practice for styling, all CSS Custom Properties from a component are located on top of the component, are prefixed with the component name and have default values (listing 5.3). The following sections discuss the most important created web components.

keep-tile

Purpose of the `keep-tile` component (figure 5.9) is a clear call-to-action section that is more prominent than a button. When using a `keep-tile`

[25] https://chrome.google.com/webstore/detail/polysearch/gchibjlnlbpgcfjpbebnlecbbjndiidj
[26] http://yeoman.io

```
<keep-tile label="Add a new Project" icon="add-circle" class="
    tile--add-project" link="/projects/new"></keep-tile>
```

Listing 5.4: Usage of `keep-tile`.

```
.tile--add-project {
  --keep-tile-icon-width: 40%;
  --keep-tile-background: #DCEDC8;
  --keep-tile-label-background: #8BC34A;
  --keep-tile-icon-color: #689F38;
  --keep-tile-label-color: var(--primary-text-color);
  --keep-tile-action-btn-color: var(--primary-text-color);
}
```

Listing 5.5: Styling via CSS variables in the `app-theme.html`. These values get passed into the `keep-tile.html` where default values for the CSS variables will be overwritten.

(listing 5.4), all attributes are optional. If the `label` is missing, just the icon will be shown. In case the `icon` attribute is missing too, a default icon from the `iron-icons`[27] will be shown. This component appears on the homepage of the project (figure 5.6) and is reusable in other projects since no application-dependent data is stored. The styling customization from outer components that are using `keep-tile`, is shown in listing 5.5.

keep-label

For components outside Polymers element catalog the `keep-label` component (figure 5.10) mimics the appearance of a material design form label. The component has no special functionality, it just shows a nested text in the color of the CSS Custom Property `--secondary-text-color`.

paper-datatable

As there was no implementation for material design data tables[28] in the *Polymer* element catalog, the project uses `paper-datatable`, a web component created by David Mulder[29] for showing a list of projects (figure 5.11). It is open source and available via *Bower*. The `paper-datatable` was wrapped in another component: `expandable-list` where a `iron-ajax` component fetches the project data from the backend and populates the `paper-datatable`, which is highly customizable. Date formatting is done

[27] https://elements.polymer-project.org/elements/iron-icons?view=demo:demo/index.html
[28] https://material.google.com/components/data-tables.html#data-tables-structure
[29] https://github.com/David-Mulder/paper-datatable

```
<expandable-list data-url="http://localhost:10010/projects">
  </expandable-list>
```

Listing 5.6: Usage of `expandable-list` which is a wrapper element for `paper-datatable`.

```
<keep-chip is-deletable value="{{item.name}}"></keep-chip>
```

Listing 5.7: Usage of `keep-chip` with delete icon and a data-bound value.

```
:host {
  --keep-chip-background: #e0e0e0;
  --keep-chip-color: #424242;
  --keep-chip-delete-icon-color: #e0e0e0;
  --keep-chip-delete-icon-background: #a6a6a6;

  --keep-chip-hover-background: var(--keep-chip-color);
  --keep-chip-hover-color: #fff;
  --keep-chip-hover-delete-icon-color: var(--keep-chip-color);
  --keep-chip-hover-delete-icon-background: #fff;
}
```

Listing 5.8: CSS Custom Properties for a `keep-chip`.

with *moment.js*[30] and if the project release date is the current year, a `paper-badge` will be added. The `expandable-list` simply uses the REST API URL as attribute (listing 5.6). Since the `expandable-list` contains application-specific data, its reusability scope is limited to this project.

keep-chip

This is an implementation of basic material design chips[31] (figure 5.8), which is used for artists and tags. If the `is-deletable` attribute is set, a delete icon will be added (listing 5.7). A click on the icon fires a `delete` event with the value of the chip. The chip itself doesn't contain any application-specific data. Managing a list of chips is done in the `keep-tags` component (section 5.3.2). Various CSS Custom Properties allow styling from the outside (listing 5.8).

[30] http://momentjs.com/
[31] http://www.google.com/design/spec/components/chips.html

```
1  <keep-tags id="piTags" tags="{{project.tags}}"></keep-tags>
```

Listing 5.9: Usage of `keep-tags` with a data-bound array of tags.

```
1  <template is="dom-repeat" items={{project.stills}}>
2    <keep-thumb class="keep-thumb" src="{{item}}"></keep-thumb>
3  </template>
```

Listing 5.10: Usage of `keep-thumb` for displaying all project stills.

keep-tags

When designing a component, the decision has to be made whether it knows about the application-specific API or not. An illustrative example is the management of tags in the project detail (figure 5.12). The goal was to create a reusable component for managing a list of tags and avoiding the fact that every component has to know about the backend API. After several iterations the communication diagram in figure 5.13 evolved. A chip just stores the name and shows the delete icon which fires a *delete* event on click. The `keep-tags` component stores a list of `keep-chip` elements and has a form input for adding a new one. If a new tag gets added or deleted, the corresponding event fires and the internal list of tags will be updated (listing 5.9). Last but not least, the `project-info` observes the `keep-tags` and is the only component which knows about the API for persisting the tags and fetching an id for a newly added tag. For artists it isn't possible to use the exact same `keep-tags` component because the data structure of an artist is different to a tag.

keep-thumb

This component shows an image thumbnail with additional controls for fullscreen and delete events (figure 5.14). As `src` attribute an image path is passed. Internally, `iron-image` component displays the image as thumbnail and an overlay with the buttons is added. The buttons just fire events `delete` or `fullscreen` with the image path in the event detail, so listening parent components can immediately work with the event data. `keep-thumb` is used in the project detail view (listing 5.10).

project-info

Being the most comprehensive component of the frontend, the `project-info` handles the project detail view (figure 5.8). It is designed to act as mediator for its containing components (mostly form elements) and communicates with the backend API to persist the entered data (listing 5.11). This way the

```
1   <project-info data-url="http://localhost:10010/projects" tag-url="
      http://localhost:10010/tags" id="projectInfo"></project-info>
```

Listing 5.11: Usage of `project-info` providing backend URLs for projects and tags.

child components can be kept application independent because they just fire events and then the `project-info` component takes care of application specific actions. Once loaded, the property `project` reflects the whole project as an object. If any form fields change, the `hasChanged` property will be set to true and the *Save* button indicates a non-persistent state to the user. Depending if the project is existing, a *POST* or *PUT* AJAX call will be executed when clicking the *Save* button. When altering tags or stills, a *PATCH* operation will ensure persistence in the database. All API operations are handled via *iron-ajax* components.

As the `project-info` component is also used for creating new projects, it listens to an event `paramsset` fired by *page.js*. If the event contains a project id, an AJAX request will fetch the existing project from the API. If not, an empty form will be presented and the project will be created at the first click on the *Save* button.

Focusing on a single component, it is tempting to forget about the reusability in other applications. It strongly depends on the component purpose. Some of them will turn out as application-dependent, but the amount of those should be kept on a minimal level. Further discussion on designing an independent component with certain benefits, consequences and limitations follows in chapters 6 and 7.

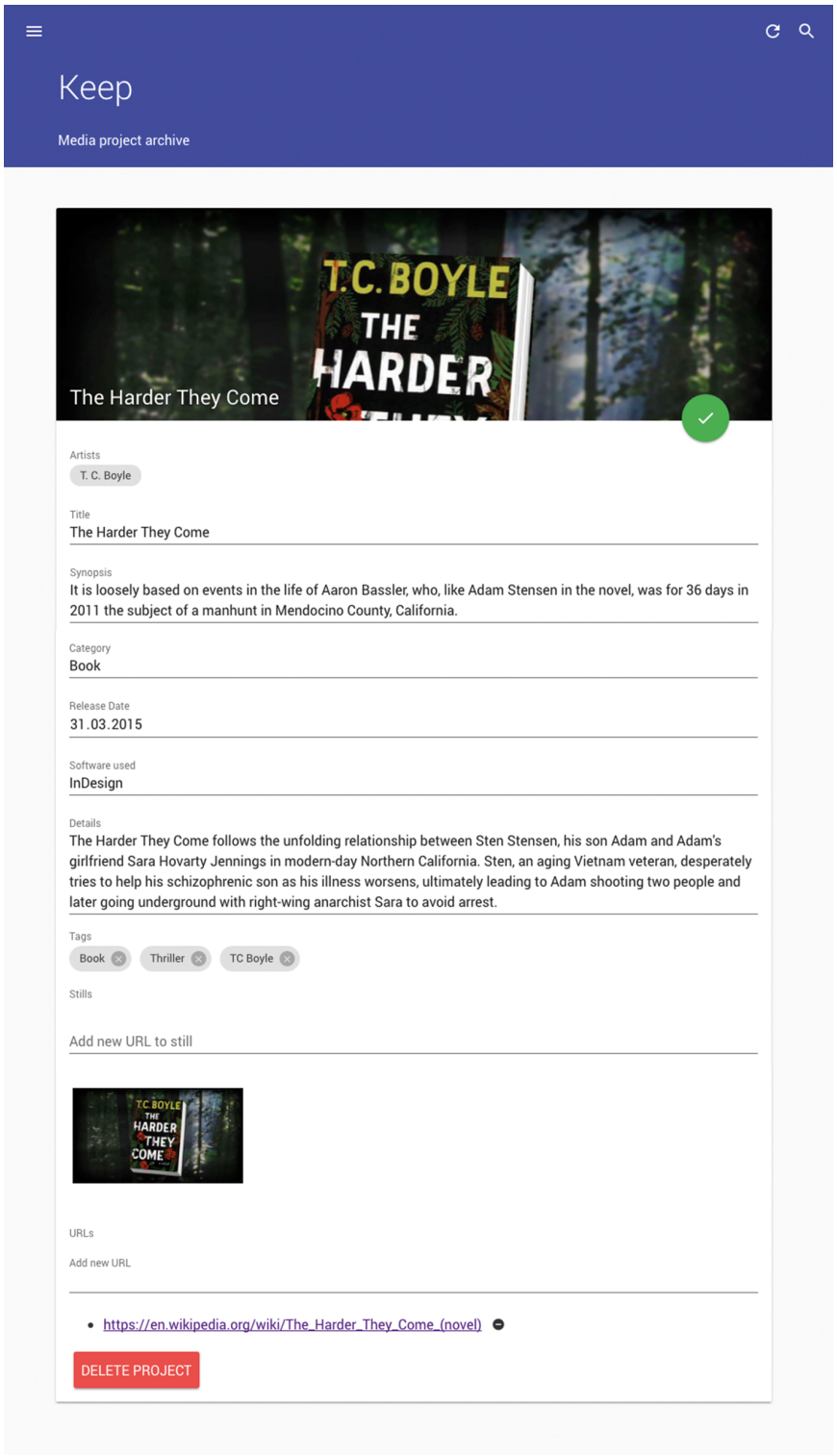


Figure 5.8: Keep: Editable detail view of a project. The `project-info` component serves as a mediator for the form elements to keep them application-independent.

Figure 5.9: Example of a `keep-tile` component.

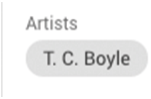


Figure 5.10: Example of a `keep-label` plus a `keep-chip`.

Figure 5.11: `paper-datatable` component with additional `paper-badge` elements.

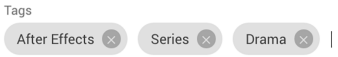


Figure 5.12: `keep-tags` component: manages a list of multiple `keep-chip`.

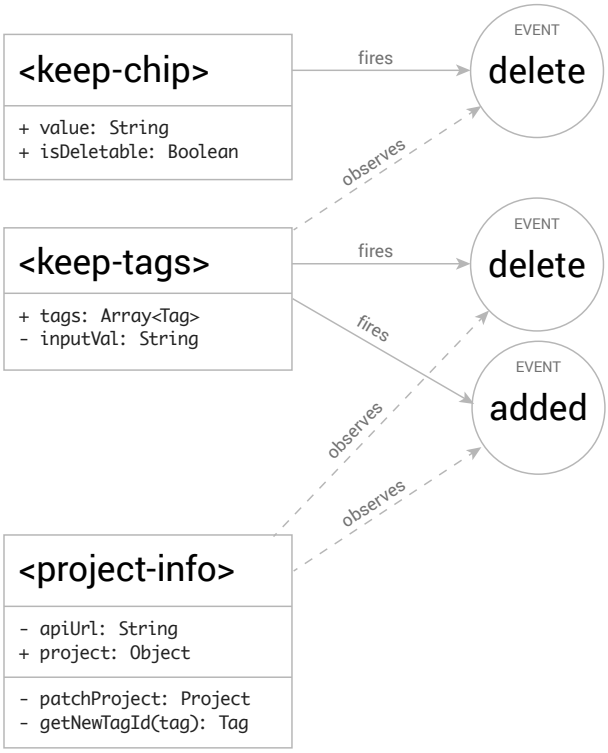


Figure 5.13: Event diagram for tag management inside the project detail.

Figure 5.14: `keep-thumb` component: gets an image as argument and displays it with additional fullscreen and delete controls.

Chapter 6

Architectural design concepts

As mentioned in chapter 3, there are many libraries and frameworks around web components. A framework lifetime is hard to predict and it is often uncertain if the development still continues in a year or two. The funding may stop or more performant competitors will overtake the chosen framework. The web landscape is changing fast and this is why it is important to extract and formalize concepts from practical implementations. Interoperability between components should be possible without depending on the implementation behind the components interface. This chapter deals with one research question of this thesis: *What are the important aspects when designing a web component regarding architecture and reusability?*.

6.1 A modular mindset

One of the first things to consider when getting into modular architecture is the mindset. Web architects can shift their mindset from creating a predefined, fixed structure to creating smaller modules which are usable in different contexts, changed, moved and assembled to establish a complete environment. Creating flexible modules *can* be complex, but flexibility doesn't necessarily come with complexity. Small components can be used together in order to represent more complex functionality, which aligns with the concept of *atomic design* [10]. With less code and less functionality a component is easier to maintain.

The purpose of a component should be clearly defined and limited. If the component has too many purposes ("How about we use component A for purpose X?"), it can get complex and bent out of its original shape. Considering a new component (which can be composed out of other components) is one possibility of avoiding that dilemma or as Doug McIlroy said in 1978 [4, p. 1902–1903]:

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.

Furthermore, it is important to *limit the reusability scope of the component*. This aspect is closely related to the purpose of the component. During the implementation process of the thesis project it was helpful to split components in two categories:

- *application-independent* components are general purpose elements used across websites (e.g. buttons, inputs, icons) and
- *application-dependent* components handle application-specific data structures and functionality.

Application-independent components have a larger reusability scope and can be used across different web projects. For application-specific behaviour the application-dependent components come into play, which can be composed out of application-independent ones and simply add the needed functionality to keep independent components independent. These are still valuable since application-dependent components can be reused *within* the application. For example, a component for showing user info can occur more than once across the same application. Striving for independent components is favourable, but sometimes application specific data structures occur and application-dependent components are necessary.

6.2 Managing content in components

A main purpose of websites is the visualization of content. The content should be presented in an appropriate, easily understandable way for the reader. Web components can be used as medium to communicate content to the recipient. Instead of directly presenting data in rendered HTML, the raw data can be passed to a component and the component decides about the presentation. This supports the separation of content and presentation and helps in terms of accessibility.[1] The `<template>` HTML element can be used for creating templates inside a component where the passed data will be inserted. For example, if a `<v-card>` component prints a business card, somehow the data has to be passed to the `<v-card>`. A common technique are HTML data attributes (listing 6.1). Now the attributes can be fetched and shown inside the component with a combination of the `<template>` element and JavaScript (listing 6.2). But since HTML tags are nestable, it's also possible to nest tags (listing 6.3).

Inside the `<v-card>` component (listing 6.2) a `<content>` tag would have to exist as insertion point for nested tags. The `<skype-call-button>` and

[1] https://www.w3.org/TR/WCAG20-TECHS/G140.html

```
1  <v-card firstname="Klaus" lastname="Fischer"></v-card>
```

Listing 6.1: Using HTML data-attributes to pass data to a component.

```
1  <template id="v-card-template">
2    <strong>Name</strong>: <slot name="firstname"></slot> <slot name="
     lastname"></slot><br>
3  </template>
```

Listing 6.2: Template part of the v-card component. The slots define the render spot of the passed data.

```
1  <v-card firstname="Klaus" lastname="Fischer">
2    <skype-call-button username="skypeuser"></skype-call-button>
3  </v-card>
```

Listing 6.3: Nested components. The `v-card` template has to specify an insertion point to render nested content, otherwise it won't be rendered. In listing 6.2 the `<skype-call-button>` won't be rendered because the template is missing a `<content>` tag.

other DOM inside the `<v-card>` tag would be placed at the `<content>` insertion point. Theoretically, a really large DOM tree could be placed inside the `<v-card>`. Pratically, it can happen that the `<v-card>` isn't able to render a large DOM correctly because it is not designed for it. Too much content could be cut off, if the `<v-card>` is size-limited with CSS. With documentation and a clearly defined purpose (see section 6.1) this problem can be tackled. The `<content>` element is already deprecated[2] and replaced by the `<slot>` element. The `<slot>` element[3] succeeds the `<content>` element to allow multiple named insertion points instead of one insertion point with `<content>`. With slots the user of a component decides *which* content is passed into the component and the component itself decides *where* to place the received content. In case the author Ben writes the page layout and passes content into the slots, Lucy, who is developing the component, decides where the slot content will be placed. Benefits and consequences about that circumstance are discussed in section 7. For practical content insertion patterns in JavaScript refer to chapter 5 or Addy Osmani's - Learning JavaScript Design Patterns (chapter: *Modern Modular JavaScript Design Patterns*) [5].

[2] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/content
[3] https://webkit.org/blog/4096/introducing-shadow-dom-api/

6.3 Communication between components

A communication between components has several similarities to a conversation between people. To ensure a productive conversation, the participants have to find a common technique of understanding each other (language, gestures, drawing). Someone expresses a feeling or desire, which will (hopefully) be noticed and other members of the conversation react or respond to it. The same cultural background of all members and correct understanding are not guaranteed. Therefore, it is important to settle on a comfortable way of communicating. The communication issues also map to web components.

The internal implementation of a web component can be written vanilla JavaScript or use a library like *Polymer* or similar. Either way, the components have to communicate with each other and express their state, so multiple components can act together. Focusing on the reusability of web components, one architecture goal is to strive for *loose coupling* (see [2, p. 36]) between components, so dependencies are minimized and a component can be reused in another project. The three most obvious ways to communicate are:

- *CSS classes* which express styling-related information,
- *HTML data attributes* to indicate logical functionality or logical state and
- *JavaScript events* for emitting changes and objects rather than simple values.

Sticking to that categorization can avoid confusion when altering the element, so the removal of a CSS classes concerns only the visual styling and won't break the components functionality. For example, a rendered TODO list could communicate in multiple ways (listing 6.4). If a `<todo>` changes state from *undone* to *done* or vice versa, the `<todo>` component sets its data-attribute `is-done` and addtionally emits an event. The event can contain information about the changed `<todo>` element. There are no naming conventions for CSS classes or data attributes, namespacing or which additional data an event should emit. So the developer has the freedom of choice how to name everything and what data will be emitted when a event fires. However, it is possible to consider proposals, guides or best practices for naming.[4] Existing components can be analyzed and with an increased usage of web components possibly more best practices will emerge.

Using the *shadow DOM* (see 2.2.4) enables encapsulation of scripts and styles which helps avoiding scope conflicts. JavaScript variables are scoped to the component and functions act as API of the component. More concepts about styling are discussed in section 6.5.

[4] http://getbem.com, https://smacss.com/, https://www.polymer-project.org/1.0/docs/devguide/events

```
1  <todo-list class="todolist--large" todo-counter>
2    <todo is-done>Read</todo>
3    <todo>Understand</todo>
4    <todo>Write</todo>
5  </todo-list>
```

Listing 6.4: Todo list with web components. Uses CSS classes for expressing styling-related states and HTML data attributes for logical functionality. The `todo-counter` marks the appearance of a counter for open tasks and the `is-done` attribute denotes a task as completed.

6.3.1 API design

When building a component the API makes it possible for fellow components to access data and observe changes. It should be easy for other components (and developers) to understand the API and use the component without extensive study of the documentation. Marijn Haverbeke [3, p. 200] recommends 3 aspects when designing an API for JavaScript modules that can also be mapped to web components:

- *Predictability:* if it is possible to predict how the interface works, looking up the documentation isn't necessary and it's easier to build a mental model of the module.
- *Composability:* functions should do a single, clear thing. Returning strings, arrays and standard JavaScript data structures make it easier for other modules to work with the return values.
- *Layered Interfaces:* depending on the usage of the module, it may be necessary to expose a simple *high level* interface (e.g. `todo.setDone()`) or a *low level* interface with additional customization options (e.g. `todo.setStatus(status)`). Both can be done efficiently if the high level functions are utilizing the low level functions.

Using an already built web component, this highlights the importance of documentation. With a documented component API the developers who are using the component have to spend less time studying the behaviour but can refer to the documentation. Tools like JSDoc[5] or the *Polymer* toolchain are able to generate documentation from commented source code.[6] Staying consistent with naming and providing a documented API improves the reusability factor of a component and eases communication with other components.

[5] http://usejsdoc.org/
[6] https://www.polymer-project.org/1.0/docs/tools/documentation

6.4 Semantics and accessibility

With the existing set of standardized HTML elements[7] it is possible to write semantically meaningful code. Elements like `<video>`, `<button>`, `<audio>` are very distinct and it is obvious which content they represent. Looking at elements like `<header>`, `<section>` or `<article>` the name alone isn't always sufficient enough to tell the exact purpose of the element. This is intentionally, since these elements aim are used for structuring a HTML document and structures vary from site to site.

6.4.1 Element naming

With custom elements (see section 2.2.1) the semantic meaning is even more endangered because the rules of custom element naming[8] are loose. `<asdf-elem>` is as valid as `<data-list>` without providing much semantic meaning. Semantics can get even more fuzzy with inheritance, so developers are responsible for proper naming and documentation of the elements. There are not just bad examples, custom elements can also improve the understanding of content like in listing 6.4. A common practice to avoid naming conflicts are prefixes in the element name. In case of *Polymer* all elements following the material design are prefixed with `paper-`. With scoped styles all CSS classes inside a component only affect elements inside the component and won't bleed into other components. This results in less naming conflicts and best practice CSS naming concepts like BEM[9] or SMACSS[10] may lose their importance. What will be more important is the naming of *CSS custom properties* [21] (often times just called *CSS variable*. If a component uses a CSS variable for the font color (e.g. `--todo-color`), parent components can assign a new value to the variable and the child component uses this value. A problem can occur, if a parent component assigns a value to a CSS variable `--background-color` and this variable is used differently in 2 child components. This can lead to unexpected results as unintentionally other components are affected. Prefixing the CSS variable with the element name is a possible solution (e.g. `--todo-background-color`).

6.4.2 Accessibility

In order to meet the WCAG 2.0 guidelines[11] several enhancements can be made at custom elements. As with common HTML elements it is possible to add `tabindex`, `aria` attributes and the `role` attribute. A `tabindex` can

[7] https://developer.mozilla.org/en-US/docs/Web/HTML/Element
[8] https://www.w3.org/TR/custom-elements/#valid-custom-element-name
[9] http://getbem.com/
[10] https://smacss.com/
[11] https://www.w3.org/TR/WCAG20/

make a component focusable, the `role` and `aria` attributes enhance its semantics. Adding an event listener for the `keydown` event enables keyboard accessibility. For visual accessibility it helps to use relative size units (`em` or `rem`) in CSS because everything will scale according to the `font-size` property. High color contrast is also part of the accessibility guidelines. Various tools like the Chrome Extension High Contrast[12] allow testing for enough contrast and expose weak spots in black and white mode. Also z-index conflicts may arise when nesting components. Encapsulated components don't know about z-indices of their child components, so it's possible that a child component with a low z-index value may be invisible due to the parent element with a higher z-index. Jason Strimpel proposes an application-wide z-index manager [6, p. 33] as one solution.

6.5 Visual design

One amazing power of the web is the ability to visualize content with very little code. This power comes with great responsibility because changing just a few characters in the markup or styling may result in a completely different rendering. From small to complex, there are many great resources[13] about building component-driven design systems. A main goal is the creation of multiple small components which are working together in a design system to create a uniform user experience. The result is a pattern library where components can be explored and combinations can be tested.

Consistency across components is a big challenge, not just in terms of how a component looks, but also how it feels and the whole user experience of a component. Components have to be flexible in application and responsive to their environment. Sooner or later, in the components design process the styling and visual behaviour of a component has to be defined. The visual design of a component may restrict the usage across different projects because it won't fit visually in another environment. There are several ways of tackling the challenges of modular design.

First of all, at the beginning of the design process it is important for the designers to establish a common language to collaborate in a common domain [16]. During the concept phase of a component, it also makes sense to limit the scope and functionality, like discussed in section 6.1, to minimize the complexity of components. Afterwards, in the implementation and iteration process tools like *Pattern Lab*[14] reflect changes in a component to the whole design system. For designers and developers it is easier to test and

[12] https://chrome.google.com/webstore/detail/high-contrast/djcfdncoelnlbldjfhinnjlhdjlikmph?hl=en

[13] http://airbnb.design/building-a-visual-language/, http://atomicdesign.bradfrost.com/, http://styleguides.io/books.html

[14] http://patternlab.io/

monitor the impact of a change.

For *application-independent* components the *Polymer* team has its own strategy to develop their elements catalog.[15] The *Iron Elements*, which are functional core elements, provide very little styling and allow extensive customization options. *Paper Elements* use these *Iron Elements* and style them according to the *Material Design Guidelines*. The idea is to create bare-metal functional components and separately themed components which are making use of the unstyled ones. This concept is similar to the *layered* API approach discussed in 6.3.1. With this strategy the components are more sensitive to updates since an update of the core element can result in an additional necessary update of the styled element.

In the implementation phase the styles of a component are scoped to the component which means they are available *just for that component* and nobody else in the design system. One way to apply the *DRY*[16] principle is declaring *global styles* which can be imported and used by components. Additional *component-specific* styles can be added inside the component without affecting other components (figure 6.1). This comes with benefits and consequences discussed in section 7.

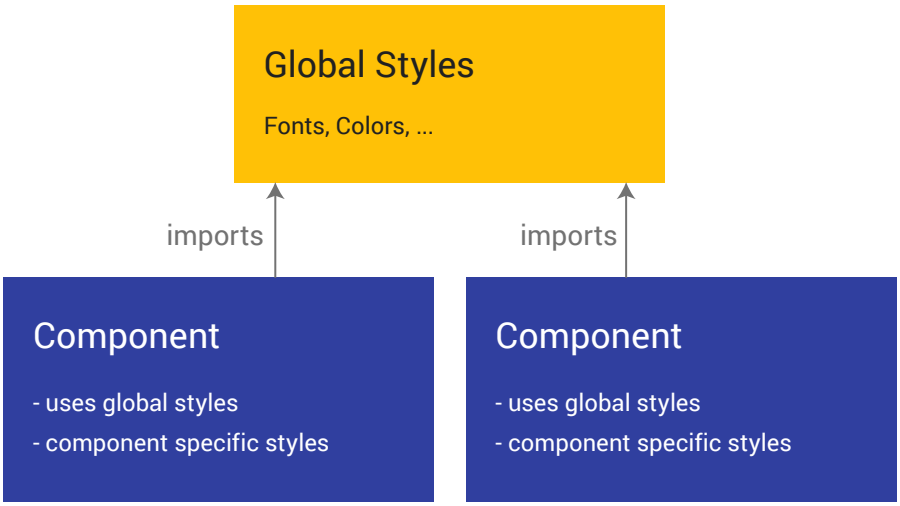


Figure 6.1: Application wide styles are declarated once, then components import and use them. In addition component-specific styles can be added inside the component.

[15] https://elements.polymer-project.org/
[16] Do not repeat yourself.

6.6 Testing

Several challenges when writing or using web components are

- *testing* to ensure compatibility and stability by running tests,
- *updating* a component to a new version and
- *deploying* new versions of the component.

For a full-featured testing tool the *Polymer* Team has created the *Web Component Tester*(WCT).[17] It utilizes

- *mocha*[18] as a test framework,
- *chai*[19] for assertions,
- *async*[20] for aiding with asynchronous testing,
- *sinon*[21] for mocking a server to test XHR requests,
- *test-fixture*[22] for testing scoped `<template>` elements and
- *accessibility-developer-tools*[23] for automated accessibility tests.[24]

Tests are written in JavaScript and can be executed via the command line or in the browser. To reset the DOM state between tests, the `<test-fixture>` element can be used.[25] For remote cross-browser testing *WCT* supports *SauceLabs*[26] integration. Also *Travis CI*[27], which is free for open source projects, is an option for automated testing. Of course, all parts of the *WCT* can be also used for testing non-Polymer web components. With the approach of *test-driven development* it would be interesting to validate existing third-party components by writing tests before employing them. *AVA*[28] runs multiple tests concurrently at once and would be interesting as future testing framework.

Embracing a test-driven workflow is crucial for the evolution of a web component. If the implementation of a component changes, failing tests report upcoming issues. In case the changes are applied and the component will be updated, the failing tests are a source for a migration guide, so users of the component know in which cases the component will break and which changes are necessary for using the new component version.

[17] https://www.polymer-project.org/1.0/docs/tools/tests
[18] http://mochajs.org/
[19] http://chaijs.com/
[20] https://github.com/caolan/async
[21] http://sinonjs.org/
[22] https://github.com/PolymerElements/test-fixture
[23] https://github.com/GoogleChrome/accessibility-developer-tools
[24] https://github.com/Polymer/web-component-tester
[25] https://github.com/Polymer/web-component-tester/blob/master/README.md#test-fixture
[26] https://saucelabs.com/
[27] https://travis-ci.org/
[28] https://github.com/avajs/ava

6.7 Updates

To create a versionable web component there is no way around a version control system. Looking at available web components online[29] the most popular VCS is *Git*.[31] Maintaining the codebase as a *Git* repository (in short: repo) eases the versioning and distribution process on many stages. When developing a web component, the *git-flow*[32] branching model can be used in combination with *semver*[33] for labeling the releases.

6.8 Deployment

Once the web component exists in a *Git* repo the next topic is the distribution, so other projects can use it. To encourage open-source development the repo can be pushed to *GitHub*[34] where people can view, clone or fork the repo and raise issues for bugs. Publishing only to *GitHub* would mean that every user has to update manually by checking if there's a new version available and then pull all changes. To avoid that, it makes sense to publish the web component repo to a package manager like *Bower*[35] or *npm*.[36] With the web component published, projects can install the web component as dependency. When installing a package with `bower install <packagename>` or `npm install <packagename>`, the user has a range of possiblities to choose which version should be installed.[37] Version numbers in the `package.json` use prefixes to lock the installed version, allow patches (non-breaking changes), minor updates, major updates or always the lastest version.

Now, everytime the build process of a project triggers `bower install <packagename>` or `npm install <packagename>` the package version specified in the `bower.json` or `package.json` will be installed.

Customelements.io is crawling *Bower* and *npm* package registries for specific keywords in the package description to provide a listing of available web components. The whole publishing ecosystem is illustrated in figure 6.2.

If the web component is closed source, it is possible to use a private *GitHub* repository, a private *npm* package or *npm enterprise*[38] to run *npm* infrastructure behind a companys firewall. Installing from a locally cached *npm* modules is also possible.

[29] https://customelements.io/,[30]
[31] https://git-scm.com/
[32] http://nvie.com/posts/a-successful-git-branching-model/
[33] http://semver.org/
[34] http://github.com
[35] http://bower.io
[36] http://npmjs.com
[37] https://docs.npmjs.com/getting-started/semantic-versioning
[38] https://docs.npmjs.com/enterprise/index

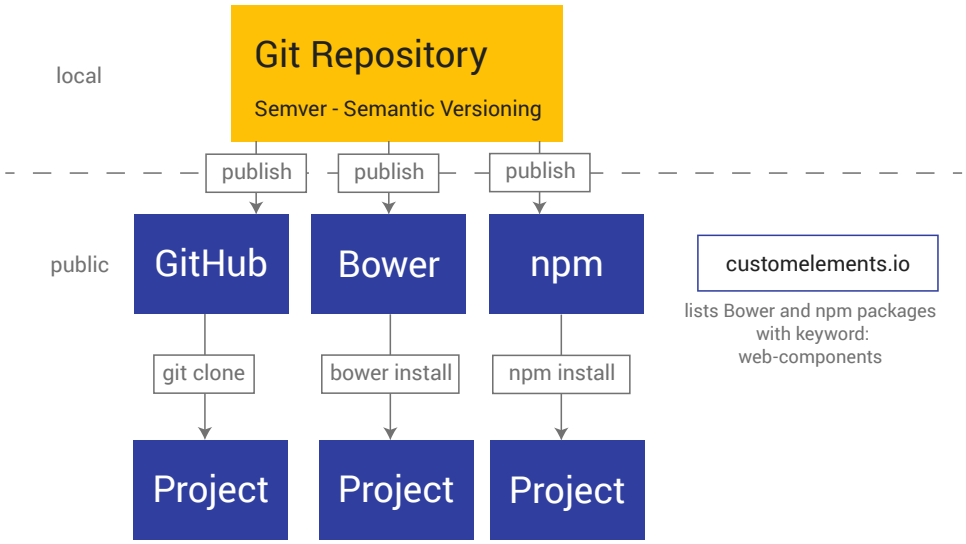


Figure 6.2: The web component development starts in a local *Git* repository which can be published to *GitHub*, *Bower* and *npm*. Projects can clone the *GitHub* repo and update manually (not recommended) or install the web component via a package manager and update automatically. Additionally, packages with the `web-component` keyword are listed in customelements.io.

The upcoming chapter discusses the benefits and consequences when employing web components in a project.

Chapter 7

Benefits, consequences and limitations

Before discussing benefits, consequences and limitations of web components it is important to distinguish between 2 fields of usage.

The first one is the usage of a small amount of web components on a site, like a `<google-map>` or a `<analog-clock>` to add a piece of functionality. To build a whole site by nesting and combining web components is the second field of usage. This chapter focuses mostly on the latter, because additional aspects are coming up when using an architecture of components and it also discusses the second research question of this thesis: *What are the consequences and limitations when employing an architecture based on web components?*

7.1 Components instead of pages

One of the first things when diving into component-driven development is the shift from designing fully-layouted pages to granular components (section 6.1). This shift has to happen in the conceptual, visual design and implementation phase. Putting the content in first place and developing the necessary components to communicate the content takes time. Thinking about the flexibility and reusage of components needs additional effort in the beginning and should pay off in the implementation phase and flexibility of the system. People working on a component-driven project have to be familiar with the concepts, to create a component or pattern library where web components can be derived.

Once the components are built, it is possible craft a new page just by assembling existing components and filling in the content without developing a new page from ground up. So the inital time to develop the design system can be seen as a downside, which pays off later in production for more agility and faster changes.

The flexibility of nesting components comes with the caveat that nested content may be improperly displayed if it doesn't fit the parents container. Here the *purpose of a component* can help as described in section 6.1.

By organizing files component-based, all files concerning a component are in one place. All scripts, styles and markup for a component are bundled in one folder. Updating a component requires less cognitive load than in projects with global scripts and styles since the mind can focus on the component folder and scoping problems are reduced.

7.2 Architectural decisions

Utilizing components comes with an own application architecture. On the one hand there's a pool of components which will be developed and on the other hand the sites are just assembled with those components and filled with content. This implies reusable components, flexible enough to be used with different content in different places. Flexibility comes with a price and a component can get complex just because it is used in too many variations. It helps to limit the components purpose to keep the complexity low (see section 6.1).

More components also mean more dependencies. A `<contact-form>` component most likely depends on different input and button components. To strive for loose coupling, the `<contact-form>` component *imports* its dependencies via *HTML imports* (see 2.2.3), so on top of each component all dependencies are clearly visible. The browser will request all imports from the server which will result in additional loading time. Browsers are smart and build a dependency graph before requesting the dependencies, so every component will be requested once, no matter how often it occurs. Server-side merging of imports before delivery is also a strategy that may be obsolete in the future when using HTTP/2 (end of section 2.2.3). In conclusion, the amount of dependencies are a disadvantage, but granular architecture eases the replacement of single components.

Standard HTML elements are a great way to markup content in a meaningful way and can be sufficient already. Web components are extending the set of available HTML elements to provide new ways for exploring content. They should make it easier to author HTML pages by hiding implementation detail inside the component. A component can be used for complex matters (like a `<google-map>`) or just as a wrapper to represent content more verbose (like a `<blog-post>`). Either way a web component provides encapsulation via *shadow DOM*. That avoids many cross-scope conflicts. CSS styles are only applying to the component and won't bleed out, the same goes for scripts.

The more tricky question is how to cross the encapsulation border from the outside and pass content, scripts and styles into a component. Currently,

a component can be styled from the outside by exposing CSS custom properties. The set of exposed CSS custom properties controls the customization degree of the component. It is only as customizable as the component author allows.

If a component *stays verbose*, it is easier to detect the state and changes for outside components. The component attributes can be reflected to the data attributes of the DOM element. Custom events with useful detail data can be fired in addition to the lifecycle callbacks (see 2.2.1), so outer components can use the custom events as hooks for taking further actions. In the standardization process are techniques how to pass content into a component. This is clearly a limitation for now. A simple component can define *one* spot (with the deprecated `<content>` element) where to insert nested content. No matter which DOM will be nested, it will be inserted at that spot. This allows only very limited control and a more sophisticated concept with *named slots* is in the standardization process (see 6.2).

Using the `<template>` element inside a component provides a way to create reusable markup. There are no templating languages or DOM-building via JavaScript necessary. A `<template>` is a chunk of DOM which can be filled with content and then cloned into the document DOM.

7.3 Choosing a framework

At the time of writing the native browser support of the four web component technologies increases steadily (see section 2). Luckily, polyfills are existing to use web components in current browsers as of today.

Going for native JavaScript web components can be tedious, functionality like data-binding, event-handling and dealing with scope needs additional development time. Various libraries or frameworks (see section 3) implement those features and offer ready-to-use components that can be customized. This provides a benefit in development speed, but comes with the downside of being bound to a library or framework. A big bonus of web components is the compatibility, so a vanilla JavaScript component can be used in any framework. Integrating a *Polymer* component which uses library-specific functionality can also be integrated in other applications, but additional payload for *Polymer* itself adds up. Writing native JavaScript web components with *ES2015* syntax is in advance and provides easier understanding of components.

7.4 High quality components

No matter if the choice are vanilla JavaScript components or a library, it is always necessary to use the right component for the right job. When implementing a new functionality, one has to choose between:

- including a existing component (search online),
- combinding already included components or
- building a new component.

Including a new existing component requires the search for it. The website customelements.io lists vanilla JavaScript web components as well as components from other element libraries. With the freedom of choice sometimes it is hard to predict which component fits the job. As the search results link to *GitHub* the only way to judge a component are the stars, forks, issues and commit activity which is not really a reliable source. Considering big players (currently *Google* with *Polymer* and *Angular 2*) is also an option. More budget and a bigger team can result in more effort per component and longer support.

"Use the platform" is a credo populated by the *Polymer* team when combining components or building new components.[1] It encourages the usage of the standardized JavaScript API implemented by browser vendors instead of building own workarounds and hacks. Relying on standards should ensure a longer lifetime of components.

Most importantly, testing and updating components with a reasonable workflow is necessary for a stable and high-quality component.

7.5 Updating components

Updating a single component in a ecosystem of components can have major consequences. An update may fix the component in one place, but break the system in another place. In a world of responsive design where components should be as flexible as possible, the context of a component can vary broadly. Therefore, extensive testing is unavoidable. This is also why each component should have a public interface seperated from the internal implementation to allow easier testing (see section 6.3.1).

From the consumer side, when using 3rd party components from *Bower* or *npm*, the projects' build process may update components with `bower install` or `npm install` to the latest version which won't fit visually anymore or break because of a changed interface. Also major releases of components or component libraries (like *Polymer*) may result in additional refactoring to ensure a working system.

Another challenge is the consistency of the pattern library. Once a pattern library is built (for example with a first major release), updates will follow and shouldn't interfer with the consistency.

In conclusion, the benefits start with a modular mindset and the reduced cognitive load when focusing on a single component at a time. Reusing components avoids repetition and saves time. Encapsulation minimizes scope

[1] https://www.polymer-project.org/1.0/docs/browsers

conflicts in web applications. Building components in vanilla JavaScript, especially with the ES2015 syntax, will change with the advance of the W3C specs and eases the development. Choosing a framework instead of vanilla components comes with the downside of introducing a large dependency, but enables component catalogs, syntactic sugar and compatibility. Building a system of multiple small components also requires testing and updating of every component to keep the application secure and up-to-date. Instead of testing a monolithic application, it is possible to test its components first.

The last chapter of this thesis focuses on the experiences during the thesis project and provides an outlook to empower the usage and inspire the community around web components.

Chapter 8

Conclusion

The web changes fast and it is risky to employ a framework that may be out-of-date two years later. This is why long-lasting standards exist and web components are built on standards. At the time of writing, the standardization process is in progress and it is very likely that the implementation will change until the browser vendors implement the final standardized solution. Hence the chapters *Architectural Design Concepts* and *Benefits, consequences and limitations* are extracting formal, long-term concepts independent from the implementation. Moving the mindset from pages to components, intentionally limiting a component and building a communicative as well as extendable component leads to more reusable web components.

Thinking about those concepts and applying them to web components improves their quality, no matter if vanilla JavaScript or a framework is used. To discuss the third research question (*How can a component be validated for reusability?*) Andrew Rota proposes a best practice for component design on the React.js Conf 2015 [19]:

- small,
- extremely capsulated,
- as stateless as possible,
- performant.

In combination with testing these 4 keypoints lead to versatile components with a high degree of reusability.

8.1 Outlook

There is still much room for improvement when building and using web components. The *Custom Elements v1 spec*[1], as explained by Eric Bidelman[2], allows a far more readable code with ES2015 syntax. Once the browser

[1] https://www.w3.org/TR/custom-elements/

[2] https://developers.google.com/web/fundamentals/primers/customelements/

takes care of registering custom elements another obstacle is removed from the development process. Also *Shadow DOM v1 spec* introduces *slots* to pass content into a component.[3] If a web component is only needed on certain screensizes, it would be a performance gain to register the web component conditionally like with a CSS media query. Easing the process of building framework-independent components would be a main future goal for further development. Also security holes would be worth investigating. For example, if sensitive login data is passed into the Shadow DOM. Jeremy Keith expresses his concerns about web components in his blog post *The extensible web* [15]:

> First of all, ask the question "who benefits from this technology?" In the case of service workers[4], it's the end users. They get faster websites that handle network failure better. In the case of web components, there are no direct end-user benefits. Web components exist to make developers lives easier. That's absolutely fine, but any developer convenience gained by the use of web components can't come at the expense of the user—that price is too high.

Keith also highlights the fail-well approach: if a web component does not render as expected, content should still be visible (in a less rich user experience) and backwards compatibility is necessary.

As closing words I would like to encourage people around the web community to really consider componentized systems and use web components as of today. It is a huge gain in productivity and consistency if existing code can be reused and browser support is available. Embrace best practices around the web community, stay informed about new developments in the field of web components and now go componentize your application!

[3] https://developers.google.com/web/fundamentals/primers/shadowdom/?hl=en

[4] Web technology for adding offline support to a web application

Appendix A

CD Contents

Format: CD-ROM, Single Layer

A.1 Thesis

Pfad: /

ReusabilityOfWebComponents.pdf Master thesis

Pfad: /latex

. . . . . . LaTeX source

Pfad: /latex/images

. . . . . . all images used in the thesis

A.2 Resources

Pfad: /resources

adactio.pdf . . . . . [15]
addy-road.pdf . . . . . [18]
ebidel-ce-v0.pdf . . . . [8]
ebidel-ce-v1.pdf . . . . W3C: Custom Elements spec v1
ebidel-template.pdf . . [9]
frost-atomicdesign-1.pdf [10]
frost-atomicdesign-2.pdf [10]
frost-atomicdesign-3.pdf [10]
frost-atomicdesign-4.pdf [10]
frost-atomicdesign-5.pdf [10]

krug-communcation.pdf	Michael Krug and Martin Gaedke: AttributeLinking
la-designlanguage.pdf .	[16]
polymer-behaviours.pdf	Polymer: Shared Behaviours
polymer-lifecyclecallbacks.pdf	Polymer: Lifecycle Callbacks
polymer-shadydom.pdf	Polymer: Shady DOM
react-wc.pdf	React & Web Components
savkin-angular2-template-syntax.pdf	[20]
w3c-css-variables.pdf .	[21]
w3c-custom-elements.pdf	[11]
w3c-html-imports.pdf .	[13]
w3c-htmltemplates.pdf	W3C: HTML Templates spec
w3c-shadow-dom-v0.pdf	[12]
w3c-shadow-dom-v1.pdf	W3C: Shadow DOM spec v1
w3c-slot-proposal.pdf .	[22]
w3c-templatelement.pdf	[23]
w3c-webcomponents.pdf	W3C: Web Components Overview

A.3 Listings

Pfad: /latex/listings

01-nativewebcomponents/	Listing 1
02-shadowdom/	Listing 2
03-templateelement/ . .	Listing 3
04-completetooltip/ . .	Listing 4
05-xtag/	Listing 5
06-angular2/	Listing 6

A.4 Thesis Project

Pfad: /thesisproject

keep/	Frontend
keep-api/	Backend

Appendix B

Listings

B.1 Web components with plain JavaScript

B.1.1 my-tooltip.js

```
1 // 1. Create a prototype inheriting from a basic HTMLElement
2 var MyTooltipProto = Object.create(HTMLElement.prototype);
3
4 // 2. Give my-tooltip a foo() method.
5 MyTooltipProto.foo = function() {
6   alert('foo() called');
7 };
8
9 // 3. Define a property "bar".
10 Object.defineProperty(MyTooltipProto, "bar", {
11   value: 5,
12   writable: true
13 });
14
15 // 4. lifecycle method executed everything when
16 // a <my-tooltip> element will be added in the DOM
17 MyTooltipProto.createdCallback = function() {
18   console.log("i was created");
19   // assign a random number as id
20   this.bar = window.crypto.getRandomValues(new Uint32Array(1))[0];
21 }
22
23 // 5. Register x-foo's definition.
24 var MyTooltip = document.registerElement('my-tooltip', {prototype:
       MyTooltipProto});
```

B.1.2 index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>01 - Native Web Components</title>
  <script src="bower_components/webcomponentsjs/webcomponents-lite.js">
    </script>
  <script src="elements/my-tooltip.js"></script>
  <script>
    document.addEventListener('DOMContentLoaded', function() {

      var tt = document.querySelectorAll('my-tooltip');
      tt.forEach(function(tooltip) {
        console.log(tooltip.bar);
      });
    });
  </script>
</head>
<body>
  Hello Tooltip!
  <my-tooltip>Hey, i am number 1.</my-tooltip>
  <my-tooltip>Hello there, i am number 2!</my-tooltip>

</body>
</html>
```

B.2 Shadow DOM

B.2.1 my-tooltip.js

```
// 1. Create a prototype inheriting from a basic HTMLElement
var MyTooltipProto = Object.create(HTMLElement.prototype);

// 2. Give my-tooltip a foo() method.
MyTooltipProto.foo = function() {
  alert('foo() called');
};

// 3. Define a property "bar".
Object.defineProperty(MyTooltipProto, "bar", {
  value: 5,
  writable: true
});

// 4. lifecycle method executed everything when
// a <my-tooltip> element will be added in the DOM
MyTooltipProto.createdCallback = function() {
  console.log("i was created");
  // assign a random number as id
  this.bar = window.crypto.getRandomValues(new Uint32Array(1))[0];
  this.addShadowDOM();
}

```

```
24 MyTooltipProto.addShadowDOM = function() {
25   // 6. add shadow root element
26   var shadow = this.createShadowRoot();
27   // style element is scoped inside the shadow dom
28   shadow.innerHTML = "<style>p { text-transform: uppercase; }</style>";
29   shadow.innerHTML += "<p>I'm the Shadow DOM!</p>";
30 }
31
32 // 5. Register my-tooltip's definition.
33 var MyTooltip = document.registerElement('my-tooltip', {prototype:
       MyTooltipProto});
```

B.2.2 index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>02 - Shadow DOM</title>
6    <script src="bower_components/webcomponentsjs/webcomponents-lite.js">
       </script>
7    <script src="elements/my-tooltip.js"></script>
8    <script>
9      document.addEventListener('DOMContentLoaded', function() {
10
11       var tt = document.querySelectorAll('my-tooltip');
12       var ttar = Array.prototype.slice.call(tt);
13       ttar.forEach(function(tooltip) {
14         console.log(tooltip.bar);
15       });
16     });
17   </script>
18   <style type="text/css">
19   /* won't affect p elements inside shadow dom */
20   p {
21     color: goldenrod;
22   }
23   </style>
24 </head>
25 <body>
26   <p>Hello Tooltip!</p>
27   <my-tooltip>Hey, i am number 1.</my-tooltip>
28   <my-tooltip>Hello there, i am number 2!</my-tooltip>
29   <p>Another paragraph</p>
30
31 </body>
32 </html>
```

B.3 HTML template element

B.3.1 index.html

```
<!-- Based on the example under: https://www.w3.org/TR/html5/
     scripting-1.html#the-template-element -->
<!DOCTYPE html>
<html>
<title>03 - Template element</title>
<script>
  // Data is hard-coded here, but could come from the server
  var data = [
  { name: 'Pillar', color: 'Ticked Tabby', sex: 'Female (neutered)',
    legs: 3 },
  { name: 'Hedral', color: 'Tuxedo', sex: 'Male (neutered)', legs: 4 },
  ];
</script>
<body>
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Colour</th>
        <th>Sex</th>
        <th>Legs</th>
      </tr>
    </thead>
    <tbody>
      <template id="row">
        <tr><td><td><td><td>
        </template>
      </tbody>
    </table>
    <script>
      var template = document.querySelector('#row');
      for (var i = 0; i < data.length; i += 1) {
        var cat = data[i];
        var clone = template.content.cloneNode(true);
        var cells = clone.querySelectorAll('td');
        cells[0].textContent = cat.name;
        cells[1].textContent = cat.color;
        cells[2].textContent = cat.sex;
        cells[3].textContent = cat.legs;
        template.parentNode.appendChild(clone);
      }
    </script>
  </body>
</html>
```

B.4 Complete tooltip example

B.4.1 my-tooltip.html

```
<!-- all behaviour -->
<script src="my-tooltip.js"></script>

<template id="tt">
  <style>
    :host {
      position: relative;
      text-decoration: underline solid gray;
    }
    .tt {
      display: none;
      position: absolute;
      top: -2.5em;
      left: 0;
      padding: 0.4em 0.5em;
      border-radius: 3px;

      font-family: sans-serif;
      font-size: 12px;
      white-space: nowrap;
      background-color: #444;
      color: #fff;
    }
    :host(:hover) {
      cursor: pointer;
    }
    :host(:hover) .tt {
      display: inline-block;
    }
  </style>
  <span class="tt"></span>
  <content></content>
</template>

<script>
  document.registerElement('my-tooltip', {prototype: myTooltip});
</script>
```

B.4.2 my-tooltip.js

```
// encapsulated functionality of the tooltip
var myTooltip = (function() {

  var insertIntoDocument = (function() {
    "use strict";
    var importDoc;
```

```
7     // reference to the my-tooltip.html
8     importDoc = (document._currentScript || document.currentScript).
      ownerDocument;
9     // current my-tooltip DOM element, id of template tag in
      my-tooltip.html
10    return function (obj, idTemplate) {
11      var template = importDoc.getElementById(idTemplate),
12          clone = document.importNode(template.content, true);
13
14      // fill template
15      fetchInfo(obj, clone);
16      console.log(clone.querySelector(".tt"));
17      // attach full template to shadow root if text is set
18      if (obj.text) {
19        var sd = obj.shadowRoot;
20        // if the object has a shadowRoot, empty it
21        if(sd) {
22          while(sd.firstChild) {
23            sd.removeChild(sd.firstChild);
24          }
25
26        } else {
27          // if the object has no shadowRoot create one
28          sd = obj.createShadowRoot();
29        }
30        // append filled template to shadowRoot
31        sd.appendChild(clone);
32
33      }
34    };
35  }());
36
37  // el = <my-tooltip> DOM element
38  // clone = current instance of template
39  var fetchInfo = function(el, clone) {
40    var dataText = el.getAttribute('data-text');
41
42    if (dataText) {
43      el.text = dataText;
44      clone.querySelector(".tt").textContent = dataText;
45    }
46  };
47
48  var proto = Object.create(HTMLElement.prototype);
49
50  // every element should have a property for tooltip text
51  Object.defineProperty(proto, "text", {
52    value: undefined,
53    writable: true
54  });
55
56  proto.attributeChangedCallback = function(attrname, oldval, newval) {
57    console.log("attr changed", attrname, oldval, newval);
58    insertIntoDocument(this, "tt");
```

```
59   };
60
61   // insert into document, if a <my-tooltip> element will be created
62   proto.createdCallback = function() {
63     // 'this' accesses the currently created DOM element <my-tooltip>
64     insertIntoDocument(this, "tt");
65   };
66
67   // return prototype for registering the element in my-tooltip.html
68   return proto;
69 })();
```

B.4.3 index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>04 - Complete tooltip</title>
6    <script src="bower_components/webcomponentsjs/webcomponents.js">
       </script>
7    <link rel="import" href="elements/my-tooltip.html">
8    <script>
9      document.addEventListener('WebComponentsReady', function() {
10       console.log("Imports loaded & elements registered!");
11     });
12   </script>
13 </head>
14 <body>
15   <h1>Hello Tooltip!</h1>
16   <p>I am sample text and a <my-tooltip data-text="This is super secret
       info">keyword</my-tooltip>has a tooltip.</p>
17   <p><my-tooltip data-text="more secret info">Me too!</my-tooltip>
       Check it out via hovering the text.</p>
18
19
20 </body>
21 </html>
```

B.5 X-Tag example

B.5.1 movie-quoter.html

```
1 <!-- boilerplate from https://github.com/webcomponents/hello-world-xtag
      -->
2 <!-- Imports x-tag -->
3 <script src="../../bower_components/x-tag-core/src/core.js"></script>
4
5 <!-- Defines element markup -->
```

```
<template>
    <style>
    :host([theme="fancy"]) p {
        color: purple;
        font-family: cursive;
    }
    :host([theme="minimal"]) p {
        color: gray;
    }
    </style>

    <blockquote>
        <p>Awesome quote</p>
        <footer>
             <cite><span class="who"></span>, <span class="movie">
    </span> </cite>
        </footer>
    </blockquote>
</template>

<script>
(function(window, document, undefined) {

    var quotes = [
        {
            "quote": "All those moments will be lost in time... like
    tears in rain.",
            "who": "Roy Batty",
            "movie": "Blade Runner"
        },
        {
            "quote": "Frankly, my dear, I don't give a damn",
            "who": "Rhett Butler",
            "movie": "Gone with the Wind (1939)"
        },
        {
            "quote": "I love lamp",
            "who": "Brick Tamland",
            "movie": "Anchorman (2004)"
        },
        {
            "quote": "I'm gonna make him an offer he can't refuse.",
            "who": "Don Corleone",
            "movie": "The Godfather (1972)"
        },
        {
            "quote": "I'm as mad as hell, and I'm not gonna to take
    this anymore!",
            "who": "Howard Beale",
            "movie": "The Network (1976)"
        }

    ];

```

```
57      /**
58       * @param  {number} upper limit
59       * @return {number} random number from 0 to max
60       */
61      function getRand(max) {
62          var rand = (window.crypto.getRandomValues(new Uint32Array(1))
        [0]) % max;
63          return rand;
64      }
65
66      // Refers to the "importer", which is index.html
67      var importer = document;
68      // Refers to the "importee", which is src/movie-quoter.html
69      var thisDoc = document._currentScript.ownerDocument;
70      // Gets content from <template>
71      var template = thisDoc.querySelector('template').content;
72
73      xtag.register('movie-quoter', {
74          lifecycle: {
75              created: function() {
76                  this.bquote = template.querySelector('blockquote');
77                  this.quote = template.querySelector('p');
78                  this.who = template.querySelector('.who');
79                  this.movie = template.querySelector('.movie');
80                  // Creates the shadow root
81                  this.shadowRoot = this.createShadowRoot();
82                  this.setQuote();
83              },
84              attributeChanged: function() {
85                  console.log("attribute changed");
86              }
87          },
88          accessors: {
89              theme: {
90                  attribute: {},
91                  get: function(){
92                      return this.getAttribute('theme') || "minimal"
93                  },
94                  set: function(value){
95                      this.xtag.data.theme = value;
96                  }
97              }
98          },
99          methods: {
100             setQuote: function() {
101                 var quoteNum = getRand(quotes.length);
102                 var q = quotes[quoteNum];
103
104                 this.quote.textContent = q.quote;
105                 this.who.textContent = q.who;
106                 this.movie.textContent = q.movie;
107                 // Removes shadow root content
108                 this.shadowRoot.innerHTML = '';
109                 // Adds a template clone into shadow root
```

```
110                 var clone = importer.importNode(template, true);
111                 this.shadowRoot.appendChild(clone);
112             }
113         },
114         events: {
115             'click': function (event) {
116                 console.log('a movie-quoter was clicked');
117             }
118         }
119     });
120 })(window, document);
121 </script>
```

B.5.2 index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>05 - X-Tag</title>
6    <script src="bower_components/webcomponentsjs/webcomponents-lite.js">
       </script>
7
8    <link rel="import" href="elements/movie-quoter.html">
9  </head>
10 <body>
11   <h1>Welcome to our simple X-Tag example</h1>
12   <movie-quoter theme="fancy"></movie-quoter>
13   <p>Change the attribute "theme" from the tag above to "minimal"</p>
14   <button id="btnToggleStyle">Toggle Style</button>
15   <button id="btnNewQuote">Random Quote</button>
16
17   <script>
18     document.querySelector('#btnToggleStyle').addEventListener('click',
        function(e) {
19       var quoter = document.querySelector('movie-quoter');
20       if (quoter.getAttribute("theme") === "fancy") {
21         quoter.setAttribute("theme", "minimal");
22       } else {
23         quoter.setAttribute("theme", "fancy");
24       }
25     });
26     document.querySelector('#btnNewQuote').addEventListener('click',
       function(e) {
27       var quoter = document.querySelector('movie-quoter').setQuote();
28     });
29   </script>
30 </body>
31 </html>
```

B.6 Angular 2 example

B.6.1 app.component.ts

```
1 import {Component} from 'angular2/core';
2 import {NiceGreeterComponent} from './nice-greeter.component';
3 import {MovieQuoterComponent} from './movie-quoter.component';
4
5 @Component({
6     selector: 'my-app',
7     template: `<h1>My First Angular 2 App App</h1>
8           <nice-greeter>content</nice-greeter>
9           <h3>Another H3</h3>
10          <movie-quoter></movie-quoter>`,
11    directives: [NiceGreeterComponent, MovieQuoterComponent]
12 })
13 export class AppComponent { }
```

B.6.2 nice-greeter.component.ts

```
1 import {Component} from 'angular2/core';
2 import {ViewEncapsulation} from 'angular2/core';
3
4 @Component({
5     selector: 'nice-greeter',
6     template: `<h3>Hi {{name}}</h3>
7     <button (click)="logMyName()">Log my name</button>`,
8     styles: [`
9       h3 {
10        color: red;
11      }
12    `],
13  // activate native shadow dom.
14  // inspect the result, comment this line and inspect again
15  encapsulation: ViewEncapsulation.Native
16 })
17 export class NiceGreeterComponent {
18  name: String;
19  constructor() {
20    this.name = 'Max';
21  }
22  logMyName() {
23    console.log(this.name);
24  }
25 }
```

B.6.3 movie-quoter.component.ts

```
1  import {Component, Input} from 'angular2/core';
2  import {ViewEncapsulation} from 'angular2/core';
3
4  @Component({
5      selector: 'movie-quoter',
6      template: `<blockquote attr.theme="{{theme}}" (click)="changeQuote
       ()">
7                      <p>{{quote}}</p>
8                      <footer>
9                           <cite><span class="who">{{who}}</span>, <span
       class="movie">{{movie}}</span> </cite>
10                     </footer>
11                 </blockquote>
12                 <button (click)="changeTheme()">Change Theme</button>`,
13     styles: [`
14       blockquote[theme="fancy"] p {
15             color: purple;
16             font-family: cursive;
17         }
18         blockquote[theme="minimal"] p {
19             color: gray;
20         }
21         blockquote(:hover) {
22             cursor: pointer;
23         }
24     `],
25     // activate native shadow dom.
26   // inspect the result, comment this line and inspect again
27     encapsulation: ViewEncapsulation.Native
28 })
29 export class MovieQuoterComponent {
30
31     private quotes = [
32         {
33             "quote": "All those moments will be lost in time... like
       tears in rain.",
34             "who": "Roy Batty",
35             "movie": "Blade Runner"
36         },
37         {
38             "quote": "Frankly, my dear, I don't give a damn",
39             "who": "Rhett Butler",
40             "movie": "Gone with the Wind (1939)"
41         },
42         {
43             "quote": "I love lamp",
44             "who": "Brick Tamland",
45             "movie": "Anchorman (2004)"
46         },
47         {
48             "quote": "I'm gonna make him an offer he can't refuse.",
```

```
49              "who": "Don Corleone",
50              "movie": "The Godfather (1972)"
51          },
52          {
53              "quote": "I'm as mad as hell, and I'm not gonna to take
        this anymore!",
54              "who": "Howard Beale",
55              "movie": "The Network (1976)"
56          }
57
58      ];
59
60      quote: string;
61      who: string;
62      movie: string;
63      theme: string;
64
65      private getRand(max) {
66          var rand = (window.crypto.getRandomValues(new Uint32Array(1))
        [0]) % max;
67          return rand;
68      }
69      private assignQuote(num) {
70          this.quote = this.quotes[num].quote;
71          this.who = this.quotes[num].who;
72          this.movie = this.quotes[num].movie;
73      }
74
75      constructor() {
76          this.assignQuote(0);
77          this.theme = "minimal";
78      }
79      changeQuote() {
80          this.assignQuote(this.getRand(this.quotes.length));
81      }
82      changeTheme() {
83          if (this.theme == "minimal") {
84              this.theme = "fancy"
85          } else {
86              this.theme = "minimal";
87          }
88      }
89  }
```

References

Literature

[1] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000 (cit. on p. 26).

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994 (cit. on pp. 4, 5, 50).

[3] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press, 2012 (cit. on p. 51).

[4] Doug McIlroy, E. N. Pinson, and B. A. Tague. "UNIX Time-Sharing System". *The Bell System Technical Journal* 57.6 (1978), pp. 1899–1904 (cit. on p. 47).

[5] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, 2012 (cit. on p. 49).

[6] Jarrod Overson and Jason Strimpel. *Developing Web Components: UI from jQuery to Polymer*. O'Reilly Media, 2012 (cit. on pp. 15, 16, 53).

Online sources

[7] Bosonic Authors. *Bosonic Project*. 2015. URL: http://bosonic.github.io/documentation/getting-started/faq.html (visited on 09/08/2016) (cit. on p. 20).

[8] Eric Bidelman. *Custom Elements*. Aug. 2013. URL: http://www.html5rocks.com/en/tutorials/webcomponents/customelements/ (visited on 09/08/2016) (cit. on pp. 7, 65).

[9] Eric Bidelman. *Custom Elements*. Aug. 2013. URL: http://www.html5rocks.com/en/tutorials/webcomponents/template/ (visited on 09/08/2016) (cit. on pp. 7, 65).

[10] Brad Frost. *Atomic Design*. 2016. URL: atomicdesign.bradfrost.com (visited on 09/08/2016) (cit. on pp. 47, 65).

[11] Dimitri Glazkov. *Custom Elements*. 2015. URL: https://w3c.github.io/webcomponents/spec/custom/ (visited on 09/08/2016) (cit. on pp. 3, 6, 66).

[12] Dimitri Glazkov and Hayato Ito. *Shadow DOM*. Dec. 2015. URL: http://w3c.github.io/webcomponents/spec/shadow/ (visited on 09/08/2016) (cit. on pp. 3, 9–11, 66).

[13] Dimitri Glazkov and Hajime Morrita. *HTML Imports*. 2016. URL: http://w3c.github.io/webcomponents/spec/imports/ (visited on 09/08/2016) (cit. on pp. 3, 66).

[14] Google Inc. *Lifecycle callbacks in Polymer*. 2016. URL: https://www.polymer-project.org/1.0/docs/devguide/registering-elements.html#lifecycle-callbacks (visited on 09/08/2016) (cit. on p. 24).

[15] Jeremy Keith. *Extensible web components*. 2016. URL: https://adactio.com/journal/11052 (visited on 09/08/2016) (cit. on pp. 64, 65).

[16] Alla Kholmatova. *The Language of Modular Design*. 2015. URL: http://alistapart.com/article/language-of-modular-design (visited on 09/08/2016) (cit. on pp. 53, 66).

[17] Scott Miles. *What is shady DOM?* May 2015. URL: https://www.polymer-project.org/1.0/articles/shadydom.html (visited on 09/08/2016) (cit. on p. 24).

[18] Addy Osmani. *JavaScript Application Architecture On The Road To 2015*. Dec. 2014. URL: https://medium.com/google-developers/javascript-application-architecture-on-the-road-to-2015-d8125811101b (visited on 09/08/2016) (cit. on p. 65).

[19] Andrew Rota. *The Complementarity of React.js and Web Components*. 2015. URL: http://andrewrota.github.io/complementarity-of-react-and-web-components-presentation/#/33 (visited on 09/08/2016) (cit. on p. 63).

[20] Viktor Savkin. *Angular 2 Template Syntax*. 2015. URL: http://victorsavkin.com/post/119943127151/angular-2-template-syntax (visited on 09/08/2016) (cit. on pp. 22, 66).

[21] W3C. *CSS Custom Properties*. 2015. URL: https://www.w3.org/TR/css-variables/ (visited on 09/08/2016) (cit. on pp. 5, 10, 13, 52, 66).

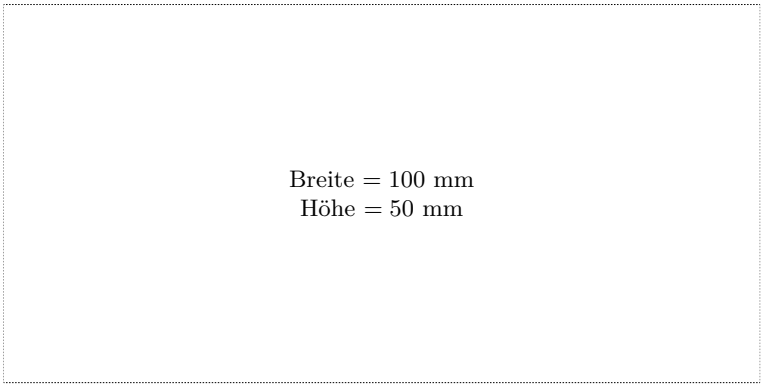
[22] W3C. *<slot> HTML element proposal*. 2015. URL: https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md (visited on 09/08/2016) (cit. on pp. 16, 66).

[23] W3C. *The Template Element*. 2015. URL: http://www.w3.org/TR/html5/scripting-1.html#the-template-element (visited on 09/08/2016) (cit. on pp. 3, 7, 9, 66).

[24] W3C. *Web Components*. 2015. URL: http://www.w3.org/standards/techs/components (visited on 09/08/2016) (cit. on p. 3).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —