Browsernative Microservices

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

Januar 2017

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.

Contents

Simplicity and the web

> Simplicity is prerequisite for reliability. - Edsger W. Dijkstra

Computers can scale, humans can't. Ever since a program or complex system made by humans has been constrained by humans mental capabilities. Like in the analogy of juggling balls, our brain can just "juggle" a few things at a time. Rich Hickey, the inventor of the programming language Clojure gave an inspirational keynote on the topic of **simplicity**.[1] In every sphere of a humans life, simplicity aligns perception with our mental capacities.

Derived from the ancient Latin word **simplex**, simple can be understood as "literally, uncompounded or onefold"[2] which points directly to the unidimensional aspect. While complexity describes the multilayered und entangled nature of conditions, simplicity empowers the human brain to reason about issues in a straightforward manner. It certainly has some overlapping's with easy, but while easy is more of a relative nature, simple can be laid out as a objective manner and therefore universally applicable.

Software development is undoubtedly rich in complexity and full of subtle pitfalls. In a typical scenario, a piece of software evolves over time in one or another opinionated direction. Layers of new abstractions wrestling with old legacy abstractions and mutation becomes untraceable. Subtle bugs start to creep in. Eventually the small piece of software may end up in a highly complected monolith which will determine future design decisions to a painful degree. Future strategies of the company/organization will be highly determined by the current state in the need of "keeping the lights on".

On the other side of this dystopian scenario, the truly modular system architecture abandons many of those inconsistencies. The whole system is divided in pluggable parts, object mutation is either traceable or avoided altogether in favor of immutable data structures. As Rich Hickey argues, design decisions should be made under the **impression of extending, substitution, moving, combining and repurposing**. The ability to reason about the program at any given time is crucial for future decisions and implementations. Recalling again the unidimensional nature of simplicity.

---

[1] Rails Conf 2012 Keynote: Simplicity Matters by Rich Hickey

[2] Etymology Dictionary

Simplicity in "the web", read as a loose generalization of "everything that runs in the browser", is certainly a story full of misconceptions and will be explained further along. While simplicity in the backend is mostly a matter of principles and patterns, any browser-based frontend is restricted on the highly deterministic nature of the browser platform.

In the last four years the average transfer size of a webpage doubled to currently around 2.5 MB.[3] Subtracting images, fonts or other content the size of HTML, CSS and JS sums up to a total average of 550 kb. One character weights around 1 byte which means an average webpage is delivering 550.000 character or around 125 pages of single-spaced text. Frederic Filloux analyzed the payload on different newspaper websites and came to the conclusion, that only round about 5-6 % of the transferred characters made for human consumption.[1]

Having an 95 % overhead is rather undesirable for both the consumers and creators of the website. Since it's a widespread problem without a single point of failure one can argue the platform itself is the failure. By design, every pageload results in a monolithic DOM tree managed by the browser engine. Whether rendering just a bunch of static text nodes or an ever changing webapp the underlying global nature of the DOM tree remains the same. Every additional piece of code added to the webpage will invisibly add another fold of complexity to this global object.

In an non-deterministic runtime environment, encapsulation and modularization is a typical pattern to make complexity manageable and accommodate future uncertainty.[2, p. 1] Since years the average JS payload is steadily rising which can be interpreted as a trend towards more dynamic websites. The demands to the browser platform changed from a static page renderer to a **dynamic UI machine** without changing the underlying architecture significantly. Under the current situation only additional layers of abstraction can wrestle complexity.

In the recent years many **frameworks**, libraries and methodologies approached the global nature of the DOM by scoping assets and design rules into maintainable components. While the DOM can't be scoped, JS can. Many frameworks, like ReactJS, AngularJS or VueJS just to name a few, ditched the old rule of separated HTML, CSS and JS in favor of an additional layer of abstracted JS components (containing content, markup and styling). Quiet often those frameworks mimic a MVC pattern on top of the browser engine which is a reasonable simple design pattern to build graphical user interfaces. While frameworks are a valid approach to build scalable web applications they remain highly opinionated, embody inherent complexity themselves and can change and break over time. Another downside is code inflation which is a crucial point for performance and third-party libraries are no exception on that. A standardized way for creating complex UI interfaces painlessly requires new build-in browser capabilities.

In the year 2013 thinkers, creators and browser vendors joined together to

---

[3]HTTPArchive Trends

propose *The Extensible Web Manifesto*.[4] The claim of the manifesto was to enhance the current web platforms with new low-level capabilities. Those features should empower creators of the web to write more declarative code and therefore overcome known bottlenecks and artificial abstractions. Four years later, the enhancement of JavaScript leapfrogged and many new low-level APIs brought to life. With this new APIs at hand a vivid web developer can create robust websites with less code and less additional libraries. This paper is an approach to unfold these **browsernative** technologies to create overall simple and resilient **microservices** for the browser

Disclaimer: This paper introduces many new browser build-ins with the focus on try and test. As the time of writing, many examples can be tried frictionless in the console of the latest versions of **Google Chrome, Opera and Apple Safari**.[5] On Mozilla Firefox technologies work behind a flag and Microsoft Edge implementation is unfortunately far behind. But Browser implementation changes quickly and soon technology adoption won't be an issue. Meanwhile new standards can be used through **polyfills** even on legacy browsers.

Microservices

In search of a better, simpler web architecture we might look on already established pattern that proofed to fulfill enterprise needs. Microservices are a good approach for tearing big monolithic systems into fine-grained simple services with explicit defined boundaries. In a nutshell a microservice is a small, autonomous service that works together with other services seamlessly.[3, p. 2] Or with the words of Fowler and Lewis: "... the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."[4] Just yet at this point a reader might spot some similarities with microservices and the browser-based development: Both wrestling the problem of monolithic architecture and both using lightweight communication mechanisms. In fact, many big companies of "the web" like Amazon or Netflix successfully transformed their monolithic system into a service based system which gives a glimpse of the power behind microservices.[4]

Microservices incorporate many ideas from developing scalable software, like *domain-driven design* where it pursues the incorporation of real world structure in the code.[3, p. 2] Or making use of *continuous delivery* for pushing software rapidly through *automated deployment* mechanisms into production.[4] Furthermore, microservices transcendent the technical perspective and reaches into the team organization. As a primary source of truth this paper relies on the work of Sam Newman, who has written a comprehensive guide called *Building Microservices* and the work of Fowler and Lewis. The purpose of this section

---

[4]The Extensible Web Manifesto
[5]Are we componentized yet?

is to gain confidence about the microservices architecture in the context of the browser platform.

Componentization via Services

"A **component** is a unit of software that is independently replaceable and upgradeable." [4] Components are the building blocks of microservices. And microservices are the building blocks of applications. Essentially the difference between microservices and components is just the level of abstraction. Whether a concrete microservice or a much more generic component, both share a similar set of principles. Therefore this paper referring to both parts when talking about **services**.

The first principle of services is the **loose coupling principle**: changing and deploying one service shouldn't result in changing other parts of the system.[3, p. 30]. Mutations or shadowed variables, which is happening a lot in CSS, making it hard to keep changes ought to only affect one place in the application. A *browsernative microservice* therefore pushing encapsulation and avoiding variable mutations outside its scope as much as possible. Practically, CSS will be scoped and JS fosters immutable JS entities and avoids variables leaking into the global namespace.

The second principle of services is the **high cohesion principle**: Whether designing a microservice or it's components we want related behavior sit together, and unrelated behavior to sit elsewhere.[3, p. 30] High cohesion can be expressed in a dynamic way as the *Single Responsibility Principle*: "Gather together those things that change for the same reason and separate those things that change for different reasons."[5] In a very quick and dirty code quality analysis, the quality can be measured just by counting the places changes in the code occur in order to implement a functionality. An arbitrary threefold MVC system should require a maximum of three changes to implement or change functionalities. The problem in browser based development is not only the global paradigm which makes changes deliberately unpredictable. The high cohesion principle is violated by the traditional separation along the technological entities **HTML, JS and CSS**. Understanding the relation of HTML markup and another CSS file adds incidental complexity. Different approaches emerged over the recent years to join the forces. Most notable the dedicated frontend language *elm*[6] where model, view and controller sit in one place. A browsernative service aims for a combinement of the web native trinity HTML, JS and CSS.

Traditional web development relies on **libraries** to enhance the service capabilities of the web platform. Compared to libraries, a component service offers multiple advantages for building, deployment and shipping. As expressed earlier a *components for the web*, or web component, is **self-contained** which means it embodies all needed functionality to get it's job done. Therefore it has a much

---

[6]elm lang

better evolution mechanism in the service contracts. Changing functionality won't break other services. A component can progressively enhanced which guarantees functionality throughout different versions. A library is only loosely coupled to the implementation and therefore hard to track in functionality. Changing a library may result in an unforeseen amount of time fixing implementations. It is not unusual to see websites embodying different versions of the same library to guarantee functionality but lowers page performance significantly.

Another issue where web components stand out is related to performance and especially the critical first page rendering. Libraries for the browsers are traditionally "shipped" as non static immediately-invoked functions run immediately on page load. Following the *Google RAIL* model, a user-centric performance measurement, a page load ought be less than 1 second to catch up user's attention.[6] There are many ways for optimizing the critical first render but as a rule of thumb a build-in web component might be always superior to libraries in terms of first rendering. The fine-grained lifecycle methods which will be described in the technical section of this paper give the developer far reaching optimization opportunities.

The last argument in favor of components over libraries is the more explicit interface.[4] While the functionality of a library needs documentation to be accessible a component functionality is exposed via the components' signature. The markup language HTML is by design equipped with an expressive syntax and steering a web component using attributes and values should be most straightforward even to unexperienced web developers.

Organized around Business Capabilities

> "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations". [7]

Emphasizing the human factor in microservices is a key feature. Microservices are a product of real-world usage.[3, p. 1] Instead of splitting team structures along the technology stack (UI Experts -> Middleware -> Database) a microservice approach model teams around **business capabilities**.[4] Consequently every team is capable of planning, designing, implementing, testing and maintaining their very own microservice. Along the technology stack every member gains high competence about the functionality of the service architecture which can be critical for evolving the service over its whole lifecycle.

Real-world domains tend to be complex and multifaceted. To unfold their complexity, domains can subdivided into **bounded context**.[3, p. 31] For example, customer service is a business domain but with varying bounded context. One context can be sales, another context could be support. Every context makes different assumptions about the underlying model and draws an explicit interface where it decides what to share with other contexts.[3, p. 30]

Evaluating each bounded context within each business domain will eventually shape the data structures and services in question. This methodology can be iterated over and over again. A sales service for example might be evaluated to different sales context resulting in differing interfaces on devices shaped by browsernative microservices.

By assigning service responsibility to a team, the so called **Definition of Done** (DoD) shifts from "accomplishing projects" to "accomplishing products". This new paradigm not only changes the administrative overhead like budgeting or resource allocation. It creates a kind of responsibility connection from the team to the service. Expectedly, those teams are more motivated within their very own service and exhibit a more sophisticated iteration time.[4]

For many companies working in the spheres of the internet the client side is highly important for their business. In fact, business goals and capabilities can be derived from the front-end needs. The state of the web is not only a story of numerous artifacts, it is also a story of an highly fragmented market along devices, operating systems, differing sizes and functionalities. Different devices again have different assumptions about the technology stack. Splitting teams along the stack results in an slow paced back and forth negotiation for every change to be made. As browser technologies, design guidelines and devices change frequently it makes absolutely sense to shift responsibility towards the teams altogether. By focusing the service into clear defined business boundaries, it is easier to define a smart API of the service.

Smart endpoints and dumb pipes

To ensure the microservice functionality among teams and different services requires thoughtful decentralization. Emphasizing once more the real-world capabilities of microservices a message channel architecture can be derived from patterns known from traditional postal services. A physical letter has only two smart endpoints entitled to read and process the message while packaging likewise the connection itself is maximized towards unification. *Smart endpoints and dumb pipes* is coined to the approach of designing communication mostly decoupled and as cohesive as possible.[4] Applying this rules to the web platform can result into building unified JSON message objects passed along "dumb" middleware components. A typical browser event comes close to this definition and suits arguably well for in-memory communication between web components and microservices.

Microservices heavily rely on simple HTTP request-response with resource APIs and lightweight messaging.[4] REST APIs are technology agnostic which aligns well with the idea of decentralization.[3, p. 247] The advantage of this overall simple communication model is the suitability for both frontend-backend likewise backend-backend communication. A service therefore can evolve from an heavy backend with a lot of network roundtrips to a leaner backend seamlessly. The

browser build-in fetch API which is essentially a HTTP request can be heavily incorporated into a browsernative microservice to ensure communication between services.

Decentralized Governance

Microservices are separate entities and decentralization is important to ensure autonomy. Distributed service responsibility owned by the team promoting shared governance and technology agnostic communication channels work towards a mostly decoupled service architecture.

This distributed nature allows teams to create their own technology stack, tools and services designed in the spirit of language- and platform independence and share their knowledge with other parties.[4] In the recent years many big companies like Facebook, Google, Netflix and others followed that spirit and published their ideas and implementations open source. The previously mentioned ReactJS for example is a product of Facebooks need to ensure a consistent frontend experience. In fact, many tools and techniques are byproduct of vital interaction of concrete domain problems and their implementations.

The spirit of freedom can't be applied universally to *browsernative microservices* as the browser and its underlying DOM will be the limitation factor to a certain degree. Talking about the browser, a reader might be tempted to narrowly thinking of the obvious VIEW layer only - which is not true anymore. In the recent years the browser engines grows to a kind of virtual machine: there are connectors to build-in databases, multithreading support, an ever-growing JS build-ins and even speech synthesis. So-called *Progressive Web Apps*[7], a bunch of criteria for building good browserapps, can achieve a similar look and feel like native apps. Services like NativeScript[8], effectively compiling "the web" to native code, lower the boundary between native and browser code even further.

JS is the widely accepted language of the web. Nevertheless, a microservice engineering team might choose another language for various reasons. Transpiling languages to JS as target language isn't exotic anymore. Languages like TypeScript, ClojureScript or PureScript compile to JS even exclusively. Once web components hit a critical mass there will be most likely some library support or foreign function interface towards ES6 modules (which are mandatory for the new specifications). With the rise of WebAssembly, a new low-level programming language for the browser, the determination on JS will hypothetical deteriorate and new quasi native languages for the web might gain traction.

Another more real life decentralization aspect derives from the easiness of deployment in a safe, sandboxed environment. Web components virtually ship no overhead or require obscure build tools. This makes them ideal candidates

---

[7]Progressive Web Apps
[8]NativeScript

for sharing and open source publishing. Webcomponents.org[9] is a registry for ready-to-use components of every scale and purpose where for even Google shares a lot of their material design elements.

Decentralized Data Management

Data Management in a microservice follows the same modular philosophy like the service implementation. As mentioned earlier different bounded contexts make different assumptions of the underlying models. A *browsernative microservices* takes this idea even further and expands it to the fragmented world of electronic devices. Decentralized decisions about conceptual models demand for decentralized data storage decisions.[4] Todays web architectures aims to leverage an increasing amout of processing to the client to avoid time-consuming roundtrips especially in mobiles networks.[10] Since network roundtrips are costly it is a good advice to only query as much data as needed and cache as much as possible. The build-in LocalStorage or its successor IndexedDB are mature persistence technologies and libraries like PouchDB[11] even offer backend syncing out of the box.

"Microservices prefer letting each service manage its own database."[Fowler2014] Ben Issa, chief architect of ING Australia emphasizes this pragmatism on APIs in a conference talk. At ING the frontend demands tailor the backend APIs, APIs may be produced automatically and not even Issa knows how many APIs exists.[8] They are using a pattern called **backend for frontends** empowering the team working to craft their UI and backend in a one-to-one relationship.[3, p. 72]

To see this pattern in the field a reader might have a look at Facebooks GraphQL[12]. GraphQL is a query language for the frontend. The backend solely replies on the frontend needs. Another well documented example in the field is Cognitects Datomic[13], where parts of the database will be reflected to the client. A so-called Transactor ensures ACID compliance.

The simplified microservice example later in this paper assumes a generic build-in API accompanied by build-in frontend components. Instead of gluing frontend and backend together on runtime the microservice is designed holistically containing both front- and backends. For the sake of simplicity data management won't be explored into depth throughout this paper.

---

[9]Webcomponents.org
[10]Latency numbers: https://gist.github.com/jboner/2841832
[11]PouchDB
[12]GraphQL
[13]Datomic

Infrastructure Automation

In the global nature of web development the development couldn't completely decoupled from the production environment. This circumstance left developers switching back and forth between files developing tricky opinionated (and more often biased) ways to glue related parts together. With scoped components, code blocks can be developed more sane and conveyed into production without headache. Even more, in the scripted environment of the browser this can be implemented rapidly and continuous.

Previously mentioned Ben Issa, described the ING standard workflow. Every component deserves a own **git repo** containing

- Internationalization conformity (i18n)
- Accessibility conformity (a11y)
- Tests for the component
- Demos of the component
- Blueprints to mock the one to one APIs
- Docs

Every check-in is handled as release candidate and can be independently deployed by a fully automated deployment machinery.[8] Even though this example is an opinionated perception it gives a sense of a mature component build for the web. Due to an exhaustive amount of testing and deployment tools for JS an automated infrastructure shouldn't be a problem.

Design for failure

In theory a microservice is designed with a lot of emphasizes on real-time monitoring for both the architectural elements and business relevant metrics.[4] Due to the modular structure weak points can occur in the orchestration of the services. Testing and automation is a feasible task but failures may occur in another end users setting undetected. Legacy browsers for example remain a problem for enhancing websites with new technologies.

Regarding the evolution of the web, the "next billion" internet users most likely using Android, have decent specs mobile phones, use an evergreen browser but won't have a reliable internet connection.[9] While *Progressive Enhancement* was once coined on the principle to build websites both for Browsers with JS and HTML only, the new *Progressive Enhancement* tends towards an "**offline first**" build principle. A *Browsernative Microservice* therefore not only tries to cache data as much as possible, it should also bring in a lot of program logic as described in the previous chapters.

Working in a JS heavy infrastructure demands for optimization to avoid unexpected side-effects like the *flash of unstyled content (FOUC)*. Googles Polymer

propagates the a general-purpose **PRLP pattern**[14]:

- Push critical resources for the initial route
- Render initial route
- Pre-cache remaining routes
- Lazy-load and create remaining routes on demand

Following this pattern a critical resource can detect browser functionalities beforehand and switch to a **polyfill** instead of the latest browser optimized version. After the initial paint, critical resources like top-level microservices or other app logic can be loaded and registered.

Evolutionary Design

Microservices tend to become smaller over time. An evolutionary design approach puts emphasizes on decomposition and scrapping the service. "The key property of a component is the notion of independent replacement and upgradeability."[4] Therefore we can safely change and chop services. Lazy parts of the system which won't change often should be separated from parts undergoing a lot of churn.[4] Parts that needs coupled changes could should be moved together or should be even merged.

This flexible approach fits good in the world of browser based development. In the last decade we have seen a lot of of changes in the way we develop for the web. New approaches like the virtual DOM approach found their way into mainstream web development followed by frameworks and libraries.

Browsernative microservices should be perceived as complementary technology. Contrary to Angular, React and other frameworks they have a strong interop with existing systems and can be used with them together.

Andrew Rota, developer at Wayfair, came up with the idea to compose small web components around a managing React system.[10] As web components are native elements there is no difference to use a native `button` or a native `custom-button`. With this pattern a web developer can make use of the encapsulated advantages of web components while still making use of the declarative event management from various frameworks. Whatever new framework will be on the rise within the next years, this approach allows rapid decomposition and reassembling towards a new system.

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

---

[14]PRLP pattern

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[11]. Some people prefer the three specs [12], but the majority advocating the four specs variant, which is listed on the quasi-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 introducing the *Single Responsibility Principle* to the browser. In essence, they provide a way to create **custom HTML tags** subsuming behavior, design and functionality. An obligatory **HelloWorld** will give a flavor about the spec:

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

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

This example should be self-explanatory. Notably, custom elements come in the fashion of *ES6 Classes*[15] in favor of the normal JavaScript prototype-based inheritance model. This class must inherit the base `HTMLElement` interface which "ensures the newly created element inherits the entire DOM API and any properties/methods that you add to the class become part of the element's DOM interface."[13] Like any other *ES6 class*, the new element can be specialized further on using 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, custom elements ship their functionality PRIOR initialization of the element. The so called *fat-arrow* (`=>`) is just a new ES6 syntax feature for an anonymous function declaration.

---

[15] JavaScript Classes

After declaration, the new HTML element needs to be registered in the global build-in `customElements` object with an tag name like `<hello-world>` acting as key to the element. 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 rendered inside the DOM. Typically this method is called straight after the `constructor()` on insert. Typically, this method contains setup code, such as fetching resources or rendering elements according to attributes.[13] For a fast initial render of the page, it is highly preferable to put many proceedings in favor of the constructor.
- `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` interface. Therefore, the new element inherits base properties and methods commonly used in all HTML elements like `id, class, addEventListner` . 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 {
 constructor() {...}
 set sayhello(val) {
  this._hello = val;
  console.log(this._hello);
 }
```

```
 get sayhello() {
  return this._hello;
 }
});
customElements.define('hello-world', HelloWorld);
// Instantiation
var el = new HelloWorld();
el.sayhello = "earth";
el.sayhello;//"earth"
```

Native DOM properties, like `id` or `onclick`, reflect their values between HTML and JS.[14, Para. 2.6.1] For example declaring the HTML `<hello-world id="hello">` like this equals to assign the ID in JS like `Node.id = "hello"`. This behavior won't work out-of-the-box with methods or properties defined setters. For example declaring `<hello-world sayhello="mars"></hello-world>` would't call the `sayhello` function in the previous setup.

A common workaround to bind HTML and JS behavior together is archived by using the aforementioned lifecycle method `attributeChangedCallback` to **bind changing HTML attributes to JS properties** and to map JS attributes to HTML with `this.setAttributes(...)` respectively. On insertion time HTML attributes might trigger custom JS methods retrieving HTML attributes using `this.getAttributes(...)` method.

Concluding this section, a reader might already discover the **mental model** behind *web compontents*. 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 `HTMLButtonElement` interface. While this functionality is perfectly spec'd it is strongly rejected by some browser vendors.[16] 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)

A *shadow DOM* is basically an isolated DOM tree living inside an another (hosting) DOM tree. The spec refers the hosting tree as *light DOM tree* and the

---

[16]https://github.com/w3c/webcomponents/issues/509

attached DOM as *shadow DOM tree*. Conceptually, the *shadow DOM* issues a single important topic in software development: **Encapsulation**. While the first spec *custom elements* provides a sufficient way to encapsulate JS behavior, *shadow DOM* coined strongly to in the direction of style encapsulation.

With an ever increasing complexity of an single-page application, the global nature of the DOM creates a daunting situation for code organization and leads over times to highly fragmented bits of CSS and obscure CSS selectors or html wrappers. 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() {
  ...
  this.attachShadow({mode: 'open'});
  shadowRoot.innerHTML = '<p>hello</p>';
 }
}
```

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

Slots

Contradicting to the simplified `HelloWorld` example, a *shadow DOM* shouldn't contain any ~~valuable~~ content. While technical possible any change of an element would require deeply nested calls from the *light DOM* to the *shadow DOM* to update the element in place. That's why *shadow DOM* should be perceived more as **static HTML template** and provide therefore a kind of internal frame for the render engine. `Slots` are placeholders for *light DOM* nodes used to mark the endpoints in question.

Technically, the *light DOM* nodes are not moved inside the *shadow DOM*. Their just rendered in place. It's an subtle but important difference towards handling a node. JS behaviour and CSS styles applied in the *light DOM* will still be valid in the *shadow DOM*. The render engine literally taking the nodes and putting

15

them inside the `slot`. This procedure is commonly referred as **flattening** of the DOM trees.

Named slots

A named slot is the preferable way for clear code organization. Taking for example `<slot name="hello">Drop me a "hello" node</slot>` targets all direct *light DOM* child nodes of the hosting node matching the slot name like `<div slot="hello"></div>`. Writing a little documentation inside the `<slot>` tag is considered as a good practice as it will be rendered only if no matching *light DOM* node is available. This functionality makes a *custom element* pretty much self-explanatory.

Unnamed slots

Inside a so-called *default slot* which looks like `<slot>Unnamed content goes here</slot>`, the render engine expands all direct *light DOM* children without a `slot` attribute. In case of multiple default slots, the first slot takes it all.

Styling

As mentioned in the last section, there is a distinct difference about the nature of nodes. Nodes declared and rendered exclusively in the *shadow DOM* are not affected by any styling from outside. Nodes which are declared outside and distributed via `slots` will be styled in the *light DOM* and can be additionally painted in the *shadow DOM* through the new CSS-Selector `::slotted()`.

Note that styles from the outside have an higher specify than styles assigned after distribution. Therefore it is generally a good advice to minimize the global stylings to some base styling for uniformity of the web site while leaving the specific stylings to the component. Due to the cascading nature of CSS, styles will still "bleed in" from ancestors to the *light DOM* nodes.

Regarding the importance style encapsulation, a couple of new CSS rules emerged that are exclusively targeting the *shadow DOM*. The table below outlines styling possibilities for the use INSIDE the *shadow DOM*:

- ::slotted(selector)
  Applies to distributed nodes and repaints them after distribution. `Slotted` won't override outsides styles but can complement them with unset style rules.
- :host
  The host property will add styles or change inherited ones inside shadow DOM. Using `all: initial;` will ensure browser defaults only.

- :host(condition)
  Like the previous one this node will style the shadow DOM but this time based on attributes/conditions assigned to the hosting node.
- :host-context(condition)
  Like the previous one this node will style the shadow DOM but will look after context set at the host node or even at the host ancestor.

Using the *functional selector* of `:host()` or even the only-functional `:host-contest()` allows the creation of **context-aware custom elements**. A possible use case would be "theming" a component (example taken from [15]):

```
> index.html
<body class="darktheme">
  <fancy-tabs>
    ...
  </fancy-tabs>
</body>

> fancy-tabs shadowRoot
<style>
:host-context(.darktheme) {
  color: white;
  background: black;
}
</style>
```

JS Behavior

As mentioned earlier any logic applied to *light DOM* nodes stays with the node even after redistribution. For the sake of separation of concerns the business logic should be part of the *custom element* (the *light DOM*) and not the part of the *shadow DOM*. On the other hand there are numerous scenarios where JS is just concerned with **styling or animation of an element**.In this case it might be more straightforward to apply JS inside the *shadow DOM* to avoid mixing with business logic handlers.

Drilling down to a *light DOM* node from an *shadow DOM* context is not possible with querying the node directly with `.querySelector()` or `.getElementById()` as the node is not part of the context. To get a distributed node in question it needs the way over the slot node and call `slot.assinedNodes()` to receive an array of distributed node(s) which can be accessed and manipulated like any other node. Calling `.assignedNodes()` on an empty `slot` returns an empty array.

Wrapping up this section, *shadow DOM* provides a non-hacky way to create uniform looking *custom elements* and even enhance styling possibilities without adding much overhead. Still, for smaller components with only one or two child

nodes, just a little styling and/or no structured redistribution a *shadow DOM* might be to hard to reason about. Eventually it all depends on the question of "how hard is it to implement it without shadow DOM" - which can't be answered universally. For a more in-depth guide, Google Engineer Eric Bidelman wroten a great primer on *shadow DOM*[15].

So far, there is still a missing link between *light DOM* and *shadow DOM*. The observant reader may have already noticed the weak point in the `HelloWorld` example: how to "vitalize" the *shadow DOM*. Recapturing the last `HelloWorld` example a string of markup was assigned to the `shadowRoot.innerHTML` property. While it works perfectly fine in this simple case, a string of markup is rather cumbersome and error-prone and doesn't scale well. When putting quotes inside another quotes things break quickly. It makes the life hard for developers to work with it because it requires manual indentation and is out of syntax highlighting. That's the time templates come into play.

HTML Templates (w3c)

Among all other new standards *HTML templates* are the most mature and adopted standard in the browser environment. All major browsers, except from Internet Explorer, support this standard.

One core concept in templates is efficiency: Whatever dropped inside a `template` tag ~~bucket~~ will be parsed on runtime - but not constructed into the *content tree*. It remains plain HTML Markup sitting somewhere in the document until the time of activation.

Activation typically takes four steps:

1. **Querying the template node in question**
   const node = document.querySelector('template');
2. **Parsing the content and preparing the templates' content**
   const content = node.content;
   -> Returns a *DocumentFragment* object. Handling is straight forward
   content.querySelector('img').src = 'logo.png';
3. **Optional: Cloning the *DocumentFragment* for multiple use**
   const clone = content.cloneNode("deep");
4. **Appending the clone/original to destination**
   document.body.appendChild(clone);

As easy and minimal *HTML templates* are, they're missing out a crucial feature other template implementations usually have. As templates are basically just dump containers for HTML Markup, there is no way to define some logic as **placeholders** where content should appear. Of course, with heavy use of JS things could be modeled this way. The idiomatic way tends more towards a *Shadow DOM & HTML templates* symbiosis.

18

```
> index.html
<hello-world>
 <p id="sendto" slot="placeholder">
  Hello World Web Component
 </p>
</hello-world>

<!-- COMPONENT STARTS HERE -->
<template id="hello">
 <!-- STYLES -->
 <style>
  #stylewrapper {
   font-weight: bold;
   color: orange;
  }
 </style>
 <!-- CONTENT -->
 <div id="stylewrapper">
  <slot name="placeholder">
   Named placeholder
  </slot>
 </div>
</template>

<script>
 // Switched to anonymous class notation
 // for keeping associated code together.
 customElements.define('hello-world',
  class extends HTMLElement {
   constructor() {
    super();
    this.attachShadow({mode: 'open'});
    const helloTemplate = document.querySelector('#hello');
    this.shadowRoot.appendChild(helloTemplate.content);
   }
  });
</script>
```

The updated `HelloWorld` component looks already pretty mature. It combines all the previous mentioned standards into one blob of HTML. *Custom Elements* serves the logic, *Shadow DOM* scopes the styles and *HTML Templates* efficiently glues DOM and *Shadow DOM* together. This separation of concerns comes with a huge gain in flexibility. In a real world scenario `HelloWorld` would contain/reference multiple *HTML Templates* and could switch them around without any fuss. A developer might to split up templates into **STYLE** templates and **CONTENT** templates to increase reusability even further.

The last standard in the row of four is not concerned with the internals of a *web component*. *HTML Imports* serves the need for an efficient distribution mechanism of components.

HTML Imports (w3c)

Importing the `HelloWorld` component is a one-liner:

```html
<link rel="import" href="Hello.html" async>
```

The `async` flag is optional but like in any other fetching event, strongly recommended. Once the imported HTML document comes into scope, activation follows a very similar pattern like the aforementioned *HTML templates*:

1. **Querying the link node**

2. **Parsing the content and preparing the render**
   const content = linknode.import; -> Unlike the *HTML template* the content a fully equipped document object.

3. **Optional: Cloning some nodes for multiple use**

4. **Appending the clone/original to destination**

This again is the imperative way to handle a generic *HTML Import*. In the declarative world of *web components* a component is activated, parsed and anchored solely by its' tag name `<hello-world></hello-world>`. Preliminary, the component needs proper configuration. The next section will elaborate the right configuration and composition of a component to work out-of-the-box.

Despite from being just a practical document importer *HTML imports* acts like a fully fledged dependency manager for the browser. Multiple resources, ranging from stylesheets, scripts, documents, media files and even other `imports` can be grouped together in a logical `import` statement. Internally, the browser engine keeps track for every imported resource so it won't be loaded twice. The inherent complexity is in fact a stumbling block for wider browser adoption. Currently only Googles blink web engine supports *HTML Imports* as they are the driving force behind the *web components* spec in general. Mozilla and Apple imposed distaste for *HTML Imports* as a whole. One reason for this can be found in the incompatibility of the spec with the upcoming *ES6 module loader*.[17]

Despite the discrepancies among browser vendors *HTML Imports* should still be part of the paper as no other native browser technology can bundle up CSS, JS and HTML that efficient.

---

[17]https://hacks.mozilla.org/2014/12/mozilla-and-web-components/

Appendix A: Custom Events (whatwg)

Events are first-class citizens in the browser world and *Custom Events* are no exception. The *Custom Elements* interface is part of the DOM since years but with the rise of *Custom Elements* they will most likely become an indispensable building block of *web components*.

```
> index.html
<hello-world>
 <button>Launch CustomEvent</button>
</hello-world>
<!-- COMPONENT STARTS HERE -->
<script>
 customElements.define('hello-world',
 class extends HTMLElement {
  constructor() {
    super();
   // Craft a CustomEvent e
    const e = new CustomEvent('hello-world', {
    bubbles: true, //important!
    detail: 'Contains scalar or object'
   });
   // Launch e on child button click
   this.addEventListener('click', click => {
    this.dispatchEvent(e)
    click.stopPropagation();
   });
   // Catch e. Typically done by some parent
     // node. Message in detail property
   this.addEventListener('hello-world', e => {
    console.log(e.detail)
   });
  }
 });
</script>
```

Naming events after the emitting tag makes the API almost self-explanatory. The `detail` property can be loaded with primitives as well as objects. The further sections will elaborate a feasible architecture for building a scaling microservice architecture.

Chaining and aggregating events from child nodes should be practiced and exercised quiet frequently. As mentioned earlier in the *Custom Elements* section, the pattern of **extending native elements** should be somewhat dismissed as it may never implemented outside of Chrome. Nevertheless, it is possible to create own kind of quasi native buttons when chaining a *CustomEvent* directly after the native click event.

Another typical *custom element* use case can be as an actor on (native) child elements. In this case, the *Custom Element* catches events from children, buffers or rebuild them and eventually fires an event towards the document root.

Unfortunately events only work "upstream" towards parent nodes. Still the web platform offers plenty of possibilities to talk back to child nodes.

Appendix B: Web Worker (whatwg)

Like *Custom Events*, *Web Workers* had been around for a long time and therefore enjoy full support among major browsers. They emerged at around 2009 when discussions about browser performance was still in the early days. Nevertheless, the addressed problem of *Web Workers* is a fundamental language problem of JS itself.

JS runs in a single-threaded language environment. Every script in the browser environment, from handling UI events to query and process larget amounts of API data and manipulating the DOM, runs on the same thread[16]. Putting a lot of work to the single main thread can slow down the web service significantly. From time to time scripts can block or fail for whatever reason which leads to a frozen or crashed UI. A worker can overcome the bottleneck of the single-threaded nature with spawning new **background threads** which allows the UI to stay responsive even when computation-heavy tasks should be carried out. Furthermore, a worker thread adds a performance advantage embracing the multi core CPU architecture most devices running on today. To grasp the full potential of workers, a reader might dive deeper into the Angular 2 architecture, where most of the application layer is abstracted from the main rendering thread into worker threads.[18]

A *Web Worker* spawns a new background thread where scripts can run concurrent to the main thread. Usually a worker is loaded from a workers dedicated file to embrace this kind of separation.

```
const worker = new Worker('worker.js');
```

After initialization a worker communicates over a simple **message based interface** with the main thread.

```
> main.js
// Send to worker
worker.postMessage('Hello World');
// Receive msg from worker
worker.addEventListener('message', e =>
 console.log('Worker said: ', e.data));

> worker.js
// Receive msg and echo back
```

---

[18]Angular 2 Rendering Architecture

```
this.addEventListener('message', e =>
 this.postMessage("Echo " + e.data));
```

Building an browsernative microservice

After getting confidence in microservice principles and technical background the paper should bring them both together to form a *browsernative microservices*. Needless to say the described example is overall simplified to only illustrate the connection between browsernative technologies and microservice patterns. Googles library Polymer is a good place for learning about web components in depth and make use of their simple command line tools. One of their most famous proof of concept is the so-called Polymer Shop which is a fully-fledged online shop nested within a single root element `<shop-app>`. This app made of several main views and many more invisible custom elements for routing, service worker caching, theming, etc. The whole shop runs as a single application only fetching and sending resources and switching views. Let's assume we work in a sales engineering team of the Polymer Shop and need to rebuild the checkout microservice.

The current checkout can be found at https://shop.polymer-project.org/checkout. Currently the checkout is a single, 671 lines of code long Polymer component incorporation all required fields for sign in, shipping, billing and summarizing the order. In the spirit of microservices we will split up the microservice into fine grained components. The shopping cart data is pulled out of a local storage JSON entity set up previously by another custom-element.

Checkout microservice

Like a usual checkout this example has 3 to 4 steps:

1. Sign in or Sign up
2. Shipping details
3. Payment details
4. Review and place order

Translated into a raw **Custom Element** HTML structure, the top-level microservice might look like the following snippet:

```
>shop-checkout.html
<shop-checkout>
  <sign-in></sign-in>
  <shipping-details></shipping-details>
  <payment-details></payment-details>
  <place-order></place-order>
</shop-checkout>
```

Yet already we see the simplicity arouse from web components as they persue a clear structure. Each of the child components should act independently over other child nodes utilizing the loose coupling principle. Each child ships all the HTML, CSS and JS code needed to fulfil its work following the high cohesion principle. Each component may contain different views to accommodate different bounded contexts resulting from different devices. And last but not least, all of them communicate over an unobstrusive message bus via the service root component `shop-checkout`.

Before diving deeper into implementation, its worth to clarify the **architectural pattern** behind components. Any reader of the paper came across ReactJS / Redux, the concept of components may look familiar. Dan Abramov, the creator of Redux, once defined a simple dichotomous pattern for creating UI components. Firstly, he came up with the idea of **presentational components** only related with the concern about *how things look*. This component literally doesn't know anything about the service in question which makes the component highly flexible and reusable. They are controlled solely from the outside, receiving data and dispatching unbiased events on user interaction.[17] Most probably every presentational component embodies more HTML/CSS markup and less JS code. It should encapsulate its styles from bleeding out and protect its styles being overwritten. Furthermore, it may contain several templates to change it's look on different demands.

Secondly, Abramov described components he refers as **containers**. A container component is concerned with *how things work*.[17] Containers acts as invisible wrappers around presentational components acting more in the sense of UNIX filters. Their job is to fetch data from child nodes, aggregating events, interacting with the model and push state back to the presentational components. Consequently, they might contain more JS and less, if any, HTML markup. We probably don't need to utilize ShadowDOM as no styles are involved.

Last but not least, there are **build-in components**, which is every build-in HTMLElement like the HTMLButtonElement. They are solely controllable and styleable from the outside and must therefore wrapped in presentational components and/or containers.

Lets start the service description top-down beginning with the **root container** managing the overall service.

Root container

The root container `shop-checkout` is basically just an encapsulation layer in terms of service functionalities. Encapsulation of CSS won't be necessary at this point as no styling is involved. On initialization the `shop-checkout` might perform following tasks:

1. Importing child components with **HTML Imports**

2. Initialize the service in a logical top to bottom order like in the snippet above
3. Initializing a **model thread** with `new Worker('checkout-model.js');`
4. Setting up a `postMessage` stub to dispatch messages to the model
5. Setting up an event listener for messages from the model
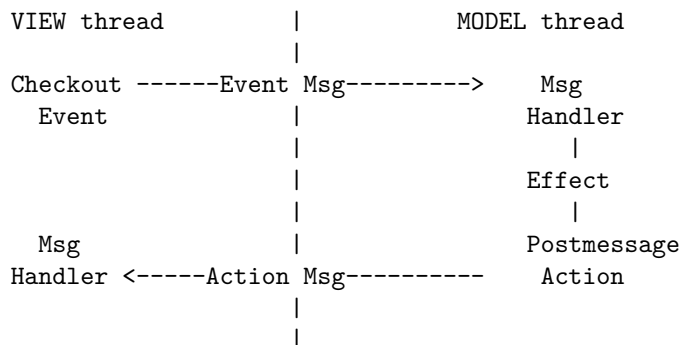6. Setting up an event listener for **custom events** from the child nodes

On runtime the root container acts mostly like a **message dispatcher** implementing the microservice idea of smart endpoints and dumb pipes.

Every custom child node is eligible aggregate subordinate events from their children, buffer them and interact with the model trough via a **dumb message system**. To talk to the model any child node can dispatch a **unified event containing a unified message object** like the following example illustrates:

```
// Within every child node class
set dispatch(msg) {
  this.dispatchEvent(
    new CustomEvent('checkout', {
      bubbles: true,
      detail: msg
    });
  );
}
```

Calling `this.dispatch(msg)` within a child node will trigger an event bubbling upstream. The service root will implement a simple event listener for `checkout` events. In a second step the service root add information about the original target (like *tag-name#id.className*) to the message so the message can be processed in the model thread and eventually pushed back correctly to the sender.

Summarizing the unidirectional model-view communication from the service root perspective looks like following sketch:

```
VIEW thread           |              MODEL thread
                      |
Checkout ------Event Msg--------->    Msg
  Event               |             Handler
                      |                |
                      |             Effect
                      |                |
  Msg                 |             Postmessage
Handler <-----Action Msg----------    Action
                      |
                      |
```

Effects are yielded by the asynchronous operation of messages and create actions returned to sender. Effects can be created by external messages, like subscription to an WebSocket, too. Effects may be created with additional resources from

the server or syncing with local storage like in the polymer-shop.

While the msg handlers are basic "dumb" switches the **smartness** solely arouse from intelligent controllers processing the message. Every component must implement a **unified socket** in it's service contracts like a property on the custom element:

```
// Within every child node class
set receive(msg) {
  // do something smart
}
```

This communication flow is designed to **fail graceful** as simply nothing happens if one or another switch is not defined. Furthermore it incorporates lots of possibilities for **evolutionary design** as child nodes can be loosely dropped in or teared apart. Middleware container wrappers can be pulled in between child nodes and service root to do some extra work and and even the message object can be enhanced freely as long as it won't break with the message conventions. Changing or enhancing any functionality requires only to touch the controllers in the child nodes in question and the endpoint section at the model. Furthermore, **infrastructure automation** might be achieved by dynamically evaluating the mounted child nodes and registering the "backend" model switches on build time. Due to it's standardized message system the components are testable within standardized tests.

To work in a larger scale, we might define a base component which has the default message and socket properties in its guts and extend every new element from this base class.

Another yet more powerful approach can be achieved by calculating the new shape of the DOM in the worker directly and just patch the DOM in the view layer.[19] This approach is certainly more complicated but worth to evaluate further more.

Presentational components

Thinking further

cohesive HTML JS CSS

Most obvious ist the gathering of all related code under the umbrellar of a single HTML tag.

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

---

[19] A good example of this approach can be found at Introducing Pokedex

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.

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"[4] 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.[8] 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.

dichotome Pattern

Ein üblicher eventgesteuerter Webservice setzt sich aus unterschiedlichsten Komponenten zusammen, die wiederum unterschiedlichste Eventlistener & -emitter in sich subsummieren. 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.

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

MVC Pattern

Pure frontend vs heavy backend

[1] F. Filloux, "Bloated html, the best and the worse," 2016 [Online]. Available: https://mondaynote.com/bloated-html-the-best-and-the-worse-cac6eb06496d

[2] C. Y. Baldwin and K. B. Clark, "Modularity in the Design of Complex Engineering Systems," in *Complex engineered systems: Science meets technology*, D. Braha, A. A. Minai, and Y. Bar-Yam, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 175–205 [Online]. Available: http://dx.doi.org/10.1007/3-540-32834-3{_}9

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

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

microservices.html

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

[6] M. Kearney, "Measure performance with the rail model." 2016 [Online]. Available: https://developers.google.com/web/fundamentals/performance/rail

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

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

[9] N. Lawson, "Progressive enhancement isn't dead, but it smells funny." 2016 [Online]. Available: https://nolanlawson.com/2016/10/13/progressive-enhancement-isnt-dead-but-it-smells-funny/

[10] A. Rota, "React.js conf 2015 - the complementarity of react and web components." 2015 [Online]. Available: https://www.youtube.com/watch?t=124&v=g0TD0efcwVg

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

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

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

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

[15] E. Bidelman, "Shadow dom v1: Self-contained web components." 2016 [Online]. Available: https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom

[16] E. Bidelman, "The basics of web workers." 2010 [Online]. Available: https://www.html5rocks.com/en/tutorials/workers/basics/

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