Browsernative Microservices

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

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

Simplicity and the state of 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 for the human brain. 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 example, the truly agile system architecture is laid out in a fine-grained manner. Software becomes pluggable, mutations traceable and modules interchangeable. 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.

The **state of the web** is certainly a different kind of complexity beast. "The web" is coined to "everything that runs in the browser". While simplicity in the backend is mostly a matter of principles, any frontend developer is highly 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 calculated the ratio of real content 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 service added to the webapp embodies another fold of complexity.

In an non-deterministic runtime environment, encapsulation and modularization is a typical pattern to make complexity manageable and accommodate future uncertainty.[2, p. 1] Today many websites tend towards becoming complex single-page applications (and therefore the average JS payload is soaring too). Consequently, the demands to the browser platform changed from a static page renderer to a **complexe UI machine**. Under the current situation only additional layers of abstraction can handle 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. Therefore 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

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

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. Furthermore, code inflation in front-end 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 2013 thinkers, creators and browser vendors joined together to propose *The Extensible Web Manifesto*.[4] The claim of the manifesto is 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 scoped and highly reusable microservices without additional libraries even directly in the browser console.

Disclaimer: This paper introduces many new browser build-ins with the focus on try and test. As the time of writing, examples can be tried frictionless in the consoles 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

Calling the case for *Browsernative Microservices* brings up the question about the concept of microservices in general. In fact the concept of microservices has many facets, stretching beyond disciplines and technical boundaries. It lacks a formal standardization but subsumes certain ideas emergine from this pattern. 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 is to match their ideas against the browser platform as targeted runtime.

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] Microservices incorporate many other ideas, like *domain-driven design* where it pursues the incorporation of the real world in our code.[3, p. 2] Or making use of *continuous delivery* for pushing software rapidly through *automated deployment* mechanisms into production.[4] And, last but not least, microservices utilizes the idea of small teams with a lot of product knowledge working mostly autonomous on their very own service with their very own set of tools and techniques.

Componentization via Services

"A **component** is a unit of software that is independently replaceable and upgradeable." [4] Components are the building blocks of the microservice but in fact they even share a lot of similarities just on a smaller scale. One of the similarity is the *loose coupling principle*: changing and deploying one service shouldn't result in changing other parts of the system.[3, p. 30]. Same goes for the component which should know as little as it needs about the service to fulfil it's job.

Any reader of the paper coming from the ReactJS / Redux world this concept of components might look familiar. Dan Abramov, the creator of Redux, once defined a simple dichotomous pattern for creating UI components. In his believe, there are **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.[5]

On the other hand Dan Abramov outlines components which he calls **containers**. A container component is concerned with *how things work*.[5] Containers acts as invisible wrappers around presentational components. Their job is to fetch data from child nodes, aggregating events, interacting with the model and push state back to the presentational components. In contrast to their presentational counterparts, containers can't life on its own without children. Conceptually, they show another important principle of components and microservices in general: *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 perceived as the *Single Responsibility Principle* defined by Robert C. Martin: "Gather together those things that change for the same reason and separate those things that change for different reasons."[6] 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 an arbitrary MVC system a `button` might be inserted in the VIEW, the CONTROLLER needs some adjustment and maybe the the MODEL, too. Three places for adjustment is a reasonable

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

easy task for the brain, everything more violates the concept of simplicity.

Using libraries in web development is common sense. But compared to libraries, a component service offers multiple advantages. 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 (like with JQuery). Another issue with libraries is dead code elimination which means the process of removing code that is never going to be executed. Newer build tools for the web, like Webpack 2 or Rollup offer this feature which relies heavily on the static structure on ES6 modules.[7] Contrary libraries for the browsers are traditionally "shipped" as **immediately-invoked functions**. Bootstrapping those closed functions is much harder to archive as even dead code is actually executed on load.

Web components are necessary static ES6 modules and any capable bundler can choose which component is needed at runtime. Even more granular, a good web component is composed of static JS blocks which can be strapped away too.

A web component is self-contained which means it embodys 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 components. A component can progressively enhanced which guarantees functionality throughout different versions.

Compared to libraries componentized services offers a more explicit interface.[4] While the functionality of a library needs documentation, a component functionality is exposed via the components' signature which comes in the fashion of an HTML element. For example, a button can consume attributes and becomes a primary buttons just by assignment.

```
<my-botton isPrimary>Primary</my-botton>
```

Organized around Business Capabilities

> "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations". [8]

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.

Real-world domains tend to be complex and multifaceted. To unfold their complexity, domains can subdivided into bounded contexts.[3, p. 31] For example, dealing with customers is a business domain. Customers again have different contexts depending on their demands. One context can be sales, another context could be support. Every context makes different assumptions about the underlying model. Each bounded context draws an explicit interface where it decides what models to share with other contexts.[3, p. 30] Each context can derived into multiple microservices or, talking about *browsernative microservices*, interfaces to the customers.

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 in a fast paced manner it makes absolutely sense to shift responsibility towards the teams altogether. A *browsernative microservice* embraces commitment over its whole lifecycle.

Smart endpoints and dumb pipes

Microservices for the browsers aim for side-effects like changing the UI given to input parameters or emitting browser events. An isolated microservice won't make too much sense after all. To ensure communication between services, flexible yet powerful communication channels must be established.

Smart endpoints and dumb pipes is coined to the approach of designing communication mostly decoupled and as cohesive as possible.[4] In the analogy to the real world a message channel should look like sending a letter: Two smart endpoints (sender and receiver) and a mostly unified "dumb" letter format and channel. Each endpoint owns their specific domain logic. In the browsernative context we already emphasized the role of presentational components and containers. The "smartness" of presentational components is altering the UI and reacting on users input. The logic of a container follows the idea of an filter in a Unix sense

- receiving a request, applying logic as appropriate and producing a response.[4]

Cited earlier in this paper microservices often rely on simple HTTP request-response with resource API's and lightweight messaging as communication protocols.[4] Those technologies highly accessible and widely adopted and most probably usable for a browsernative service. In the implementation part of this paper a reader will explore the combination of REST protocols and a lightweight browser message bus in action.

Decentralization

Decentralization is a meta concept of microservices. As written earlier decentralization is part of the organizational structure by assigning service responsibility solely to the team. Or by gluing independent service bricks to a whole distributed system talking over the network protocols.

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 brainchild of Facebook. 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 to *browsernative microservices* as the browser and its underlying DOM will be the limitation factor. Still there are some decentralization aspects around creating microservices for the browser worth to mention.

JS is the widely accepted language of the web. Nevertheless, a microservice engineering team might choose another language for different 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 interfaces 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 of *browsernative microservices* which makes them a good fit for open source. They are small, leightweight and can be tested live in the browser without any additional tooling. Webcomponents.org for example is a registry for ready-to-use components of every scale where even Google shares a lot of their material design elements.

Decentralization will be achieved just because of the encapsulation of the service. Grouping together HTML, JS and CSS Code in a safe, sandboxed environment exposes the possibility to build more cohesive and understandable services. In the typical global nature of web development those three pillars are separated. This circumstance left the developer switching back and forth between code bases developing a tricky (and sometimes biased) way to glue related parts together.

Infrastructure Automation

Design for failure

Chapter about progressive enhancement

Evolutionary Design

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

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 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.[9] A very common structure is formed by the technology stack (UX Designers, Frontend- & Backend Developers) or by separating teams along the product lifecycle (development, testing, deployment).

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

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

W3C specifications

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

Depending on the context, some people argue for only two specs which essentially make it possible to create a scoped component but not caring too much on it's distribution[10]. Some people prefer the three specs [11], but the majority advocating the four specs variant, which is listed on ~~the official~~ webcomponents.org website. For the purpose of this article, all four specs will be discussed briefly to provide a rough understanding. It is not meant to cover all bits and pieces.

Custom Elements (w3c)

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

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

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

Most obvious, this spec relies on the new *ES6 Class Syntax* in favor of the original prototype-based inheritance model. "Extending `HTMLElement` ensures the custom element inherits the entire DOM API and means any properties/methods that you add to the class become part of the element's DOM interface."[12] Like any other *ES6 class*, *custom elements* can be sub-classed further on with the typical `extends` inheritance.

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

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

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

- `connectedCallback()`
  Called upon the time of *connecting or upgrading the node* which means the moment the node is rendered inside the DOM. Typically this method is called straight after the `constructor()` on insert. This block of code contains setup code, such as fetching resources or rendering elements according to attributes.[12] For performance issues it's highly preferable to put much code in here.
- `disconnectedCallback()`
  Called upon the time of *node removal*. Cleanup code like removing eventListeners or disconnecting websockets can be put here.
- `attributeChangedCallback(attrName, oldVal, newVal)`
  This method provides an *Onchange handler* that runs for certain attributes called with three values as defined in the signature. It is meant to control an elements' transition from on `oldVal` to a `newVal`. Due to performance issues, this callback is only triggered for attributes registered in an *observedAttributes* array.
- `adoptedCallback()`
  Called when moving the node *between documents*.

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

```
> main.js
class HelloWorld extends HTMLElement {
 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"
```

While getters and setters work great in the JS world they fail crossing the boundaries to the corresponding HTML node. Declaring `<hello-world sayhello="mars"></hello-world>` would't work in the previous setup. A common workaround is archived by using the previous mentioned `attributeChangedCallback` lifecycle method to **reflect changing HTML attributes to JS** and/or map JS attributes to HTML with `this.setAttributes(...)` respectively. On **insertion time** html attributes might raise their hand with `this.hasAttributes(...)` and `this.getAttributes(...)`. Native DOM properties will reflect their values between HTML and JS automatically.[13, Para. 2.6.1]

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.[6] Most likely the spec will change in future in one or other way on this issue and therefore **customized build-in elements** left out of this paper intentionally.

Shadow DOM (w3c)

The second most important concept of *web components* rewards to the *shadow DOM* spec. In terms of complexity this spec outpacing all others by far. 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 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 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

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

`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. Therefore it's strongly recommended to begin every *shadow DOM* with a ***CSS reset:*** `:host {all: initial;}`.

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 [14]):

```
> 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*[14].

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.

```
> 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 template =
     document.querySelector('#hello');
    this.shadowRoot
     .appendChild(template.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:

```
<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 previously mentioned *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 as the last `HelloWorld` example wouldn't work like it is currently. 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*.[7]

Despite the discrepancies among browser vendors *HTML Imports* should still be part of the paper and future *web components* as no other native browser technology can bundle up CSS, JS and HTML that efficient. As of today, January 2017, only Googles Chrome and related Opera browser supporting the full *web components* spec and, despite from *HTML Imports*, all other browser vendors most likely will catch up with *Custom Elements* and *Shadow DOM* within this year. In the meantime, the full *web components* stack can be **polyfilled** and used across all browsers.

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 string 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>
```

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

Naming events after the emitting tag makes the API almost self-explanatory. Fortunately the `detail` property can transmit even objects witch allows the developer to craft almost all functionality inside one event. 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 be 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 eventuall 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 embrace 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[15]. 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 to whatever reason which leads to a frozen UI on the users side. A worker can overcome the bottleneck of the single-threaded nature with spawning a new **background thread**. Most of the browsers work can be leveraged to this new thread. Todays web architectures aims to leverage an increasing amout of proccessing to the client to avoid time-consuming roundtrips especially in mobiles networks.[8] While many native APIs like `fetch` work seamlessly in the new thread, a worker has no access to the DOM at all.

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');
```

[8] Latency numbers: https://gist.github.com/jboner/2841832

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

Having no access to the DOM can be seen as hinderance,but i

Macroperspektive / Composition

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

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

containers

Anatomy of an browsernative microservice

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

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

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

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

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

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

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

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

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

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

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