Bachelor Thesis
Vue vs Vugu

Ferdinand Wegerif And Martin Overelv
Universitetet i Stavanger

Spring, 2021

Contents

Abstract

In this thesis, we compare the usability and performance of the front-end frameworks, Vue and Vugu. This is done, to see how experimental WebAssembly technology compares to a well-established front-end technology. As Vugu is experimental technology, this implies that the framework has not obtained a lot of attention, thus this thesis' attention is valuable to its potential to affect front-end development. To compare these frameworks, various small web-applications were built in both frameworks, with focus on different performance and usability aspects as comparison metrics. The result from these comparisons concludes our results. This thesis has value, as comparisons between old and new technology is a way to determine their viability in the future of web development.

Acknowledgements

For our guidance throughout this thesis, we would like to give our sincere thanks to our guidance councilor, Dr. Leander Nikolaus Jehl. For the opportunity to write this thesis, we would like to extend our gratitude to the institution, Universitet i Stavanger. It is through this institution and through our guidance councillor's' help that we were able to produce the research and work needed to write this thesis. Without the available resources retrieved from these parties, our thesis would not be possible.

Chapter 1

Introduction

In 2015 [1], WebAssembly, a compiled low-level programming language focused on near-native performance [2] was announced. Since then, there have been several frameworks and libraries that intend to take advantage of the language's performance [3]. This thesis attempts to explore how Vugu, an experimental WebAssembly framework, compares to Vue, a well-established JavaScript framework. Comparing Vue and Vugu has value as the future of programming is continuously adapting to our growing technological needs; comparisons between old and new technology is a way to determine their viability in the future of web development.

"Browsers were originally a document viewer for HTML [...]. Today your browser, both the one on your phone and on your desktop, is not just an HTML document viewer. It is a full run-time for web applications. [...]"
Brad Peabody, creator of Vugu, GopherCon 2020, 12.11.2020 [4]

Is modern day JavaScript suitable for these web applications or are there better alternatives? The purpose of this thesis is to ascertain the advantages and disadvantages Brad Peabody's work, Vugu, has over conventional user interface frameworks, such as Angular, React and Vue. This is meant to give insight into the potentials of his work and shed light onto the current state of his competition. In this thesis we will take a closer look at the libraries Vugu and Vue, by comparing their performance and usability in different aspects.

1.1 Comparison Metrics

The purpose of front-end frameworks is to create highly interactive user interfaces while not reducing the developers' efficiency, as a result our comparison metrics are focused on performance and usability. Comparing the performance and usability aspects of the frameworks is done by decomposing these topics to help us pinpoint and evaluate differences in these aspects more precisely.

Performance attributes we look at in this thesis are, file sizes; investigating file sizes produced by the frameworks and consider if their download speed may affect developers or end-users substantially. In addition, we compare advantages or disadvantages that occur with the respective frameworks coding language. The environments used by the frameworks differ, we attempt to find out how this affects performance by comparing different computational tasks.

Usability is an abstract term. We decompose usability into two subtopics, the developer usability, and the end-user experience. With end-user experience we ask the question; does the framework provide the end-user a satisfactory experience when using the web application? With developer usability we explore how easily the developer can create functionalities with differing specifications and needs. Some aspects we look at in this thesis include: what resources the developers have available, how the frameworks allow for reusing code, the functionalities the frameworks support and how well the development environments assist developers.

1.2 Background Vugu using WebAssembly with Go

"If we talk about modern web development [...] most of [the development] and specifically user interfaces [...], those are written in JavaScript. However, it is not necessarily because JavaScript is the best solution, it is the solution that has been working. So, I would say that JavaScript is chosen by necessity and not necessarily by choice. Now WebAssembly changes the whole game [...]"
Brad Peabody, creator of Vugu, GopherCon 2020, 12.11.2020 [2]

WebAssembly addresses the problem of safe, fast, portable low-level code on the Web. [63] The portable compiled code is then converted into machine code on the target device [15]. This compact binary form enables the instructions to be transmitted on the web, meaning well-established low-level languages can be used for front-end development. It is up to debate whether WebAssembly will be competing with JavaScript over market dominance in front-end development [16]. For now, WebAssembly requires JavaScript to interact with any Web-API, like the DOM (Document Object Model), with future plans to allow it to interact directly [17].

The Vugu project is described in the following manner in its documentation: "Vugu is [a] Go library which makes it easy to write web user interfaces in Go." [13]. Even though Go is compiled to WebAssembly, this does not necessarily mean that Go is fully supported by WebAssembly, as Mozilla states that supporting garbage collected languages is still a "high-level goal" in the future [20].

1.3 Background Vue.js

The definition of the adjective; *progressive*: "happening or developing steadily"
- OxfordDictionaries.com 20.01.2021 [21].

This is the word/adjective that Vue.js uses to describe their JavaScript framework. Vue.js is an open-source front-end JavaScript framework for building user interfaces and single-page-applications [22]. The framework was created by Evan You, and is maintained by him and the rest of the active core team members [23]. It is well-established amongst the front-end user interface frameworks, used by several large companies, and has large amounts of available resources for developers [65]. On the Vue home-page it is described as an approachable, versatile and high performance framework that help create a maintainable and testable code-base [24]. Vue.js claims that as a progressive framework, it can be built upon existing server-side application for richer and better interactive experiences [22].

1.4 Outline

Chapter 1 introduced the background for this thesis' relevancy and introduces the compared frameworks.

Chapter 2 & 3 is focused on the similarities and differences found in the frameworks' development environment, language differences, concepts and structure. These chapters purpose is primarily to show-case differences and similarities between the frameworks, while making the reader more accustomed to how the frameworks work.

Chapter 4, 5, 6 & 7 are focused on the comparison of the frameworks, where features the frameworks have are used to observe how they compare to each other.

Chapter 8 & 9 concludes our work, lists the thesis references and lists the future work this thesis should focus on.

Chapter 2

Architecture and Structure

In this chapter we will look at some shared similarities and concepts that build the Vugu and Vue frameworks. We compare how they implement these concepts and the differences in their implementations. First we compare the languages and give an introduction to Vugu and Vue code. Then we go deeper into what components are, and how they are implemented. We continue this by explaining the concept of a Virtual DOM and finish the chapter by going over how the frameworks achieve a reactive design.

2.1 Language Differences

Vugu and Vue use objects to store data, methods to create functionality to objects, and separate HTML templates for the presentation of data from these objects. This shared user interface structure that we find in both libraries mainly differ in language syntax and supported library functionality.

In most Vugu use-cases, an object is initialized alongside the render-er (that updates the DOM), the build environment (that supports core mechanisms for the framework to work) and the mount point for WebAssembly. The initialized objects structure and methods are then used to contain and update data from the DOM.

```
<script type="application/x-go">
import "fmt"

type Root struct {
    messageNum int
}

func (c *Root) logMessage(message string) {
    c.messageNum++
    fmt.Printf("LogEntry#%d :\t%s\n", c.messageNum, message)
}
</script>
```

Listing 2.1: The structure found in a root.vugu example file without the HTML template. A Root object is an example of the object initiated by WebAssembly to contain data.

Similar to Vugu, when initializing Vue, a JavaScript object is defined. The Vue instance uses this object to contain data, that are both sent and retrieved between the DOM and itself. Note that the structure of the defined object is not strict in JavaScript, as JavaScript allows for loosely-written objects.

```
<script type="text/javascript">
    let vueInstance = new Vue({
        el: '#root',
        data: {
            messageNum: 0
        },
        methods: {
            logMessage: function(message) {
                this.messageNum++;
                console.log(`LogEntry#${this.messageNum} :\t${message
    }`);
            }
        }
    })
</script>
```

Listing 2.2: Example of an equivalent framework instance in Vue without the HTML template.

The most noteworthy differences between frameworks written using Vue and Vugu are not based upon syntax or structure, but rather how Vugu and Vue are able to use the written syntax and structure to manipulate the DOM. This is primarily done through components.

2.2 Components

The DOM is made up of nodes that use a hierarchical design to structure user interface elements [5]. One of the key characteristics Vue and Vugu share is grouping these elements into constructs that explain both design and interactivity of the website. In other words, components are objects that the frameworks instance uses to group functionality, data and design [6]. The frameworks separate these structures by representing the component as a single DOM element and defining this single DOM element as part template, part code to explain the behaviour and design of this single element. This means that the templates the frameworks use, utilize abstraction to link segments together.

Vugu and Vue structure commonly consist of a root component with nested child components, not unlike the hierarchical design the DOM uses [12]. This way of forming abstraction and grouping behaviour and design is pragmatic as it lets one reuse code and segment the DOMs complex behaviour and design into smaller simpler segments.

```
<!-- adder.vugu -->
<div>
    <div>
        <button @click="c.Increment()">Increment</button>
    </div>
    <input type="text" :value='c.value'>
</div>

<script type="application/x-go">
type Adder struct {
    value   int
}

func (c *Adder) Increment() {
    c.value++
}
</script>
```

Listing 2.3: Example of a component in Vugu. Note that the object and template are connected and independent of other DOM structures.

```
let adder = Vue.component('adder', {
        template: `
        <div>
            <div>
                <button v-on:click="increment">Increment</button>
            </div>
            <input type="text" v-bind:value='value'>
        </div>
        `,
        data() {
            return {
                value: 0
            }
        },
        methods: {
            increment() {
                this.value++;
            }
        }
    });
```

Listing 2.4: Example of a component in Vue. Note the similarities between the objects found in Vugu and Vue. The key differences are how the template is stored in data.

While the concept behind components is simple, there exists a wide variety of needs related to them. As a result, well-established libraries like Vue contain several types of mechanics connected to components. Currently, the most prominent mechanics found in Vue are supported by Vugu; explicitly the Vugu documentation supports: Static Component referencing, Dynamic Component referencing, slots, Component Life Cycles, Component Events and Modification Tracking [7].

A statically referenced component is a way to explicitly place a component type into the DOM in Vue and Vugu. The separation has no flexibility and only the specified component type can be placed at the reference point. This is in contrast to dynamic components that allow for various component types to be placed.

```
1  <div vg-for='_, adder := range c.AdderComponents'>
2      <vg-comp expr="adder"></vg-comp>
3  </div>
4
5  <script>
6  type Root struct {
7      AdderComponents  []*Adder
8  }
9  func (c *Root) Init() {
10     c.AdderComponents = make([]*Adder, 5)
11 }
12 </script>
```

Listing 2.5: Example of a statically referenced component in Vugu. Note that only a component of type Adder may be placed inside the AdderComponents.

```
1  <body id="root">
2      <adder v-for='num in this.adderAmount' :key='num' v-bind:ref="'
   adder' + num"></adder>
3  </body>
4
5  <script>
6  let app = new Vue({
7          el: '#root',
8          data: {
9              adderAmount: 5,
10         },
11         components: {
12             adder
13         }
14     });
15 </script>
```

Listing 2.6: Example of a statically referenced component in Vue. Note that adder components are created at the specified HTML element.

Note how Vue uses custom HTML tags to indicate the type and placement of components, as seen in Listing 2.6, Line 2. This is done differently in Vugu, where attributes are used to specify the component in a “vg-comp” custom HTML tag, this is the default way to place Vugu components, as seen in Listing 2.5, Line 2. The reason behind Vugu’s decision to avoid specific custom HTML tags for each component over specifying the placed component as an attributes, is unclear. A possible reason behind Vugu’s choice for avoiding custom HTML tags to reference components could be potential conflicts between custom HTML tags.

Another key difference is in how for-loops work in the frameworks. As seen in the Listings 2.5 and 2.6, the frameworks use an abstract form of for-loop to iterate over multiple components. The placement of the attributes that cause this iteration is different between Vue and Vugu. In Vue, the iteration occurs on the specified element. In Vugu, the iteration occurs inside the specified element.

2.3 Component Life-cycle

When building a web application, the components go through a list of initialization steps from the components creation to its destruction. During the initialization process of components, the component-specific life cycle hooks are run. Life cycle hooks are keywords for methods that allow the developer to run specific code at certain time stages relating to the component's life cycle.

Shown in Vue's model of the component life cycle diagram at Figure 2.1 The life cycle hooks are displayed as red colored rounded rectangles.

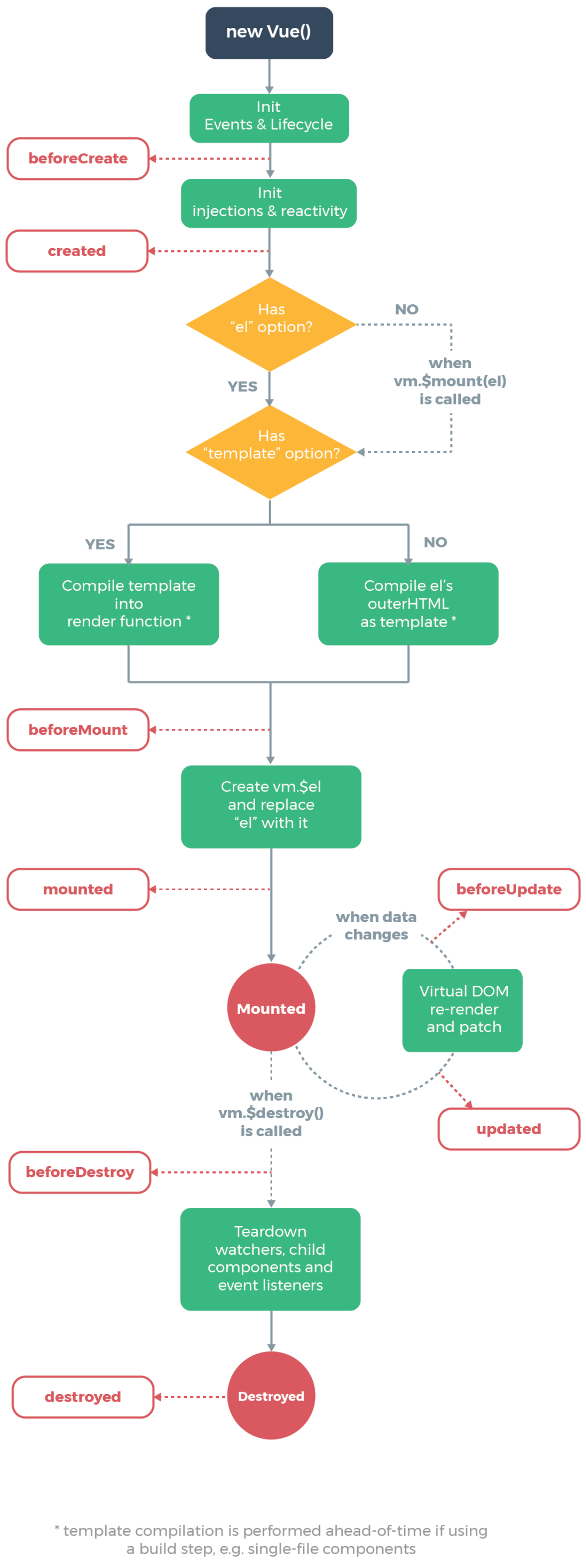

new Vue()
Init Events & Lifecycle
beforeCreate
Init injections & reactivity
created
Has "el" option?
NO
when vm.$mount(el) is called
YES
Has "template" option?
YES
NO
Compile template into render function *
Compile el's outerHTML as template *
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
updated
when vm.$destroy() is called
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed
* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

Figure 2.1: Vue's illustration of its component life cycle. Retrieved from Vue documentation [61].

We explain each of the life cycle hooks with the following description:

Component Life Cycle Hooks, Vue	
Hook	Description
beforeCreate()	This hook is executed when the component is initializing. This can be seen in Figure 2.1, the method is triggered before any Vue reactivity has been configured. (See Section 2.6.1, for explanation of reactivity in Vue.)
created()	This hook is executed after the reactivity of the component has been configured, the Virtual DOM is yet to be mounted, meaning that the component is yet to be rendered to the real DOM. (See Section 2.5, for explanation of the Virtual DOM.)
beforeMount()	This hook is called prior to rendering the DOM based on the Virtual DOM structure.
mounted()	This hook is called after the Virtual DOM has rendered the real DOM. The component is now fully accessible in the web-page.
beforeUpdate()	*beforeUpdate()* and *updated()* refers to life cycle hooks related to when data is changed in the component. *beforeUpdate()* is called before the changes in the component has been applied to the actual DOM.
updated()	This hook is triggered after repainting the real DOM is completed.
beforeDestroy()	This hook is triggered before this Vue component is destroyed.
destroyed()	This hook is called after the component is destroyed.

Table 2.1: Component Life Cycle Hooks in Vue, inspired by [62].

The Vugu documentation does not include a model that demonstrates the life cycle hooks provided, though they have given a description of available life cycle hooks. The given description in Table 2.2 [7].

Component Life Cycle Hooks, Vugu	
Hook	Description
Init()	[This hook] is called when a component is created before any other callbacks and gives an opportunity to initialize.
Compute()	[This hook] is called each time before the output of a given component is built. It provides an opportunity to compute any necessary information for each render pass.
Rendered()	[This hook] is called after each render pass has completed and all DOM elements have been synchronized with the page.
Destroy()	[This hook] is called when build/render pass occurs, and it is discovered that a component is no longer needed and gives an opportunity to destroy any resources no longer used.

Table 2.2: Component Life Cycle Hooks in Vugu, retrieved from [7].

2.3.1 Comparison of Life-cycles

Vue provides eight component life cycle hooks, while Vugu provides four. The frameworks provide similar life cycle hooks, though Vue has some extra hooks available. The life cycle hooks between Vue and Vugu that appears to be similar are listed below in Table 2.3. These life cycle hooks execute at similar timestamps:

Similar Life Cycle Hooks

Vugu	Vue
Init()	*created()*
Compute()	*beforeMount()*
Rendered()	*mounted()*
Destroy()	*beforeDestroy()*

Table 2.3: Component Life Cycle Hook Comparison

Though Vue and Vugu have similar life cycle hooks, Vugu lacks life cycle hooks related to when data in a component changes, and after the component is destroyed.

2.4 Passing Information between Components

2.4.1 Passing Information between components in Vue

To reference nested components in Vue and Vugu, the frameworks operate differently. While the Vue application may have child components, their behavior are independent, meaning that a parent and its child are segmented. The communication between these segmented parts have restrictions. Vue allows communication between a parent and child through two primary methods:

Props, these are properties of the child component, set by the parent component at initialization. Their purpose is to pass initialization data to customize components based on parameters from the parent. As can be seen from Listing 2.7, Line 4 and 11, an initial value is passed by binding data to the prop. This does not provide two way communication, but is instead used to initialize the component.

Event-triggering, by invoking methods found in a component, outside structures are able to cause changes to the component. This is done differently between child and parent, as the reference the component has to its parent, differs from the reference the parent has to its child components.

```
1  <body id="root">
2      ...
3      <!-- Child Components -->
4      <adder v-for=’num in this.adderAmount’ :key=’num’ v-bind:ref="’
   adder’ + num" v-bind:startvalue="initialvalue"></adder>
5      <!-- Parent Computed -->
6      <div>{{totalSum}}</div>
7      ...
8  </body>
9  <script>
10 let adder = Vue.component(’container’, {
11         props: [’startvalue’],
12         template: ‘
13         <div>
14             <button type="button" v-on:click="increment">Increment</
   button>
15             <input type="text" v-bind:value=’value’>
16         </div>
17         ‘,
18         data() {
19             return {
20                 value: this.startvalue
21             }
22         },
23         methods: {
24             increment() {
25                 this.$parent.incrementShared();
26                 this.value++;
27             }
28         }
29     });
30 
31 let app = new Vue({
32         el: '#root',
33         data: {
34             adderAmount: 0,
35             sharedSum: 0,
36             initialvalue: 10
37         },
38         methods: {
39             incrementAll: function() {
40                 Object.values(this.$refs).forEach(adderComponent => {
41                     if(adderComponent != undefined) adderComponent[0].
    increment();
42                 });
43             },
44             addAdder: function() {
45                 this.adderAmount++
46                 this.sharedSum += this.initialvalue;
47             },
48             incrementShared: function() {
49                 this.sharedSum++;
50             }
51         },
52         components: {
53             adder
54         },
55         computed: {
56             totalSum: function() {
57                 return (this.sharedSum != 1)? this.sharedSum + " Clicks
     Total" : this.sharedSum + " Click Total";
58             }
59         }
60     });
61 </script>
```

Listing 2.7: Vue example where information is passed between parent and child.

The example displayed in Listing 2.7, makes updates from child components to the parent, and from the parent to the child components. It should be noted that while the parent has information regarding what component types it has, as seen in Listing 2.7 Line 53, it does not specify how many components there are present. The amount of components initialized is decided by the developer's application logic and triggered DOM events.

The parent contains a list of references to children. This list is used to reference all component information of every child components the parent has. To interact with the correct child component the list needs to interpreted. An example of how this may be done is seen in Listing 2.7 Line 40. While the example from Listing 2.7, does not contain differing component types, this may pose a problem if the list of references also contain other components. How would Vue then know which references are “adder” components and which are not?

A solution to finding the correct children is by binding a reference name to the component. This is done in Listing 2.7, Line 4, however not properly implemented to show that it is not strictly necessary to reference child components by reference name. Direct component referencing would be done by the parent using the set component references found in Line 4. Listing 2.8, displays the needed changes to switch to direct component referencing.

```
    let referenceName;
    for(let i=0; i<this.adderAmount; i++){
        referenceName = "adder" + i;
        this.$refs[referenceName][0].Increment();
    }
```

Listing 2.8: Direct Component Referencing can be done by replacing the code contained inside Listing 2.7 Line 39 with the above code.

Communication from a child component to its parent is simpler, as there is only one parent. As can be seen in Listing 2.7, Line 25, the parent is updated in a similar manner by a child component, where a parent’s method is triggered.

Noticed how these objects are updated, instead of updating the values stored directly, the values are updated through methods. This is intentional as component values should not be increased directly, but indirectly through triggering child component methods, like “increment()”, as seen in Line 41, in Listing 2.7. The reason the parent and child do not directly change each others data values in Vue is connected to how the framework ensures reactivity. This is further explained later on in this chapter, but a quick explanation is that the value change would not trigger an update of the DOM.

2.4.2 Passing Information between Components in Vugu

Vugu does not have the same restrictions for component referencing as Vue. In Vugu, there are no concerns for how the components are updated. While there is no strict need to trigger updates by calling methods, it would be a better practice in most cases, as the way the component is updated, is often better suited to be done through methods to standardize the way an object is interacted with.

```
<!-- root.vugu -->
<!-- Child Components -->
<div vg-for='_, adder := range c.AdderComponents'>
    <vg-comp expr="adder"></vg-comp>
</div>
<!-- Parent Computed -->
<div vg-content='c.TotalSum()'></div>

<script type="application/x-go">
import "strconv"
type Root struct {
    AdderComponents  []*Adder
    SharedSum        int
}
func (c *Root) Init() {
    c.AdderComponents = make([]*Adder, 0, 5)
    c.SharedSum = 0
}
func (c *Root) AddAdder() {
    c.AdderComponents = append(c.AdderComponents, &Adder{value:0,
root:c})
}
func (c *Root) IncrementAll() {
    for _, adder := range c.AdderComponents {
        adder.Increment()
    }
}
func (c *Root) TotalSum() string {
    if c.SharedSum == 1 {
        return strconv.Itoa(c.SharedSum) + " Click Total"
    }
    return strconv.Itoa(c.SharedSum) + "  Clicks Total"
}
</script>

<!-- adder.vugu -->
<div>
    <button type="button" @click="c.Increment()">Increment</button>
    <input type="text" :value='c.value'>
</div>

<script type="application/x-go">
type Adder struct {
    value   int
    parent  *Root
}
func (c *Adder) Increment() {
    c.value++
    c.parent.SharedSum++
}
```

```
    </script>
```

Listing 2.9: Example of how information may be passed between components in Vugu. It includes code snippets from both “root.vugu” and “adder.vugu”.

In Listing 2.9 Line 12, the parent stores references to child components as “*Adder” objects. As a result, there will be no need to assign reference names for components or ensure that the object is of the correct component type. As the list of Adder components are segmented from other components and there is no need to keep track of how many Adder components there are present, as the length of the component list already does so.

As long as the component has a reference to the target it wants to update, it may do so. This is seen in Listing 2.9 Line 24, where the parent triggers a child component method to increment its value, or in Line 48, where the child component updates its parents value directly.

2.5 Virtual DOM

The DOM is an API (Application Programming Interface), where a document is represented through hierarchical objects. The DOM is initialized by the browser with the help of an HTML or XML file, where the structure of the file includes a set of HTML or XML elements. Each element in the Markup language represents a corresponding JavaScript object.

The DOM can be interacted with JavaScript and is highly connected to JavaScript [64]. This implicates that if any other programming language wants to interact with the DOM, it often requires a JavaScript wrapper to communicate with the DOM. DOM Operations include adding content or events, deleting content, or modifying the content of the document. As in-line styling and attributes are contained by the DOM, this includes modifying attributes of elements or the CSS styling.

The Virtual DOM is an abstraction of the real DOM, and is structured as a tree data structure, resembling the real DOM. The goal of the Virtual DOM is to realize a virtual and lightweight representation of the actual DOM API. The Virtual DOM can be used in multiple coding languages, either with respect to the coding language used in the front-end framework, or the language the Virtual DOM is implemented in. For instance, in Vue applications Virtual DOM consists of virtual nodes of JavaScript objects. While in Vugu applications the virtual nodes are Go objects.

Utilization of the Virtual DOM is more efficient than communicating with the real DOM in the browser [57]. The Virtual DOM is generated based on the implemented web application, where virtual nodes are functioning corresponding to how the actual DOM node would look.

When a change occurs in the application, for instance a value is updated, or some styling changes. The framework generates an updated Virtual DOM with the new changes. The framework now has two virtual DOMs, the newly generated one and the old one. The framework utilizes a “diff algorithm” to find the least amount of changes necessary to keep the real DOM up to date. The results from the “diff algorithm” returns information of what nodes that needs to be updated in the real DOM. The “diff algorithm” has to be efficient and productive to ensure a good end-user experience. Its speed is critical as the comparisons are made after every change. [58]. Using the Virtual DOM and the “diff algorithm” prevents re-rendering of the real DOM after every new change, and results in re-rendering of only the specific nodes returned by the algorithm.

In Vue the virtual DOM is built in JavaScript, the nodes are called VNodes and are Vue components, a JavaScript object that extends the Vue instance. The virtual DOM is mounted to the HTML DOM using the property “el”, if the Vue instance does not have a parent node then it is the root node.

In Vugu the virtual DOM is built in Go, and nodes are called VgNode. The Virtual DOM is automatically generated from the .vugu-files where the view of the web application is implemented. The generated file is pure Go code, and represents the Virtual DOM.

In the Vugu documentation there was no information regarding the implementation of the Virtual DOM. In the Vugu GitHub repository, the package titled “domrender” has the job of taking all the information from the virtual DOM and rendering it to the actual DOM. This happens in cycles, so the render loop keeps the web-application in tact, keeping everything synchronized with the real DOM. It should be noted that as the “diff algorithm” and the render often is run at the end of a task, these updates often occur in batches.

2.6 Reactive Design

2.6.1 Reactivity in Vue

One of the core reasons to utilize a JavaScript framework over plain vanilla JavaScript when developing an application, is modern frameworks reactivity systems. The goal of a reactivity system is to ensure that data is synchronized throughout the application, while trying to make this data synchronization as non-intrusive as possible for the end user.

The way Vue achieves a reactive design is through tracking changes done to data in the application. This is done when inserting and modifying data in a Vue application. Vue does this by going through all the datas properties and convert the inputs of this data to “getters”/“setters” using the ES5 JavaScript “Object.defineProperty” [25]. These “getters”

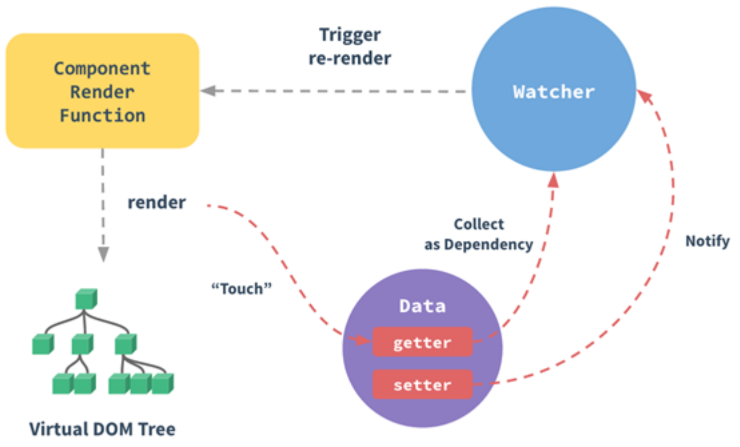


Figure 2.2: Dependency-tracking and change-notifications in Vue. Figure retrieved from Vue documentation. [25]

and "setters" are invisible to the user and act as if the data is static. It is through this mechanisms that Vue is able to perform dependency-tracking and change-notifications. "Object.Defineproperty" is a static method that defines a new property directly on an object, or modifies an existing property on an object. It returns the object afterwards [26].

```
let data = { price: 5, quantity: 2 }
Object.defineProperty(data, 'price', {  // For just the price property
      get() {                           // Create a get method
        console.log('I was accessed')
      },
      set(newVal) {                     // Create a set method
        console.log('I was changed')
      }
  })
data.price                              // This calls get()
data.price = 20                         // This calls set()
```

Listing 2.10: Vue's example code that demonstrates how a getter and setter may work [27].

"Object.defineProperty" makes it possible to implement "getters" and "setters" for a property. When a "getter" or "setter" for instance is accessed or a change happens to the property, the dependencies that are correlated to this property will change. Every component instance has its corresponding watcher instance. The watcher instance records if any of the properties are "touched". When a dependencys setter is triggered, the watcher is notified, triggering a re-render of the component [25].

After Vue as updated the relevant variables and data in the Vue instance, the watcher is informed that an update has happened via the "getter" or "setter". If a new variable is created, the watcher will store all of that variables dependencies. Then if it gets changed all of the dependencies gets updated. After this update has happened, the watcher will trigger a re-render of the Virtual DOM. After this new updated version of the Virtual DOM is created, Vue will compare the new Virtual DOM tree with the old one. If it detects a change Vue will only re-render the parts that are different in the actual DOM.

While using “Object.defineProperty” will detect changes on object changes, there are also limitations to what changes can be detected. Specifically, array changes cause problems [56], for such cases, the implemented “Vue.set()” function or Array.Prototype functions are required to ensure reactivity.

Data Binding

With Vue you can bind your application logic to the DOM two ways:

- One-way data binding: *v-bind*
- Two-way data binding: *v-model*

The former way of binding data is single-directional connecting the application logic to the DOM, meaning that if a change happens in the application logic, the DOM changes correspondingly. The latter provides double-binding meaning that it has the same functionality as the single-directional connection, but includes functionality that updates changes both ways. Meaning DOM changes make the corresponding data change as well.

2.6.2 Reactivity in Vugu

While the structure of components and data are quite similar in Vue and Vugu, there are also a lot of differences between the inner workings of the frameworks. In regards to reactivity the goal is simple; when a change in the instance object data occurs, the dependent representation of that value in the DOM should change correspondingly. The way Vue achieves this reactivity is with watchers and through “get” and “set” functions. This is not the same case for Vugu.

First of all, it is important to establish how Vugu works. In Vugu, the WebAssembly instance is responsible for handling incoming function calls. While the structure of the components may be similar to Vue, the way these components are monitored differ. As the Vugu documentation does not go into specifics regarding how these components are monitored, our understanding is lacking in how these updates occur.

From our understanding, changes to the “Virtual DOM” trigger re-rendering of components. The changes to the virtual DOM occur from the monitored data field found in the components themselves. This is related to the modification tracking Vugu does of components. The modification tracking is done on a component implemented basis and if unimplemented, the Virtual DOM is reconstructed with the new present data. Brad Peabody states that the default way Vugu tracks if a component has changed is that it scans all the fields of the component and utilizes hashing and other unspecified mechanics to keep track of these changes [14].

As a result of the modification tracker triggering re-renders of the DOM, updates in the component data-fields are caught in the render-cycle, concluding that one-way binding of data is achieved. There are no two-way data-binding in Vugu, however it can be implemented by the developer. Causing changes in the DOM trigger an event instead, and the data-fields updates.

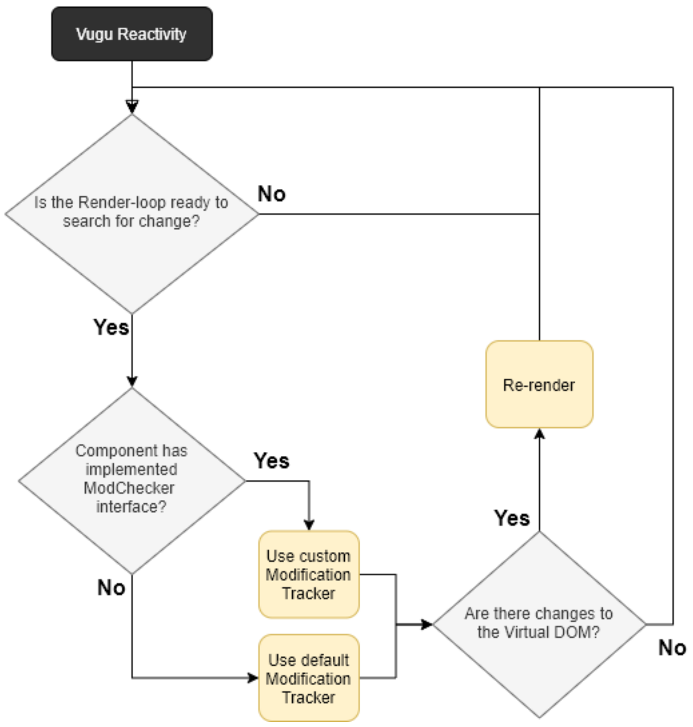

Vugu Reactivity
Is the Render-loop ready to search for change?
No
Yes
Component has implemented ModChecker interface?
Yes
No
Use custom Modification Tracker
Use default Modification Tracker
Are there changes to the Virtual DOM?
Yes
No
Re-render

Figure 2.3: Our understanding of the reactivity cycle in Vugu. The ModChecker interface is responsible for how component modification tracking is done.

Chapter 3

Framework Development

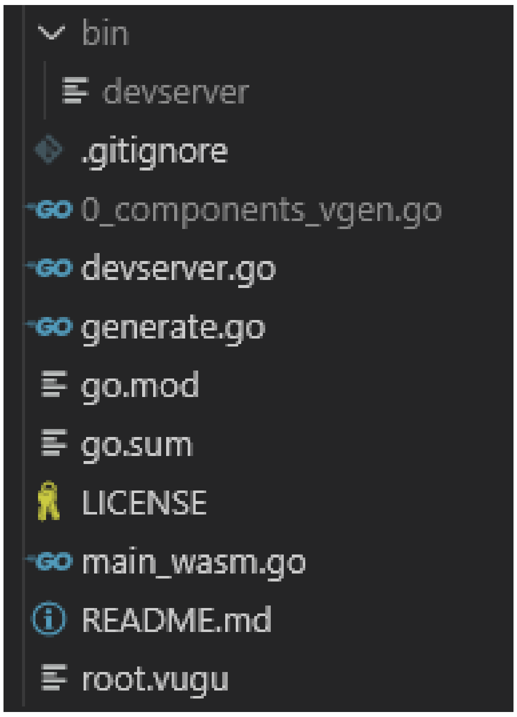
One of the most important topics for a developer is the environment they develop in. An unsuited environment may affect productivity and the code produced. In this chapter we explain how the environment created for the framework development look. This is meant to both give insight into how the frameworks are put together in a broader sense, as well as evaluate the development environments provided by the frameworks. To do this, we look closer at the development environments provided by Vugu and the Vue CLI tool, the minimum requirements needed to develop in Vue. In addition we look at the available resources provided in both the frameworks documentation to further evaluate the developer usability.

3.1 Starting Vugu Development

Due to the fact that WebAssembly needs to be compiled, developing Vugu projects require more back-end functionality. As a result, a lot of the focus of Vugu is centered around what the WebAssembly will compile and what files will be served. This can seen from the files included in the “Getting Started” project Vugu suggest using as a template for developing Vugu projects.

By following this documentation, the Vugu vgrun tool is downloaded and used to create an example project for Vugu. In the documentation some of the main files in the project are explained [47], this explanation is however limited to the point that a developer can create, change and modify components and the back-end http server, to test out how the framework works. Unfortunately, it does not explain the behavior of the developing environment in depth, meaning that the user is left without deeper knowledge of how the different files used to compile the project into WebAssembly work. From our experience, this left us in the dark on how the framework worked as a whole, which reduced our ability to properly work on the different files the project use for the development environment.

The Vugu template that is downloaded through the Vugu “vgrun” tool is inferred to be the suggested platform to develop a new Vugu project. It consists of several mechanics designed to create an easy start point for Vugu projects. While the project consists of several files, the majority of these files are not required for the back-end to work, but instead required for the front-end WebAssembly to operate as intended. The following files were downloaded by the tool to create a development environment for Vugu:



```
bin
  devserver
.gitignore
0_components_vgen.go
devserver.go
generate.go
go.mod
go.sum
LICENSE
main_wasm.go
README.md
root.vugu
```

Figure 3.1: Files included in the “vgrun” example.

- “devserver.go”, this is a http server and the code responsible for performing the WebAssembly compilation. The compiler references the Vugu repository and outside of calling prerequisites found in the file by default, there is no need for extra configuration for the compiler to work properly. This creates a distance between the developer and the intricacies of compiling WebAssembly, reducing the knowledge needed to create a front-end framework in WebAssembly. The focus in this file is mostly on the configuration of the http server. In Vugu development it is run using the command: “vgrun devserver.go”.
- A main “root.Vugu”-file is created. This file includes the root component that serves as an introduction to the Vugu application.
- A Go generate handler with some specified parameters is included in the file “generate.go”. This is used to specify how go files should be generated.

- A WebAssembly target file is included, called “main_wasm.go”, this file serves as the WebAssembly start point and it is through this file the browsers WebAssembly is run, it is responsible for building the Vugu environment, initiating a parent component and is responsible for the render-loop that calls re-renders of the web-page. This file was left untouched during our testing and exploration of Vugu, as there was no required configuration in this file for Vugu to operate as expected.

- Extra files and files from the source GitHub repository are also added, such as: Go checksum and module files; go.mod and go.sum, a Vugu License file; LICENSE, files specifying how to start the project; README.md, a gitignore file; .gitignore.

The development server works by generating .go component files from .vugu files using the vugugen call from the vgrun tool, it is not a required function call, as “vgrun devserver.go” will also generate the files. In Figure 4.1, the file “0_components_vgen.go” is one of these .go generated files. This file is generated from the “root.vugu”-file and is interpreted by the “main_wasm.go” re-render loop.

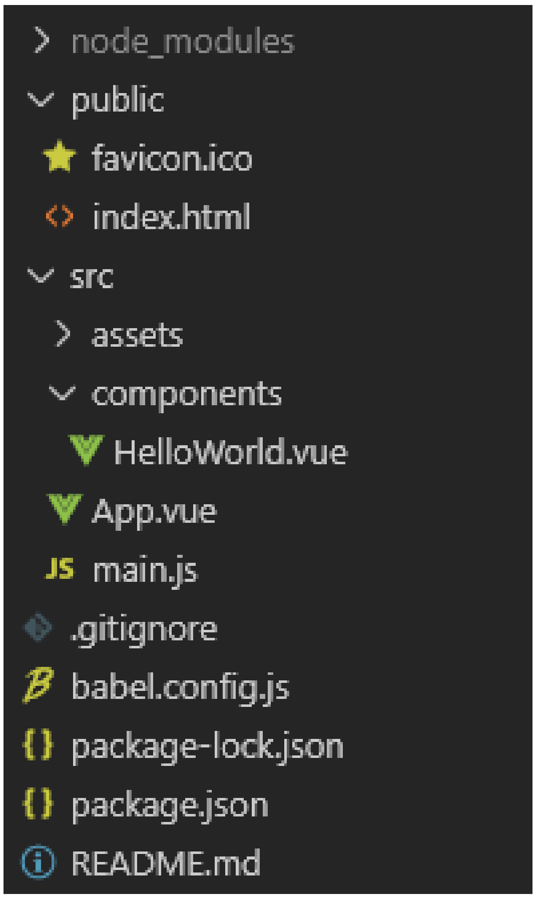
The served WebAssembly file is compiled from the .go files in the working directory based on first-line comments in .go files. To better understand this, the following example makes it more clear: as “devserver.go”’s purpose is to act as a development server, it should not be included in the WebAssembly, as such, the first line of the file states: “// +build ignore”. This tells the WebAssembly compiler to ignore the file when compiling. Other examples include “main_wasm.go”’s first line; “// +build wasm” that specifies its requirement for the WebAssembly file. Adding these comments are optional, but a .go file will be added to the WebAssembly if it is in the working directory and there is no objection in the form of a first line comment.

3.2 Starting Vue Development

Vue, in contrast to Vugu has no need to be compiled as the framework’s code-base is written in JavaScript. This simplifies the job of the back-end development server, meaning that unless other dependencies are required from the back-end development server, only file serving is required for the server to operate as intended. Currently there are two different ways to create Vue applications, through the Vue CLI by creating a Vue project and organizing the served files using .vue files or by creating Vue objects in JavaScript and serving the connected JavaScript.

3.2.1 Vue Development using Vue CLI

The former method for creating a Vue project works by organizing components into ".vue"-files in a similar manner found in Vugu. Similar to Vugu's "vgrun" tool, Vue's "Vue CLI" tool can be used to develop Vue applications. Vue CLI is dependent on the npm (Node Package Manager)[49] in order to create a Vue project through this method, both Vue CLI and npm are required. This thesis will not go into depth about how these resources are installed, instead given their installment a Vue example project can be made through the following console command: "vue create hello-world" [50]. The console will prompt the user to choose between different versions of the example project. In our case, we chose the default, Vue version 2, with babel and eslint. After this, a directory containing an example project is created.



```
> node_modules
v public
   favicon.ico
   index.html
v src
  > assets
  v components
     HelloWorld.vue
   App.vue
   main.js
 .gitignore
 babel.config.js
 package-lock.json
 package.json
 README.md
```

Figure 3.2: Files created by Vue CLI, as a template project.

The “hello-world” template created by Vue CLI, is built on top of the Node Package Manager environment, as a result, the project includes a long list of available resources to develop the front-end alongside Vue. In Figure 3.2, these are contained inside the directory “node_modules”, but because of the length of resources, they are not displayed. The files “babel.config.js”, “package-lock.json” and “package.json” are also connected to Node dependencies that can be used by the Vue CLI development.

The relevant files for Vue development in a Vue CLI project are found inside the directories “src” and “public”. A similar method to Vugu is used to structure components in these directories. Each components is separated by file, this is done to organize the different aspects of projects better. Separating components is done to attempt to fulfill the need larger projects have for organized structure. From a development perspective, structure may be highly beneficial for usability. When several developers work on a common project, the boundaries of changes and dependencies become unclear without these organized structures.

The .vue files found in the template, serve to describe components, because components describe the DOMs functionality and design, the .vue files are separated into the following sections:

- “<template>” to describe the HTML template used by the component.
- “<script>” to describe the functionality of the component.
- “<style>” to describe the CSS used by the component.

Node may in the end serve as back-end for a Vue project. The .vue files are used to build the served framework instance of Vue through JavaScript [55]. As a result, while building a Vue JavaScript file is required for projects that use the Vue CLI tool, Vue projects are in practice not dependent on .vue files nor the Vue CLI, this means that Vue also can be written in JavaScript without the need for Vue CLI.

3.2.2 Vue Development using JavaScript

Our focus has primarily been on creating smaller Vue projects to test functionality and usability, in these projects, the need for dependencies are small and the structure of the served components are not as important as in large projects. A more lightweight approach to Vue development, than through Vue CLI, is writing Vue directly in served JavaScript. This is the approach we have opted to use in our thesis examples.

HTML documents have by the default the support for added scripts. While the best practice for structuring a project would be to differentiate JavaScript and HTML, this means that the JavaScript required to utilize Vue on a web-page may be included in the .html file sent.

```html
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Single File Vue Project</title>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
    <!--Document Structure-->
</body>
</html>
<script type="text/javascript">
    let app = new Vue({ Vue Instance Object });
</script>
```

Listing 3.1: Example of a Single File Vue Project. Note the required Vue script included in the document head.

The possibility for such light-weight Vue projects is a clear advantage Vue has over Vugu, where the WebAssembly has more complex requirements to be run in a browser. As mentioned in Listing 3.1, retrieving the Vue.js source-file is the only requirement needed to start Vue development through this approach.

3.3 Framework Documentation Resources

Learning a new framework is a big part of the developer usability. Developing a new web-application require information of how the selected tools get utilized. Learning to use a new framework is much easier with a community and support system to assist in troubleshooting problems. Where one may be stuck, others may have found solutions to both specific and general problems. With a growing more prominent userbase these resources also create themselves, through communities like for example *StackOverFlow*. Nevertheless, in-depth documentation and available examples are important features to how well the technology is adopted by programmers.

3.3.1 Vue Resources

Referring to the Developer. Mozilla gives a list of resources regarding Vue [66].

- **Vue Docs**: The main Vue site. Contains comprehensive documentation, including examples, cookbooks, and reference material.
- **Vue Github Repo**: The Vue code itself. This is where you can report issues and/or contribute directly to the Vue codebase.
- **Vue CLI Docs**: Documentation for the Vue CLI. This contains information on customizing and extending the output you are generating via the CLI.
- **Vue Mastery**: A paid education platform that specializes in Vue, including some free lessons.
- **Vue School**: Another paid education platform specializing in Vue.

This in connection with resources like *StackOverflow* help both new and advanced users overcome difficult obstacles.

3.3.2 Vugu Resources

In Vugu, available resources are limited to the Go (for Go related resources in Vugu), Vugu documentation and the Vugu repository on GitHub. Except the former mentioned documentation, the user is left to learn the rest on their own. This resulting in problems becoming more time consuming to solve. Lacking and inaccurate documentation are problems that are hard to overcome for developers using experimental technologies like Vugu. Time spent is critical for development teams and focus on other matters are more important. If the Vugu framework grows, this will drastically improve discussion in online coding forums. Resulting in an increase of resources developers can utilize for questions regarding Vugu, which in return improves the developer usability.

As the Vugu framework is heavily linked to the Go community, the Vugu community attempt to ascertain better developer usability, by relying on the already implemented Go packages. A benefit Vugu gets from its connections to Go, is that it means that developers with knowledge of Go will not have major problems getting accustomed to Vugu. This aspect of usability is an important advantage for experimental technology, as it will be easier for developers to adapt to its requirements.

3.3.3 Resources compared

Vue has a bigger platform than Vugu because of Vugu's experimental nature. Vue therefore has a big advantage over Vugu in terms of developer usability. Overcoming problems when writing code using the Vugu framework is harder compared to Vue, explained by the insufficient resources Vugu has available compared to Vue.

Chapter 4

Dual Stack

This chapter centers on a web-application architecture technique that aims to increase the developer usability. The first part explains the Dual Stack term. The second part demonstrates a web application example with developer usability benefits because of the Dual Stack architecture. The third segment of this chapter aims to express more precisely how Dual Stack was utilized in the example.

4.1 Dual Stacking Explained

In this segment the term dual stack describes the ability to reuse code in both stacks of web applications, the front-end and the back-end. The prerequisite is that the front-end and parts of the back-end that will use dual stack is written in the same coding language. JavaScript is the most prominent scripting language in front-end of a web application [51]. Though if the back-end server is not written in JavaScript, the possibility of using dual stack is diminished. In Vugu both the front-end and back-end is effectively implemented in Go, making dual stacking a viable option for standardizing code. This does not mean that implementing Dual Stacking in a Vue application is impossible. By utilizing Node.js [54] as a run-time environment, JavaScript code is used both on both endpoints of the application. As a result Dual Stacking can implemented.

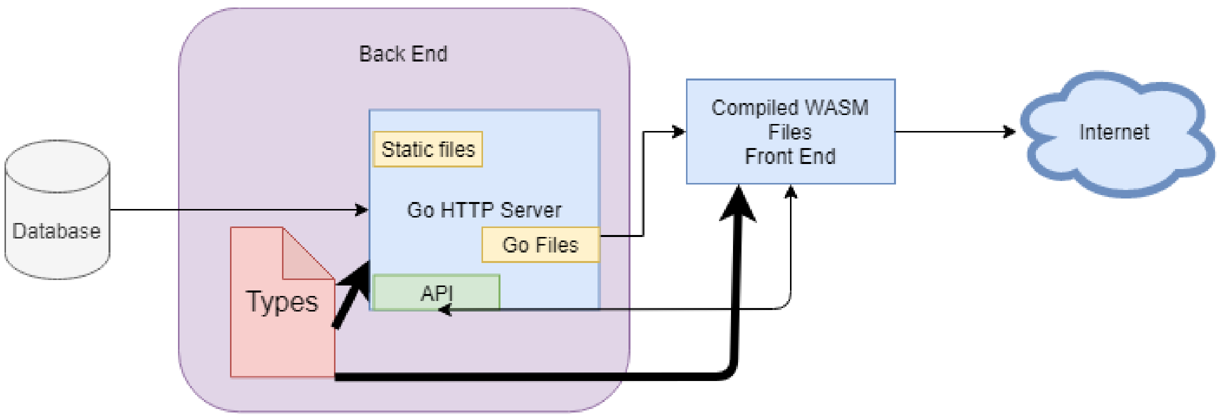
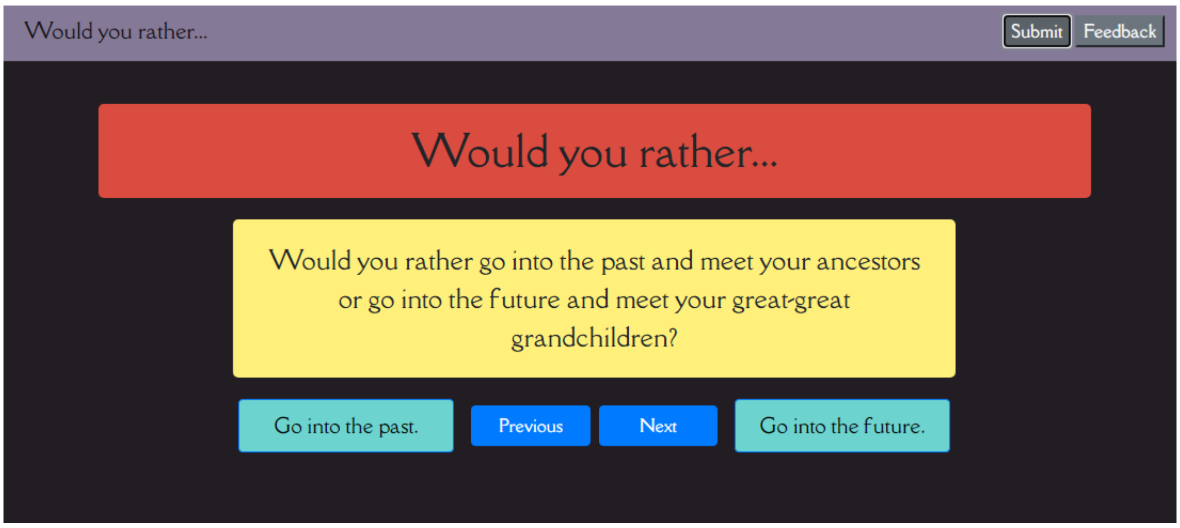


Figure 4.1: Example of how dual stacking works. Note the shared types between front-end and back-end code.

Figure 4.1, describes the architecture of how a Vugu web application using dual stack may look. The architecture has a back-end http server that serves compiled WebAssembly files, static files and serves an API that accepts http requests from clients. Dual stacking in this architecture provides re-usability of the types package in the back-end and front-end part of the application. These shared data-structures get encapsulated as a package and allows the developer to reuse types in both the front-end and the back-end.

4.2 "Would You Rather?" Example



Would you rather go into the past and meet your ancestors or go into the future and meet your great-great grandchildren?

Figure 4.2: The Would You Rather example is used to show-case the use-cases for dual stacking in front-end frameworks.

Figure 4.2 demonstrates a small web application created in both Vugu and Vue, used to compare dual stacking and AJAX. Showing the effectiveness of dual stacking in terms of re-usability of code. The Figure displays the user interface of the web-application "Would You Rather", the dual stack approach is limited to Vugu, as the server used to serve both examples is written in Go.

"Would you rather", is a game where one is asked a question, and prompted to choose between two answers. The web app emulates this by displaying the question and options on the web page, and the user clicks the answer they would rather happen. In our example, the question is displayed within a yellow label, and two answers are disclosed as buttons underneath in light-blue. Optionally, the user may also change question by clicking the two dark-blue buttons. Every question comes with two different outcomes/answers. When the outcome is chosen the client clicks on the corresponding button. This answer is sent to the server as JSON data, utilizing AJAX (Asynchronous JavaScript And XML), afterwards the user is prompted with the next question.

This small applications back-end server is a Go HTTP server, that serves both the compiled web assembly files, static files as well an API in that the clients send requests where the server acts as a RESTful API [52] as it responds with JSON data.

```go
type Server struct {
  Router      *devutil.Mux
  statements *t.StatementList
}
```

Listing 4.1: "Server" data-structure

Both the statement-list and the statements are data-structures belonging to the type package. A "Server" data-structure is used to handle API calls. statementList constitutes a part of this data-structure, as seen in Figure 4.1.

When the server receives the JSON data that contains answers at it's endpoint, the server increases the total number of votes with respect to the client's specified answer. Meaning all clients can see the total amount of votes on each of the answers of every question.

The application allows clients to submit questions to the server. The server takes the submitted question and appends the question to the "statementList" object of the "Server" data-structure given the data that is submitted is a valid "statement type". A valid statement object is valid only if the method *IsValid()* returns true.

4.3 Dual Stacking Example's Experiences

The benefits from using Dual Stacking is seen when validating answers, as the validation can be done on both end-points, the client can reduce POST-requests that are invalid by running the validation before sending it. While validating the request has to be done on the back-end for security purposes, reducing invalid traffic has the possibility of creating performance advantages for servers with large amounts of traffic.

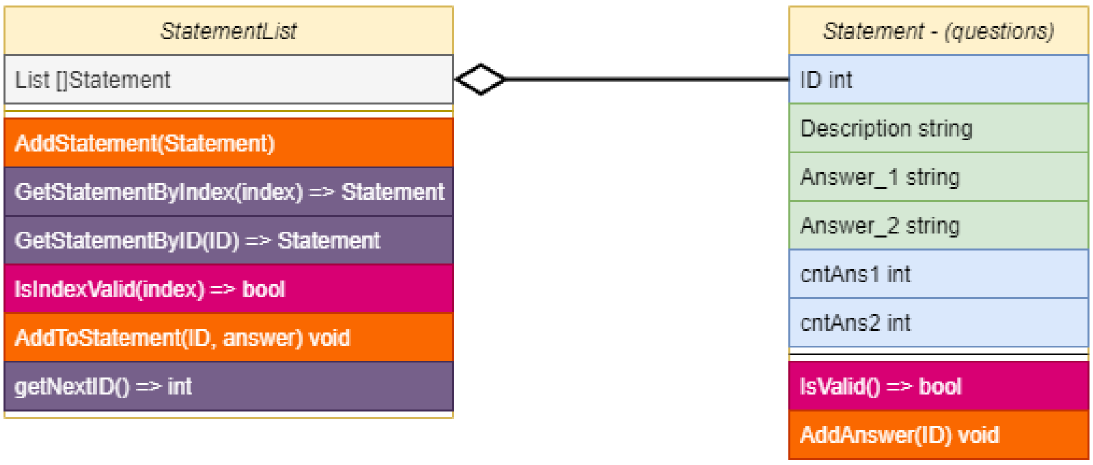
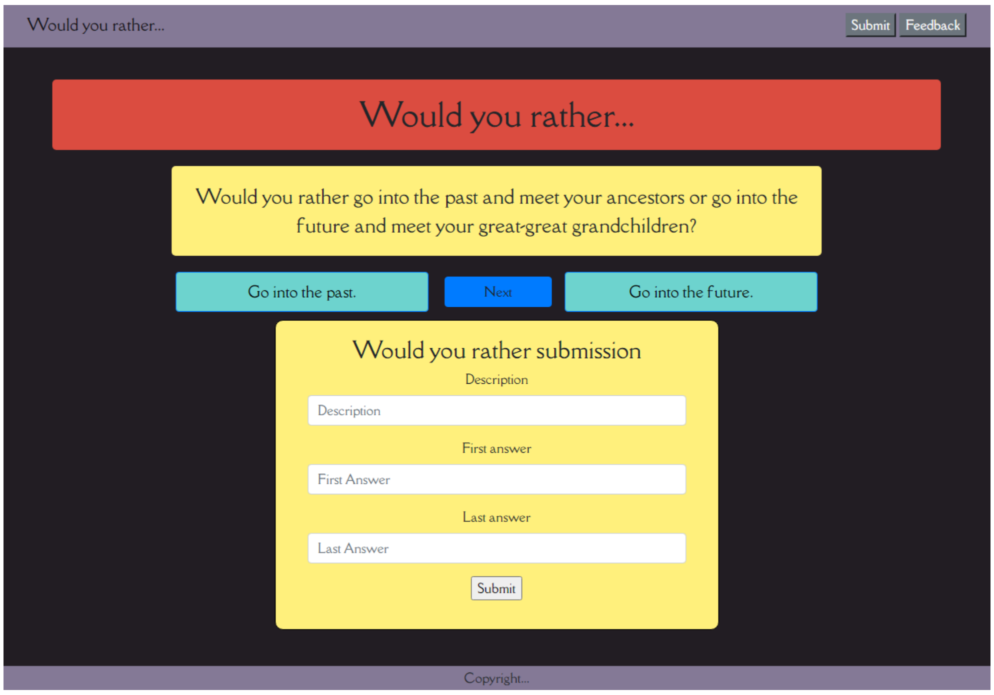


Figure 4.3: Class diagram of relationship between data-structures in the type package



Would you rather go into the past and meet your ancestors or go into the future and meet your great-great grandchildren?

Would you rather submission

Description

First answer

Last answer

Figure 4.4: The Would You Rather question submission option.

In the Vue implementation of this example the back-end server is slightly modified version from the original Vugu server. The difference is that the Vue server does not serve web assembly files, but JavaScript instead. The Vue version of the "Server" data-structure is equivalent with the Vugu version.

The problem with using Vue with a Go server compared to Vugu is that it cannot use Go defined data-structures in the type package as a JavaScript class prototypes. Therefore, we will have to recreate the same logic that these back-end classes already had implemented. This poses a problem for developers, as maintaining code is more difficult when one has to ensure functionality works similar in two different code bases. This can be especially time consuming for big classes with complex code.

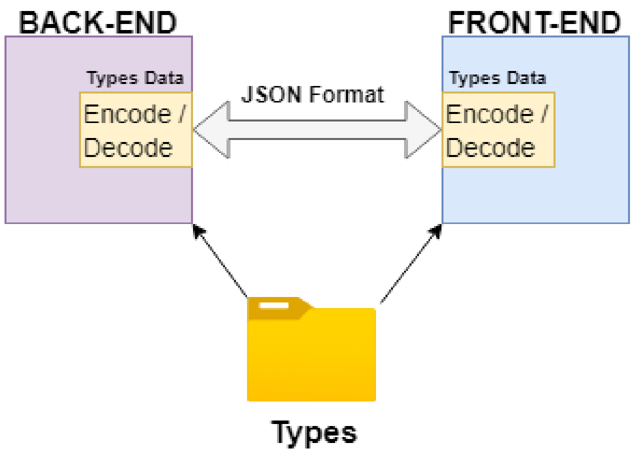


Figure 4.5: The Would You Rather example sharing a "types" object through JSON.

Our personal experience with JSON encoding Go structures are positive. The Go JSON library is able to easily decode JSON data to Go structures, and to encode Go structures to JSON data with the help of the Go JSON package [53]. Through JSON encoding, the "types" data-structure is transfered as described in the Figure 4.5.

An example of the Go code required to request objects through Go can be seen in Listing 4.2. This code requests the "statementList" object.

```go
func getStatements() (t.StatementList, error) {
    var statement_list t.StatementList
    response, err := http.Get("/api/allStatements")
    if err != nil {
        return statement_list, err
    }
    defer response.Body.Close()
    err = json.NewDecoder(response.Body).Decode(&statement_list.List)
    return statement_list, err
}
```

Listing 4.2: Code used in front-end to retrieve statement_list.

Chapter 5

AJAX

This chapter centers on AJAX and contains four parts. The first part gives an introduction to AJAX and the background of the technology. The second part demonstrates how AJAX can be used with JavaScript/Vue. The third part demonstrates how AJAX is used with Vugu, and the tasks connected one can expect in the process of implementing AJAX with Vugu. The final part of the chapter compares AJAX performance between the two frameworks.

5.1 AJAX Explained

AJAX (Asynchronous JavaScript and XML) is a technology that enables the possibility to perform asynchronous calls to retrieve data from external resources. Resources includes other web-servers, API's or the back-end hosting the web page.

In JavaScript, you can use AJAX by the help of popular methods like for example the XMLHttpRequest object, or the Fetch API. Described by MDN Web Docs XMLHttpRequests are *objects that are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a web page to update just part of a page without disrupting what the user is doing. [41]*

In classic web applications prior to AJAX, the user activity as described in Figure 5.1 was heavily dependent on the server system processing amount that was required. Reasoning was that when the server had processing to do the user could not interact with the web application before the server-side processing had finished. Creating a lack of availability of the web application as a result.

The concept of AJAX is so that you do not need to request whole web pages when getting new information from a server. Instead of requesting a whole page of data from a server. AJAX allows to request for specific information from the web server, for example contacting some endpoint hosted on a web page the user can contact this without having to retrieve all the same HTML information multiple times, increasing efficiency and reducing the processing overhead. Prior to AJAX retrieving information from webservers was inefficient in comparison.

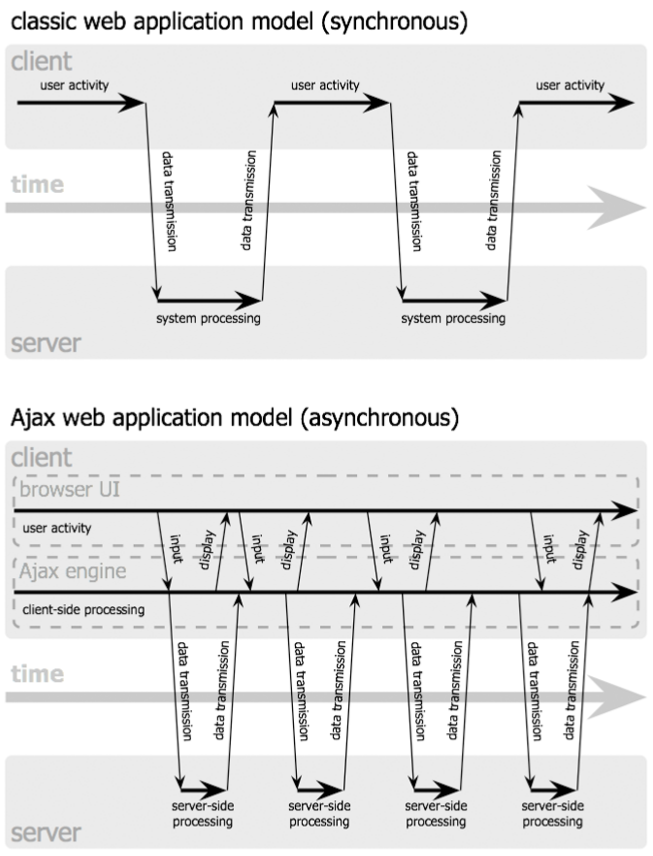


Figure 5.1: Classic Web application model vs AJAX web application model. Figure retrieved from: [40].

5.2 Vue.js

In Vue.js we utilized tools like XMLHttpRequest or Fetch to perform AJAX calls, because these are JavaScript tools. From a personal usability perspective Fetch and XMLHttpRequest are easy to use, there is a lot of resources online, explaining how because of the popularity of JavaScript and these are some of the most popular tools according to an article online presenting some ajax libraries [42].

```
updateAllStatements: function() {
    fetch("/api/allStatements")
        .then(Response => Response.json())
        .then((data) => {
            this.statements = data
        })
}
```

Listing 5.1: AJAX call to back-end server in Vue.js

This is a Vue instance method inside of the would you rather game which was explained earlier in chapter 4. The method utilizes fetch to make an asynchronous call for a resource from an API hosted the backend server. The fetch API returns a promise, the object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value [43].

If this promise is resolved we want to set the response representation to JSON, which is done in Line 3 in Listing 5.1. We use the the json method on the response because the fetch response objects body is not in JSON format and is therefore not accessible. Converting the response to JSON format returns another promise object.

```
 1 let app = new Vue({
 2     el: '#app',
 3     data: {
 4         statements: [],
 5         submit_form: false,
 6         currentStatement: 0,
 7         answered: 0,
 8         form_question: {
 9             'ID': 0,
10             'Description': '',
11             'Answer_1': '',
12             'Answer_2': ''
13         }
14     }
15 }
```

Listing 5.2: Vue.js instance data variables

If this promise is resolved, which happens if in fact this data is convertible to JSON format, we take this data which is all the statements fetched from the server API and store it inside of the statements Vue.js instance data variable. Which is shown in Listing 5.2 on Line 4.

5.3 Vugu

In Vugu code that is Go ran in the front-end of the application, meaning we will have to operate with AJAX in a statically typed language, with other code libraries and rules of what is possible to do in JavaScript.

Running code asynchronously in Vugu is achieved by using Goroutines. Goroutines are a way to execute go code asynchronously by utilizing threading, which is managed by the go run-time. So, when you write a function that is ran asynchronously the run-time does not wait for the function to finish before it continues to the next code. Though using Goroutines may seem easy to do, but if you run multiple Goroutines simultaneously and they interact with the same variables a data race may occur.

5.3.1 Data Race in Vugu

A data race refers to a situation that occurs when Go-routines operates with memory. For example a data race may happen if one of two goroutine fetches data from a web server, with the goal to update one of the front end variables, and the second goroutine accesses the soon to be updated variable before the first goroutine finishes fetching the data of that variable, it causes a conflict. Resulting in either the retrieved data being overwritten, or the second goroutines modification of said variable being purposeless.

A data races are undesired because it shows that the program containing it has some unexpected behaviour and possibilities for different end-results every time the program is ran. This unexpected behaviour causes difficulty to both debugging the program, and also increasing the correctness of the program.

5.3.2 Locks

To solve this race condition that occurs Vugu uses locks, locking the possibility of threads interacting with the same variables at the exact same time. Locked areas introduce synchronous behaviour by enforcing a limit of threads that may access the variables. In the Vugu documentation it is recommended that for operations that involve blocking, which may involve fetching data from a server, where blocking occurs while waiting for server response, or could also occur waiting for I/O operations. Vugu recommends utilization of both a goroutine and locking/unlocking during variable assignment. [46]

```go
// EventEnv provides locking mechanism to for rendering environment to
    events so
// data access and rendering can be synchronized and avoid race
    conditions.
type EventEnv interface {
  Lock()         // acquire write lock
  UnlockOnly()   // release write lock
  UnlockRender() // release write lock and request re-render

  RLock()   // acquire read lock
  RUnlock() // release read lock
}
```

```go
// EventEnvImpl implements EventEnv
type EventEnvImpl struct {
  rwmu            *sync.RWMutex
  requestRenderCH chan bool
}
```

Figure 5.2: Vugu's definition of the Event Environment interface, and implementation

Locking before variable assignment happens by using the special variable event, which is of type vugu.DOMEvent, which corresponds to event data that is sent to the frontend application by the browser. Included in the vugu-DOMEvent is the ability to create a new EventEnv(), which has the required methods for doing locking and unlocking. Vugu states that these methods behave like an RWMutex; a type in the sync Go package [46]. This is because RWMutex is a composite of the EventEnv() type, meaning an implementation of the EventEnv interface at Figure 5.2 uses composites of the RWMutex type.

Priorly stated in JavaScript we could use fetch API or Async Await with the help of promises, in Vugu we use the http package to make HTTP requests. With the http package using for example the get method, causing a GET request to the requested end point(url) returns two variables a response and error. You make these calls inside goroutines which makes the web application not having to wait for the requests to finish.

Example

In this example we will look at a specific part in the previous would you rather game application, where we used AJAX with Vugu as well as locking variables before assignment.

```
1  func getStatements() (t.StatementList, error) {
2    var statement_list t.StatementList
3    response, err := http.Get("/api/allStatements")
4    if err != nil {
5      return statement_list, err
6    }
7    defer response.Body.Close()
8    err = json.NewDecoder(response.Body).Decode(&statement_list.List)
9    return statement_list, err
10 }
```

Listing 5.3: Vue.js instance data variables

Previously in we referred to the Listing 5.3, this code shows a function in Go that requests for information with the http package on Line 3, and handles the potential error that comes with the response. After decoding the retrieved JSON to the Go object statement_list it returns the statement_list with potential errors. To use this function asynchronously and not cause blocking while waiting for the response we use this function inside of a goroutine.

```
1  func (c *Root) Init(ctx vugu.InitCtx) {
2    //Initializing the web application variables
3    c.Show_submission = false
4    c.Answer = 0
5    c.Curr_statement = 0
6    go func() {
7      statement_list, err := getStatements()
8      if err != nil {
9        panic(err)
10     }
11     ee := ctx.EventEnv()
12     ee.Lock()
13     c.Statements = statement_list
14     ptr, err := c.Statements.GetStatementByIndex(c.Curr_statement)
15     c.Statement = *ptr
16     ee.UnlockRender()
17     if err != nil {
18       fmt.Println(err.Error())
19     }
20   }()
21 }
```

Listing 5.4: Init lifecycle function using *getStatements()* asynchronously.

We use the function shown in Listing 5.3 inside a goroutine, running the function displayed in Listing 5.4 at Line 7. This is the Init function of the Vugu application, one of the life-cycle hooks, here we also initialize the default data.

Using this function we ensure that it happens asynchronously by putting it inside the go function. The event environment which ensures locking and unlocking reading and writing to the variables ensures that no data race will happen.

5.3.3 Prerequisites of Vugu

The result of utilizing asynchronous behaviour in Go requires the developer to implement locks into their applications, thus having to learn more about how this concept works. From a new developers perspective it seems like a negative that you have to use this logic, as it is not related to the application logic, and more like code that is required so that the application does not break, and that it works properly. Resulting in increased amount of code that needs to be written, and also interchanging inside the same what might be high level logic and low level locking.

For a new developer coming into Vugu it might become overwhelming when getting started if you are not familiar with Vue, React or other popular JavaScript frameworks. When first getting started with Vugu in the documentation there seems to be a lot of assumptions regarding what the user already should know.

5.4 AJAX Performance

To test the performance of AJAX in both Vue and Vugu, we used the would you rather example, first we contacted one of the endpoints of the server API with a get request. In the second performance test we tried to The goal was to fetch all statements from the web server and time if there was any significant difference between JavaScript or WebAssembly when it comes to comparing the efficiency in the web assembly compiled from go to pure JavaScript.

5.4.1 GET Requests

We ran one of the get request functions inside the would you rather example, the function has the exact same functionality, request the server for information and then check chrome devtools for how long the time was before we get a server response, simply to check if there is any significant difference of both frameworks. So we created a button that was made to test this performance. Clicking the button made a HTTP request to the server, then we went into chrome devtools onto the network tab to check what requests has been made from the browser. Then checked the timing tab, and recorded the result into a JSON file.

The button was clicked 5 times creating 5 HTTP GET requests in this first example. Performing the DOM Event called the function in Listing 5.5 in the Vugu version. This function calls the *getStatements()* function that is shown in, Listing 5.3.

```go
func (c *Root) ajaxTest() {
    go getStatements()
}
```

Listing 5.5: GET Request Vugu

```javascript
ajaxTest: function() {
    this.updateAllStatements()
}
```

Listing 5.6: GET Request test Vue

Listing 5.6 shows the Vue version of the code. Which links to 5.7

```
updateAllStatements: function() {
    fetch("/api/allStatements")
        .then(Response => Response.json())
        .then((data) => {
            this.statements = data
        })
}
```

Listing 5.7: Vue Get Test

Mean value was calculated for each of the categories, increasing accuracy in the data. The data was written manually into a JSON file. After recording the data, we calculate the mean value of each of the attributes representing different timers that calculated the speed of the frameworks, we use these values to determine the speed of the AJAX call. Calculating the mean values, we end up with these values:

These terms are explained by chrome devtools [48]

- **Request Sent**: The request is being sent
- **Waiting (TTFB)**: The browser is waiting for the first byte of a response. TTFB stands for Time To First Byte. This timing includes 1 round trip of latency and the time the server took to prepare the response.
- **Content Download**: The browser is receiving the response.

GET-request mean values			
	Vugu	Vue	Unit
Request Sent	73.4	69.2	μs
Waiting(TTFB)	0.806	0.706	ms
Content Download	0.958	0.778	ms

Table 5.1: The calculated mean values from 5 GET-requests from the AJAX example.

The time it took for the request to be sent is minimal, and the difference between the two frameworks is 4.2 microseconds, a ridiculous small difference, pretty much irrelevant.

5.4.2 POST Requests

When utilizing AJAX to perform HTTP requests, posting data to a webserver is an important feature of HTTP. Following the GET requests example we decided to check for noticeable differences in posting data to the backend server.

```go
func (c *Root) ajaxTestPost() {
    go postStatement("Testing Vugu's post speed", "Is it fast?", "Is it slow?")
}
```

Listing 5.8: Vugu POST Test

Leading to:

```go
func postStatement(description string, ans1 string, ans2 string) error {
    var statement = t.NewStatement(0, description, ans1, ans2)
    json_data, err := json.Marshal(statement)
    if err != nil {
        return err
    }
    response, err := http.Post("/api/newStatement", "application/json", bytes.NewBuffer(json_data))
    if err != nil {
        return err
    }
    posted_data, _ := ioutil.ReadAll(response.Body)
    fmt.Printf("Successfully posted %v to the server", posted_data)
    return err
}
```

Listing 5.9: Vugu Post Function

Go code 5.8 was ran to create example data and post it to the backend server. While in Vue we ran:

```
1  ajaxTestPost: function() {
2      let test_question = {
3          ’ID’: 0,
4          ’Description’: ’Testing the AJAX POST Speed’,
5          ’Answer_1’: ’Is it fast’,
6          ’Answer_2’: ’Or is it slow’
7      }
8      fetch("/api/newStatement", {
9              method: ’post’,
10             headers: {
11                 ’Content-type’: ’application/json’
12             },
13             body: JSON.stringify(test_question)
14         })
15         .then(Response => Response.json())
16         .then((data) => {
17             console.log(data)
18         })
19 }
```

Listing 5.10: Vue Post Test

The Vue code in Listing 5.10 achieves similar results to the Go/Vugu implementation. First an object is created with some attributes and values, Line 2 in the JavaScript figure, Line 3 in the Vugu figure. This is posted with fetch in JavaScript as seen on Line 8 in Listing 5.10, in Vugu the data is posted with the go http package as seen on Line 7 in Listing 5.9

POST-request mean values			
	Vugu	Vue	Unit
Request Sent	66.6	55.8	μs
Waiting(TTFB)	0.510	0.498	ms
Content Download	1.010	1.018	ms

Table 5.2: The calculated mean values from 5 POST-requests from the AJAX example.

Concluding from a performance perspective utilizing AJAX to fetch or post data to a webserver should not possess any noticeable performance differences based on the tests performed.

Chapter 6

Performance

This chapter will evaluate how Vugu and Vue perform under different types of stress scenarios. We evaluate the frameworks under different types of stress scenarios, as the frameworks serve a variety of needs to web-pages, as a result, there may be different aspects where the framework are best suited. This chapter consists of a part where we compare the frameworks run-time performance, a comparison of how fast the frameworks are able to perform DOM updates, and our attempt at increasing the DOM update performance of Vugu using custom modification tracking.

6.1 Run-time Performance

One of the main advantages WebAssembly advertises is its potential performance increase over JavaScript. To test if the potential performance increase impacts the performance of the front-end frameworks, we use the Tower of Hanoi puzzle to evaluate the computing speed of the frameworks languages.

The Tower of Hanoi puzzle [10] is a logical problem consisting of a cone made up by a specified number of stone slabs of varying sizes, and three platforms the slabs can be placed at. The goal of the puzzle is to move the cone from one platform to another. The puzzles rules are:

- Only one slab may be moved at a time.
- A slab may not be placed upon a smaller slab.

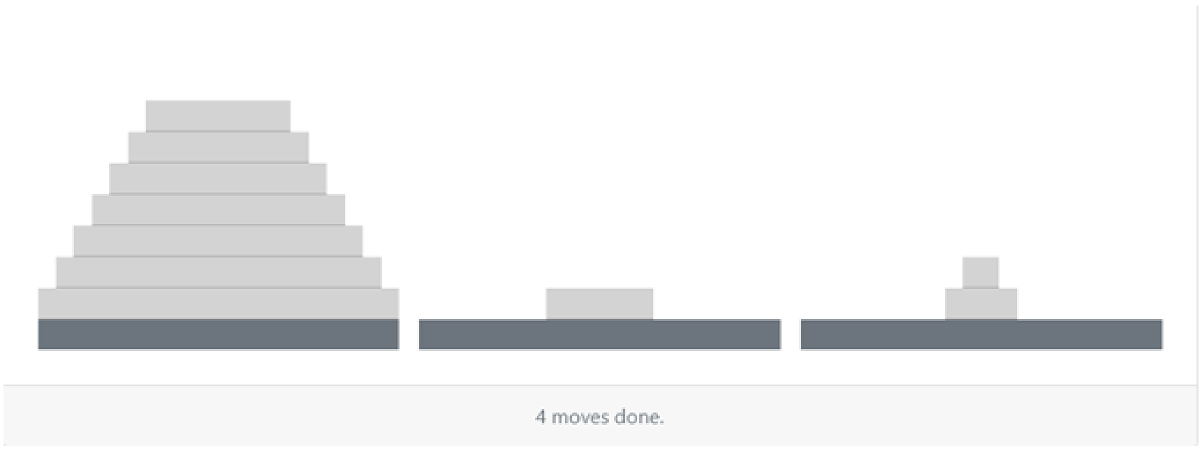


Figure 6.1: Graphical design of Tower of Hanoi example.

This is suitable for testing the languages performance as a metric for the performance of Vugu and Vue as the optimal route requires moving $2^n - 1$ slabs. This means that the run-time for the function will be affected by the number of slabs needed to complete the puzzle by O(2^n). In other words, by increasing the number of slabs, the number of moves needed are increased in an exponential growth. The puzzle was chosen because large amounts of calculations will make the difference in time it takes to complete computing the calculations more noticeable.

Our Tower of Hanoi examples include a DOM event that will trigger a function that uses the optimal algorithm for solving puzzle. While the function that solves the puzzle executes function-calls to move tiles one by one, the DOM updates will not show this in both Vugu and Vue. This is because of the way re-renders of the DOM are handled in the frameworks, by batching them together periodically. As a result, while increasing the number of slabs will have an exponential effect on the run-time of solving the towers, time used to re-render the page will not be affected as much. The time it takes to perform these DOM changes will affect the time it takes for the function to solve. As such, the example can only be used to look for trends in the data when comparing the execution time of the two languages.

Hanoi Example n=20			
Type	Vugu	Vue	
Scripting	429	21722	ms
Rendering	7	7	ms
Painting	7	9	ms
System	12	15	ms
Loading	0	-	ms

Table 6.1: Tower of Hanoi, Example Results

Table 6.1 displays the results from our testing environment in Google Chrome on the same device for both tests. It was retrieved using a Google run-time performance analysing tool [8]. The complete referenced data is included [A] and can be imported and analysed using the same run-time performance analysing [8].

The results show a large difference in scripting, while the differences between the other metrics are negligible. This is in accord with the advertised performance advantage of WebAssembly over JavaScript. While the applied bottleneck is an extreme case for computational need from a web browser, the performance benefit from using a compiled target over a dynamic target for computational processing has clear benefits.

6.2 DOM Update Performance

While the computational power of the Vugu environment is faster than the Vue environment, this does not necessarily mean that Vugu is a better alternative. One of the main purposes of both Vue and Vugu is to update the DOM. Therefore, it is relevant to not only compare the computational performance of the environments the frameworks run in themselves, but also how they interact with the DOM.

As previously mentioned, WebAssembly is unable to interact with the DOM directly. Currently to interact with the DOM, WebAssembly must interact through JavaScript [17]. This is a big disadvantage for communication between WebAssembly and the DOM, as it imposes extra computational costs on the interaction. This would not be a big issue if the interaction did not require a lot of excess computation. As JavaScript and WebAssembly do not share data types, Link Clark explains there is a need for type conversion with every interaction between WebAssembly and JavaScript [28]. As a result, the interaction between WebAssembly and the DOM is really an interaction between WebAssembly, a JavaScript type converter, a JavaScript function that performs the DOM changes and the DOM. How costly this extra computation is, is hard to calculate in practice, as it requires a deeper understanding of the JavaScript function-calls WebAssembly call.

In the previous example, the required DOM interactions were minimal, but what happens when large amounts of DOM interactions are required? A solution to test this is by evaluating the time it takes to make big DOM changes. The following examples create 10 000 components in their respective frameworks. It works by using a DOM Event to trigger a function that adds 10 000 minor components to the DOM. In this example the 10 000 components consist of a label indicating a contained counted value, with a button added to increment the components stored value.

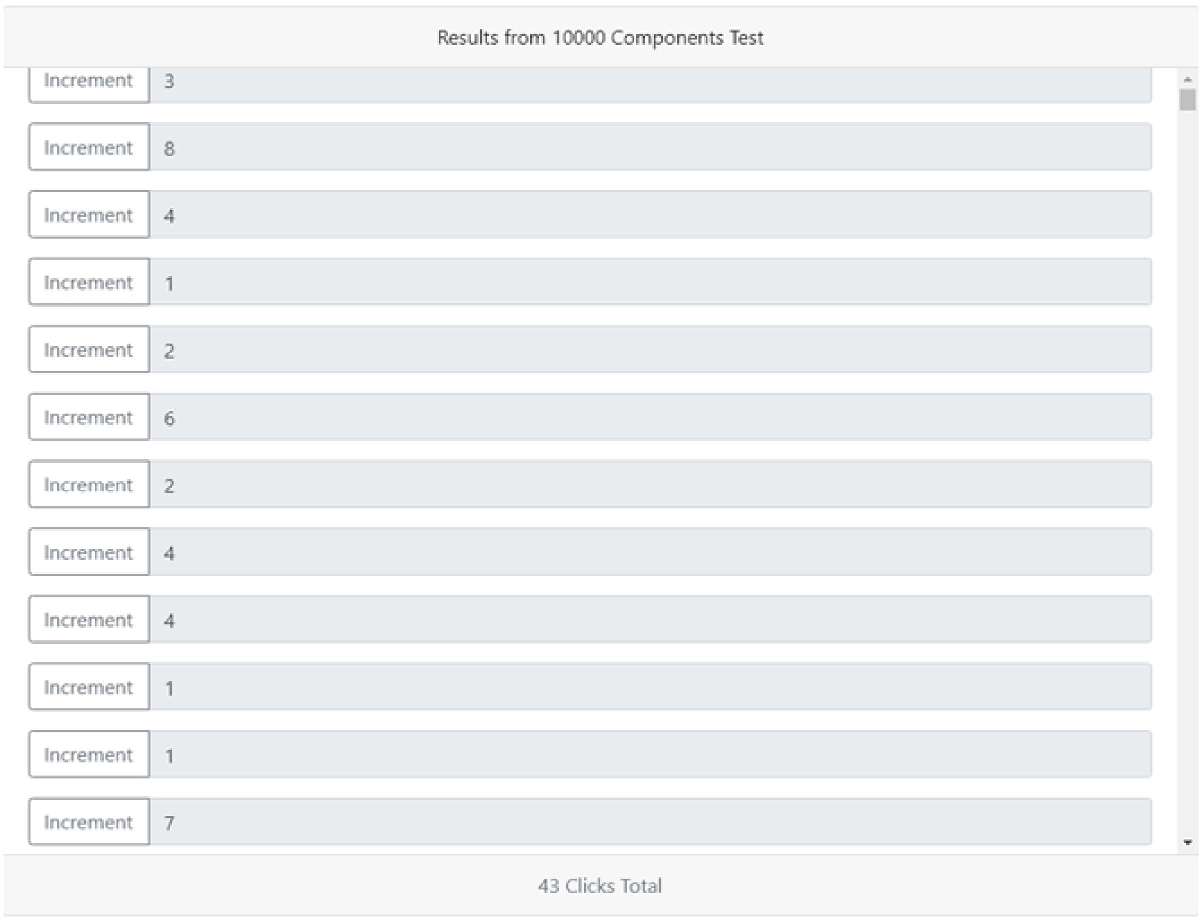


Figure 6.2: Graphical design of components 10 000 example. The DOM elements consisting of the “Increment”-button and labeled value are the 10 000 components added in the example.

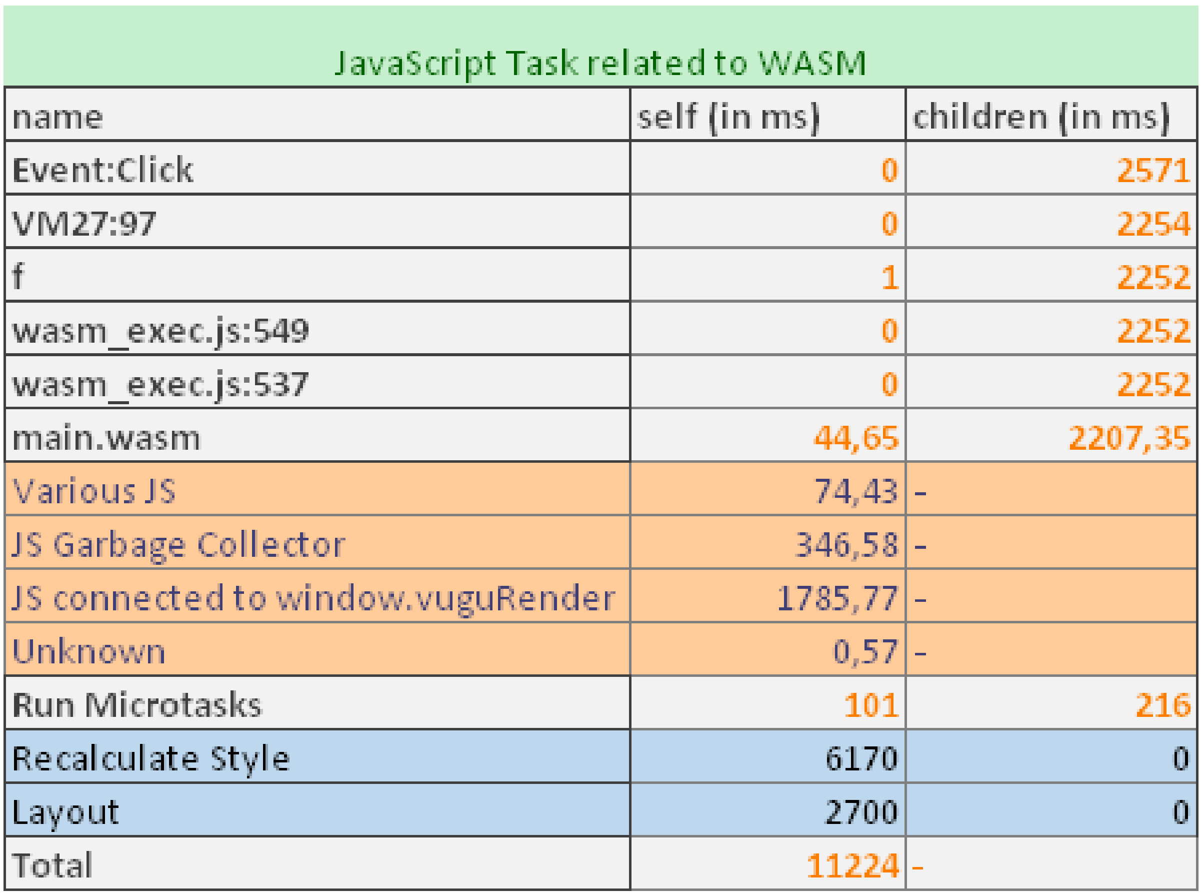
Through the Google runtime performance analysing tool [8], the call stack is evaluated against time consumption as a performance metric. Table 6.2 shows the results from the complete run of our testing environment in Google Chrome on the same device for both tests. The results were retrieved by using the same Google runtime performance analysing tool [8] used in the Tower of Hanoi example. It consists of triggering the DOM event and retrieving the summarization of the consumed time. The complete referenced data is included [B] and can be imported and analysed using the same run-time performance analysing tool [8].

Components Example Creating 10 000 Components			
Type	Vugu	Vue	
Scripting	3714	1592	ms
Rendering	9255	9186	ms
Painting	58	137	ms
System	93	118	ms
Loading	0	0	ms

Table 6.2: Components 10 000, Example Results

Both Vugu and Vue call Recalculate Style and Layout, these are calls to update the DOM and recalculate the appropriate styles for DOM elements [10]. As both examples create 10000 equally styled elements, they are independent from the performance of the tested frameworks and can therefore be neglected from the evaluation of Vugu and Vue. Further we can see that the variations between these show no clear trends in performance gains or losses between Vugu and Vue. The call for Painting is also irrelevant, as its purpose is to display the rendered page properly in the browser. For this reason, Painting is only affected by the updates that occur and as the updates are the same for both cases, the differences between these variables are neglected. Our results show that scripting is the biggest difference between the two frameworks. This confirms that it is a lot more time consuming to do large DOM updates in Vugu. While this does not confirm the hypothesis that this is caused by the overhead WebAssembly interactions require, it is the expected outcome if this is true.

To better understand if the hypothesis is valid or not, a better representation of the Vugu overhead is required. To create a representation of this overhead found in the 10000 components Vugu example requires a distinction between what is done in WebAssembly and JavaScript. While the Google run-time performance analysing tool [8] allows one to view the time a function and its function-calls take to compute, the tool itself does not distinguish scripting between WebAssembly and JavaScript in its summary. To better distinguish between the function-calls that are done in JavaScript and WebAssembly, the DOM event behind the 10000 components example was therefore analysed manually, by analysing the type, source and consumed time, in the Google run-time performance analysing tool [8]. Manually going over large amounts of data has a greater potential for mistakes than doing so through automatic tools, because of this, there may be deviations from reality in this collected data [C]. These deviations are still presumed to be minor, and the broad strokes of the results are therefore considered by us to outweigh the flaws.



JavaScript Task related to WASM

name	self (in ms)	children (in ms)
Event:Click	0	2571
VM27:97	0	2254
f	1	2252
wasm_exec.js:549	0	2252
wasm_exec.js:537	0	2252
main.wasm	44,65	2207,35
Various JS	74,43	-
JS Garbage Collector	346,58	-
JS connected to window.vuguRender	1785,77	-
Unknown	0,57	-
Run Microtasks	101	216
Recalculate Style	6170	0
Layout	2700	0
Total	11224	-

Figure 6.3: Results from manual inspection of Vugu components 10 000 results.

To sort this data, the JavaScript task called by the DOM Event that contained the WebAssembly call was analysed. The analysis of this data consisted of looking at 5 different parts of the JavaScript Task:

- The function-calls that lead to the WebAssembly call and the time it took for them and their children to execute. This consist of functions prior to “main.wasm”.
- The time it took for Non-Scripting related function-calls to execute. These are highlighted in blue.
- Micro-tasks, which are a way for JavaScript to queue additional secondary function-calls after a DOM Event function-call [36].
- Non-WebAssembly function-calls called by “main.wasm”. These functions are highlighted in orange, and are grouped as follows:
  - Function-calls without specified language and without clear purpose are tagged as “UNKNOWN”.
  - JavaScript Garbage Collector function-calls are tagged as “JS Garbage Collector”.

  - Function-calls to the JavaScript function “window.vuguRender” are grouped together with this functions child functions, as this was the most called and time consuming function called by “main.wasm”. These were grouped as “JS connected to window.vuguRender”.
  - Other JavaScript functions not called by “window.vuguRender” were grouped as “Various JS”.

- WebAssembly function-calls are grouped as “main.wasm”, to calculate their time consumption, the parent WebAssembly functions total time was subtracted by the total non-WebAssembly function-calls called by “main.wasm”s run-time, resulting in the time WebAssembly functions used in the JavaScript Task.

While the function-calls that lead to the WebAssembly part of the Vugu framework are unexplained, they are also not the biggest performance affecting part of the framework. The most run-time consuming parts are the parts related to non-Scripting related function-calls and non-WebAssembly functions called by “main.wasm”. The latter is the reason behind the big difference previously found between Vugu and Vue when comparing their performance of large DOM manipulations. If the data is correct, then it means that while the computation required in WebAssembly may take around 45 ms, the time consumption of the JavaScript functions are too large for Vugu to have a performance advantage over Vue for large DOM manipulations.

The JavaScript functions connected to “window.vuguRender” [29] called by WebAssembly impose a large performance loss, but what are the purposes of JavaScript functions connected to “window.vuguRender”? “window.vuguRender” can be narrowed down to the following pseudo-code:

```
window.vuguRender = function () {

    //assert that a buffer of bytes exist to receive from WASM
    //assert that required classes to decode and read the buffer exists
    //assert that various variables related to a state object for DOM
   manipulation
    //  are configured or fallback to some valid configuration

    //cycle instructions from buffer in an endless loop

        //decode OperationCode
        //switch on OperationCode
            //case: instruction to end decoding
                //break cycle
            //case: instruction to perform JS operation
                //perform JS operation on state
                //break switch
            //case: instruction to change state of vuguRender
                //update state
                //break switch
            //default:
                //throw error for invalid OperationCode
                //return

}
```

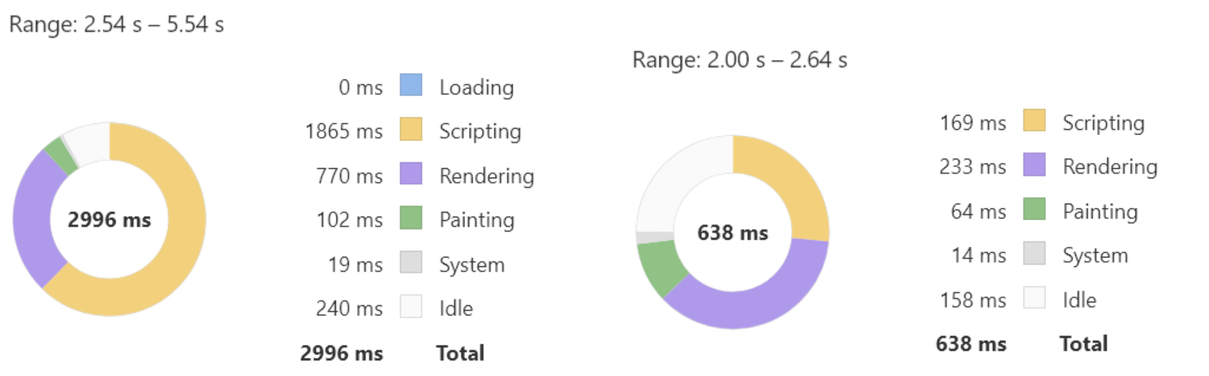
Listing 6.1: The “window.vuguRender” pseudo-code.

The functions job is to decode information from WebAssembly, convert it to JavaScript accepted type formats and perform the requested operation on the DOM. Functions connected to these operations are in other words the source to the overhead discussed [28]. These are also the operations Link Clark [28] wrote about affecting WebAssembly DOM manipulation. As our results align with Link Clark’s explanation, it is unclear whether or not the results found from the manual analysis are affected by confirmation bias; prioritizing information that support personal expected outcomes. In any case, the overhead that WebAssembly experiences when interacting with the DOM, may improve if future WebAPI’s allow WebAssembly to interact directly with the DOM.

6.3 Modification Tracking

Another problem that we found when comparing Vugu and Vue using the 10 000 components example in Section 6.2, was how long it takes to increment one component when the list of components are 10 000. Through the Google runtime performance analysing tool [8], the time consumption of updating one component out of 10 000 components were compared between Vue and Vugu. To create the data we recorded, the following was done in Vugu and Vue:

1. Start development server using “vgrun devserver.go” in Vugu or “go run startweb-server.go” in Vue.
2. Access the Vugu 10 000 Components example’s web-page.
3. Click the “10 000 Components Test”-button.
4. Access the Google run-time performance analysing tool [8] and click record.
5. Click the “Increment”-button of a component.
6. Stop the recording when the web-page has performed the increment.



0 ms	🟦	Loading
1865 ms	🟨	Scripting
770 ms	🟪	Rendering
102 ms	🟩	Painting
19 ms	⬜	System
240 ms	⬜	Idle
2996 ms		**Total**

169 ms	🟨	Scripting
233 ms	🟪	Rendering
64 ms	🟩	Painting
14 ms	⬜	System
158 ms	⬜	Idle
638 ms		**Total**

Figure 6.4: Comparison of the result from clicking “Increment” on a component in Vugu and Vue with 10 000 components present. Vugu is seen to the right of the figure. Vue is seen on the left. The datasheets used [H] to represent this data can be found and analysed using the same performance analysing tool used to retrieve this data [8].

When find that when updating a component with 9 999 siblings, there is a substantial difference in the time it takes to perform this increment in Vugu and Vue, as seen in Figure . This may be explained from how the DOM updates are performed by the frameworks.

DOM updates reflect the changes to the data stored in the framework. In Vue this is done through the concept of Reactivity, where the DOM is updated as a reaction to the components triggered methods. In Vugu this is done differently, as the DOM is not updated by the components methods, but by modification tracking from the Render-loop. The difference we found in the frameworks, may be because the modification tracker attempts to explore if the other components are changed, while Vue's method of detecting changes only occur for components that trigger change. There is no real proof for our conjecture, instead this is a problem Vugu experiences for large amounts of components that we would like to explore deeper to see if our conjecture holds.

How this modification tracking is done is unclear from Vugu's documentation. The documentation for how modification tracking works is lacking and consist of 3 connected source. [7, 59, 60] This has caused problems for our implementation, as there is little to go of to implement custom modification tracking to find out if our conjecture is correct. It is stated that implementing ones own modification tracking is unnecessary, however it is an optional feature for projects where the graph of objects become too large and things get slow [7].

This is exactly what happens in the 10 000 Components Example when we increment one component. Because of this, we impose the question of whether implementing ones own modification tracking would improve the DOM update performance of Vugu, where large amounts of components are present.

The Vugu GitHub repository includes an example of a modification tracker that works by counting changes when the component's methods are called and confirming change to the modification tracker based on whether of not the counter has increased. To apply custom modification tracking, the documentation and figure tells that the ModChecker interface needs to be implemented by the component.

```go
func (c *ChangeCounter) ModCheck(mt *ModTracker, oldData interface{}) (
   isModified bool, newData interface{}) {
  oldn, ok := oldData.(ChangeCounter)
  return (!ok) || (oldn != *c), *c
}
```

Listing 6.2: An example of how the ModChecker interface may be implemented to perform custom modification tracking of Vugu components by the Vugu GitHub repository [59].

When attempting to implement this ModCheck method in our example, we were met with build errors, the .vugu-files were unable to generate .go-files properly. This was caused by the *Modtracker object not being recognized. As a result, in our 10 000 Components Example, the following changes were done to implement the ModChecker interface through implementing the data structures ModCheck method.

```go
func (c *Adder) ModCheck(mt *vugu.ModTracker, oldData interface{}) (
    isModified bool, newData interface{}) {
    fmt.Println("Child Accessed To Detect Changes")
    return true, *c
}
```

Listing 6.3: Added Adder method in adder.vugu. Note the changes to the parameter mt of type *vugu.ModTracker.

```go
func (c *Root) ModCheck(mt *vugu.ModTracker, oldData interface{}) (
    isModified bool, newData interface{}) {
    fmt.Println("Parent Accessed To Detect Changes")
    return true, *c
}
```

Listing 6.4: Added Root method in root.vugu. Note the changes to the parameter mt of type *vugu.ModTracker.

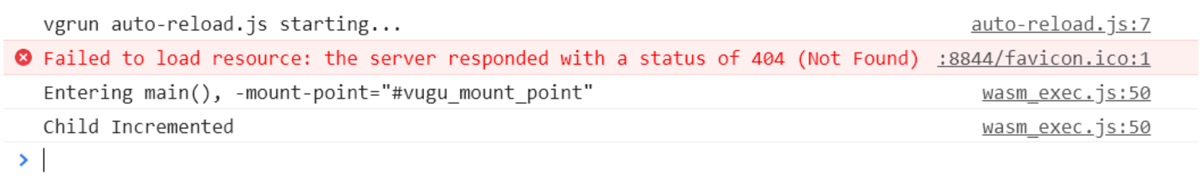
The main purpose of these methods were to ascertain that the methods are run to check for modification. To ascertain that using "fmt.Println()" would print console logs, the Adder data structures Increment() function was updated to the following:

```go
func (c *Adder) Increment() {
    fmt.Println("Child Incremented")
    c.value++
}
```

Listing 6.5: Control method: Increment(). This method was updated to ascertain that "fmt.Println()" would produce console logs in the browser.

To test how the modification tracking would execute the methods, the following was done.

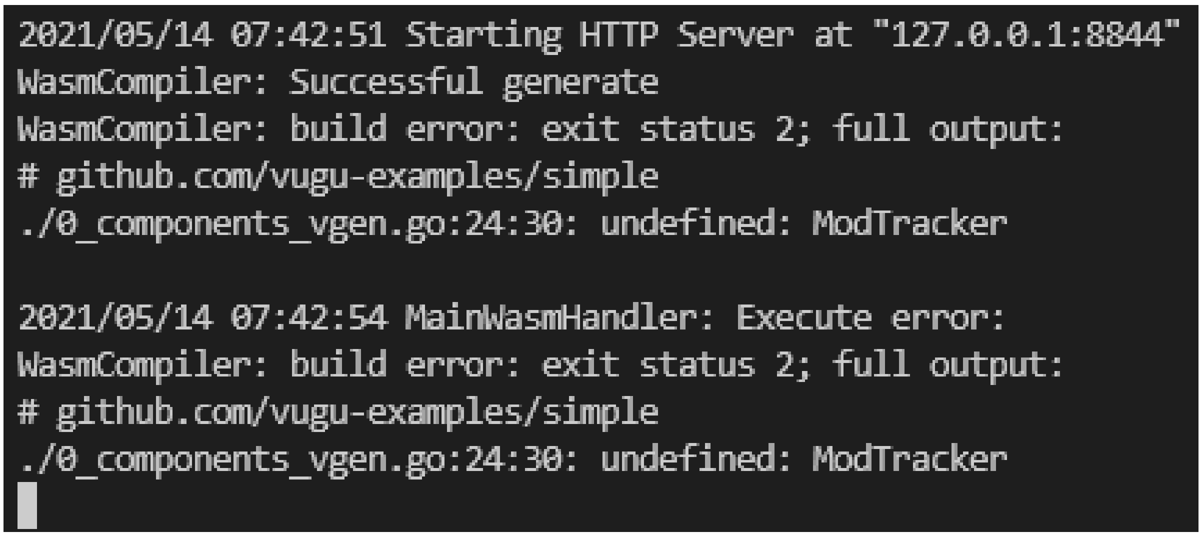
1. Start development server using "vgrun devserver.go".
2. Access the Vugu 10 000 Components Modification Tracking example's web-page.
3. Create a component by clicking the "Add New" button.
4. Click the newly created components "Increment" button.
5. Access browser's console to read the console logs.



```
vgrun auto-reload.js starting...                                                  auto-reload.js:7
Failed to load resource: the server responded with a status of 404 (Not Found)  :8844/favicon.ico:1
Entering main(), -mount-point="#vugu_mount_point"                                 wasm_exec.js:50
Child Incremented                                                                 wasm_exec.js:50
>
```

Figure 6.5: Results from the tested Modification Tracking.

The console logs displayed by Figure 6.3 show that while “fmt.Println()” does produce console logs from the “Increment()” method, the ModCheck methods are not called. Due to lacking documentation, we are unsure if our implementation of custom modification tracking is wrong or if the modification tracker interface “ModChecker” is not currently being used by the Vugu project. We were unable to find a solution to this problem. As there is no clear mistake. Using the ChangeCounter examples implementation, where a “*ModTracker” parameter is used for the method instead of the “*vugu.ModTracker” may be the solution, however our testing environments were unable to build with this parameter.



```
2021/05/14 07:42:51 Starting HTTP Server at "127.0.0.1:8844"
WasmCompiler: Successful generate
WasmCompiler: build error: exit status 2; full output:
# github.com/vugu-examples/simple
./0_components_vgen.go:24:30: undefined: ModTracker

2021/05/14 07:42:54 MainWasmHandler: Execute error:
WasmCompiler: build error: exit status 2; full output:
# github.com/vugu-examples/simple
./0_components_vgen.go:24:30: undefined: ModTracker
```

Figure 6.6: Server Console Logs, returned when using “*ModTracker” instead of “*vugu.ModTracker”.

Chapter 7

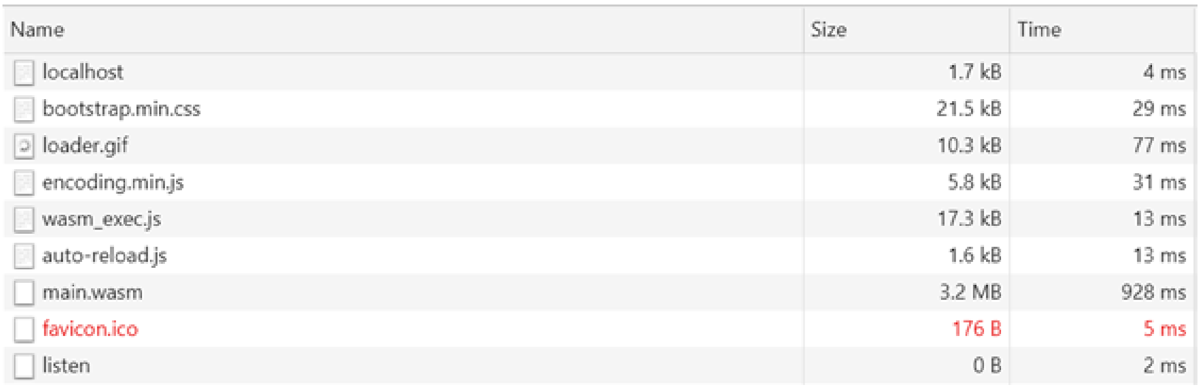
Alternative WebAssembly Compiler

This chapter takes a closer look at the loading times of web-pages that use Vugu and Vue. As WebAssembly can be compiled using different compilers, the goal of this chapter is to explore how this load time can be reduced through reducing the file size served and the computation done to serve the WebAssembly file. First the time it takes to load a similar Vugu and Vue project is compared. Then we attempt to reduce load time by reducing the served file size of WebAssembly. This is done by switching WebAssembly compiler to an experimental project; TinyGo. Finally, an attempt at reducing computation needed to serve the WebAssembly file is done by switching from a development environment to a production environment of Vugu.

7.1 Load Times

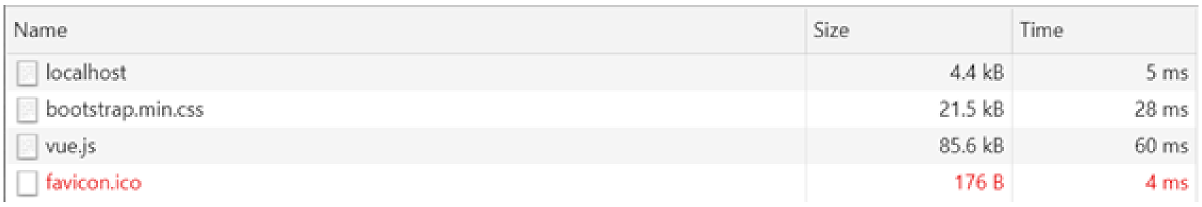
Throughout the work on the thesis, we noticed a difference in loading times for the Vugu and Vue projects when the pages initially load. More specifically, we experienced that Vugu projects take a lot more time to fully load web-pages. To better analyse what the page loads, a comparison between the resources Vugu and Vue require in the 10000 Components example, was performed. As the size of required resources affect how fast the page can load, the differing load times could be connected to the size of the files required for the environments. To analyse the required resources each respective test example used, the Google Chrome network activity analysis tool was used [33]. To capture the correct data from this tool, we did the following:

- The web-page was loaded in our testing environment using Google Chrome.
- The tool was opened in the Chrome browser and turned on.
- The web-page was refreshed with the combination CTRL+F5. (In contrast to refreshing the page with other methods, all cached content is ignored, meaning all necessary resources are requested [34].)



Name	Size	Time
localhost	1.7 kB	4 ms
bootstrap.min.css	21.5 kB	29 ms
loader.gif	10.3 kB	77 ms
encoding.min.js	5.8 kB	31 ms
wasm_exec.js	17.3 kB	13 ms
auto-reload.js	1.6 kB	13 ms
main.wasm	3.2 MB	928 ms
favicon.ico	176 B	5 ms
listen	0 B	2 ms

Figure 7.1: Results from network activity when performing a refresh of the Vugu 10 000 Components example.



Name	Size	Time
localhost	4.4 kB	5 ms
bootstrap.min.css	21.5 kB	28 ms
vue.js	85.6 kB	60 ms
favicon.ico	176 B	4 ms

Figure 7.2: Results from network activity when performing a refresh of the Vue 10 000 Components example.

The complete referenced data is included [D] and can be imported and analysed using the same networking activity analysis tool [33] used in our example.

Our results show the Vugu examples resources require more transferred data. The load times for the necessary resources also reflect our personal experiences throughout this thesis, where the time it takes to load Vugu sites are more time consuming than loading Vue sites. Load times are a crucial metric for modern websites, as it affects end-user experience, where the user experience is improved for sites with faster response times. For users with slower internet connections, it also affects the ranking of query results for the Google Search Engine [31]. In the 10 000 Components Vugu example, the loading time for necessary resources are primarily dictated by the time it takes to receive the WebAssembly file. Vugu attempts to fix this by first presenting the “loader.gif” found in the initial response request while the missing required files are fully downloaded. This does not fix the issue with more time-consuming load times, instead it is a workaround to let the user know there is progress occurring.

7.2 Compiling Smaller WebAssembly Files

A solution to the problem with large, required resources in Vugu is changing compiler. The Vugu documentation supports the TinyGo compiler; an experimental compiler focused on compiling Go to smaller WebAssembly file sizes. While there is support for the TinyGo compiler, the Vugu documentation explicitly states the possibility for incompatibility between the two projects. The documentation furthermore states expectations of improved compatibility in the future. Currently the documentation claims Vugu compiled using TinyGo supports the basic features of Vugu [32].

One of the prerequisites for changing compiler of a Vugu project to TinyGo requires one to install the TinyGo compiler. Installing the compiler itself is outside the scope of this thesis. However, given that the compiler is installed, the following changes are needed to switch compiler for Vugu projects [32, 35]:

The Vugu Go generator needs to be informed the TinyGo compiler is used. This is done by updating the following lines in the Vugu projects generate.go file:

```go
package main

//go:generate vugugen -s
```

Listing 7.1: “generate.go” without using the TinyGo compiler.

```go
package main

//go:generate vugugen -s -tinygo
```

Listing 7.2: “generate.go” using the TinyGo compiler.

The back-end server needs to compile using the correct compiler. While both the WASM compiler and the TinyGo compiler can be used by the Vugu back-end under a common interface, the compilers themselves operate differently and therefore have different requisites to compile correct. This results in necessary configurations on the TinyGo compiler instance that are not found in the WASM compiler instance. For our example, the following changes were made to the back-end go server (devserver.go) to enable compiling to WebAssembly using the TinyGo compiler (without compiling to TinyGo through Docker):

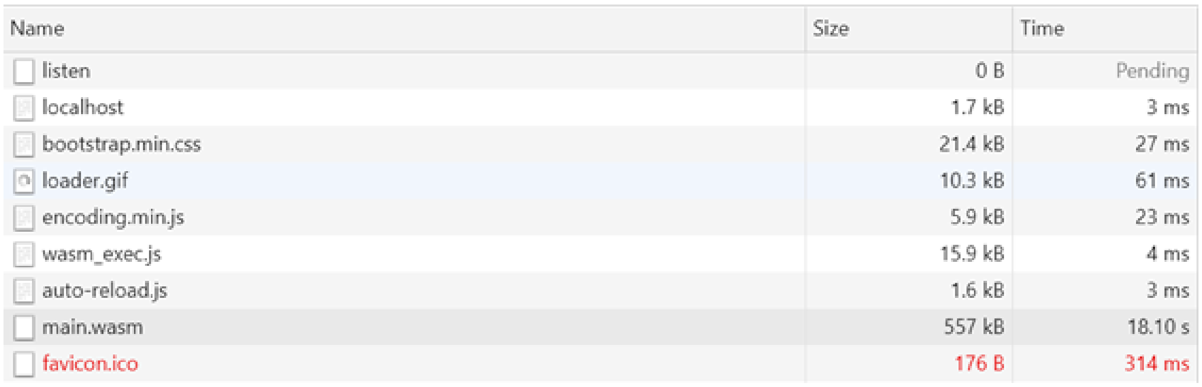
```go
func main() {
        ...
        //WASM Compiler and requirements
        wc := devutil.NewWasmCompiler().SetDir(".")
        ...
}
```

Listing 7.3: “devserver.go” code that ensures WASM compiler is used.

```go
func main() {
        ...
        //TinyGo Compiler and requirements
        wc := devutil.MustNewTinygoCompiler().SetDir(".")
        defer wc.Close()
        wc.AddGoGet("go get -u github.com/vugu/vugu github.com/vugu/vjson
    ")

        //Required to avoid calling TinyGo with Docker
        wc.NoDocker()
        wc.AddGoGet("go get -u -x github.com/vugu/vjson github.com/vugu/
    html  github.com/vugu/xxhash")
        ...
}
```

Listing 7.4: “devserver.go” replaced code that ensures TinyGo compiler is used.

By switching to TinyGo, we got the following results using the Google Chrome network activity analysis tool [33]:

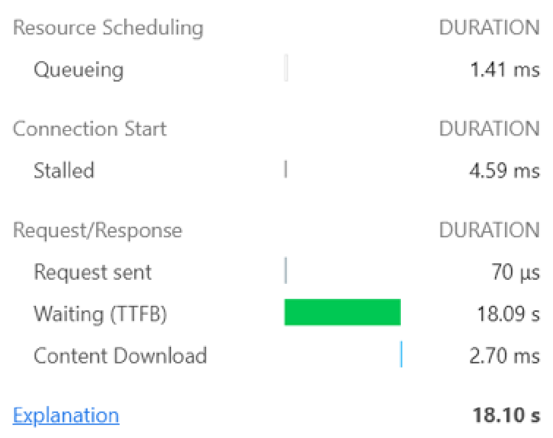


Name	Size	Time
listen	0 B	Pending
localhost	1.7 kB	3 ms
bootstrap.min.css	21.4 kB	27 ms
loader.gif	10.3 kB	61 ms
encoding.min.js	5.9 kB	23 ms
wasm_exec.js	15.9 kB	4 ms
auto-reload.js	1.6 kB	3 ms
main.wasm	557 kB	18.10 s
favicon.ico	176 B	314 ms

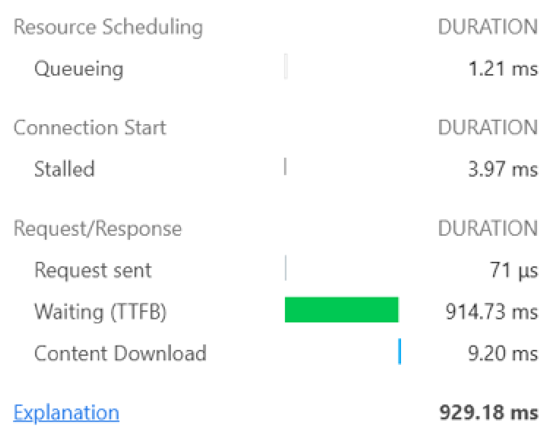
Figure 7.3: Results from network activity when performing a refresh of the Vugu TinyGo Components 10 000 example.

The complete referenced data is included [E] and can be imported and analysed using the same networking activity analysis tool [33] used in our example.

A far smaller WebAssembly file is transferred when Vugu is compiled with the TinyGo compiler. $557\text{kB} / 3.2\text{MB} \approx 0.17 \approx 1/6$
Decreasing the WebAssembly file size did not decrease the time needed to transfer the file, instead it became far more time consuming. This implies that there are other factors that affect the time it takes to receive the WebAssembly file, aside from downloading the file. The tool used to analyse the network activity also has features to subdivides time-consumption based on the status of the file transfer, on a file-by-file basis:



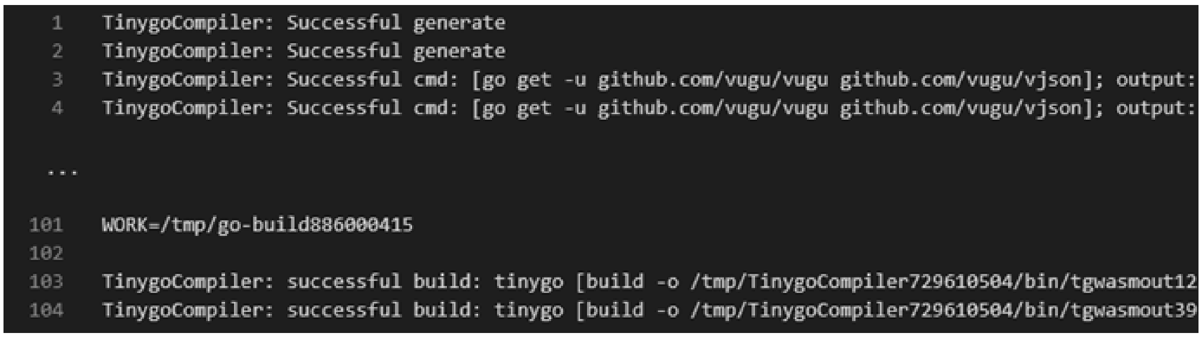
```
Resource Scheduling          DURATION
  Queueing                    1.41 ms
Connection Start             DURATION
  Stalled                     4.59 ms
Request/Response             DURATION
  Request sent                  70 µs
  Waiting (TTFB)              18.09 s
  Content Download            2.70 ms
Explanation                   18.10 s
```



```
Resource Scheduling          DURATION
  Queueing                    1.21 ms
Connection Start             DURATION
  Stalled                     3.97 ms
Request/Response             DURATION
  Request sent                  71 µs
  Waiting (TTFB)            914.73 ms
  Content Download            9.20 ms
Explanation                 929.18 ms
```

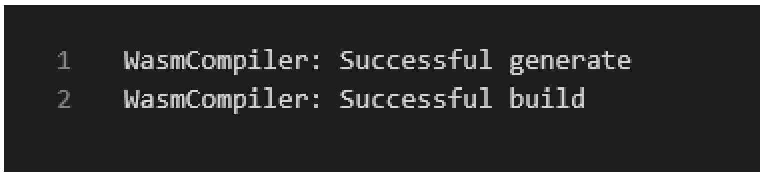
Figure 7.4: Comparison between compiled WebAssembly downloads, TinyGo(l.h.s) compared to WASM(r.h.s).

From this information, it is transparent that large WebAssembly file sizes are not the root issue of long load times in Vugu compiled with WASM and TinyGo compilers in our test environment. Instead, the request is delayed by the back-end server, signified by the request-wait-time.



```
TinygoCompiler: Successful generate
TinygoCompiler: Successful generate
TinygoCompiler: Successful cmd: [go get -u github.com/vugu/vugu github.com/vugu/vjson]; output:
TinygoCompiler: Successful cmd: [go get -u github.com/vugu/vugu github.com/vugu/vjson]; output:

...

WORK=/tmp/go-build886000415

TinygoCompiler: successful build: tinygo [build -o /tmp/TinygoCompiler729610504/bin/tgwasmout12
TinygoCompiler: successful build: tinygo [build -o /tmp/TinygoCompiler729610504/bin/tgwasmout39
```

Figure 7.5: TinyGo console log, when loading a web-page.



```
WasmCompiler: Successful generate
WasmCompiler: Successful build
```

Figure 7.6: WASM console log, when loading a web-page.

During the loading of the two examples, there were several console logs from the TinyGo compiler [F]. These console entries occurred during most of the web-page load time and are similar to the updates found on the server side, when loading WASM compiled Vugu projects. The main take-away from the entries are that the WASM and TinyGo compilers are called when the web-pages are requested. For servers in production this is inefficient resource utilization, however our testing environments are development environments, in these environments this feature may have other benefits.

From the console log information, there is a possibility that the long waiting times for a response from the back-end are related to the WebAssembly compilers. The previously mentioned feature Vugu has for development servers, that updates the front-end when changes occur on the back-end, would require a recompiled WebAssembly file. To properly find out what occurs on the server side while the client side waits to receive the WebAssembly file, more debugging information is needed from the Vugu projects. However, from the console logs that are presented during Vugu development, there are grounds to assume that a WebAssembly compilation could be occurring for every request. If this is the case, then the loading time may not be as large in Vugu projects that are in production, where recompiling files would likely not be done. Nevertheless, utilizing TinyGo to reduce the WebAssembly file-size imposes a very noticeable delay in loading times for Vugu development projects. A page load time that takes several seconds is sub-optimal for developers that want to test their changes.

7.3 Deploying WebAssembly for Production

For production environments, the Vugu documentation includes an example for how to build a WebAssembly file for distribution and an example of a bare-bone environment to serve a WebAssembly file [37]. The distribution code is built to compile Vugu using the WASM compiler, it depends on the Vugu “distutil” package [38]. Using this package is the suggested approach to build and distribute a Vugu application, yet it is not built with intended support for TinyGo compilation as there are no instructions for how the differing TinyGo dependencies should be implemented to build the WebAssembly file. Nor are there functions in the “distutil” package built to handle the TinyGo compiler. The following is the suggested code for building WebAssembly files:

```
// +build ignore

package main

import (
  "flag"
  "fmt"
```

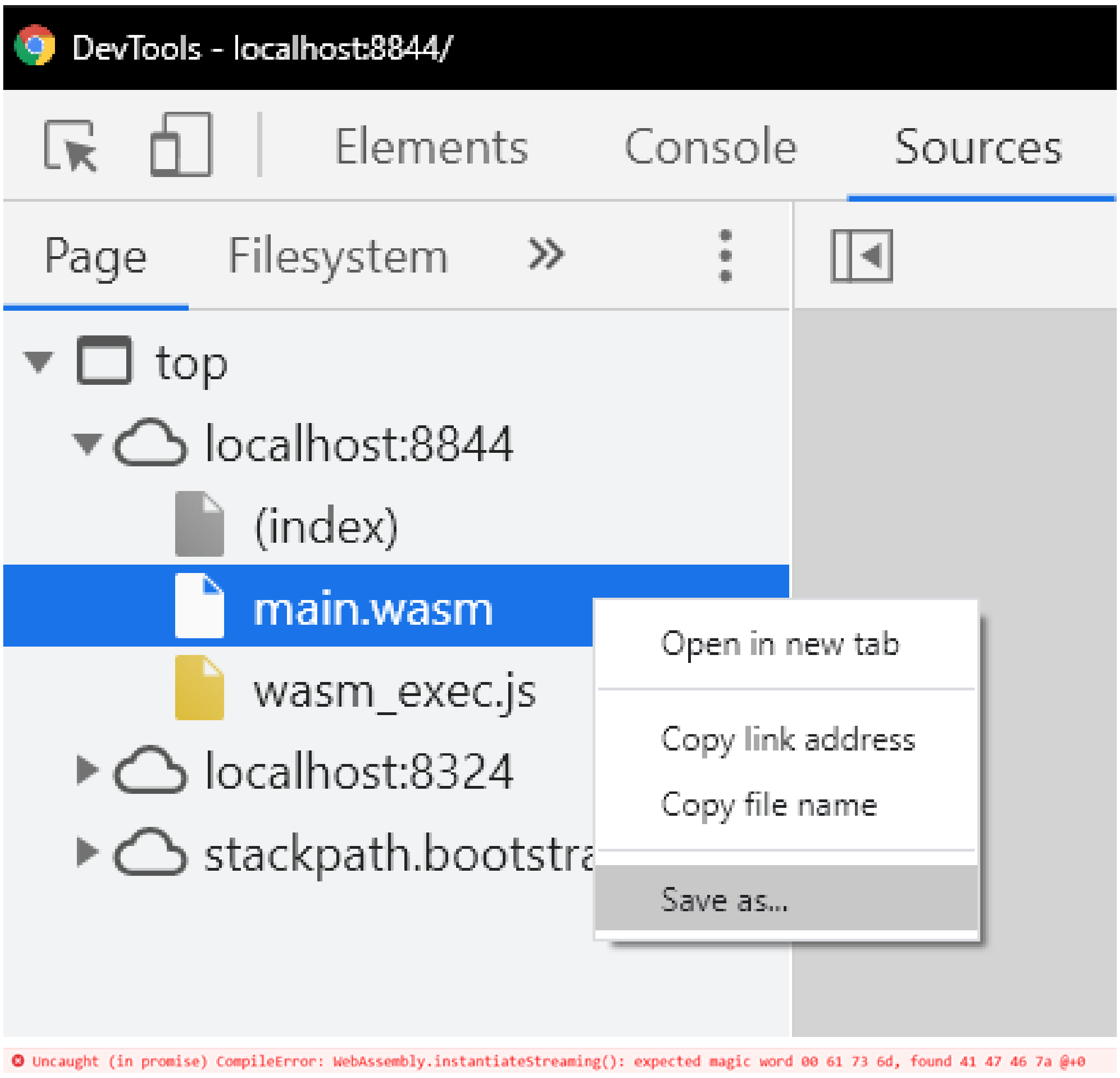
```go
  "log"
  "os"
  "os/exec"
  "path/filepath"
  "time"

  "github.com/vugu/vugu/distutil"
)

func main() {

  clean := flag.Bool("clean", false, "Remove dist dir before starting")
  dist := flag.String("dist", "dist", "Directory to put distribution
   files in")
  flag.Parse()

  start := time.Now()

  if *clean {
    os.RemoveAll(*dist)
  }

  os.MkdirAll(*dist, 0755) // create dist dir if not there

  // copy static files
  distutil.MustCopyDirFiltered(".", *dist, nil)

  // find and copy wasm_exec.js
  distutil.MustCopyFile(distutil.MustWasmExecJsPath(), filepath.Join(*
   dist, "wasm_exec.js"))

  // check for vugugen and go get if not there
  if _, err := exec.LookPath("vugugen"); err != nil {
    fmt.Print(distutil.MustExec("go", "get", "github.com/vugu/vugu/cmd/
   vugugen"))
  }

  // run go generate
  fmt.Print(distutil.MustExec("go", "generate", "."))

  // run go build for wasm binary
  fmt.Print(distutil.MustEnvExec([]string{"GOOS=js", "GOARCH=wasm"}, "
   go", "build", "-o", filepath.Join(*dist, "main.wasm"), "."))

  // STATIC INDEX FILE:
  // if you are hosting with a static file server or CDN, you can write
    out the default index.html from simplehttp
  // req, _ := http.NewRequest("GET", "/index.html", nil)
  // outf, err := os.OpenFile(filepath.Join(*dist, "index.html"), os.
   O_CREATE|os.O_TRUNC|os.O_WRONLY, 0644)
  // distutil.Must(err)
  // defer outf.Close()
  // template.Must(template.New("_page_").Parse(simplehttp.
   DefaultPageTemplateSource)).Execute(outf, map[string]interface{}{"
   Request": req})

  // BUILD GO SERVER:
  // or if you are deploying a Go server (yay!) you can build that
   binary here
  // fmt.Print(distutil.MustExec("go", "build", "-o", filepath.Join(*
   dist, "server"), "."))

  log.Printf("dist.go complete in %v", time.Since(start))
}
```

Listing 7.5: “dist.go” the suggested method for creating and building WebAssembly and other requisites for Vugu in production.

The Vugu documentation notes that the distribution code does the following [37]:

- Creates a new folder and copies the static files into the folder.
- Copies the WebAssembly dependent file wasm_exec.js to the folder.
- Ensures the tool vugugen is installed.
- Runs go generate.
- Builds the WebAssembly file.
- Optionally writes out a template file.
- Optionally builds a server executable.

This is a good tool for building and retrieving the required files for Vugu in production, it ensures the developer does not need knowledge of how compile a file using WebAssembly, meaning that the required knowledge to put a Vugu project into production is low. However as there is no added support for doing this with TinyGo, projects that want to take use of the advantages the TinyGo compiler may have over the WASM compiler are out of luck. Our attempt at a work-around to be able to serve TinyGo compiled Vugu was to extract the file downloaded by the browser from the Vugu development environment:



```
Uncaught (in promise) CompileError: WebAssembly.instantiateStreaming(): expected magic word 00 61 73 6d, found 41 47 46 7a @+0
```

Figure 7.7: Attempted method of extracting the TinyGo Compiled WebAssembly file. Serving this file in place of "main.wasm" ended up not working, throwing the error underneath.

Another attempt was focused around modifying the suggested code for building the WebAssembly file, but this attempts also ended up in failure as well, as we were unable to supply the dependencies the TinyGo compiler had properly. In this regard, others may be successful where we failed.

To serve a built Vugu project, the Vugu documentation includes bare-bone code that needs to be modified to fit the requirements of the project. The following code can be found in the Vugu documentation, and serves as a template for serving the required files:

```go
// +build !wasm

package main

import (
  "flag"
  "log"
  "net/http"
  "os"
  "path/filepath"

  "github.com/vugu/vugu/simplehttp"
)

func main() {
  dev := flag.Bool("dev", false, "Enable development features")
  dir := flag.String("dir", ".", "Project directory")
  httpl := flag.String("http", "127.0.0.1:8877", "Listen for HTTP on
   this host:port")
  flag.Parse()
  wd, _ := filepath.Abs(*dir)
  os.Chdir(wd)
  log.Printf("Starting HTTP Server at %q", *httpl)
  h := simplehttp.New(wd, *dev)
  log.Fatal(http.ListenAndServe(*httpl, h))
}
```

Listing 7.6: “server.go” the suggested method for serving Vugu in production. This does not serve Vugu files properly without extra configuration.

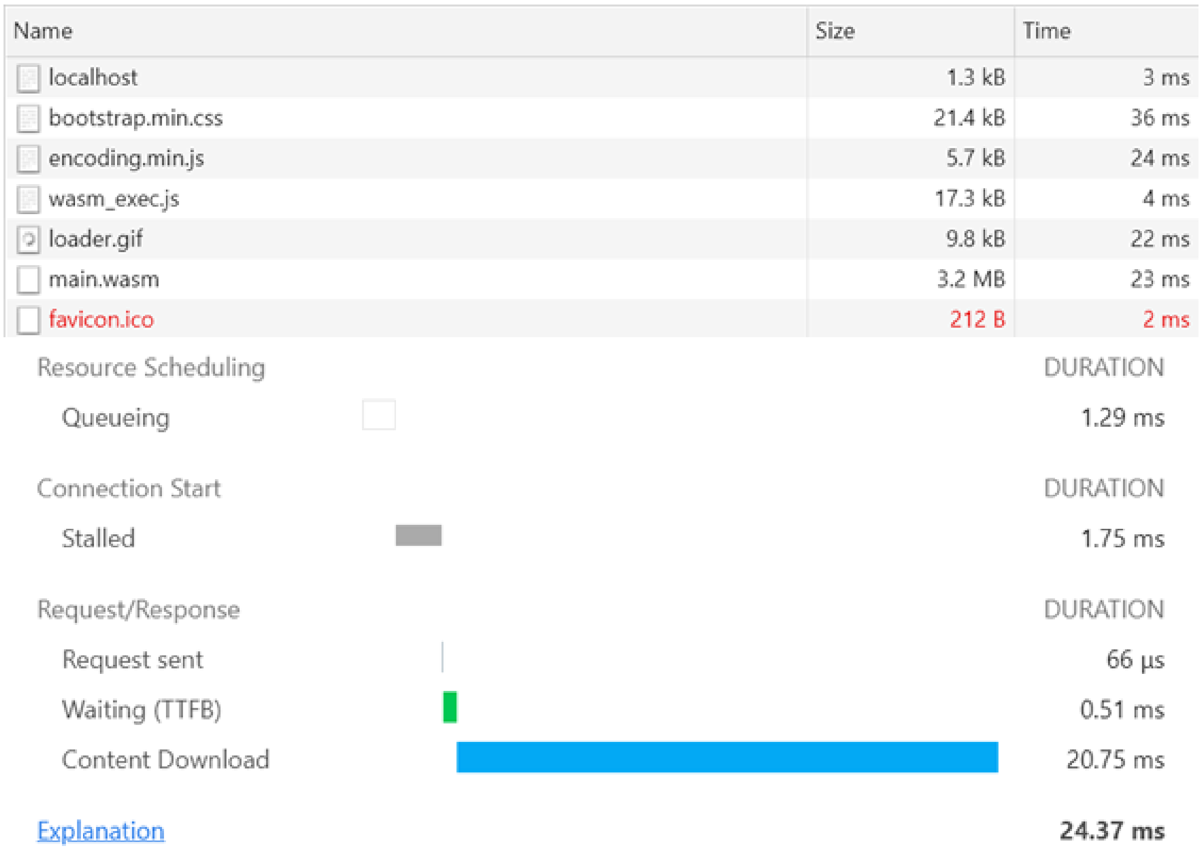
This code uses the package simplehttp, a Vugu created package with functions to serve different types of files. The documentation for how one uses this package is lacking, instead the developer is required read the code found in the Vugu GitHub repository [40]. This affects the developers efficiency, as more effort must be put into understanding how one should implement the package properly. Nevertheless, it is a nice addition that assists developers and removes a lot of the focus from developing back-end code, when a developer wants to use Vugu in their web-development.

In our comparison of the load time between Vugu using the WASM compiler in a production environment and Vue, we modified the following bits of code to serve the built WebAssembly file:

```go
// Modified from Vugu's documentation
content, err := ioutil.ReadFile(wd + "/index.html")
if err != nil {
  log.Fatal(err)
}
h.PageHandler = &simplehttp.PageHandler{
  Template:         template.Must(template.New("_page_").Parse(string
  (content))),
  TemplateDataFunc: simplehttp.DefaultTemplateDataFunc,
}
// --------------------------------
```

Listing 7.7: In our “server.go” the following was added between Line 23. and 24. in the initial "server.go".

As can be seen, our example required us to modify the simplehttp object to serve the correct HTML document. This was required to link external dependencies like CSS (Cascading-Style-Sheet). Using the same methodology previously used to capture load times, the Google Chrome network activity analysis tool [34], we got the following results [G]:

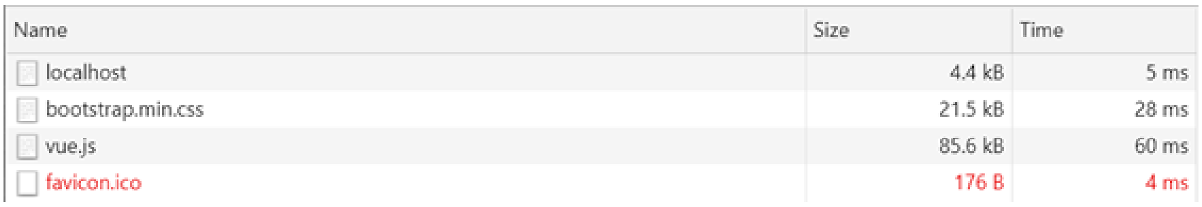


Name	Size	Time
localhost	1.3 kB	3 ms
bootstrap.min.css	21.4 kB	36 ms
encoding.min.js	5.7 kB	24 ms
wasm_exec.js	17.3 kB	4 ms
loader.gif	9.8 kB	22 ms
main.wasm	3.2 MB	23 ms
favicon.ico	212 B	2 ms

Resource Scheduling	DURATION
Queueing	1.29 ms
Connection Start	**DURATION**
Stalled	1.75 ms
Request/Response	**DURATION**
Request sent	66 µs
Waiting (TTFB)	0.51 ms
Content Download	20.75 ms
Explanation	**24.37 ms**

Figure 7.8: Results from network activity analysis of page-load request from 10 000 Components in production with Vugu. Summary above, and sub-divided time-consumption below.

The waiting time that previously caused large load times is now reduced below 1ms, significantly faster than previously captured request-wait-time for Vugu examples. The WebAssembly file sizes are still problematic, deploying a server Vugu project does not fix this issue. As our testing environments retrieve files from the same computer as they are served (without competing requests from other clients or bottlenecks in the network connectivity between the server and client), the speed at which the WebAssembly file is downloaded may be misleading. Opensignals recorded average cellular download speed for Norway was in 2020, 47.5 Mbps [44]. Downloading a 3.2 MB file would take:
3.2MB $/$ 47.5Mbps $= (3.2$MB $*$ 8b/B$)$ $/$ 47.5Mbps ≈ 0.54s
For the average user, waiting 0.54s to interact with the web-page would affect user experience. For slower download speeds it may be a large inconvenience. 0.54s would also most likely affect Google search query indexing [31], which in turn would affect the number of interactions the site sees.



Name	Size	Time
localhost	4.4 kB	5 ms
bootstrap.min.css	21.5 kB	28 ms
vue.js	85.6 kB	60 ms
favicon.ico	176 B	4 ms

Figure 7.9: Results from network activity when performing a refresh of the Vue Components 10 000 example.

Compared to the results of the same example written in Vue, the file size of the main Vue file “vue.js” is significantly smaller (3.2 MB is a large file when compared to Vues 85.6 kB), meaning that in terms of transfer speed, there are advantages when using Vue.

Note, the request-wait-time that caused problems for non-deployed Vugu examples is not compared to the request-wait-time of the “vue.js”-file as it is irrelevant, because the file is served from a third party and can therefore only be improved by sending the file from the server instead of the third party. This does not refute the fact that switching server may affect load times. However, it is not tested as the most optimal solution for Vugu projects; deploying a Vugu project to decrease request-wait-time and compiling with TinyGo to decrease file size, is expected to potentially still result in larger load times than the experienced load times found in our Vue testing example. As we were unable to combine the effects of deploying a Vugu project to decrease request-wait-time and compiling with TinyGo to decrease file size, this claim is uncertain, and would require future work to verify the integrity of the claim.

Chapter 8

Conclusion

In this thesis we presented the two front-end frameworks Vue and Vugu. This chapter will give a conclusion to the task we formulated at the introduction of this thesis. How does the frameworks differ in terms of usability (end-user experience and developer usability), and performance.

8.1 Thesis Conclusion

From our experience with the frameworks, the biggest difference between them is the developer usability. Vugu is experimental technology, because of this problems are often harder to overcome, as there is a lack of available resources to help troubleshooting. This ends up affecting the developer usability, as more time is consumed troubleshooting problems in Vugu, when solutions for the same problem are available in Vue.

The development environments provided by the Vugu "vgrun" tool works well. It is made similar to the project structure used by the "Vue CLI" tool, where components are grouped by file to make the project structure scale-able and organize-able. Vue is not bound to the project structure found in "Vue CLI" and can be reduced to one JavaScript file, while Vugu is compiled to WebAssembly and therefore has requirements that its development environment support its compilation. This versatility found in Vue is an advantage in terms of developer usability, as it is not tied down by large requirements.

There advantages to Vugu, as its written in Go, a language suitable for back-end development, the Dual-Stack feature becomes a viable option to re-use code in both the front and back-end. This has its advantages for developers usability. The Go packages are also useable in the front-end, ensuring support for matters like asynchronous HTTP calls and console logging.

In terms of performance, Vugu has an advantage when tasked with computational heavy tasks, because of WebAssembly's performance advantages over JavaScript. Though, WebAssembly has big disadvantages when communicating with the DOM. In edge-cases, large amounts DOM updates therefore impose a large overhead. With Mozilla's future plans to allow WebAssembly to talk directly to the DOM API this may yield different results, but currently this means that Vue is able to perform better when tasked with large DOM updates. Vugu also has problems with performing well when making small DOM updates, when the amount of components become substantial, as reactivity is ensured by a modification tracker that does not scale too well. Custom modification tracking is advised for these scenarios, but we were unable to verify if this improves the performance.

One of the bigger problems Vugu faces is WebAssembly's large file sizes, these may affect the end-user experience for users with poor internet connectivity. The Vugu documentation refers to the experimental alternate WebAssembly compiler TinyGo as a potential solution. However this comes at the cost of the developer usability, with large load-times for loading web-pages under development, and unclear guidelines for distributing the project using TinyGo.

From a case study, our opinion is that Vugu has potential to become viable for developers and end-users, but the gap between the project and well-established front-end frameworks like Vue is still large, with several obstacles in the way.

8.2 Future Work

While our thesis deadline is over, there are still things that could have been done to improve the quality of our thesis. These things will most likely not affect the overall observations we have come to in our conclusion, but could help improve the thesis with a more thorough comparison of Vue and Vugu.

Evaluating Scalability of the frameworks was not done, as our projects have avoided larger projects in the frameworks. This was not compared in our thesis, as our focus has been on evaluating the frameworks through smaller mechanism-focused examples. Its importance is still important and should be evaluated to get a better understanding for how suitable Vue and Vugu are in the front-end development.

Front-end Routing is one of Vue's more important features, it is used by larger applications and its purpose is to alleviate the routing from the back-end by doing it in the front-end. The reason it is more suitable for larger applications is that it is created with focus on multi-page web-sites. We did not prioritize comparing this feature between Vue and Vugu, as our focus was on smaller single-paged projects. Its importance is still relevant for the evaluation of the frameworks, and should be done for a more thorough comparison of the frameworks.

Evaluating missing Vugu features should be more focused. While Vugu covers the more prominent features of modern front-end frameworks, it is still experimental and may therefore miss support for some features that are covered by more developed front-end framework like Vue. While this is niche, it may affect developers decisions on deciding the most suitable framework for their need.

Implementing custom modification tracking did not work in our attempts. As this feature may affect the results we found when comparing DOM updates, it could affect the conclusions we came to in our thesis. A successful implementation of custom modification tracking in the 10 000 Components example should reduce the checked components by returning only modified components to the modification tracker. The effects of a successful implementation of this modification tracker should afterwards be compared to an unimplemented version in Vugu and its equivalent implementation in Vue.

Distributing TinyGo to production did not yield success in our attempts. Its development counterpart showed promise in the served file-size, but used too large load-times to be useful in a development environment. As distributing WASM to production reduced the load-times needed by the compiler, this option could be viable for production environments using Vugu. With success, this should be evaluated against Vue's load times.

8.3 Final Comments

The attached datasheets used by projects and examples are included alongside these projects and examples in our publicly view-able GitHub repository, available to fork, found at:
`https://github.com/ferdinandwegerif/bachelor`
This GitHub repository will not be changed after our thesis deadline, 15.05.2021, with the exception for potentially changing repository README.md to clarify repository structure for future employers.

References

[1] By Ben James, Published 04.04.2019
`https://hackaday.com/2019/04/04/webassembly-what-is-it-and-why-should-you-care/`,
Retrieved 06.01.21.

[2] By Mozilla.org, WebAssembly Concepts
`https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts`,
Retrieved 06.01.21.

[3] By Liron Navon, Published 01.06.2019
`https://codesight.medium.com/10-frontend-frameworks-that-are-written-with-backend-languages-f73fa29fcf78`,
Retrieved 06.01.21.

[4] By Brad Peabody at GopherCon 2020, Published 22.12.2020
`https://www.youtube.com/watch?v=JIsUb1t6un0`,
Retrieved 13.01.21.

[5] By Wikipedia, Document Object Model
`https://en.wikipedia.org/wiki/Document_Object_Model`,
Retrieved 09.02.21.

[6] By VueJS.org, Vue Components Documentation
`https://v1.vuejs.org/guide/components.html`,
Retrieved 09.02.21.

[7] By Vugu.org, Vugu Components Documentation
`https://www.vugu.org/doc/components`,
Retrieved 09.02.21.

[8] By Kayce Basques, Google Runtime Analytics tutorial, Published 06.04.2017
`https://developers.google.com/web/tools/chrome-devtools/evaluate-performance`,
Retrieved 15.02.21.

[9] Asked by try-catch-finally, edited by msanford, answered by Tim Arney, Published 09.01.2016
`https://stackoverflow.com/questions/34698433/what-is-involved-in-chromes-recalculate-style-event`,
Retrieved 15.02.21.

[10] By Wikipedia, Tower Of Hanoi
`https://en.wikipedia.org/wiki/Tower_of_Hanoi`,
Retrieved 18.03.21.

[11] By Kayce Basques, Google Runtime Analytics tutorial, Published 06.04.2017
`https://developers.google.com/web/tools/chrome-devtools/rendering-tools`,
Retrieved 18.03.21.

[12] By Wikipedia, Document Object Model
`https://en.wikipedia.org/wiki/Document_Object_Model`,
Retrieved 18.03.21.

[13] By Vugu.org, Vugu Documentation
`https://www.vugu.org/doc`,
Retrieved 18.01.2021.

[14] By Brad Peabody at GopherCon 2020, Published 22.12.2020
`https://www.youtube.com/watch?v=JIsUb1t6un0&t=2312s`,
Retrieved 20.01.2021.

[15] By Mahdhi Rezvi, Published 28.09.2020
`https://blog.bitsrc.io/a-complete-introduction-to-webassembly-and-its-javascript-api-3474a9845206`,
Retrieved 20.01.2021.

[16] By Dwayne Charrington, Published 09.09.2020
`https://ilikekillnerds.com/2020/09/will-webassembly-replace-javascript/`,
Retrieved 20.01.2021.

[17] By Mozilla.org, WebAssembly API Goals
`https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts#porting_from_cc`,
Retrieved 20.01.2021.

[18] Blog post by Brad Peabody, Published 25.12.2019
`https://blog.gopheracademy.com/advent-2019/writing-go-user-interfaces-with-vugu/`,
Retrieved 18.01.2021.

[19] By vugu.org, Vugu Main Page
`https://www.vugu.org/`,
Retrieved 20.01.2021.

[20] By Mozilla.org, WebAssembly use
`https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts#how_does_webassembly_fit_into_the_web_platform`,
Retrieved 20.01.2021.

[21] Oxford Dictionaries resources, Oxford Learner's Dictionaries
`https://www.oxfordlearnersdictionaries.com/definition/american_english/progressive_1`,
Retrieved 20.01.2021.

[22] By VueJS.org, VueJS Introduction
`https://vuejs.org/v2/guide/index.html#What-is-Vue-js`,
Retrieved 20.01.2021.

[23] By VueJS.org, VueJS Team
`https://vuejs.org/v2/guide/team.html`,
Retrieved 20.01.2021.

[24] By VueJS.org, VueJS Introduction
`https://vuejs.org/v2/guide/#`,
Retrieved 20.01.2021.

[25] By VueJS.org, VueJS Reactivity
`https://vuejs.org/v2/guide/reactivity.html`,
Retrieved 20.01.2021.

[26] By Mozilla.org, JavaScript Define Property use
`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty`,
Retrieved 22.01.2021.

[27] By vuemastery.com, in-depth reactivity concept
`https://www.vuemastery.com/courses/advanced-components/build-a-reactivity-system`,
Retrieved 20.01.2021.

[28] By Lin Clark, Mozilla.org post about WebAssembly API interactions, Published 21.08.2019
`https://hacks.mozilla.org/2019/08/webassembly-interface-types/`,
Retrieved 19.03.2021.

[29] Source Code of JavaScript DOM Renderer for Vugu, from the official Vugu repository.
`https://github.com/vugu/vugu/blob/272a0542a687dbd0bc3cab6a626f1a0929d4f439/domrender/renderer-js-script.js#L156`,
Retrieved 06.04.2021.

[30] By Brad Peabody, GoLab conference, Published 15.12.2020
`https://www.youtube.com/watch?v=6vAE7XGiD7E`,
Retrieved 15.03.2021.

[31] By Addy Osmani and Ilya Grigorik, Updated 23.09.2019
`https://developers.google.com/web/updates/2018/07/search-ads-speed`,
Retrieved 08.04.2021.

[32] By vugu.org, Vugu Tiny Go Documentation
https://www.vugu.org/doc/tinygo,
Retrieved 08.04.2021.

[33] By Kayce Basques, Published 08.02.2019
https://developer.chrome.com/docs/devtools/network/,
Retrieved 21.04.2021.

[34] Asked by Awais Qarni, edited by Peter Mortensen and Freek de Bruijn, and answered by Pavel Podlipensky, Published 21.12.2011
https://stackoverflow.com/questions/8589760/difference-between-f5-ctrl-f5-and-click-on-refresh-button,
Retrieved 21.04.2021.

[35] Vugu Repository File, Updated 21.06.2020
https://github.com/vugu-examples/tinygo/blob/master/devserver.go,
Retrieved 21.04.2021.

[36] Mozilla DOM Microtasks documentation, Updated 29.12.2020
https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide,
Retrieved 21.04.2021.

[37] By vugu.org, Vugu Distribution and Serving Documentation
https://www.vugu.org/doc/build-and-dist,
Retrieved 26.04.2021.

[38] Vugu distutil Documentation, Published 21.04.2021
https://pkg.go.dev/github.com/vugu/vugu/distutil,
Retrieved 26.04.2021.

[39] Vugu “simplehttp” Package Source Code, Published 21.04.2021
https://github.com/vugu/vugu/blob/24d8d4332371618c6f2b8881ae9f8d52fab8e3ca/simplehttp/simple-handler.go#L427,
Retrieved 26.04.2021.

[40] Ajax: A New Approach to Web Applications, By Jesse James Garrett, Published 18.02.2005
https://web.archive.org/web/20150910072359/http:/adaptivepath.org/ideas/ajax-new-approach-web-applications/,
Retrieved 26.04.2021.

[41] XMLHttpRequest Documentation, By Mozilla, Last Modified 27.04.2021
https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest,
Retrieved 26.04.2021.

[42] Top JavaScript Libraries for Making AJAX Calls, By Manjunath M, Published 26.01.2018
https://dzone.com/articles/top-javascript-libraries-for-making-ajax-calls,
Retrieved 26.04.2021.

[43] Promise, By Mozilla, Last Modified 27.04.2021
`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`,
Retrieved 28.04.2021.

[44] Wikipedia Article, List of countries by Internet connection speeds
`https://en.wikipedia.org/wiki/List_of_countries_by_Internet_connection_speeds`,
Retrieved 28.04.2021.

[45] Vugu Dom Events Documentation
`https://www.vugu.org/doc/dom-events`,
Retrieved 28.04.2021.

[46] Go Sync Package Documentation, Read/Write Mutual Exclusion Lock
`https://golang.org/pkg/sync/#RWMutex`,
Retrieved 28.04.2021.

[47] Vugu Getting Started Documentation
`https://www.vugu.org/doc/start`,
Retrieved 29.04.2021.

[48] Google Network Analysis Development Tool referencesheet
`https://developer.chrome.com/docs/devtools/network/reference/#timing-explanation`,
Retrieved 30.04.2021.

[49] Node Package Manager
`https://www.npmjs.com/`,
Retrieved 30.04.2021.

[50] Vue CLI create project template
`https://cli.vuejs.org/guide/creating-a-project.html#vue-create`,
Retrieved 30.04.2021.

[51] JavaScript's use in the front-end
`https://w3techs.com/technologies/details/cp-javascript`,
Retrieved 1.05.2021.

[52] What is REST API, by Alexander S. Gillis, Last updated 20.09.2020
`https://searchapparchitecture.techtarget.com/definition/RESTful-API`,
Retrieved 1.05.2021.

[53] Go JSON package Documentation
`https://golang.org/pkg/encoding/json/`,
Retrieved 01.05.2021.

[54] Information about Node.js, from Node.js
`https://nodejs.org/en/about/`,
Retrieved 01.05.2021.

[55] Vue forum question about the difference between .vue files and vue.js, by wysiwyg, answered by woodberry, posted and answered –.10.2017
https://forum.vuejs.org/t/what-is-the-difference-between-vue-component-and-a-vue-file/19374,
Retrieved 08.05.2021.

[56] Article about the limitations of Vue reactivity, from medium.com, by Milad Meidanshahi, published 09.06.2019
https://medium.com/@miladmeidanshahi/update-array-and-object-in-vuejs-a283983fe5ba,
Retrieved 10.05.2021.

[57] Virtual DOM and *diff algorithm*, Posted by Claire on 24.08.2019
https://programmer.ink/think/virtual-dom-and-diff-algorithm-in-vue.html,
Retrieved 11.05.2021

[58] Building User Interfaces Using Virtual DOM [Page 27-28], By Marianne Grov, Spring 2015
https://www.duo.uio.no/bitstream/handle/10852/45209/7/mymaster.pdf,
Retrieved 11.05.2021.

[59] ChangeCounter, Modification Tracking Example, Vugu GitHub repository, By Bradley Peabody, 07.09.2019
https://github.com/vugu/vugu/blob/v0.3.4/change-counter.go#L16,
Retrieved 14.05.2021.

[60] Vugu ModChecker Documentation, Published 01.04.2021
https://pkg.go.dev/github.com/vugu/vugu#ModChecker,
Retrieved 14.05.2021.

[61] Vue's component life cycle diagram, https://vuejs.org/v2/guide/instance.html,
Retrieved 14.05.2021

[62] Design and Implementation of Reusable Component for Vue.js [Page 5], Published 01.03.2021
https://www.theseus.fi/bitstream/handle/10024/496696/Tikhonova_Anastasiia.pdf?sequence=2&isAllowed=y,
Retrieved 14.05.2021.

[63] Bringing the Web up to Speed with WebAssembly, Published June 2017
https://dl.acm.org/doi/pdf/10.1145/3062341.3062363,
Retrieved 14.05.2021

[64] DOM Documentation by Mozilla, Last modified 09.04.2021
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction,
Retrieved 14.05.2021

[65] Vue.js GitHub Repository
https://github.com/vuejs/vue

[66] MDN Web Docs, Vue Resources
`https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Vue_resources`,
Retrieved 15.05.2021

[A] Attached Datasheets: TowerOfHanoiVugu & TowerOfHanoiVue

[B] Attached Datasheets: Components10000CreatedVugu & Components10000CreatedVue

[C] Attached Datasheet: Components10000CreatedVuguManual

[D] Attached Datasheets:
Components10000PageLoadNetworkActivityVugu & Components10000PageLoadNetworkActivityVue

[E] Attached Datasheet: Components10000PageLoadNetworkActivityVuguTinyGo

[F] Attached Datasheets: Components10000TinyGoConsole & Components10000WASMConsole

[G] Attached Datasheet: Components10000PageLoadNetworkActivityVuguWASMCompiledProduction

[H] Attached Datasheet:
Components10000IncrementOneVugu & Components10000IncrementOneVue