International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 04 Issue: 10 | Oct -2020 ISSN: 2582-3930

Component-based UI Web Development

Omendra Singh Rathore, Ms. Charu Puri

Department of Computer Science
P.G.D.A.V. College
Delhi University

Introduction

Internet is no longer a place surrounded by documents with static con-tent. As a big target platform, it has many applications with massive infrastructures. During the past years, many drafts and approaches for building web applications were introduced. One of the most used ones was using the server-side API (*Application Programming Interface*) glued together with lightweight JavaScript UI (*User Interface*) library called jQuery. It was the right solution, but only for specific applica-tions. As applications were growing and many new ones were being introduced, certain limitations were discovered. Applications were hardly scalable and maintainable, prototyping was not easy neither quick. However, jQuery and its model had a high impact on which direction web front end development moved.

In the year 2013, a JavaScript library for building UIs called React was introduced by Facebook. React's infrastructure, like many other library infrastructures, e.g. Angular's, Vue's, LitElement's, Stencil's, etc., was designed around reusable, self-contained, semantic build-ing blocks. These semantic building blocks, also called components, were based on very similar ideas as Web Components specification introduced by Alex Russell at Fronteers Conference in 2011 with these words:

Quote: Web Components are some new work that we are doing to make the DOM extensible. Russell [1]

By the time of writing this thesis, Web Components specification was not fully standardised into the Web platform yet, and libraries like React took advantage of it, rapidly grew in the popularity and de-facto determined the new way of writing web applications by the method called component-based UI web development. Even though the core concepts of components build with React seem to be similar to the specification ones, they differentiate after an in-depth examination.

Using components to build UIs for the web applications solves many problems that were web developers facing in the past, but it also creates some new ones. One of the main issues of the component-based UI web development, and arguably the most discussed one, is managing the common components state. This act is also referred to as the state management. Because the Web is a very resilient and flexible platform, many state management solutions were already introduced for React as well as for other libraries. On the other hand, due to the ongoing standardisation process, Web Components themselves are not widely used yet, and so just few state management solutions for pure Web Components exist.

The goals of this thesis are:

- Provide a basic overview of component-based UI web develop-ment.
- Emphasise the core principles and differences between the Web Components specification, Web Components libraries and React components.
- Provide an introduction to some of the available state manage-ment solutions for React.
- Discover whether some of the state management solutions for React can be used within Web Components.

This thesis is divided into five chapters. In the first one is a basic overview of component-based UI web development with description of its key principles. Next three of them are connected and are describ-ing the fundamental principles and differences of Web Components, LitElement (a superset of Web Component specification) and React.js library (as one of the popularisers of component-based UI web devel-opment). The fifth chapter is named State management and it focuses on providing a basic overview of state management problems and available solutions.

1 Component model

Quote: Component-based software engineering (CBSE), also called as component-based development (CBD), is a branch of software engineering that emphasizes the sepa-ration of concerns with respect to the wide-ranging func-tionality available throughout a given software system. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software. [2]

Nowadays, web architecture is changing every year. New ideas, approaches, frameworks and libraries are appearing almost every day. In other words, it is a mess, but it does not necessarily mean that it is a bad thing. The web is a very adaptive and resilient platform that enables to choose tons of different approaches and have numerous possibilities.

In the last years, the increasing popularity of component-based UI development changed the way of writing UIs. This model pushed out some more traditional approach as MVC (*Model View Controller*) or MVVM (*Model View ViewModel*). In this chapter components as the basic building blocks of component-based web architecture are described.

1.1 Predecessors

Web Components specification is not the first attempt to popularise the component model in markup languages. In the past, there were few other attempts as i.e. HTCs (*HTML Components*) from 1998 by Microsoft or XBL (*XML Binding Language*) from 2001 by Mozilla.

1. Component model

1.2 Single page application

The component model goes well with Single page application model. Single page application is an application that dynamically reacts to the user input without any need of refreshing the whole page and thus retrieving it each time from the server. This approach allows mainly better UX (*User Experience*) by avoiding the interruption between suc-cessive pages, making the Web application look and feel more like a desktop or a mobile one.

1.3 Separation of concerns

Quote: It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. Dijkstra [3]

One of the most important design principles in component-based architecture is SoC (*Separation of Concerns*). It focuses on the encapsu-lation on the level of components, making components self-contained and loosely coupled with high cohesion. Thus making the software as much adaptable for change, extensible, maintainable and reusable as possible.

1.4 Single responsibility

Another important principle that component-based architecture en-courages is the single responsibility principle. In the context of com-ponents, single responsibility principle means that each component should have only one function and should do it well.

2 Web Components specification

In 2011, Alex Russell, a man behind one popular JavaScript UI library called Dojo, proposed a vision of the future. He asked himself the following questions:

Quote: What if DOM were extensible? - We could make our extensions crawlable? - DOM object didn't "feel" foreign to JS? Data binding and templating were part of the platform? Russell [1]

Based on these questions, he mentioned four different draft speci-fications:

- Custom elements - a way of defining and using custom HTML elements in the DOM (*Document Object Model*).
- Scoped CSS - CSS encapsulation mechanism for a subtree of elements defined in its scope.
- Shadow DOM - a method of combining multiple subtrees into one hierarchy and a way how these trees interact with each other within a document.

- HTML Templates - a method for declaring DOM subtrees and a way of manipulating them to instantiate different document fragments with identical contents.

These specifications together were initial steps of creating new W3C (*World Wide Web Consortium*) meta-specification called *Web Com-ponents* introduced by Alex Rusell by these words:

Quote: Web Components are some new work that we're doing to make the DOM extensible. Russell [1]

The current version (1) of the Web Components specification con-sists of the following four specifications:

- Custom elements

2. Web Components specification

- HTML Template
- Shadow DOM
- ES Modules

2.1 Custom elements

Note: *Living standard, maintained in the HTML Standard and DOM Standard (WHATWG).*

Quote: Custom elements provide a way for developers to build their own fully-featured DOM elements. Although authors could always use non-standard elements in their documents, with application-specific behavior added after the fact by scripting or similar, such elements have histor-ically been non-conforming and not very functional. By defining a custom element, authors can inform the parser how to construct an element properly and how elements of that class should react to changes. [4]

```
▼<td class="Bu bAn">
   <div class="Bt" style="width: 268px;"></div>
  ▼<div class="nH if">
    ▼<div class="nH V8djrc byY">
      ▶<div class="hj">…</div>
      ▼<div class="nH">
        ▶<div class="ha">…</div>
          <div class="dJ"></div>
        </div>
```

Figure 2.1: Screenshot of G-mail application markup

The figure above (fig. 2.1) shows a markup of one of the most used applications in the world - G-mail[1]. On first sight, it is not apparent

1. http://mail.google.com

what the nested divs with generated class names mean. It is hardly understandable and readable for human either for a machine. In this case, it is not a semantic document anymore. Custom elements are solving this issue, allowing authors to write their own semantic HTML tags. In comparison, the example with G-mail, might after rewriting with custom elements look somehow like this (fig. 2.2).

```
<gmail-app>
  <side-bar></side-bar>
  <email-thread id="123">
    <email-content from="sender@gmail.com" to="receiver@gmail.com">
      <!-- content -->
    </email-content>
  </email-thread>
</gmail-app>
```

Figure 2.2: Semantic G-mail markup proposal

One of the main advantages of semantic elements is their reusabil-ity. When building new applications, the whole components written in this way can be easily understood, edited and reused.

2.1.1 Autonomous custom element

An autonomous custom element is an element extending generic HTMLElement interface. For demonstration purposes, a simple autonomous custom element, for switching between languages, is introduced (fig.
2.3). Whenever the parser sees language-icon tag, it creates a new instance of LanguageIcon class, sets language attribute to the internal state of the class and watches for changes to this attribute. From now on, this custom element is functional, and it does only one thing - whenever its attribute changes, it calls an alert function saying which attribute changed displaying its old and new value.

2. Web Components specification

```
// The goal is to  use a custom element in a html file declaratively
// like this:
<language-icon language="en-us"></language-icon>

// JavaScript class extending HTMLElement interface has to be created.
// HTMLElement interface represents any HTML element.
class LanguageIcon extends HTMLElement {

  // getter returning an array of attributes that should be observed
  static get observedAttributes() { return ["language"] }

  // whenever observed attribute changes, this callback is called
  attributeChangedCallback(name, oldValue, newValue) {
    this._language = newValue;
    alert(`Attribute '${name}' changed from '${oldValue}' to '${newValue}'`)
  }
}

// After class is created, it can be used to define a custom element thanks to
// customElements.define() function .
customElements.define("language-icon", Icon);
```

Figure 2.3: Autonomous custom element

There are also two other options on how to create a custom element (fig. 2.4):

1. programmatically using DOM API
2. programmatically using Custom element constructor

2. Web Components specification

```
// using DOM API
const languageIcon = document.createElement('language-icon')
languageIcon.setAttribute('language', 'en-us')
document.body.appendChild(languageIcon)

// using custom element constructor
const languageIcon = new LanguageIcon()
languageIcon.setAttribute('language', 'en-us')
document.body.appendChild(languageIcon)
```

Figure 2.4: Two other methods to create a Custom element

2.1.2 Customized built-in element

Customized built-in elements allow to reuse behaviour of already existing HTML elements and extend them with some custom func-tionality. In contrast with autonomous custom elements, their classes are not extending HTMLElement generic interface but a more concrete interface. For example, to extend the element button, it is needed to extend a component class with HTMLButtonElement interface (fig. 2.5).

2. Web Components specification

```
// HTMLButtonElement is an interface for manipulating <button> elements
class FancyButton extends HTMLButtonElement {
  // after an element is connected to the DOM, this callback is called
  connectedCallback() {
    this.addEventListener('click', () => {
      alert('You clicked me!')
    })
  }

  // after an element is disconnected from the DOM, this callback is called
  disconnectedCallback() {
    this.removeEventListener('click')
  }
}

// After the class is created, it can be used to define a custom element
// thanks to customElements.define() function, but with also specified
// extends option.
customElements.define('fancy-button', FancyButton, { extends: 'button' })
```

Figure 2.5: Customized built-in element

There are three native possibilities to construct Customized built-in element (fig. 2.6):

1. declaratively in the HTML
2. programmatically using DOM API
3. programmatically using constructor

2. Web Components specification

```
// declarative creation in HTML
// is attribute specifies customized element tag name
<button is="fancy-button">Click this fancy button!</button>

// using DOM API
const fancyButton = document.createElement('button', { is: 'fancy-button' })
fancyButton.textContent = "Click this fancy button!"
document.body.appendChild(fancyButton)

// using custom element constructor
const fancyButton = new FancyButton()
fancyButton.textContent = "Click this fancy button!"
document.body.appendChild(fancyButton)
```

Figure 2.6: Three basic methods to create a Customized built-in ele-ment

2.1.3 Lifecycle

Every custom element has its lifecycle methods (fig. 2.7), which are invoked after certain events. The current version 1 of Custom elements specification describes four different lifecycle methods:

1. connectedCallback – invoked after an element is appended into the DOM.
2. disconnectedCallback – invoked after an element is discon-nected from the DOM.
3. adoptedCallback – invoked each time an element is moved to a new document.
4. attributeChangedCallback – invoked each time an attribute of an element is changed. For the performance reasons, the “observed” attributed has to be specified in the static get observedAttributes method.

2. Web Components specification

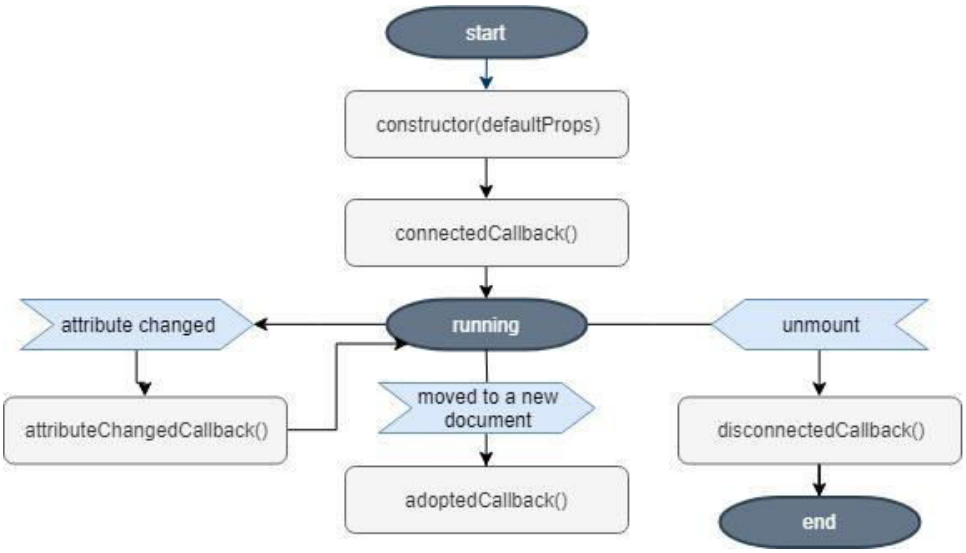

Figure 2.7: Web Component lifecycle

2. Web Components specification

2.2 HTML templates

Note: Living standard, maintained in the HTML Standard and DOM Standard (WHATWG).

As web applications grew in complexity, a requirement to display dynamically retrieved data on the UI became an inseparable part of the Front-End development and web developers were often facing a problem of how to display these kinds of data. In the following chapter, a few of the most popular approaches for displaying dynamically retrieved data are described.

2.2.1 Background example

Note: In the examples, the main focus is given to the tem-plating part of a component. That is why the rendering part of the template, the render function itself, is abstracted from the examples.

To demonstrate the syntactical differences between the selected approaches a simple example implementation of dynamic UI, based on the retrieved data, is demonstrated for each approach. Dynamically retrieved data are represented as the results object that is an array of four objects with the same properties, but with distinct values (fig. 2.8).

```
// parsed data
const results = [
  { content: "1 - Lorem ipsum dolor sit amet …" },
  { content: "2 - Lorem ipsum dolor sit amet …" },
  { content: "3 - Lorem ipsum dolor sit amet …" },
  // …
]
```

2. Web Components specification

To render these items, it is possible to put the results data into an unordered list, where each item in this list represents one element in the results array (fig. 2.9).

```
<!-- <ul> is a placeholder for results -->
<ul>
  <!-- <li> represents each result element -->
  <li>
    <p>1 - Lorem ipsum dolor sit amet ...</p>
  </li>
  <li>
    <p>2 - Lorem ipsum dolor sit amet ...</p>
  </li>
  <li>
    <p>3 - Lorem ipsum dolor sit amet ...</p>
  </li>
  <!-- ... -->
</ul>
```

Figure 2.9: Results as an unordered list

The code (fig. 2.9) contains much repetition. Each of the list item element is the same but has different data (content property). Follow-ing the DRY (*Don't Repeat Yourself*) principle, it is possible to create only one, reusable template and fill it with the data as many time as required, creating as many elements as required.

2.2.2 DOM API

Before the HTML Templates specification was standardised, there were two possible ways to create a template while using only platform tools (fig. 2.10):

1. Using the Element's innerHTML property and string concatena-tion.
2. Using DOM manipulation method.

```
// innerHTML and string concatination method
function InnerHTML(results) {
  let items = ''

  // going through each result
  data.forEach(({ content }) => {
    const li =
      '<li>' +
        '<p>' + content + '</p>' +
      '</li>'

    // string concatination - adding list items together
    items += li
  })

  return '<ul>' + items + '</ul>'
}

// abstract render function
render(InnerHTML(results))

// DOM manipulation method
function DOMManipulation(results) {
  const ul = document.createElement('ul')

 // going through each result
  results.forEach(({ content }) => {
    const li = document.createElement('li')
    const p = document.createElement('p')
    p.textContent = content
    li.appendChild(p)
    // appending list item into the unordered list
    ul.appendChild(li)
  })

  return ul
}

// abstract render function
render(DOMManipulation(results))
```

Figure 2.10: Two methods to create an reusable element

2. Web Components specification

Even though the DOM offered two different approaches to create a template, the community was not satisfied with those solutions. One of the reasons was that this approach was not well readable when coming to more complicated templates. That is why third-party templating libraries gained popularity, and the new HTML template element was introduced.

2.2.3 Mustache.js

Note: Mustache is named "Mustache" because of the use of curly braces which resemble a sideway moustache.

Mustache.js is a web templating library which had a significant impact on the creation of the HTML templates specification. It is available for many programming languages such as Closure, Go, Java, JavaScript, .NET, Python, Ruby. It is described as a "logic-less" template system because it lacks any explicit control flow state statements as conditionals and loops.

```
<!-- The browser ignores this script because the type “text/template” is
not recognised, but inner elements are still accessible -->
<script type="text/template" id="template">
  // unique Mustache double curly braces syntax
  {{#results}}
    <li>
      <p>{{content}}</p>
    </li>
  {{/results}}
</script>
<script>
  const template = document.getElementById('template')
  // abstract render function, needs also the template itself
  render(Mustache(template, results))
</script>
```

Figure 2.11: Mustache.js example

This method is getting the most of the power of HTML - its declar-ativity together with minimal DOM API usage. On the other hand, there is one major downside to this approach: dependency on the external library. In general, dependencies have many disadvantages. They might change or might not be supported in future, they might introduce some security issue, they need to be first downloaded and parsed (which is also not ideal in case of performance and initial load) and, from the developer perspective, the developers need to first learn how to use them (in case of Mustache.js the special curly braces syntax). That is why HTML templates specification was created, offering simi-lar functionality while being not dependant on any external library, just using the internal platform tools.

2.2.4 HTML Templates

> **Quote**: The HTML Content Template (<template>) ele-ment is a mechanism for holding client-side content that is not to be rendered when a page is loaded but may sub-sequently be instantiated during runtime using JavaScript. [5]

The template element does the same thing as script element from the Mustache.js example, and thus the elements inside of it are not being rendered by the browser by default. However, in the previous example, the script element was not rendered because of the unknown text/template type. This method seems to be a little "workaround", because the main purpose of the script tag, is not to hold an HTML template. In the case of the template element, it does not require any workaround to work. Not rendering itself is just its default behaviour known by the browsers. Additionally, the content of the template element is parsed by the browser while loading the page, ensuring content validity.

> **Quote:** The template contents of a template element are not children of the element itself. [6]

Which means, that content of the template element is not consid-ered to be in the document at all. For example, the content cannot be

found by document.querySelector() function, until it is cloned and appended into the document by the script.

> **Quote**: The template element is used to declare fragments of HTML that can be cloned and inserted in the document by script. [6]

```
<template id="template">
  <li>
    <p></p>
  </li>
</template>
<ul id="results"></ul>

<script>
  function htmlTemplate(template, results) {
    // main document fragment, that holds all of the results
    const ul = document.querySelector('#results')

    // going through each result
    results.forEach(({ content }) => {
      // cloning the template content
      const clone = document.importNode
      // setting the content
      clone.querySelector('p').textContent = content
      // appending list item into the unordered list
      ul.appendChild(clone)
    })

    return ul
  }

  const template = document.getElementById('template')
  // abstract render function also needs the template itself
  render(htmlTemplate(template, results))
</script>
```

Figure 2.12: HTML Templates

On the first side, this (fig. 2.12) code looks more complicated, and it is even longer than the Mustache.js example. However, a more ex-tended code is just one of the tradeoffs when the highly abstract library

is replaced with the low-level platform specification. Another tradeoff is that for each result item the cloned node has to be queried to find value placeholders (fig. 2.13).

```
1 clone.querySelector('p').textContent = content
```

Figure 2.13: Necessary querying of cloned template

That might not be an issue in this example, but in the more massive templates, where each one of the thousands cloned nodes is being queried multiple times, this might cause some performance issues and it is something to be cautious about.

2.3 Shadow DOM

Note: Working draft, maintained mostly in DOM Standard (WHATWG).

Quote: Shadow DOM is just normal DOM with two dif-ferences: 1) how it's created/used and 2) how it behaves in relation to the rest of the page. Normally, you create DOM nodes and append them as children of another ele-ment. With shadow DOM, you create a scoped DOM tree that's attached to the element, but separate from its actual children. This scoped subtree is called a shadow tree. The element it's attached to is its shadow host. Anything you add in the shadows becomes local to the hosting element, including <style>. This is how shadow DOM achieves CSS style scoping. [7]

Being able to keep markup, styles and behaviour encapsulated from other code is one of the most important aspects of component-based systems. Shadow DOM specification introduces scoped styles

to the Web platform, enabling encapsulation of styles on the level of components, removing the possibility of multiple clashing styles and keeping the code cleaner. Without tools or naming conventions, it is possible to bundle CSS with markup and encapsulate the styles from the rest of the page.

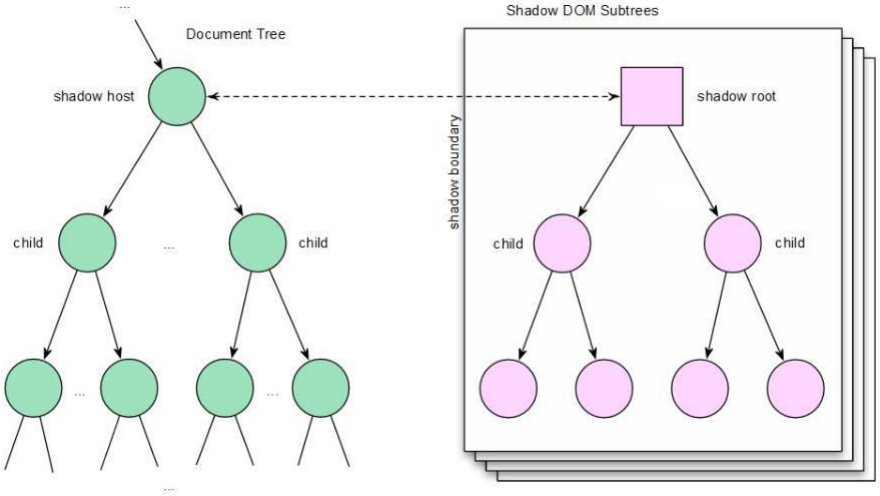

Figure 2.14: The document tree and several shadow DOM subtrees.[2]

2.3.1 Creating shadow DOM

To create a Shadow DOM, a shadow root element has to be created first. The shadow root is an element that gets attached to the host element. Host element is an element "hosting" Shadow DOM tree. To create a shadow root element, attachShadow() function has to be called on the host element as shown below (fig. 2.15).

> **Note:** Not every element is attachable! Elements like <input> and <textarea> already have their own internal shadow DOM. Also it does not make sense to attach shadow to a self enclosing element > that cannot have children e.g.
> <img>,
.

2. https://www.sitepoint.com/the-basics-of-the-shadow-dom/

```
customElements.define('shadow-host', class extends HTMLElement {
  constructor() {
    super()

    const shadowRoot = this.attachShadow({mode: 'open'})
    const insideShadowRoot = document.createElement('h1')
    insideShadowRoot.innerText = 'Hello from Shadow DOM'
    this.shadowRoot.appendChild(insideShadowRoot)
  }
})
```

Figure 2.15: Creation of Shadow DOM

2.3.2 Composition

For the composition of elements can be used a <slot> tag, which acts as a placeholder that can be filled by the authors' defined markup. A basic example of how this works is below (fig. 2.16).

2.3.3 Styles

- CSS selectors from the outer page do not apply inside the com-ponent.
- Styles defined are scoped to the host element.
- The host element can be styled using the :host selector.

2. Web Components specification

```
<!-- Shadow root of shadow-host element -->
<!-- #shadow-root -->
<h1>Title</h1>
<div class="container">
  <slot></slot>
</div>

<!-- Actual usage of of shadow-root-host element with author defined markup -->
<shadow-host>
  <h2>This element is inserted into the slot!</h2>
</shadow-host>

<!-- Final rendered DOM, with replaced <slot> element by user defined markup -->
<shadow-host>
  #shadow-root
  <h1>Title</h1>
  <div class="container">
    <h2>This element is inserted into the slot!</h2>
  </div>
</shadow-host>
```

Figure 2.16: Shadow DOM composition

```
<!--
#shadow-root
The :host allows to select and style the element hosting a shadow tree
-->
<style>
  :host {
    background: red;
  }

  .container {
    width: 500px;
  }

  .container h2 {
    color: blue;
  }
</style>

<h1>Title</h1>
<div class="container">
  <slot></slot>
</div>
```

Figure 2.17: Styling Shadow DOM

2. Web Components specification

2.4 ES modules

ECMAScript modules bring standardised modules system to JavaScript. While Node.js (JavaScript runtime built on Chrome's v8 engine) has been using CommonJS modules (a project to establish conventions on module ecosystem for JavaScript) for years, browsers never had a module system. Before ES modules were standardised, there were many other approaches to substitute the lack of modules functionality.

2.4.1 Multiple HTML script tags

It is possible to put include multiple <script> elements into an HTML (fig. 2.18), but this method has many disadvantages:

- Scripts are synchronous – each script stops further processing while it is executed.
- Multiple HTTP requests – each script creates a new HTTP re-quest, which affects page performance.
- Order of the scripts is crucial – if scripts are dependent on each other and one script fails, it breaks the further JavaScript pro-cessing. Also, each script can override global defined variables or functions, that have been defined by its precedent. It means that without the in-depth knowledge of each script behaviour it was possible to break some functionality while appending the new script.

```
1 <script src="libary-numer-one.js"></script>
2 <script src="libary-numer-two.js"></script>
3 <script src="script-numer-one.js"></script>
4 <script src="script-numer-two.js"></script>
```

Figure 2.18: Multiple HTML script tags

2.4.2 Bundling

To solve some of the mentioned issued of multiple HTML script tags many projects adopted a bundling approach. Webpack (fig 2.19), as one of the most popular module bundlers, is, i.e. an inseparable part of React.js framework. Bundlers introduce a compile step so JavaScript files can be concatenated into one file during the build time.

- Only one script.
- Reduces the number of HTTP requests.
- Allows the use of the alternative syntax as, i.e. not standardised imports, TypeScript or JSX. Before the bundling process, a tran-spile process can be run. In case of not standardised imports, the dependency tree is built to resolve all the possible dependencies and thus removes the order of the scripts problem.

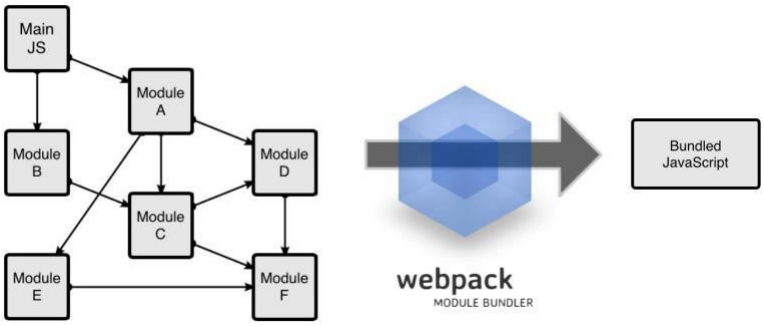

Figure 2.19: Webpack bundling process[3]

2.4.3 ES modules

ECMAScript modules allow importing one file to another, without the need to use any pre-processing. Everything inside of a module is private to the module and only explicitly selected parts are exported outside. ES Modules are bringing the following advantages:

3. http://www.pro-react.com/materials/appendixA/
3. https://developers.google.com/web/fundamentals/primers/modules

2. Web Components specification

Figure 2.20: Scripts execution[4]

- Separation-of-concerns – the module itself lives in its own world and thus can be easily reused, maintained and extended.
- Add possibility to load modules dynamically – only when are they needed.
- Strict mode is enabled by default – enables more advanced static code analysis.
- The defer attribute is used by default – enables parallel HTML and script load (fig 2.20).

The following figure (fig. 2.21) shows the basic ES modules syntax:

2. Web Components specification

```
<script type="module">
    // importing function 'func' from library 'library.js'
    import { func } from './library.js'
    var func2
    // ...
    // exporting function func2 from this module
    export { func2 }
</script>
<!-- fallback.js is executd when type module is not supported -->
<script nomodule src="fallback.js"></script>
```

Figure 2.21: ES Modules syntax

2.5 Support

Web Components are supported in almost every evergreen browser, and their support is getting better almost every day for other browsers (i.e. Microsoft Edge). For older browsers and backwards compatibility, polyfills are available. Even though they support many features out of the box, full support is not possible because some of the specifications are not fully polyfill-able (i.e. Shadow DOM).

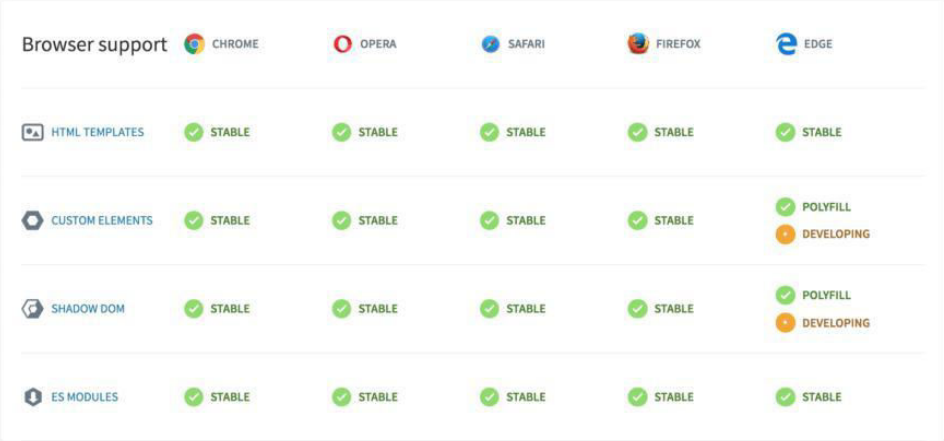

Figure 2.22: Web Components browser support[5]

5. https://dev.to/thepassle/ing--web-components-aef

2.6 Conclusion

Even though Web Components bring many advantages, writing the code like this becomes tedious and hard to maintain over time. For example, if it would be needed to add 15 properties to the component, for each of the property would have to be written getter and setter. More so, if half of the properties would need to be reflected to at-tributes as well, there would need to add a lot of repeating boilerplate code to achieve such functionality. It is possible to add everything in one component, but that would require a lot of error-prone boilerplate code. That is why other libraries are becoming popular, removing unnecessary boilerplate, providing higher abstraction and bringing many other improvements based on the low-level Web Components API with only a tiny addition of JavaScript page load sizes.

3 LitElement

Note: Released in 2017, maintained by Google.

LitElement is a library for writing Web Components on a higher abstract level, providing performance benefits, more declarative way of writing components and overall better developer experience. It follows the Web Components specification and uses only platform standardised syntax, thus makes the components reusable without any need of transpilation. Because LitElement is a superset of Web components specification, only the key differences between LitEle-ment library and Web Components specification are described in this chapter.

3.1 Components

A class extending LitElement base class represents a LitElement com-ponent. In addition to Web Components specification, LitElement supports type-safe observable properties that cause an element to up-date with many useful options. Observable properties can be written in either vanilla JavaScript or TypeScript style (in this case, components needs to be first transpiled to work correctly). Other advanced features as batching changed properties and attributes and asynchronous ren-dering are also supported, reducing the overhead and keeping the state consistent.

3.1.1 Lifecycle

LitElement implements basic Web Component lifecycle methods. In addition, to provide a higher abstraction and more functionality over the Web Components specification, it extends component update life-cycle by many by method in following call order:

1. someProperty.hasChanged
2. requestUpdate
3. performUpdate
4. shouldUpdate
5. update

3. LitElement

6. render
7. firstUpdated
8. updated
9. updateComplete

```
import { LitElement, html } from 'lit-element'

class LanguageIcon extends LitElement {
  // automatically observed properties
  static get properties() {
    return  {
      // type checking, reflecting to properties and many other
      // features supported out of the box
      language: { type: String, reflect: true }
    }
  }

  // render returns an html element defined by lit-html syntax
  render() {
    return html`<span>Current language: ${this.language}</span>`
  }
}

customElements.define('language-icon', Icon)
```

Figure 3.1: LitElement component

3.2 Templates

LitElement templates are based on Lit-html library. Lit-html is a lightweight templating library that is adding a layer of abstraction on the top of the already existing <template> tag. Lit-html solution is focusing on

the declarative usage of HTML templates in JavaScript, efficient render and update of the DOM and better overall development experience.

```
// importing html and render function from lit-html
import { html, render } from 'lit-html'

// template function using html tagged template literals
const template = (results) => html`
  <ul id="results">
    ${results.map(result => html`
      <li>
        <p>${result}</p>
      </li>
      `)}
  </ul>
`

// calling render function to render the element inside of the document's body
render(template(results), document.body);
```

Figure 3.2: lit-html template

The main idea behind this library is to use the latest ES6 feature, called *tagged template literals*. It reduces the need to query the cloned document to find the placeholders for values, while still supporting HTML syntax checking and using only platform features (unlike Mus-tache.js curly braces syntax). Lit-html also remembers the placeholders placements and only updates the parts of the template that changes. This high-level approach uses the newest platform features and does not require any other dependency than relatively small 3.5kB (com-pared to the other approaches) library.

3.3 Styles

Based on the Shadow DOM API, LitElement supports component en-capsulated styles. The recommended way of defining the styles is using the static style property. Static style property supports Constructable

3. LitElement

Stylesheets, which is a specification that is allowing browsers to parse styles exactly ones and reuse the stylesheet as many times as required for maximum efficiency.

To define a static styles property (fig. 3.3) following steps are needed:

- Import of the css tagged literal function from LitElement mod-ule.
- Creation of static style property returning a tagged template literal or an array of tagged literals with style defined in plain CSS

```
1 import { LitElement, css, unsafeCSS } from 'lit-element'
2
3 class MyElement extends LitElement {
4   static get styles() {
5     const mainColor = 'red'
6
7     // variables has to be wrapped in unsafeCSS function
8     return css`
9       :host {
10        color: ${unsafeCSS(mainColor)}
11      }
12    `
13  }
14 }
```

Figure 3.3: Styling LitElement component

3.4 Imports

LitElement uses the standard ES modules described in the previous chapter.

3.5 Support

LitElement claims to support all major browsers including Chrome, Firefox, IE, Edge, Safari and Opera.

3.6 Conclusion

Even though LitElement is still a relatively new library, it already brings many advantages over pure Web Components. It makes a tiny abstraction over Web Components specification, enabling writing Web Components with less overhead and just a tiny addition of JavaScript page load size (3,5 kB). LitElement is still strictly dependent on the Web platform, which enables usage of already built-in developer tools. On the other hand, because of this dependency on the platform, the functionality and the level of abstraction is still limited by the platform.

4 React.js

Note: Released in 2013, maintained by Facebook.

React.js library is a JavaScript library that allows the creation of reusable components. It is one of the biggest popularisers of component-based UI development, introducing many patterns and best practices possibly usable by other component-based approaches.

```
1 import React from 'react'
2 import ReactDOM from 'react-dom'
3 import PropTypes from 'prop-types'
4
5 class LanguageIcon extends React.Component {
6   // one of the ways for type checking
7   static propTypes = {
8     language: PropTypes.string
9   }
10
11  render() {
12    // this is not html but but jsx
13    return (
14      <span>Current language: {this.props.language}</span>
15    )
16  }
17 }
18
19 // rendering component
20 // name "attribute" is called name "props" in react
21 ReactDOM.render(<LanguageIcon name="en-us" />, document.body)
```

Figure 4.1: React component

At first sight, the React component (fig. 4.1) looks syntactically similar to the LitElement component (fig. 3.1). However, on the inside,

both of them work quite differently. In this chapter, React approaches for solving common component-based problems are described as a contrast to Web Components specification and LitElement library.

4.1 Main concepts

4.1.1 Compiled code

React components are not natively supported by browsers and have to be transpiled to JavaScript by tools called transpilers. Even though that transpiled React components have much common with Web Com-ponents, they are not based on Web Components specification.

> **Note**: Transpilation is a type of S2S (source-to-source com-pilation), which is a type of compilation that produces the equivalent source code in the same (i.e. older version of the language) or a different language.

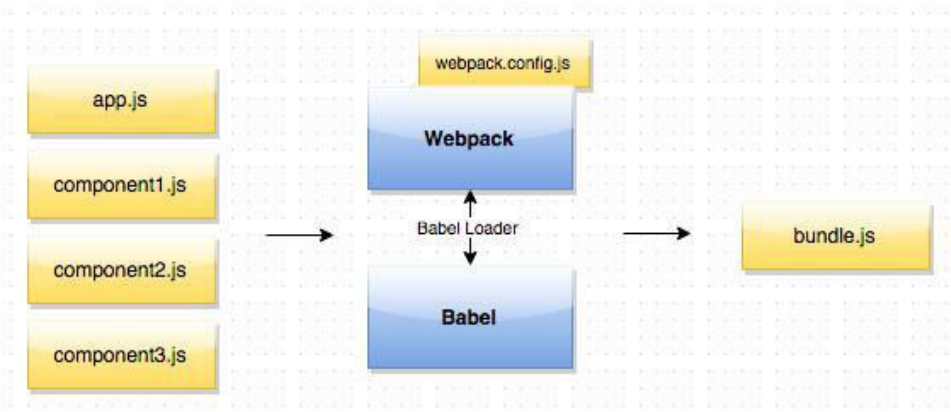

Figure 4.2: React transpilation process[1]

1. https://www.codeproject.com/Articles/1028182/ Getting-Started-with-React-ES?msg=5125189

4.1.2 Virtual DOM

VDOM (*Virtual DOM*) is a concept of storing a lightweight virtual DOM in memory used by React. It is enabling performance benefits, more declarative API and an overall higher abstraction over the na-tive DOM. VDOM is a concept to keep the UI in memory and sync it together with the real DOM only when necessary. It also only ren-ders the part of the DOM that is necessary to be re-rendered, not the whole DOM tree. This process of syncing the VDOM with the real DOM is called reconciliation and allows such things as the in-cremental rendering and thus splitting the whole rendering process into multiple chunks and spreading it over multiple frames, while increasing the rendering performance (React Fiber[2]). In contrast with the native DOM, virtual DOM abstracts out attribute manipulation, manual DOM updates and event handling.

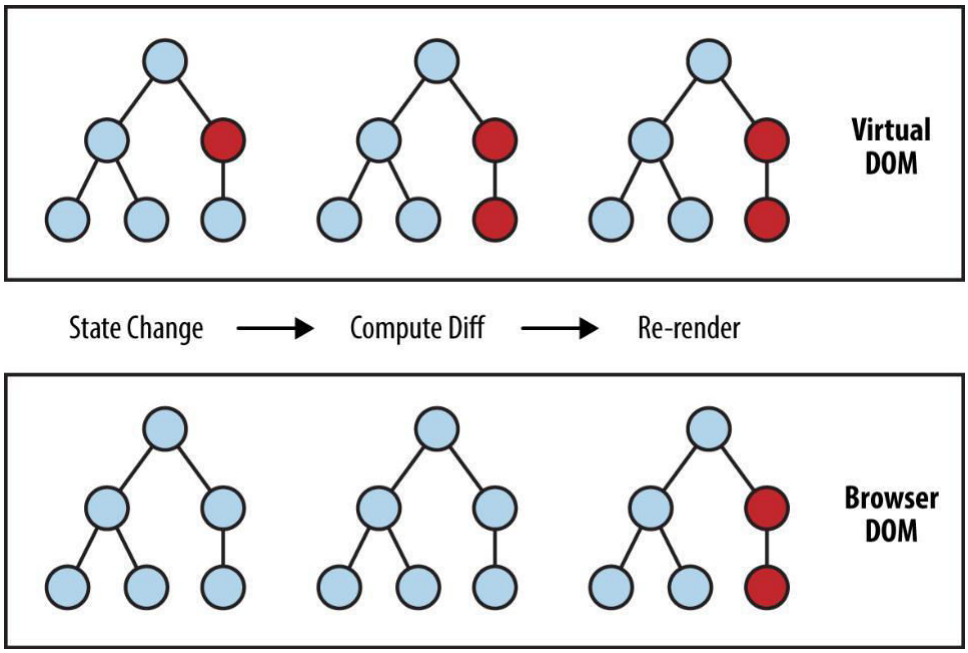

Figure 4.3: VDOM reconciliation process[3]

2. https://medium.com/react-in-depth/inside-fiber-in-depth-overview-of-the-new-reconciliation-algorithm-in-react-e1c04700ef6e

4.1.3 One-way data flow

One-way data flow (fig. 4.4) is a pattern recommended to use by React, allowing only one-way data transfer. Which means that data from a parent component can be passed to the child component only via props (React component attributes), and only in the direction parent-children. The key advantages of this pattern are:

- Less error prone – more control over data than other approaches as two-way data binding
- Easier to debug – precisely knowing which data comes from which direction
- More efficient – this approach tends to be more efficient than, i.e. two-way data binding

The disadvantage of this pattern is described as a prop-drilling problem in the State management section.

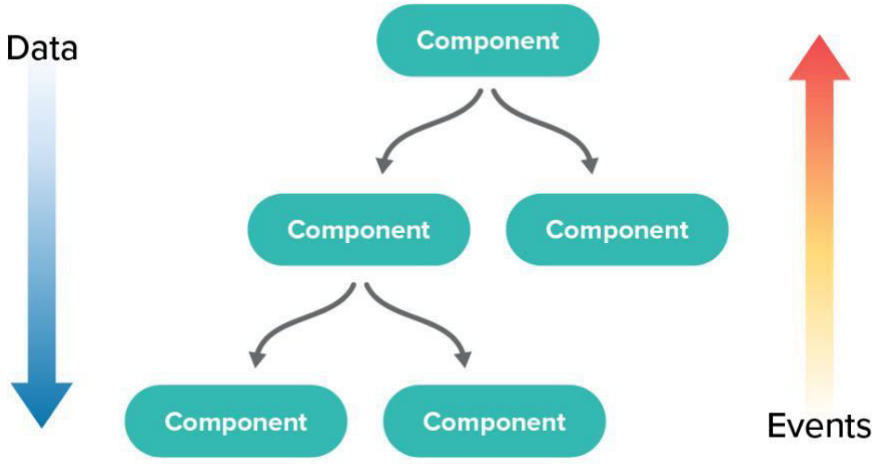

Figure 4.4: One-way data flow[4]

3. https://programmingwithmosh.com/react/react-virtual-dom-explained/

4.2 Components

React component is a component that accepts data (called props) and returns a React element describing the UI. There are two types of com-ponents: Stateless and Statefull. Stateless component is just a function accepting props and returning React element. Statefull component is either a class extending React. Component interface (or function while following new React Hooks API). In the contrast with stateless components, statefull components can manage their local state via component lifecycle methods (fig. 4.5).

4.2.1 Lifecycle

Because of VDOM and overall higher abstraction, React lifecycle (fig. 4.5) differs from Web Component lifecycle. On the other, the level of abstraction is and provided lifecycle callbacks are similar to the LitElement one.

4. https://almerosteyn.com/2017/11/id24-accessible-react-tips-tools-tricks#
/

4. React.js

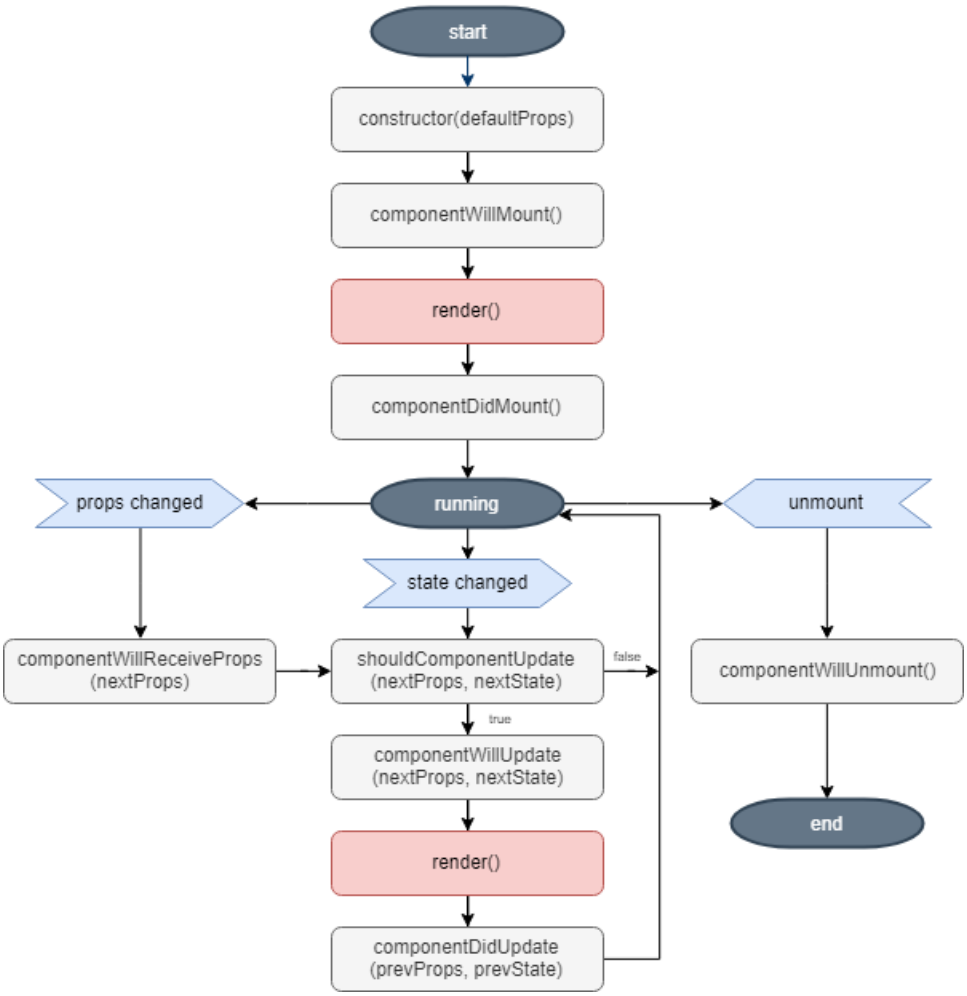

Figure 4.5: React component lifecycle

4.3 Templates

JSX is popular templating alternative introduced to React. It is an XML-like syntax extension to JavaScript. It does not necessarily need to be used with React, but it is recommended for better developer experience.

This approach is trying to move out of the platform as far as possi-ble. Introducing its simplified syntax. There is no need to use double

curly braces, neither template literals. On the other hand, this syntax all has its pitfalls. Even though it looks on the first site familiar to the JavaScript developer, it is not standardised into ECMAScript code and thus has several dissimilarities. It is not supported by browser out of the box, and a transpiler has to be used to transform it into the standard ECMAScript tokens.

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
)

const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
)
```

Figure 4.6: Two identical React elements - JSX vs React.createElement()

4.4 Styles

React does not use Shadow DOM for encapsulating styles, but because it uses Virtual DOM, styles can be defined as a JavaScript object and passed to the predefined style prop, which at the end creates the encap-sulated styles for a component and thus provide similar functionality as Shadow DOM.

```
1 import React from 'react'
2
3 class MyElement extends React.Component {
4   render() {
5     const mainColor = 'red'
6
7     const styles = {
8         color: mainColor
9     }
10
11    return <span css={styles}>red text</span>
12  }
13 }
```

Figure 4.7: Styling React component

4.5 Imports

React, uses CommonJS instead of ES Modules, which is a module specification used by Node.js for working with modules. That is why React has to be compiled before it can be shipped into the production by module bundler as i.e. widely used Webpack module bundler.

4.6 Other features

Because of the abstraction out of the platform, React supports many other features almost out of the backs, that would be hardly achievable just with pure imperative Web Components.

- Server-side rendering – rendering React component on the server and outputting pure HTML content
- Non-blocking rendering – concurrent mode introduced to Re-act watches if there is some action with higher priority than rendering itself. It there is such an action (i.e. user input), Re-act automatically pauses rendering and let other things finish before.
- Mobile platform support (React Native) – thanks to the Reacts' level of abstraction, it is possible to build mobile applications using React style JavaScript

4.7 Support

React supports all popular browsers back to Internet Explorer 9. More so, React supports rendering on the server (Node.js) and mobile ap-plication development (React Native).

4.8 Conclusion

React focuses on different things than Web Components. Even though it offers similar functionalities as Web Components libraries (i.e. LitEle-ment), it focuses on even greater abstraction out of the platform. Its

4. React.js

goal is to write components completely in a declarative way, without any need for DOM manipulation. It introduces many abstractions as Virtual DOM or JSX. Because of these abstractions, React is capa-ble of such things as non-blocking rendering, dynamic UI loading, server-side rendering or powering mobile apps almost out of the box. However, Even though Web Components and React components differ, they differ mostly on the level of abstraction of component model im-plementation. Furthermore, because the Web Components are browser standards, it is straightforward to use a Web Component inside of the React component, and also React component in a Web Compo-nent (currently with some limitations[5]), making the two of them not necessarily mutually exclusive. That is why many approaches, best practices and libraries introduced to React are possibly usable by Web Components themselves, making Web Components or rather Web Component libraries a powerful competitor or a powerful ally for React itself.

5. https://custom-elements-everywhere.com/libraries/react/results/results.html

5 State management

This chapter introduces state management problem as one of the hard-est problems regarding web component-based development. Because Web Components are relatively new technology, not many state man-agement solutions for them actually exist, and if they do exist, they are simple ports of state management solutions introduced to React. That is why are in this chapter few of the most popular state management solutions for React analysed and based on this analysis, a new state management library for Web Components is proposed.

> **Quote:** One of the hardest parts of software development is managing state. Life could be so simple if the user couldn't interact with the application at all, but that sounds like a 90s website and most of us are building interactive appli-cations, so we've got to put state somewhere. [8]

5.1 Component state

Generally speaking, a component state is any data living inside a component. In the case when a class represents a component, the state could be a set of values of properties of the class.

> **Note:** The state depends on defined constraints, level of abstraction and point of view. For example, React defined state as a state property with setState helper function, to render the element properly (because of the VDOM).

For example, this is how a counter component with a state can look like (fig. 5.1).

Many things can be achieved just with the component state. Where things start to break down, is when multiple components want to share the same state or the state has to passed down many components deep into the component tree.

1. https://kentcdodds.com/blog/application-state-management-with-react

5. State management

```
class Counter extends React.Component {
  // local component state
  state = { count: 0 }

  // increment function that calls setState function which causes React
  // component to update the state and efficiently re-render
  increment = () => this.setState(({ count }) => ({ count: count + 1
}))
  render() {
    return <button onClick={this.increment}>{this.state.count}</button>
  }
}
```

Figure 5.1: React statefull component[1]

5.2 Props drilling

> **Note:** This problem is mostly connected to solutions using VDOM or a higher abstraction over the DOM as Lit-html, because only them allow to pass pure unstringified objects through the component tree.

Props drilling (fig. 5.2, fig. 5.3) often occurs, when multiple com-ponents need to share a common state or the props need to be passed down many levels deep into the component tree.

2. https://javascriptplayground.com/context-in-reactjs-applications/
3. https://kentcdodds.com/blog/application-state-management-with-react

5. State management

Figure 5.2: Props drilling problem - graph[2]

```
class App extends React.Component {
  // inner state of the component, holding the counter value
  state = { count: 0 }

  // increment method, calling setState function, which updates the current state
  increment = () => this.setState(({ count }) => ({ count: count + 1 }))

  // render method rendering the current element with current state
  render() {
    return <InnerComponent count={this.state.count} onClick={this.increment}>
</InnerComponent>
}

class InnerComponent extends React.Component {
  render() {
    // this middle component does not need to know props count and increment,
    // but it needs to send them to the counter component
    <Counter count={this.props.count} onClick={this.props.increment}></Counter>
  }
}

class Counter extends React.Component {
  render() {
    <button onClick={this.props.increment}>{this.props.count}</button>
  }
}
```

Figure 5.3: Props drilling problem - example[3]

5. State management

The approach of passing down the props through the components that do not necessarily need them is cumbersome and introduces many possible bugs in the future while deteriorating maintainability. That is why many other patterns and approaches to solve this issue were proposed.

5.3 Singleton pattern

To create a Singleton in JavaScript it is possible to use ES modules as a state holder and export getters and setter like on the following figure (fig. 5.4).

```
1 const state = {}
2 const getState = () => state
3 const setState = newState => Object.assign(state, newState)
4 export { getState, setState }
```

Figure 5.4: Singleton pattern for managing state[4]

Rather than drilling props in the component tree, this solution enables components to import the state module and update or get the data that they need. To make this pattern work, the components need to listen for changes to appropriately re-render themselves. This option is suitable for solving props drilling problem in Web Components and React components. When coming to advanced features like server-side rendering, singleton pattern cannot be used any longer.

4. https://kentcdodds.com/blog/application-state-management-with-react

5.4 Redux

Redux is a Flux library utilising uni-directional data flow and thus solving the props drilling problem. It focuses on clean separation-of-concerns, single source of truth, immutability and functional pro-gramming concept while providing predictable state flow and state debuggability. Redux is mainly useful in large applications, where multiple people work on the same project and where testability and debugging are one of the most important criteria.

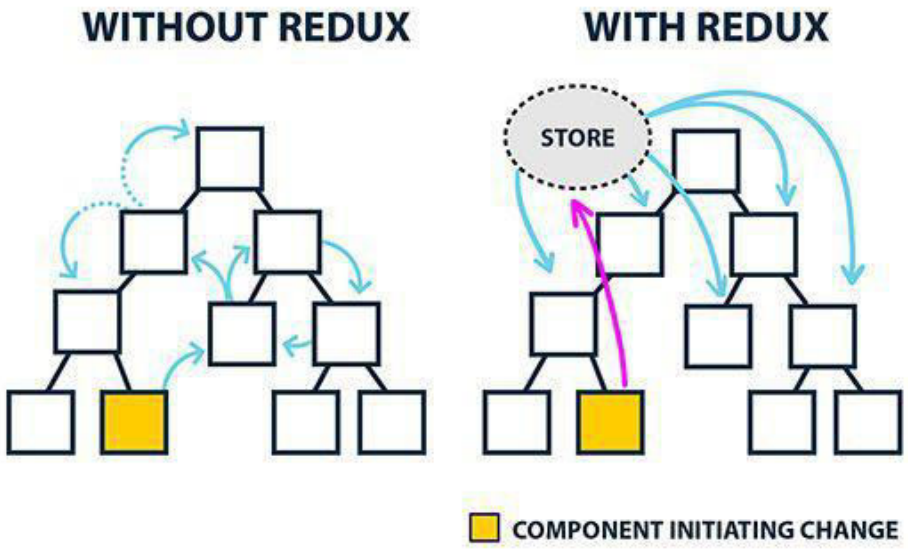

Figure 5.5: Communcation between component without and with Redux

5. https://codingthesmartway.com/learn-redux-introduction-to-state-management-with-react/

5. State management

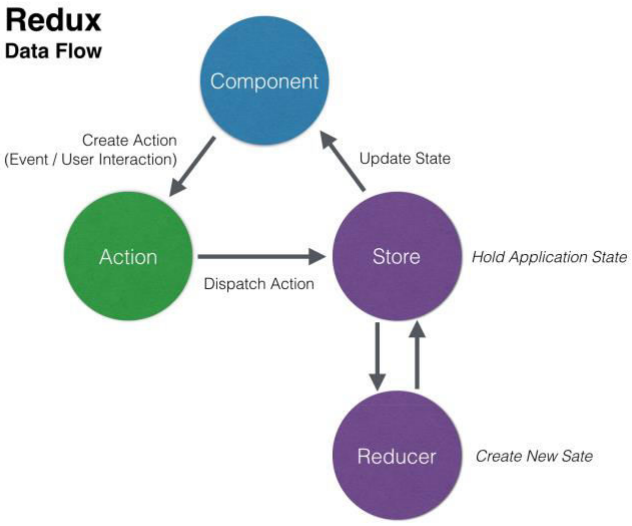

Figure 5.6: Redux data flow[5]

5.5 Context API

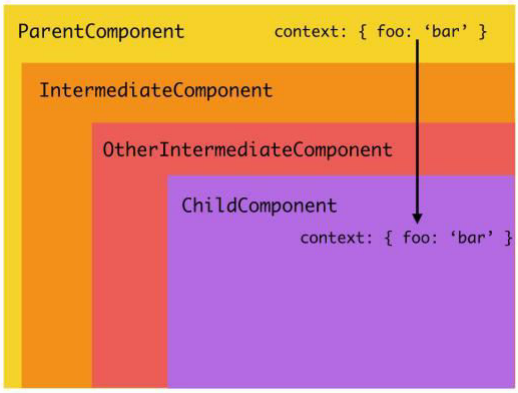

Figure 5.7: React Context API - graph[6]

6. https://javascriptplayground.com/context-in-reactjs-applications/

5. State management

Context API is an API introduced to React. It also focuses on solving the prop-drilling problem, without any need to implement emitters as in Singleton pattern way and its all declarative. However, Context API is mainly useful while passing the same props to many different components deep into the component tree structure. Context API consists of two components: Provider and Consumer. An example follows (fig. 5.8).

```
// creatin counter context with default value: 0
const CounterContext = React.createContext(0)

class Counter extends React.Component {
  render() {
    <CounterContext.Provider value={0}>
      <CounterContext.Consumer
        render={value => <h1>{value}</h1>}>
      </CounterContext.Consumer>
      <div>
        {/* Deep tree of elements, with Consumer at the end */}
        <CounterContext.Consumer
          render={value =>
            <span>Second consumer deeply nested in the tree</span>
            <h2>{value}</h2>
          }>
        </CounterContext.Consumer>
      </div>
    </CounterContext.Provider>
  }
}
```

Figure 5.8: React Context API - example

5.6 Context API for Web Components

Context API is a breakthrough in state management for React, be-cause it solves the only problem with uni-directional data flow (props-drilling) and solves it well. However, nothing like pure Context API

5. State management

was proposed for Web Components specification yet. That is why a solution usable with Web Components was designed in the scope of this thesis. It implements the same API as the React Context API ver-sion, however, it is usable with pure, standardised Web components. The solution was inspired by haunted[7] library, which implements also Context API, but only in the Hook API[8] version.

```
customElements.define('counter-provider', counterContext.Provider)
customElements.define('counter-consumer', counterContext.Consumer)

const counterContext = createContext(0)

const Counter = () => html`
  <counter-provider .value=${0}>
    <counter-consumer .render=${value => html`
      <h1>${value}</h1>
    `}>
    </counter-consumer>
    <div>
      <counter-consumer .render=${value => html`
          <span>Second consumer deeply nested in the tree</span>
          <h2>${value}</h2>
        `}>
      </counter-consumer>
    </div>
  </counter-provider>
`
```

Figure 5.9: Context API for Web Components

7. https://github.com/matthewp/haunted
8. https://reactjs.org/docs/hooks-reference.html

Conclusion

Web Components specification brings many improvements to the Web platform itself, that were either not possible to do before or were only solvable by some other, often more complicated approaches. On the other hand, Web Components are only low-level specifications and us-age of their API becomes cumbersome over time. That is why libraries like LitElement are being introduced, trying to make abstraction over the specifications with minimal increase of JavaScript page load size.

On the other side, libraries like React.js are offering similar func-tionalities as Web Components libraries (i.e. LitElement), focusing on even greater abstraction out of the platform. React's goal is to write components completely in a declarative way, without any need for DOM manipulation. It introduces many abstractions as Virtual DOM or JSX. Because of these abstractions, React is capable of such things as non-blocking rendering, dynamic UI loading, server-side rendering or powering mobile apps almost out of the box.

Furthermore, because the Web Components are browser standards, it is straightforward to use a Web Component inside of a React com-ponent and vice versa (currently with some limitations[9]), making the two of them not necessarily mutually exclusive.

One of the hardest problems to solve in component-based applica-tions is how to manage its state. React promotes one-way data flow as a more convenient way of passing data in the application. Even though this approach brings many advantages, it also has disadvan-tages. While the application grows, the component tree becomes too big and passing the data one-way hardly maintainable. That is why community and React itself answered with many possible solutions solving this problem.

Web Components and React implement a similar component model. Because of that, many of the solutions designed primarily for React can be reused with Web Components. To prove this statement, one of the newest React approach for solving state management problem called React Context API was implemented for Web Components as part of this thesis.

---

9. https://custom-elements-everywhere.com/libraries/react/results/results.html

5. State management

There are many possible ways to extend this thesis. One of them might be to include a deep comparison of rendering performance of Web Components compared to other solutions. Another possible extension might be to add a more in-depth analysis of possible state management solutions that were not mentioned in this thesis.

An appendix

A.1 Context API for Web Components

A proof of concept of Context API for Web Components, inspired by haunted library, can be found under web-component-context-api directory. This directory includes one JavaScript file exporting one function called createContext. A simple example of the usage can be found in the example folder.

Bibliography

1. RUSSELL, Alex. *Web Components and Model Driven Views* [on-line]. Fronteers, 2011 [visited on 2019-13-05]. Available from: https://vimeo.com/33430613.
2. *Component-based software engineering* [online]. Wikipedia, 2019 [visited on 2019-13-05]. Available from: https://en.wikipedia.org/wiki/ Component-based_software_engineering.
3. DIJKSTRA, Edsger W. *Selected Writings on Computing: A Personal Perspective*. Heidelberg: Springer-Verlag Berlin, 1982. ISBN 0–387–90652–5.
4. *Custom elements* [online]. WHATWG, 2005–2019 [visited on 2019-15-04]. Available from: https : / / html . spec . whatwg . org / multipage/custom-elements.html#custom-elements-intro.
5. *<template>: The Content Template element* [online]. Mozilla documenta-tion network, 2005–2019 [visited on 2019-15-04]. Available from: https : / / developer . mozilla . org / en - US / docs / Web / HTML / Element/template.
6. *HTML standard - The template element* [online]. WHATWG, 2005– 2019 [visited on 2019-15-04]. Available from: https : / / html . spec . whatwg . org / multipage / scripting . html # the - template - element.
7. *Shadow DOM v1: Self-Contained Web Components* [online]. Google De-velopers, 2019 [visited on 2019-13-05]. Available from: https:// developers . google . com / web / fundamentals / web - components / shadowdom#intro.
8. DODDS, Kent C. Application state managemnt [online]. 2019 [visited on 2019-15-04]. Available from: https://kentcdodds.com/blog/ application-state-management.
9. RIGBY, Michael. *Component-Based Software Engineering: Software Archi-tecture*. CreateSpace Independent Publishing Platform, 2016. ISBN 978-1541035614.
10. STRIMPEL, Jarrod Overson Jason. *Developing Web Components: UI from jQuery to Polymer*. O'Reilly Media, 2015. ISBN 978-1491949023.

BIBLIOGRAPHY

11. PATEL, Sandeep Kumar. *Learning Web Component Development*. Packt Publishing, 2015. ISBN 9781784393649.
12. SOUSA ANTONIO, Cassio de. *978-1484212615*. Apress, 2015. ISBN 978-1484212615.
13. SCHILP, Pascal. ING Web Components [online]. 2019 [visited on 2019-15-04]. Available from: https://medium.com/ing-blog/ing-%EF%B8%8F-web-components-f52aacc71d7a.
14. HALPERN, Ben. *Why the React community is missing the point about Web Components* [online]. Dev.to, 2018 [visited on 2019-15-04]. Available from: https : / / dev . to / ben / why - the - react - community - is - missing-the-point-about-web-components-1ic3.
15. ABRAMOV, Dan. *You Might Not Need Redux* [online]. Medium.com, 2016 [visited on 2019-15-04]. Available from: https://medium.com/ @dan_abramov/you-might-not-need-redux-be46360cf367.
16. DODSON, Rob. *Custom Elements Everywhere* [online]. 2019 [visited on 2019-15-04]. Available from: https : / / custom - elements - everywhere.com/.
17. DIMANDT, Dmitrii. *The broken promise of Web Components* [online]. 2017 [visited on 2019-15-04]. Available from: https://dmitriid. com/blog/2017/03/the-broken-promise-of-web-components/.
18. *@polymer/lit-element - A simple base class for creating fast, lightweight web components* [online]. The Polymer Project, 2019 [visited on 2019-15-04]. Available from: https://www.webcomponents.org/ element/@polymer/lit-element.
19. *lit-html - An efficient, expressive, extensible HTML templating library for JavaScript* [online]. The Polymer Project, 2019 [visited on 2019-15-04]. Available from: https://lit-html.polymer-project.org/.
20. *LitElement - A simple base class for creating fast, lightweight web compo-nents* [online]. The Polymer Project, 2019 [visited on 2019-15-04]. Available from: https://lit-element.polymer-project.org/.
21. BUCKLER, Craig. *Understanding ES6 Modules* [online]. SitePoint, 2018 [visited on 2019-15-04]. Available from: https://www.sitepoint. com/understanding-es6-modules/.

BIBLIOGRAPHY

22. BYNENS, Addy Osmani Mathias. *Using JavaScript modules on the web* [online]. Google Developer [visited on 2019-15-04]. Available from: https://developers.google.com/web/fundamentals/primers/ modules.
23. IHRIG, Colin. *The Basics of the Shadow DOM* [online]. SitePoint, 2012 [visited on 2019-15-04]. Available from: https://www.sitepoint. com/the-basics-of-the-shadow-dom/.
24. *ALL STANDARDS AND DRAFTS* [online]. W3C, 2019 [visited on 2019-15-04]. Available from: https://www.w3.org/TR/.
25. *React - A JavaScript library for building user interfaces* [online]. Facebook, 2019 [visited on 2019-15-04]. Available from: https : / / reactjs . org/.
26. VRUAB VAUGHN, Andrew Ckarj abd. *Concurrent Rendering in React* [online]. React Conf, 2018 [visited on 2019-13-05]. Available from: https://www.youtube.com/watch?v=ByBPyMBTzM0.
27. ABRAMOV, Dan. *Beyond React 16* [online]. JSConf, 2018 [visited on 2019-13-05]. Available from: https://www.youtube.com/watch?v= nLF0n9SACd4.
28. ABRAMOV, Dan. *Live React: Hot Reloading with Time Travel* [online]. React Europe, 2015 [visited on 2019-13-05]. Available from: https: //www.youtube.com/watch?v=xsSnOQynTHs.
29. RUSSELL, Alex. *Web Components: Just in the Nick of Time* [online]. Poly-mer Summit, 2017 [visited on 2019-13-05]. Available from: https: //www.youtube.com/watch?v=y-8Lmg5Gobw.
30. AZIMINIA, Ana Cidre Sherry. *Web Component Architecture and Pat-terns* [online]. We are developers, 2018 [visited on 2019-13-05]. Avail-able from: https://www.youtube.com/watch?v=hdSz1EKjK10.
31. SPLITT, Martin. *Building web applications with Web Components* [online]. Devoxx, 2017 [visited on 2019-13-05]. Available from: https://www. youtube.com/watch?v=0FstJG9t5v0.