

Browsernative Microservices

Jan Peteler, FH Würzburg-Schweinfurt, jan.peteler@student.fhws.de

Abstract—Building complex web applications nowadays require additional layers of abstraction and often heavily depend on proprietary frameworks. New specifications build right into the browserengine provide a native service API to overcome tricky abstraction constraints.

INTRODUCTION

Simplicity is prerequisite for reliability. - Edsger W. Dijkstra

Introduction: from simplicity to Microservices to extensible web manifesto to w3c specifications ... and beyond

Der Erfinder der Programmiersprache Clojure, Rich Hickey, ist ohne Zweifel eine Koryphäe auf seinem Gebiet, der Strukturierung von komplexen Systemen. In einer vielbeachteten Keynote aus dem Jahr 2012 geht er auf etymologisch-philosophische Spurensuche nach dem Wort **simplicity** aus Sicht eines Softwareentwicklers.¹ Das Adjektiv *simple* hat demnach seinen Ursprung im lateinischen Wort *simplex*, was soviel wie *einfach* oder *einzel*n bedeutet. In Gegensatz dazu stehen Eigenschaften wie *complex* oder *multiplex*. Qualitative Software ist, so Hickey, vor allem simpel - im Prozess, im Design, in der Struktur und in der Entwicklung.

Bezeichnend für diese Keynote ist, dass sie von Rich Hickey im Rahmen einer Webentwicklerkonferenz gehalten wurde. Software für den Browser war (und ist) seit langer Zeit maßgeblich geprägt von **Komplexität** auf mehreren Ebenen. Im Laufe dieser Bachelorarbeit werden diese Stru

- Das *Document Object Model* als Rückgrad jeder Webapplikation ist hierarchisch strukturiert und deren Elemente damit keinesfalls unabhängig in ihrer Darstellung und Reihenfolge.
- JavaScript als single-threaded Skriptsprache lässt sich schlecht in ihrem Verhalten isolieren, ist Fehleranfällig und hatte lange Zeit nur sehr wenig idiomatische Lösungsansätze für komplexe Probleme, wie beispielsweise Asynchronität
- CSS ist geprägt von stetigem Überschreiben vorher definierter Regeln und verletzt damit Simplität in ihrer Struktur, Design und dem Entwicklungsprozesses auf bester Art und Weise.

Betrachtet man andere populäre Webframeworks dieser Zeit, wie beispielsweise React oder Angular, lässt sich ziemlich schnell ein gemeinsames Designpattern ausmachen,

wie Komponenten intern ihre Zustände verwalten. Der Facebook Entwickler Dan Abramov hat dieses dichotome Pattern systematisch erfasst und in zwei Kategorien eingeteilt.

Nach Abramov existieren zum einen Komponenten, die alleine für das **Darstellen von Information** zuständig sind. Diese Komponenten nehmen keinerlei Einfluss auf Informationen oder anderen Komponenten um sich herum. Sie sind im wahrsten Sinne passiv und fremdgesteuert über klar definierte Schnittstellen. Diese Komponenten sind häufig flexibel einsetzbar und hochgradig wiederverwendbar. Abramov nennt diese Komponenten "Presentational Components".[1] Verortet man diese Art von Komponenten innerhalb des *MVC Pattern*, sind die Komponenten reine *View-Elemente*.

Im Gegensatz dazu stehen Komponenten, die für die **Verarbeitung von Informationen** zuständig sind. Diese so genannten "Container Components" haben oft einen internen Zustand, den sie verändern können und an ihre Kindkomponenten weiterreichen können.[1] Im *MVC Pattern* handelt es sich um die *Controller*. In der funktionalen Programmiersprache Elm werden diese Elemente *Updater* genannt, was ihre Funktion noch besser umschreibt.

Ein üblicher eventgesteuerter Webservice setzt sich aus unterschiedlichsten Komponenten zusammen, die wiederum unterschiedlichste Eventlistener & -emitter in sich subsumieren. Diese inhärente Komplexität verlangt geradezu nach einer klaren, deterministischen Struktur des Webservices, die das Zusammenspiel orchestriert. In der Analogie des Orchesters gesprochen, benötigt der Webservice (oder sogar die gesamte Webapplikation) einen Dirigenten, der für die Steuerung verantwortlich ist.

MICROSERVICES

Opening up the case for *Browsernative Microservices* brings up the question about the concept microservices in general. In fact the concept of microservices has many facets, stretching beyond disciplines and technical boundaries. It lacks a formal standardization but there are certain ideas emerge from this pattern. As a primary source of truth this article relies on the work of Sam Newman, who has written a comprehensive guide in *Building Microservices*. The purpose of this section is to match those ideas against the manifestations of web components.

In a nutshell a microservice is a small, autonomous service that works together with other services seamlessly.[2, p. 2] or with the words of Fowler and Lewis: "It is an approach to

¹Rails Conf 2012 Keynote: Simplicity Matters by Rich Hickey

developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms,..."[3] Microservices incorporate many ideas, like *domain-driven design* where we try to represent the real world in our code.[2, p. 2] Or making use of *continuous delivery* for pushing software rapidly through *automated deployment* mechanisms in production.[3] And, last but not least, microservices utilizes the idea of small teams with a lot of product knowledge working mostly autonomous on their very own service with their very own set of tools and techniques.

Technical perspective

Shift of paradigms / BFF / Platform agnostic / changes in infrastructure like APIs Databanks / Deployment

Talking about microservices in a browsernative context isn't that far fetched as microservices themselves incorporate many ideas from the web. Exemplary, microservices often communicate via an HTTP request-response with resource API's and lightweight messaging.[3] Nevertheless, while (server-side) microservices offer wide ranges of technical possibilities, we must take into account that (client-side) browsers come with certain constraints and limitations.

Fowler and Lewis issue a call for using services as components. A component is regarded as a unit of software that is independently replaceable and upgradeable. The main advantage of a component in contrast of library is the possibility of an independent deployment. It aligns perfectly with the main goal of a microservice architecture to strip away most of the dependencies in favour of clean interfaces.[3]

Components

First of all, from a technical perspective, a microservice reinforces the *Single Responsibility Principle* defined by Robert C. Martin: "Gather together those things that change for the same reason and separate those things that change for different reasons."[4] An a way this principle tackles another often cited design principle of the *seperation of concerns*. Web Components incorporate this principle in multiple ways while still remaining flexible.

Most obvious ist the gathering of all related code under the umbrellar of a single HTML tag. Grouping together HTML, JS and CSS Code in a safe, sandboxed environment exposes the possibility to build more cohesive and understandable services. In the typical global nature of web development those three pillars are separated. This circumstance left the developer switching back and forth between code bases developing a tricky (and sometimes biased) way to glue related parts together.

Secondly, the sub-standard *custom elements* introduces so called lifecycle methods and a getter/setter interface exposing the functionality to the developer. Event handling, for example, can be registered in place which is much

more declarative than assigning event listeners from the outside. Of course, this events can be pushed down to nested tags, allowing an increasingly granular system design. This approach will be explained further in the upcoming sections.

The concept of microservices incooperates not only a technical perspective. Microservice patterns are a product of real-world usage.[2, p. 1] In a real world we typical have to deal with the so called *Conway's Law*:

"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations".
[5]

Following this logic any company, whether it is web-related or not, should be devided in units grouped around a destinct business service to optimise the workflow. Fowler and Lewis outlines this approach as an "alignment of business capabilities"[3] While this kind of structure may be true for companies like Google or Amazon, there is a vast majority of companies developing for the web which are grouped around tasks.[6] A very common structure is formed by the technology stack (UX Designers, Frontend- & Backend Developers) or by separating teams along the product lifecycle (development, testing, deployment).

Advocators from the microservice approach propose a different model. best described by . Web components are one (but important) way to tie up those diciplines as one component can host a single independent business service. Combined with a flexible backend service these components can be huge gain over the cumbersome functional organizational approach.

The ideas transcending from the microservice approach offers plenty of choices and decisions how to proceed with designing a program or to structure a process.

W3C SPECIFICATIONS

For building a native microservice running on the "bare-metal" browser engine requires a bunch of new specifications and assumptions. Most importantly the quasi specification **Web Components** is needed. *Web Components* is not a real standard. It's an amalgam of APIs from multiple w3c specs which can be used independently, too. A webdeveloper may choose one spec and embrace the freedom in architecture which can be combined with other frameworks/libraries.

Depending on the context, some people argue for only two specs which essentially make it possible to create a scoped component but not caring too much on it's distribution[7]. Some people prefer the three specs [8], but the majority advocating the four specs variant, which is listed on the ~~official~~ webcomponents.org website. For the purpose of this article, all four specs will be discussed briefly to provide a rough understanding. It is not meant to cover all bits and pieces.

Custom Elements (w3c)

Custom elements are the fundamental building blocks for web components. Essentially, they provide a way to create custom HTML tags enriched with behavior, design and functionality. An obligatory **HelloWorld** will help to grasp the spec:

```
> main.js
class HelloWorld extends HTMLElement {
  constructor() {
    super(); // mandatory!
    this.onclick = e => alert("hello");
    this.addEventListener(...);
  }
}
customElements.define('hello-world', HelloWorld)

> index.html
<hello-world>say hello</hello-world>
```

Most obvious, this spec relies on the new *ES6 Class Syntax* in favor of the original prototype-based inheritance model. “Extending `HTMLElement` ensures the custom element inherits the entire DOM API and means any properties/methods that you add to the class become part of the element’s DOM interface.”[9] Like any other *ES6 class*, *custom elements* can be sub-classed further on with the typical `extends` inheritance.

The beauty of *custom elements* comes with the keyword `this` which points to the element itself. Instead of querying and assigning behavior AFTER creation of the node in question `this` functionality allows a **declarative programming style**. Assigning functionality happens right in place BEFORE creation or insertion of the DOM. The so called *fat-arrow* (`=>`) is just a new feature of ES6 and nothing more than an anonymous function().

After definition, the element needs to be registered in the new global build-in `customElements` with an tag name like `<hello-world>`. Mind the dash inside the tag name to conform the spec. Finally, the new element can go live inside the HTML Document `index.html`.

Lifecycle methods: In addition to the `constructor()`, the spec defines so called *lifecycle callbacks* for controlling the **behaviour in the DOM**. Many popular frameworks like ReactJS or AngularJS rely on similar approaches:

- `connectedCallback()`
Called upon the time of *connecting or upgrading the node* which means the moment the node is inserted and rendered inside the DOM. Typically this block of code contains setup code, such as fetching resources or rendering.[9] For performance issues it’s highly preferable to put much code in here.
- `disconnectedCallback()`
Called upon the time of *node removal*. Cleanup code like removing eventListeners or disconnecting websockets can be put here.

- `attributeChangedCallback(attrName, oldVal, newVal)`
This method provides an *Onchange handler* that runs for certain attributes called with three values as defined in the signature. It is meant to control an elements’ transition from on `oldVal` to a `newVal`. Due to performance issues, this callback is only triggered for attributes registered in an *observedAttributes* array.
- `adoptedCallback()`
Called when moving the node *between documents*.

Custom attributes: As previously mentioned, the custom elements must **extend** the `HTMLElement`. Consequently, the new element inherits properties and methods from it (and it’s parent `Element`) and things like `id`, `class`, `addEventListener`, ... run out-of-the-box. Additionally, it is possible to define custom attributes using the *custom elements’ getter / setter interface* to steer the behavior of the element.

```
> main.js
class HelloWorld extends HTMLElement {
  ...
  set sayhello(val) {
    this._hello = val;
    console.log(this._hello);
  }
  get sayhello() {
    return this._hello;
  }
}
customElements.define('hello-world', HelloWorld);
var el = new HelloWorld();
el.sayhello = "earth";
el.sayhello; //"earth"
```

While getters and setters work great in the JS world they fail crossing the boundaries to the corresponding HTML node. Declaring `<hello-world sayhello="mars"></hello-world>` would’t work in the previous setup. A common workaround is archived by using the previous mentioned `attributeChangedCallback` lifecycle method to **reflect changing HTML attributes to JS** and/or map JS attributes to HTML with `this.setAttributes(...)` respectively. On **insertion time** html attributes might raise their hand with `this.hasAttributes(...)` and `this.getAttributes(...)`. Native DOM properties will reflect their values between HTML and JS automatically.[10, Para. 2.6.1]

Concluding this section, a reader might already discover the **mental model** behind *web components*. A custom element is similar to a named function where attributes treated as **input variables**. In the hierarchical nature of DOM, input can occur either top-down via assignments and bottom-up via captured events. The same goes true when talking about output. Even though it seems obvious,

it might be helpful to keep this point in mind.

Customized build-in elements: One aspect didn't mentioned yet is the possibility of creating sub-classes of **build-in elements** by extending the native Interfaces like the `HTMLElement` interface. While this functionality is perfectly spec'd it is strongly rejected by some browser vendors.² Most likely the spec will change in future in one or other way on this issue and therefore **customized build-in elements** left out of this paper intentionally.

Shadow DOM (*w3c*)

The second most important concept of *web components* rewards to the *shadow DOM* spec. The concept isn't the easiest to wrap ones head around. Basically it's an isolated DOM tree living inside an hosting DOM tree. The spec referencing the hosting tree as **light tree** and the attached DOM as **shadow tree**. Conceptually, the Shadow DOM issues a single topic: **Style Encapsulation**.

With an ever increasing complexity of an single-page application, the global nature of the DOM creates a challenging situation for the web developer and often ends up in highly fragmented bits of CSS and obscure CSS selectors. Of course, this situation lowers code clarity and reusability dramatically. The only solution which won't break with the existing global paradigm effectively is to allow separate pieces of encapsulated code sit on top of the global DOM - introducing the shadowed DOM approach!

Enhancing the previous example the new encapsulated `HelloWorld` would like this:

```
> main.js
class HelloWorld extends HTMLElement {
  constructor() {
    super(); // mandatory!
    const shadowRoot =
      this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML = "hello";
    this.onclick = e => alert("hello");
  }
}
```

The new global method `attachShadow` adds a new document root to the `HelloWorld` which has the same properties as a normal DOM. Therefore, it's possible to call the `innerHTML` method to fill the new document (fragment) with some content. Note that `shadowRoot` is attached as **open** which ensures that some events can bubble out and outside JS can reach in the new root. Nested children and other content in the light DOM are "shadowed" by the new root and must be invited in by so called **slots**.

Slots: Macroperspective / Composition

Assumptions about custom elements: Apart from the spec'd perspective there is mental model a webdeveloper might

Creating and using webcomponents might require a new mental model how do design a

containers

<http://alistair.cockburn.us/Hexagonal+architecture>

MVC Pattern

Pure frontend vs heavy backend

PROGRESSIVE ENHANCEMENT

Chapter about progressive enhancement

[1] D. Abramov, "Presentational and Container Components - Medium." 2015 [Online]. Available: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0. [Accessed: 01-Dec-2016]

[2] S. Newman, *Building microservices*. O'Reilly Media, Inc., 2016 [Online]. Available: http://www.ebook.de/de/product/22539693/sam_newmann_building_microservices.html

[3] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," Jan. 2014 [Online]. Available: <http://www.martinfowler.com/articles/microservices.html>

[4] R. C. Martin, "The single responsibility principle." [Online]. Available: http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle

[5] M. E. Conway, "How do committees invent?" 1968 [Online]. Available: http://www.melconway.com/Home/Committees_Paper.html

[6] B. Issa, "The way of the web." Polymer Summit 2016, Oct-2016 [Online]. Available: <https://www.youtube.com/watch?v=8ZTFEhPBJEE>

[7] D. Buchner, "Demythstifying web components," 2016 [Online]. Available: <http://www.backalleycoder.com/2016/08/26/demythstifying-web-components/>

[8] A. van Kesteren, "Mozilla and web components: Update," 2014 [Online]. Available: <https://hacks.mozilla.org/2014/12/mozilla-and-web-components/>

[9] E. Bidelman, "Custom elements v1: reusable web components." 2016 [Online]. Available: <https://developers.google.com/web/fundamentals/primers/customelements/>. [Accessed: 01-Dec-2016]

[10] *HTML living standard — last updated 11 january 2017*. [Online]. Available: <https://html.spec.whatwg.org/multipage/>

²Discussion on the topic: <https://github.com/w3c/webcomponents/issues/509>