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 indispensable frameworks. While they perfectly circumstance the global paradigm of the DOM they remain highly proprietary attached to the framework. New w3c specifications build right into the browser provide a native service API in the need to create standardized web components for the browser. As of 2017 all major browsers will ship those technologies in their browsers. The paper not only provides an introduction to the specification. As an assessment criteria of it will match those specifications against the concept of microservices which is an modular system architecture to build scalable applications.

Contents

1. Simplicity and the web

Simplicity is prerequisite for reliability. - Edsger W. Dijkstra

Computers can scale, humans can't. Ever since a 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 Latin word **simplex** simple can be understood as "literally, uncompounded or onefold"[2] which points towards an unidimensional state. While complexity describes the multilayered und entangled nature of conditions simplicity empowers the human brain to reason about issues straightforward. It certainly has some overlapping's with easy, but while easy is more of a relative spirit, 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 growing software project evolves in one or another opinionated direction over time. Layers of abstractions wrestling with aged legacy code requiring layers of middleware. Mutating assets create subtle bugs and so forth. Eventually the small piece of software may end up in a highly complected monolith which will determine future design decisions to a painful degree. Opaqueness of the system will slow down innovation to a minimum in the need of "keeping the lights on".

On the other side of this dystopian scenario a truly modular system architecture abandons many of those potential inconsistencies. The whole system is divided in pluggable parts, object mutation is either traceable or avoided altogether in

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

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.

Simplicity in "the web", read as a loose generalization of "everything that runs in the browser" is certainly a story full of misconceptions. 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] Leaving images, fonts or other content aside the size of HTML, CSS and JS sums up to an 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 React, Angular or Vue 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 structure, behavior and sometimes styling). Quiet often those frameworks mimic a MVC pattern on top of the browser engine which is a reasonable simple design pattern to build UIs. While frameworks are a valid approach for building 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. All of those

[3] HTTPArchive Trends

bottlenecks in the web demand for a new standardized way for creating and evolving complex web services.

In the year 2013 thinkers, creators and browser vendors joined together to propose *The Extensible Web Manifesto*.[4] The manifestos claim 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 abandon known problems 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

2. Microservices

In search of a better, simpler web architecture one 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] 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 taste of the power behind microservices.[4]

Microservices incorporate a wide array of 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 towards microservices[3] and the work of Fowler and Lewis[4]. The purpose of this section is to gain confidence about the microservices architecture in the context of the browser platform.

[4]The Extensible Web Manifesto

2.1. 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] Picking CSS for example, where variables tend to be shadowed by higher specific values making it increasingly difficult 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. One solution would be scoped CSS / JS to avoid variables leaking into the global namespace as well as hidden implementation details to avoid mutating values.

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 enhanced towards the more dynamic **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 siloed 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*[5] where MODEL, VIEW and CONTROLLER sit in one place. A *browsernative service* sought for an combination 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 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

[5] elm lang

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.

2.2. 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 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 persistency model likewise the interface to the service. 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 differing devices shaped by browsernative microservices. By focusing the service into clear defined business boundaries, it is easier to define a smart API of the service.

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 which can be best described as **shared governance** model. "Each team collectively share responsibility for evolving the technical vision of the system."[3, p. 247] 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 frontend 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 make iteration time expensive. As browser technologies, design guidelines and devices change frequently it makes absolutely sense to shift responsibility towards the teams altogether.

2.3. 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] Newman puts his recommendation on technologic-agnostic REST APIs to free data persistence from implementation constraints.[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 to services in the backend.

2.4. Decentralized Governance

Microservices are separate entities and decentralization is important to ensure autonomy. This paper already described fragmented services bounded to singular business context choreographed by simple communication protocols developed and evolved by autonomous teams.

This distributed nature empowers 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 major browser engines grow to to a fully-fledged app deployment platform offering connectors to build-in databases, multithreading support and ever-growing JS build-ins like speech synthesis or push notifications. So-called **Progressive Web Apps**[6], a bunch of criteria for building good browser apps, can achieve a similar look and feel like native apps. And last but not least services like NativeScript[7] effectively compiling "the web" to native machine code lowering 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 is a stable solution nowadays. 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).

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 dedicated build tools. This makes them ideal candidates for sharing and open source publishing similar to the largest JS package registry

[6] Progressive Web Apps
[7] NativeScript

NPM. In the spirit of NPM web components can be perceived as frontend packages with an HTML interface instead of a JS signature. Webcomponents.org[8] 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.

2.5. 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.[9] 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[10] even offer backend adapters for 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[11]. 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[12], 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.

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

2.6. Infrastructure Automation

Microservices tend to increase complexity as this model adds a sheer number of moving parts to the system whereas requires proper orchestration.[3, p. 246] Arguably every more sophisticated web developer already came across build tools like Webpack or infrastructure automation tools like Gulp. Testing and deploying web components shouldn't be an obstacle in development.

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. Bret Victor, UI designer at Apple defined the importance of an **immediate feedback principle** for developing user interfaces.[13] In his talk he emphasizes the importance of an immediate connection between the creator of a product and product itself. Any change must results in an immediate visible feedback. Web components catch up with this principle as they allow isolated development within a single file containing all bits and pieces of the web component. Every major browser devtool offers a direct file manipulation functionality so development can be even in place.

When it comes to standardized deployment guidelines previously mentioned Ben Issa, described the ING standard workflow. Every component comes in its 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

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

2.7. Design for failure

In theory a microservice is designed with focus on 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. A microservice should track down every communication flow and provide defaults and meaningful error

[13] Bret Victor - Inventing on Principle

messages where communication stuck. Testing every single component with predefined synthetic events ensures functionality. Nevertheless, browser support may vary and legacy browsers remain a general problem for enhancing websites with new technologies and therefore demand further configuration.

Combinment of the resources in the browser always demanded for optimization to avoid unexpected side-effects like *flash of unstyled content*. Googles Polymer propagates the a general-purpose pattern called **PRLP**[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 evaluate browser maturity beforehand and switch to a **polyfill** or another fallback solution 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.

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 related to build websites both for browsers with and without JS support the demands changed tends towards an **offline first** principle avoiding network connectivity failures.[9] 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.

2.8. 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 components of the system which won't change often should be separated from parts undergoing a lot of churn.[4] And services which change for the same reason might be moved together or even could be merged.

Pursuing flexibility for web development is a selling point as innovation cycles for browser development is fast paced and technologies can change quickly. Frontend related hardware, software and methodologies innovate rapidly over time.

Browsernative microservices should be perceived as complementary technology in contrast to full-service frameworks like Angular. Being native technology pursues a strong interopt approach with existing systems. Andrew Rota for example came up with a pattern using small, encapsulated and stateless web components

[14] PRLP pattern

as leaves in the tree of React components instead of native HTML elements.[10] Even React can eventually profit from the expressiveness of custom components using a custom `<meaningful-button>` over a native `<button>`. Most likely there will be always some cutting edge framework promising advantages over native code. Whatever new framework will be on the rise within the next years native components can eliminate future uncertainty and allowing rapid reassembling towards new architectures.

3. 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. A web developer 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.

Disclaimer: This paper introduces many new browser build-ins with the focus on testability. 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**.[15] 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.

3.1. Custom Elements (whatwg)

Custom elements are the fundamental building blocks for web components introducing the **Single Responsibility Principle** to the browser. In short, they provide a way to create custom HTML tags subsuming behavior, design and functionality. An obligatory **HelloWorld** will give a flavor about the spec:

```
> HelloWorld.js
class HelloWorld extends HTMLElement {
 constructor() {
```

[15] Are we componentized yet?

```
  super(); // mandatory in constructor
  this.onclick = e => alert("hello");
 }
}
customElements.define('hello-world', HelloWorld)
```

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

This example should be almost self-explanatory in functionality. *Custom elements* come in the fashion of **ES6 Classes**[16] in favor of the normal JavaScript prototype-based inheritance model which was part of an older specification. Every valid element must extend 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 any *custom element* can be specialized further using the typical inheritance model allowing higher levels of abstraction.

The beauty of *custom elements* comes with the **bounded `this` keyword** which points to the element itself. Instead of querying and assigning behavior after creation of the node, custom elements ship their functionality on initialization of the element. The so called *fat-arrow* (`=>`) is just a new ES6 syntax feature for an anonymous function declaration.

After declaration the new HTML element needs to be registered in the global build-in `customElements` object with an dedicated tag name acting as key to the element. Mind the dash inside the tag name to conform the spec. Finally, the new element can be mounted inside the HTML document.

3.1.1. Lifecycle methods

In addition to the constructor which runs procedures on initialization, the spec defines **lifecycle callbacks** for controlling elements behavior towards DOM interaction. Many popular frameworks like React or Angular 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 if the node is inserted directly. For a faster initial render of the page it is highly preferable to put many proceedings in this method. Usually this method contains setup code such as fetching resources or rendering elements according to attributes.[13]
- disconnectedCallback()
  Called upon the time of **node removal**. Cleanup code like removing event listeners or disconnecting web sockets can be put here.

[16] JavaScript Classes

- attributeChangedCallback(attrName, oldVal, newVal)
  This method provides an **onchange handler** for certain elements attributes. This method is used to guide elements' transition from an old value to a new state. Due to performance issues this callback is only triggered for attributes registered in a dedicated array shipped with the element.
- adoptedCallback()
  Called when moving the node **between documents**. This method comes handy when using HTML imports described later.

3.1.2. Custom attributes

As mentioned earlier any custom elements must extend the `HTMLElement` interface ensuring base properties and methods used in throughout 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. Note that the keywords **get** and **set** as well as the previously used **constructor** are optional!

```
> HelloWorld.js
class HelloWorld extends HTMLElement {
  set sayhello(val) {
    this._hello = val;
  }
  get sayhello() {
    return this._hello;
  }
}
customElements.define('hello-world', HelloWorld);
// Instantiation via JS instead of HTML
var el = new HelloWorld();
el.sayhello = "earth"; // "Call" the setter
el.sayhello;// Yields "earth"
```

Native DOM properties always reflect their values between HTML and JS.[14, Para. 2.6.1] Declaring `<hello-world id="hello">` equals to the JS declaration `new HelloWorld().id = "hello"`.

This behavior won't work out-of-the-box with methods defined with setters as they are strictly JS only . Mounting `<hello-world sayhello="mars">` would't call the `sayhello` method in the previous setup. Value reflection can be implemented inside the *custom elements* using the native methods `getAttributes` and `setAttributes`. Using them exhaustively throughout lifecycles methods the new components can configured to read and listen to HTML attributes accordingly.

Designing a *custom element* this way creates HTML elements with named

attribute interfaces reaching deep into JS functionality. With this mental model in mind a web developer can create highly dynamic web components.

3.1.3. Customized build-in elements

One aspect didn't mentioned yet is the possibility of extending other build-in elements by extending other interfaces instead of the `HTMLElement` interface. While this functionality is perfectly spec'd it is strongly rejected by some browser vendors.[17] 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.

3.2. Shadow DOM (w3c)

A *shadow DOM* is just an isolated DOM tree living inside an another DOM tree. The spec refers the hosting tree as **light DOM tree** and the attached DOM as **shadow DOM tree**. Conceptually *shadow DOM* issues a single important issue for building scalable software which is namely **encapsulation**. While custom elements provide a good way to encapsulate JS behavior *shadow DOM* tends strongly to the direction of style and event encapsulation.

With an ever increasing complexity of single-page applications the global nature of the DOM creates a daunting situation for code organization and leads over times to highly fragmented bits of CSS and obscured CSS selectors. Of course this situation lowers code clarity and reusability dramatically. The only solution which won't break with the existing global paradigm of the DOM 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:

```
> HelloWorld.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, light document object. Note that the `shadowRoot` object is marked as **open** which ensures that some events can bubble out and outside JS can reach in the new root.

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

Filling the *shadow DOM* with an `innerHTML` string is rather impractical. To fill a *shadow DOM* with life usually it invites light DOM child nodes nested under the hosting node using a technique called `slots`.

3.2.1. Slots

Contradicting to the simplified `HelloWorld` example, a *shadow DOM* shouldn't contain dynamic content. Changing or interacting with the paragraph node from the example would require nested JS calls querying the hosting node, entering the *shadow DOM* and applying a function. Imported JS behavior from third-party libraries in the *light DOM* can't be used inside the *shadow DOM*, too.

That's why the shadowed documents should be more perceived as **static documents** filled and managed solely by the render engine. `Slots` are target areas for *light DOM* nodes used to mark the endpoints in question.

```
> index.html
<HelloWorld-with-ShadowDOM>
  <!--
    All child nodes will be moved inside the
    shadowRoot if shadowRoot.innerHTML = '<slot></slot>'
    -->
  <p>hello I will be scoped</p>
</HelloWorld-with-ShadowDOM>
```

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. All JS behavior and CSS styles applied in the *light DOM* will be valid in the *shadow DOM*. The render engine literally taking the nodes and drop them inside the `slot` tag. This procedure is commonly referred as **flattening** of the DOM trees.

It's possible to add semantics to the *shadow DOM* in naming the slots which frees the *light DOM* from the responsiblity to deliver nodes in a correct top-to-bottom order. Combining *shadow DOM* with HTML templates provides the web developer with a flexible **HTML template engine**.

FROM: light DOM	TO: shadow DOM
`<p slot="hello">Named</p>`	`<slot name="hello">hello only</slot>`
`<p>Unnamed</p>`	`<slot>Unnamed nodes</slot>`

Writing a little documentation inside the `<slot>` tag is considered as a good practice as it provides the developer with visual clues what nodes must be delivered. This functionality makes a *shadow DOM* pretty much self-explanatory. Inside a default slot tag the render engine expands all *light DOM* children without a named `slot` attribution.

3.2.2. 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 distributed will be styled in the *light DOM* and can be additionally painted in the *shadow DOM*.

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. It is possible to **reset all styles** inside the *light DOM* before distributing nodes using the `all: initial` reset. To ensure a consistent look between different shadow roots this technique should be used carefully.

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 previously unset style rules.
- :host
  The host property will add styles or change inherited ones inside shadow DOM. Aforementioned style resets can be placed here.
- :host(condition)
  Like the previous rule this selector will style the shadow DOM but this time based on attributes/conditions assigned to the hosting node.
- :host-context(condition)
  Like the previous rule this selector will style the shadow DOM but will look after context set at the host node or even at the host ancestor.

Using the **functional selectors** `host()` or `host-context()` allows the creation of context-aware custom elements. A possible usecase would be "theming" a component.

3.2.3. 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 used for 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 logic handlers with styling.

It seems that a *light DOM* node is in the *shadow DOM* context after distribution. Visually this may be true but logically the node stays in the normal document. To reach a distributed node from the *shadow DOM* to apply some JS behavior for styling it needs the extra way over the slot node. Calling `assinedNodes()` on the hosting slot element returns a linkage to the distributed node which can be accessed and manipulated like in the *light DOM* context.

Wrapping up this section *shadow DOM* provides a non-hacky way to create uniform looking custom elements and even enhance styling possibilities without adding overhead. For small components with just a little styling a *shadow DOM* might be over engineered. 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].

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* with slot or other element structuring instead of `innerHTML` strings. While strings 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 other quotes things break quickly. Strings make development harder because code editor features like indentation or syntax highlighting won't be supported. The HTML templates is set to fill this gap.

3.3. HTML Templates (whatwg)

Among all other new specifications *HTML templates* are the most mature and adopted standard in the browser environment. All major browsers support it since years.

One core concept in templates is browser performance. Elements inside a `template` tag will be parsed on runtime - but not constructed and rendered into the content tree. They're remaining plain HTML markup sitting in the document until the time of activation.

Activation usually 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.
3. **Optional: Cloning the fragment 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 used to have. As templates are basically just containers for HTML markup there is no idiomatic way to define **placeholders** for dynamic content. Templates could be mock up this way using JS for altering the content but the much cleaner way leverages the previously described slot technique from shadow DOM.

```
> hello-world-component.html
<hello-world>
  <template id="hello">
  <!-- styling -->
    <style>
      #hellowrap {
        font-weight: bold;
        color: orange;
      }
    </style>

    <!-- structure -->
    <div id="hellowrap">
      <slot name="placeholder">
        Named placeholder
      </slot>
    </div>
  </template>

  <!-- light DOM / typical inserted at index.html -->
  <p slot="placeholder">
    Hello World Web Component
  </p>
</hello-world>

<script>
  class HelloWorld extends HTMLElement {
    constructor() {
      super();
      this.attachShadow({mode: 'open'});
      const temp = this.querySelector('template#hello');
      this.shadowRoot.appendChild(temp.content);
    }
  }
  customElements.define('hello-world', HelloWorld);
</script>
```

The updated `HelloWorld` component looks already pretty mature. It combines all the previous mentioned standards into one HTML file. Custom elements serves the logic, shadow DOM scopes the styles and *HTML Templates* efficiently

glues light DOM and shadow DOM together. This separation of concerns comes with a surplus in flexibility. In a real world scenario `HelloWorld` could reference multiple *HTML Templates* and switch them around without any fuss. Even further a developer might split up templates into named **STYLE** templates and **CONTENT** templates to increase reusability even further.

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

3.4. HTML Imports (w3c)

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

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

<hello-world>
  <p slot="placeholder">
    Hello World Web Component
  </p>
</hello-world>
```

The `async` flag is optional but recommended like any other fetching event. Once the imported HTML document comes into scope activation follows a very similar process compared to 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* a complete document object is constructed.
3. **Optional: Cloning some nodes for multiple use**
4. **Appending the clone/original to destination**

Again this is the imperative way to handle a generic *HTML Import*. In the declarative world of web components a component is **parsed, auto-activated and anchored** solely by its' tag name `<hello-world></hello-world>` on purpose. The very own lifecycle method **adoptedCallback()** in custom elements shows the strong interconnection between those standards.

HTML imports can import everything which is wrappable with HTML markup. Stylesheets, scripts, documents, media files and even further imports statements can form a semantic *HTML import* statement. The far reaching possibilities of a single standard has its downsides. According to Mozilla *HTML imports* are not fully compatible with the dependency model of the new ES6 modules.[18]

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

Under the current situation Mozilla and Apple and Microsoft won't implement *HTML Imports* soon if any. Currently only Googles blink web engine supports *HTML Imports* as they are the driving force behind the web components specs in general. Despite the discrepancies among browser vendors *HTML Imports* are part of this paper as no other native standard can ship bundled HTML, CSS and JS which is a core concept behind web components.

3.5. Custom Events (whatwg)

Events are first-class citizens in the browser. After all they provide a neat communication channel for dynamic interactions. *Custom Events* are part of the DOM since years but with the rise of web components they will most likely become an indispensable building block of web components.

```
> hello-world-component.html

<hello-world>
 <button>Launch CustomEvent</button>
</hello-world>

<script>
 customElements.define('hello-world',
 class extends HTMLElement {
  constructor() {
    super();

   // Craft a helloevent
    const helloevent = new CustomEvent('helloevent', {
    bubbles: true,
    detail: 'Contains scalar or object'
   });

   // Launch helloevent if child button clicks
   this.addEventListener('click', click => {
    this.dispatchEvent(helloevent)
    click.stopPropagation();
   })
  }
 });
</script>
```

Naming events after the emitting tag makes the API almost self-explanatory. The `detail` property can be loaded with scalars as well as objects. Subsequent parent nodes may catch the custom event with a clear understanding about the source node.

Chaining and aggregating events from child nodes can be frequently used within web components. One use case could be creating a custom button like in the previous example. As mentioned earlier in the Custom Elements section at 3.1.3. , the pattern of **extending native elements** should be somewhat dismissed as other browser vendors may never implement it. A common workaround to eventually create a own version a button could be made with a thin wrapper around the native button and chaining a *Custom Event* after the click event.

Another common web components use case can be a **middleware** subscribing to certain child events and acting upon them. This involves catching events, buffering, destructuring and creation of an own event towards the document root. By design, events only bubble upstream towards parent nodes. For handing down information towards the child nodes we need to query the child node directly.

3.6. 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 but addressing fundamental performance bottleneck of the JS language.

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 data or manipulating the DOM runs on a single 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 needs to perform. 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.[19]

A *Web Worker* spawns a new background thread where scripts can run concurrent to the main thread. Usually a worker is loaded from dedicated file to embrace 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');
```

[19] Angular 2 Rendering Architecture

```
// Receive msg from worker
worker.addEventListener('message', e =>
 console.log('Worker said: ', e.data));
```

```
> worker.js
// Receive msg and echo back
this.addEventListener('message', e =>
 this.postMessage("Echo " + e.data));
```

4. Building a browsernative microservice

After getting confidence in microservice principles and technical background the paper should briefly join them to form a *browsernative microservices*. Needless to say the following example is overall simplified towards illustrating the connection between browsernative technologies and microservice patterns. Furthermore it is highly opinionated in development and shouldn't perceived as "single source of truth".

Googles library Polymer is a good place for learning about web components in depth and make use of their toolbox. 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 wrapper elements for routing, service worker caching, theming, etc. The whole shop runs as a single application fetching and updating remote 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. At the time of writing the checkout is a single, 671 lines of code long Polymer component including 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.

By breaking down the service the **team defined the business boundaries** within the checkout process and ended up with following granular service blocks:

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 markup. Each of the child components may 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 **unobtrusive message bus** via the service root component `<shop-checkout>`.

Before diving deeper into implementation, its worth to clarify an **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, Abramov defined a pattern around **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 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 Shadow DOM as no styles are involved.

Last but not least, the pattern can expanded for illustrational purposes to **native components** which is every build-in HTMLElement like the HTMLButtonElement. Native components are mostly deep nested elements providing the actual functionality in the browser UI. They are solely controllable and styleable from the outside and are therefore wrapped in presentational components and/or containers.

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

4.1. Service root

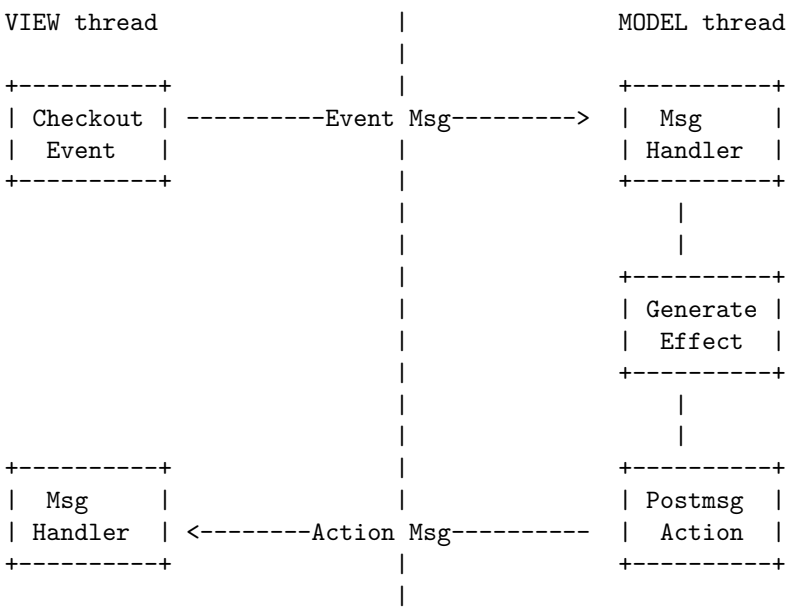
The root container `<shop-checkout>` is basically just an encapsulation layer in terms of service functionalities. Encapsulation of CSS won't be necessary at this point as no styling is involved. A simplified root container for the checkout might look like the following code snippet.

```
> shop-checkout.html
<!-- IMPORTS -->
<link rel="import" href="sign-in.html" async>
...
<!-- VIEW -->
<shop-checkout>
  <sign-in></sign-in>
  ...
</shop-checkout>

<!-- CONTROLLER -->
<script>
class ShopCheckout extends HTMLElement {
  constructor() {
    super();
    this.MODEL = new Worker('checkout-model.js');
  }

  connectedCallback() {
    // listen to msg from MODEL
    this.MODEL.addEventListener('message', [Msg Handler])
    // listen to child nodes
    this.addEventListener('checkout', e => {
      // delegate event to handler
      this._sendToMODEL = e;
    });
  }

  set _sendToMODEL(e) {
    const id = e.target.id,
          name = e.target.localName,
          load = e.detail,
          letter = Object.assign({}, { id, name, load });
    // send to MODEL
    this.MODEL.postMessage(letter);
  }
  ...
}
customElements.define('shop-checkout', shopCheckout);
```

```
</script>
```

The purpose of this simplified code snippet is to outline the **MVC threefold** in the service root. The MODEL is pushed into the worker thread, the CONTROLLER in the custom element and the VIEW in the HTML structuring. On runtime the root container acts like a **message dispatcher** implementing the microservice principle of smart endpoints and dumb pipes.

In order to interact with the MODEL every child node must implement the **dedicated `checkout` custom event** (which will be explained further on in the next section). Summarizing the unidirectional communication from the service root perspective looks like following plot:



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" switch statements the **smartness** solely arouse from intelligent controllers processing the message. This communication model offers lots of possibilities for **evolutionary design** as child nodes can be loosely dropped. Every child node can be expanded into another full microservice or split up into separate nodes without notice of the service root. Communication might **fail graceful** providing a default console log in case switches aren't defined yet.

4.2. Container components

The second layer of the checkout microservice like `<sign-in>` or `<shipping-details>` still contain mostly logic and no or less styling. Their job is to control their underlying presentational leaf components, aggregating events and to define a **set of actions towards the model** . Every container mounted directly under the service root may have a dedicated area inside the worker thread where it's fulfil his duties. For example, a typical `<sign-in>` contains merely two fields username and password and a submit button. The component listens for the submit action, aggregating the credentials and might add some semantics to it like *action: 'SIGNIN SUBMITTED'*. Fields and action message will be dispatched towards the model for further processing like initiating a authorization process.

Every container component is eligible to aggregate subordinate events from their children, buffer them and interact with the model trough via a **unified message system**. A base class, from which `<sign-in>` or `<shipping-details>` can be extended might look like this:

```
class SimpleContainer extends HTMLElement {
  // constructor, static methods ...
  set _dispatch(msg) {
    this.dispatchEvent(
      new CustomEvent('checkout', {
      bubbles: true,
      detail: msg
      })
    );
  }
  set _receive(msg) {
    // switch to methods
    console.log(`${this.localName} received ${msg}`);
  }
}
```

Extending the `SimpleContainer` will equip every container node with the unified message interface. Calling `this._dispatch(msg)` within the container will trigger an event bubbling upstream. The service root will implement a simple event listener for `checkout` events. After receiving an answer from the MODEL a service root can query the container child node in question and push forward the answer over the `_receive` property.

There might be even more middleware containers pulled in between presentational leafs and the service root to fulfil some extra work either like filters on the event or without caring about checkout events altogether. Changing or enhancing any functionality requires only the controllers in the container in question and the endpoint section at the model. **Infrastructure automation** might be achieved

by dynamically evaluating the mounted containers beneath the service root and modeling the "backend" model accordingly. Due to it's standardized message system the containers are testable within standardized tests.

4.3. Presentational components

In comparison to the former container the presentational component is build mostly around views and styling containing less or no logic towards handling the component. To extend the last `<sign-in>` container example a typical presentational component structure could look like the following top-level structure

```
<sign-in-container>
  <sign-in-presentational>
  </sign-in-presentational>
</sign-in-container>
```

The `<sign-in-presentational>` contains all required input fields and oauth connectors to Google or Facebook but **will not care about events they create**. A presentation component is usually steered via HTML attributes. Contrary to the former containers the presentational component highly utilizes **shadow DOM and HTML templates**.

```
> sign-in-presentational.html
<!-- IMPORTS -->
<link rel="import" href="facebook-oauth.html" async>
<link rel="import" href="high-res-template.html" async>
...

<!-- VIEW -->
<sign-in-presentational>
  <high-res-template>
    <facebook-oauth></facebook-oauth>
    <slot name="form"></slot>
    <slot name="button"></slot>
  </high-res-template>

  <template id="lowres">...</template>
  <template id="mobile">...</template>
</sign-in-presentational>

<script>
  class SignIn extends HTMLElement {
    constructor() {
      super();
      this.attachShadow({mode: 'open'});
      // append templates according to window.innerWidth
```

```
      // run this.getAttribute('htmlprop') to control element
    }
    attributeChangedCallback(attrName, oldVal, newVal) {
      // react on changing attributes
    }
  }
  customElements.define('sign-in-presentational', SignIn);
</script>
```

Presentational components require configuration and usually ship a long list of CSS selectors. Whereas it's possible to expand templates in place it could be also possible to define **template components** to atomize the component further and increase code readability.

tests

5. Thinking further

Following a **browsernative first** dogma this paper didn't put third-party libraries into view so far in the search of an independent approach. Nevertheless, it should be clear to this point that the example in the last section is far away from being usable in production. At first, from a technical perspective web components are relatively young and at the time of writing only usable in **newer Chrome and Opera browsers**. Most probably it will take a long time until native web components can be safely used without additional legacy support. The troublesome *HTML imports* spec will certainly need even longer which inevitable opens up the question if web components are useful from today on. And second, do web components align with the notion of simplicity if too many downsides and possible pitfalls require attention?!

Any profound discussion about web components would be incomplete without touching component alternatives like **React**. In the last couple years React introduced new concepts to web development like the idea of state, passing data as properties and declarative event management just to name a few. Those design decisions were made to overall simplify browser development with full browser support right from the start. Overall, Reacts perception about UI development created much of a hype around the library and technologies like the virtual DOM while web components never caught up in momentum at a comparable rate. According to Github Googles Polymer project is even six months older than React but has less than one third of its stars. Other notable web components projects like Bosonic[20] or Mozilla's X-tag[21] are more dead than alive, too.

It seems like the web developing community in a broader sense values **standardized procedures, tooling and browser support** in frameworks like React

[20] https://github.com/bosonic/bosonic/
[21] https://github.com/x-tag/core

(and many others to be fair). Web components and related libraries clearly missed a real selling point towards React and Angular. Providing guidelines, simpler syntax and features missed out or being to cumbersome to use at all. For the future it remains unclear if web components will eventually gain wider adoption within the UI developing community or will be adopted as low-level technology for framework development emphasized by Sebastian Markbage, one of the React creators.[18]

Even though the downsides seems heavy weight there are notable attempts to bring web components into production. A very good example is the "reactish" library Skate[22]. Following the functional rendering model from React it combines native methods with additional functionalities known from React like declarative event management. Given the focus on native technologies it only weights around 4kb (minified and gzipped) which is ten times less than React. **Small encapsulated micro UI frameworks** like Skate could certainly be the near future of web development and will hopefully pave the way of web components.

The role of web components may not result in an isolated framework but move towards a common denominator for future Reacts, Angulars, Vues... which makes it worth considering it as recyclable technology foundation. The more mature UI framework Riot already expressed their intention to gradually develop towards web components.[23]

[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.

[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:

[22] https://github.com/skatejs/skatejs/

[23] http://riotjs.com/compare/#web-components

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]

[18] D. A. Rauschmayer, "Tree-shaking with webpack 2 and babel 6." 2015 [Online]. Available: http://www.2ality.com/2015/12/webpack-tree-shaking.html