

**Classical, Current, and The Future of Software Development Technologies when it Comes to Delivering the Best Development and Browsing Experience, and Overall Performance.**

**Ayoub Maimmadi**

**Id: 76678**



**CSC 4303: Applied research**

**Supervised by: Dr. Driss Kettani, Professor of Computer Science at AUI**

## Topic Outline

<b>1- Abstract .....</b>	<b>3</b>
<b>2- Introduction .....</b>	<b>3</b>
<b>3- History of software development technologies .....</b>	<b>4</b>
<b>3-1- First era .....</b>	<b>4</b>
<b>3-2- Current era .....</b>	<b>10</b>
<b>4- Compare and contrast for two technologies.....</b>	<b>15</b>
<b>3-1 React with TypeScript .....</b>	<b>16</b>
<b>3-2 Web Assembly with Rust .....</b>	<b>16</b>
<b>3-3 Comparison and Results .....</b>	<b>17</b>
<b>5- Future of web development technologies .....</b>	<b>23</b>
<b>5-1 Development experience &amp; tools .....</b>	<b>23</b>
<b>6- Conclusion .....</b>	<b>24</b>
<b>7- References .....</b>	<b>26</b>

## **1- Abstract**

Young software engineers and developers can use a detailed history of the evolution of software development technologies to fill the gap of understanding how these tools came to be. Along with a performance investigation between two modern rival frameworks in order for computer science students to make up their own minds on what kind of technologies they wish to adopt in their software creation journey. Testing and comparing these frameworks will be achieved by building two different applications with different tools, these apps scale based on their link in order to test how much workload based on a URL a server can handle. The comparison will cover the difference of performance via server-side rendering measures and the number of requests per second achieved with different parallel connections on each of these frameworks. The aim is point out which technology is over all better performant, and to offer all the necessary information needed for newly graduated software engineers to gather as much knowledge as possible for their choice making process when it comes to tools of software development and programing language.

## **2- Introduction**

Web development technologies are the backbone of our day-to-day online experience. They have been through major evolutions from the day of this field's inception. Nowadays, there are too many tools young engineers can pick from. This paper will outline in detail the history of these technologies, so that developers can have a good understanding and expand their knowledge on how their favorite tools and frameworks came to be, and the transformation they had experience

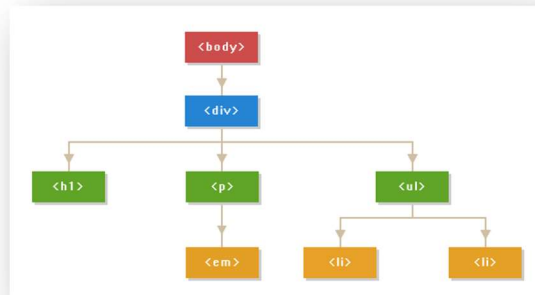
to become such a valuable technology to for them to easily use. As well as filling the gap of knowledge young software engineers and developers have when it comes to the patterns of choices made by mother companies in order to ensure the best results when it comes to their products quality, plus the survivability and dominance of their technology. This will be achieved by taking a history ride on some of the most important and revolutionary decisions made for these frameworks that forever impacted this field for the better. Most importantly, there will be a full-scale comparison between one of the currently most dominant technologies in this field when it comes to developing small and massive scale web applications by the name of React using TypeScript as the choice of programing language, against a newly adopted way of creating software and web applications with Web Assembly using Rust as choice of the programing language.

### **3- History of software development technologies**

#### **3-1 First era**

At the beginning of software creation in the early 2000's, developers have struggled to have a reliable and simple way of writing code. These times gave the birth to what is called today classical frameworks for building reliable software such as jQuery and Vanilla JavaScript. In these days, most other frameworks were at their early stages, however, jQuery has dominated the market share when it comes delivering software products and services. This reaction has happened because John Resig, a brilliant web developer and the creator or jQuery has embraced web applications for what they truly were. A tree of Document Object Model or otherwise known as DOM nodes. This has made the framework the world's most popular choice between the communities of developers and software engineers all throughout the world. Statistics shows that it is found on more than 70% of the world's top million websites, which goes to show how

heavily used and loved this technology was. The idea behind jQuery, and what John Resig was trying to achieve when he first announced the launch of jQuery back in 2006, is that JavaScript - the language used 90% of the time for building software- was very difficult and immature at the time. Therefore, he wanted people to write less of it and at the same time, achieve a matching objective or goal. Furthermore, browsers by this time had many interop issues, and jQuery provided a strong and single API to handle the edge cases across multiple browsers. That where his genius shows, by embracing websites for what the truly were, jQuery allowed the state of developer's software applications to live inside of the DOM. Whenever they wanted to update that state, they would imperatively traverse the DOM, locate the target node they want to update, and then simply update it.



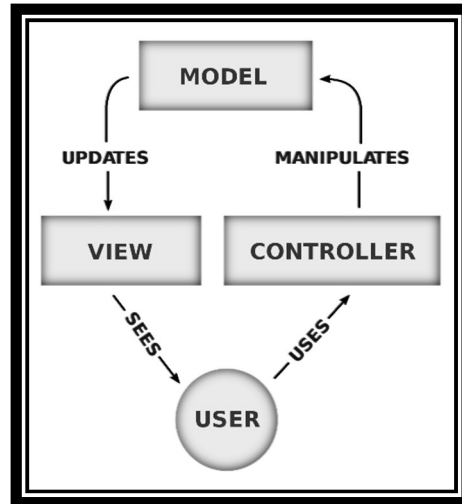
This idea has revolutionized developing software applications for the web via creating a unified, and most importantly, a simple abstraction over manipulating the DOM. Moreover, it proved to work in all existing browsers at the time. This was a huge advantage given the fact that in early stages of software development, engineers had to write different programs for different existing browsers. The cross-browser issues were incredibly aggravating for engineers. To constantly develop they applications in Firefox and have it only work there, or deal with the different code

changes from one browser to another, with different upgrades and patches was time consuming and provided a negative development experience.

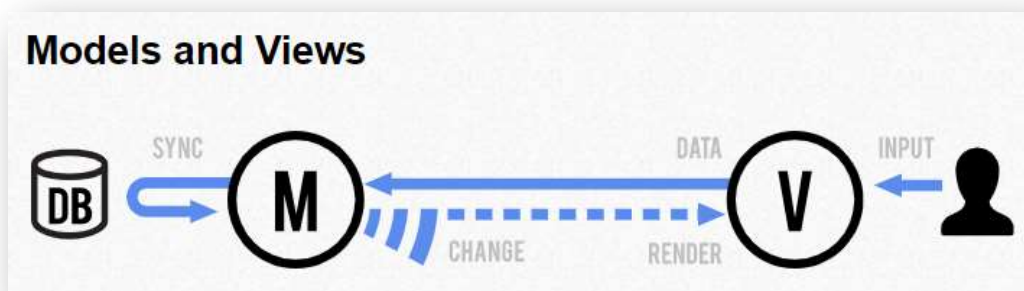
Software engineers wanted a new technology that was simple, advanced, and most importantly easy to code, and they got it via DOM traversing with jQuery. However, this framework's greatest blessing, was also its greatest curse. Developers soon observed that relying on shared mutable state was in fact a flawed idea. In simple terms, there was a risk causing a component to not properly work as intended to do when its lifetime overlaps with another component if these two components exchange the same data such as objects, or variables. What started as a simple mean to update the DOM state, evolved into a mess of multiple mutations crossing each other, that were almost impossible to keep track of or predict in huge software productions if engineers were not super careful and understand exactly what they are doing.

What developers and software engineers needed was a new technology or a tool that adopted an increasingly more structured theme. This is where backbone.js came to play. A JavaScript framework with a Representational state transfer (REST) JSON interface that is based on the model-view-controller application architecture paradigm. Overall, development with jQuery was still the norm, however, it was more and more considerably hard to adapt by people new to the field. Until 2010, when what was then called an amazing technology by the name of backbone.js was introduced to this market. To explain more why this new framework was revolutionary for its time, we need to dive deeper on the development experience of software engineers. People by then had to develop their applications and software through a very imperative application way. They had to explicitly instruct the program of the ideas they had in mind and wanted to achieve. This might easier explained with an example, and the most obvious

one when it comes to web applications is changing the content of a web page. This means that every time a change needed to happen to a page component, programs had to first find that specific content on that specific page, and then set the content to the new step. If developers wanted to create an event handler, they would have to look for and find that specific button container, and child of button, and then when there is a click event, run that handler. Plus, some of the key words used in jQuery were very much an abstraction on the built-in browser. Adding to that, the process of writing separate files for the logic and HTML components was redundant and unnecessary. Engineers complained about the over complicated method they had no choice but use, a method where JavaScript files had to reach out to HTML files, find the target elements, attach a listener to it, and when this listener is triggered, use a specific behavior. This brings us again to the newly released framework by this time, backbone.js. It was in fact the first framework to ever introduce structure to web applications for software engineers who love using JavaScript as their development language, which is the most famous and used language when it comes to software creation from the day of its birth. This framework was truly revolutionary because of this new change it has provided for building web applications, and it did it with an innovative piece of technology called the model view controller pattern. Of course, this pattern was not a new invention. It was in fact around for a very long time for desktop graphical user interfaces or commonly known as GUIs. However, no other web framework for developing modern and advanced software applications has ever introduced such a new revolutionary idea, until backbone.js. All developers had before this time was imperative coding, and jQuery, which both proved to be inferior when it comes to massive scale web applications and software services.



The idea behind the model view controller or MVC is the pattern of a model which holds the data state. As soon as an update is detected in this model, there would be a view of how the user interface would look like. The user can interact with this view on which it would travel to the controller so as to handle user data interactions, then update the model, and again and again, keep doing this loop. That's what this revolutionary framework brought to the table. It brought the entire idea of model view controller to the browser directly, which was not only the opening of a completely new way of software engineering and development, but also a groundbreaking move at the time.





This is a much clearer example of the idea that backbone.js introduced to web development. We have an input from the user, the view which is some kind of user interface that travels to the model, which then goes to the database of choice to sync the model and its change event that then updates the view which will be displayed to the user so that they can interact with, and the circle continues.

With jQuery, all the data was in the document object modifier (DOM), attached with the DOM nodes that they belonged to, so it was most common with this outdated by then outdated framework to get a reference to a DOM node and just link arbitrary data to it. For example, if an engineer intended to change the display color of a sidebar, they would have to save it as a data element on an element. Meaning, every time they want to check the state of the sidebar, they would have to check the DOM and go back from there. Which proved to be super slow and expensive to implement in massive productions. Whereas with this new technology of backbone.js, engineers would only have to keep all that state in the script, so that when an interaction is taking place with the user interface, it goes to the model which is the knowledge source. This source which backbone.js took from the DOM into the JavaScript script. Which not only was revolutionary, but it also inspired modern technologies of software engineering and development that massive companies released and uses on their own productions such as Google with Angular, and Facebook with React. The fact that backbone.js is still used until today, a near 12 years lifespan, is a testament to how respected this frameworks impact in the communities of software engineering and development. A concept that only used a couple thousand lines of code and gave power of decoupling application states from the DOM to engineers. Instead of living in the document object modifier (DOM), the state lived inside of its models. Making detected changes in the model trigger a re-render to the views that are linked to that model state. The

whole philosophy behind this pattern was to be super minimal, and as much unopinionated as it can be.

### **3-2 Current Era**

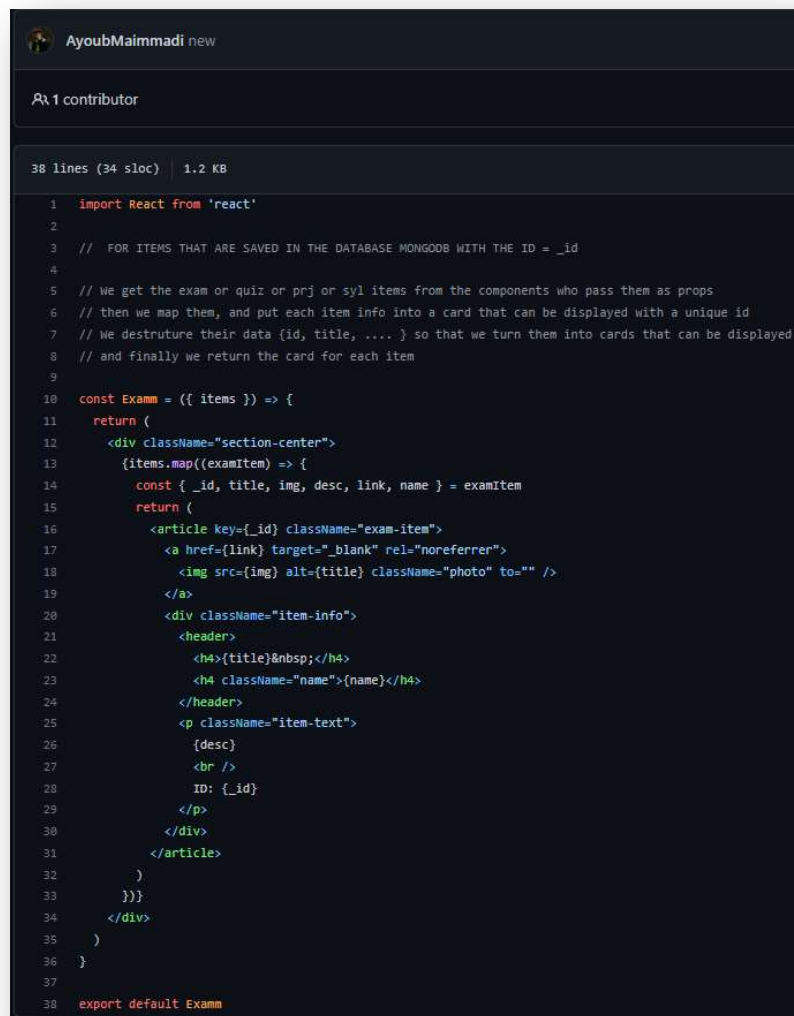
This is where new and superiorly improved technologies follow. Massive companies such as Google wondered what if the hypertext markup language -which is most used and loved in this domain- was more powerful. Comes Angular.js. A modern framework which is still one of the most popular and dominating technologies to this day. Ranking top 5 each year from the day of its birth. A technology that provided so many remarkable features straight out of the box. It changed the way people build web applications. A development engineer working on the team that released this framework said during its announcement presentation, “Angular is what a web browser would have been, had it been designed for building web applications”. Which was the perfect summation of this beautiful newly released technology for building software. An all-in-one framework that offered so many things straight out of the package. Engineers no longer needed to rely on third party tools to do the job. If a software development tool exists, Angular has it in its ocean of libraries. Everything a software engineer would ever want, which made Angular a much wiser choice for massive productions because of its massive ecosystem. After the release of Angular in 2011, with built in data filers, two-way data binding, routing, dependency injection, controllers, templates, etc. It seemed that developers had everything they will ever need all in one framework. However, we need to remember an important point. In these times, being an only front-end engineer was super uncommon in comparison to today’s standards. The nature of Angular gave back-end engineer the awkward task of developing the front-end as well. Of course, this was not as uncommon as some people might expect, and it played a massive role in expanding this framework’s popularity among others in the field of

software engineering and development. However, there was a slight problem. This new technology loved the concept of two-way data binding. It was a way of updating the view whenever the model changed and updating the model whenever the view changed. Theoretically, this was considered an improvement because developers won't have to take on the struggle of manipulating the DOM manually. However, it was practically a disaster when it came to the actual implementation of this new technology. Long lines of complicated code and hard to follow for new people working on it, and most critically damaging, hard to debug. Performance issues also have risen up because Angular had to constantly scan engineers' applications in order to detect new state changes. All tough so far, I have talked historically about only three technologies, which are jQuery, backbone.js, and Angular. Building for the web consisted of many other technologies that followed the same footsteps as the ones I mentioned. They all had one thing in common, models, and specifically, maintaining said models in sync with the view. Enters React, the currently most dominating framework in the field.

"The simplest way to build views, is to just basically avoid mutation altogether. What we found is that anytime data changes. If we can blow away the view, and just re-render from scratch, that would be so much easier. This will be done in a way that is actually performant and provides a good user experience" -Tom Ochino, an engineering manager in the React team announcement presentation in 2013.

Object oriented programming was famously adapted by many companies by this time. However, most communities of software engineers disliked this approach of programming, as Edsger W. Dijkstra famously stated, "Object oriented programs are offered as alternatives to correct ones." This is one of the main reasons why React is universally loved by so many people in this domain. The transformation of the view into something every software engineer is most familiar

with. A basic function. This view function holds the application state. All engineers have to worry about is how the state of their applications change, and the framework would handle the rest of the complicated work behind the scenes. Everything becomes much simpler because of the decelerative nature of the development. Each place data is displayed is up to date with 100% certainty, and it's done without the trouble of data binding, with no expensive module checking, and obviously, no more explicit DOM operations. As big as this may sound, this was not even the half of it. The actual revolution happens when engineers encapsulate this concept into an genuine component-based application programming interface or a component API for short. This innovation made it possible for software engineers and developers with the same way they are used to composing and creating functions in normal programs, they can now directly apply this strategy to creating and composing entire web components. Components which are reusable throughout the application and can adopt different features depending on the type of data passed to them. For example, we have an Exam component which you can see below from an Agile Software Engineering class project I have created in the spring semester of 2022, which further explains the concept of reusability.



```

1  import React from 'react'
2
3  // FOR ITEMS THAT ARE SAVED IN THE DATABASE MONGODB WITH THE ID = _id
4
5  // We get the exam or quiz or prj or syl items from the components who pass them as props
6  // then we map them, and put each item info into a card that can be displayed with a unique id
7  // We destruture their data {id, title, .... } so that we turn them into cards that can be displayed
8  // and finally we return the card for each item
9
10 const Exam = ({ items }) => {
11   return (
12     <div className="section-center">
13       {items.map((examItem) => {
14         const { _id, title, img, desc, link, name } = examItem
15         return (
16           <article key={_id} className="exam-item">
17             <a href={link} target="_blank" rel="noreferrer">
18               <img src={img} alt={title} className="photo" to="" />
19             </a>
20             <div className="item-info">
21               <header>
22                 <h4>{title}&nbsp;</h4>
23                 <h4 className="name">{name}</h4>
24               </header>
25               <p className="item-text">
26                 {desc}
27               <br />
28               ID: {_id}
29             </p>
30           </div>
31         </article>
32       )
33     )}
34   </div>
35 )
36 }
37
38 export default Exam

```

The Exam components receives different data each time in the form of a state components called items and displays the corresponding component for each item with different data. To view entire application, use this [Link](#).



To truly embrace this strategy of component-based API, the React team needed to offer a way that allowed software engineers to describe how the user interface for each component should look like from inside the component itself. At first, people disliked this approach because engineers by this time were used to having separate Script files, HTML files, and CSS files. This is called the separation of concerns principle in the field of software development. More specifically, people hated the new way of writing the HTML part of their applications with the script file. Also, React introduced JSX, which is a super set or extension of HTML where engineers can pass the programming language of JavaScript inside of the HTML using curly braces.

```
<h4>{title}&nbsp;</h4>
```

React had a completely different interpretation and new way of doing things. It offered the opinion that if anything is related with rendering a specific view, whether if it's a user interface, a state, or even giving styles to UI's, it was part of its circle of concerns. Accomplishing this in

React is done by a way of describing how the user interface of a component should look like from inside that very component. This allowed engineers to use a language which not only they are too used to using, but also flexible, and powerful to work with. It offers a way of representing logic, and interactions that are not easily done in other statically typed languages such as Java. The majority of other JavaScript technologies or frameworks allow software engineers to define their own entire user interface using only JavaScript. This made things very unpleasant and complicated for developers who are building deeply nested components in massive productions, even in relatively small ones. That's why at the React team introduced a completely new way of using HTML by the name of JSX, which stands for JavaScript XML (Extensible Markup Language). A syntax that is super familiar to engineers experienced in using templates or HTML, and yet still adapts the JavaScript language that people love and use when it comes to this field. It was a combination of accessibility and readability of HTML with the expressiveness and power of the JavaScript language. Soon, people who were already familiar and comfortable with using HTML and JavaScript found themselves easily learning this new tool and switching from whatever technologies they were using into this brand-new innovation made by the Facebook or now called Meta enterprise. Soon, the hate for this new way of developing web apps turned into love. As soon as an engineer started experimenting with React, they found themselves in love with it. It became clear to every community of software engineering that React was in fact into something new and revolutionary. What needed using thousands of lines of codes can now be done using hundreds by embracing the component-based API with the JSX syntax. Plus, what was once requiring an imperative operational code, now is abstracted behind a declarative API. Which not only offered a simple and easy experience in development, but also a massive ecosystem of third-party ready to use components.

#### **4- Compare and contrast for two technologies**

##### **4-1 React with TypeScript**

React is a JavaScript library for building user interfaces, and it is one of the most if not the most influential software and web applications building tools so far. It is a component-based API framework. Meaning, you can represent logical reusable parts of the user interface by building different components and pass different data to display different UIs. This technology is most loved by the communities of software engineers and developers because of its super simple way of writing code for its components, as a function, the most familiar concept any code writer can wish for. This function returns a Hypertext markup language, HTML for short or a user interface. This return value has to be coded using a JSX (JavaScript XML), which is an extension of HTML that allows programmers to pass the programming language of JavaScript to the HTML. However, TypeScript is the choice of programming language for this comparison. TypeScript is a statically typed super set of JavaScript. This allows it to perform much better than its sister, and offers type checking out of the box, so as not to fall into bug problems during production. To be more precise, with JavaScript, you can catch bugs only after run time. But, with TypeScript, you can catch every bug and non-existing references before it can ever be shipped to production. This makes it the preferred language when building massive scale applications, because not only is it development friendly, but it's also much faster than JavaScript.

##### **4-2 Web Assembly with Rust**

Web assembly or WASM for short, is a new way of developing software and web applications using programming languages that were not designed for building web applications. Meaning, a tool

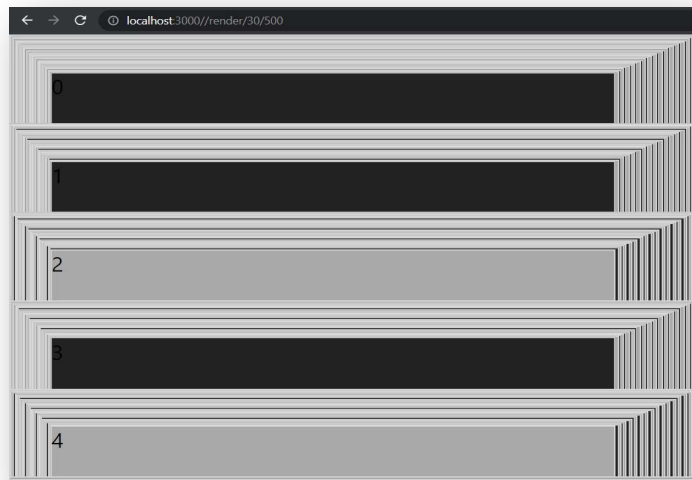


that offer developers the use of languages other than JavaScript, TypeScript or Python for building web application. Languages such as C++, Go, and Rust, that will deliver the software in a browser to the user with no installations, and very good performance. This new way of building web applications came to life in the year of 2019. The year where WASM officially became part of the W3C standard (The World Wide Web Consortium Standards organization). Web assembly consists of a low-level language that looks just like regular assembly represented text but can be converted into a binary format that can run in pretty much every modern browser.

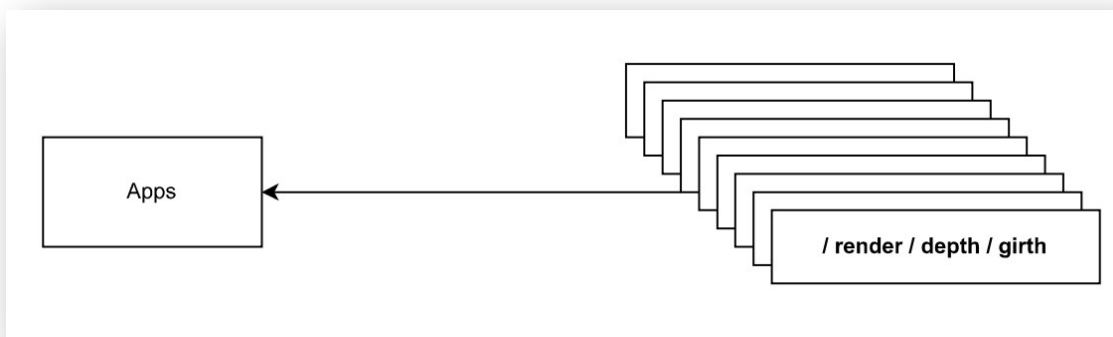
Of course, engineers will not be writing the low-level language of web assembly directly. They will in fact use it as a target for compilation by programs written in different statically typed and multi-paradigm programming languages such as Rust in this case. An example of this is building a 3D game with the C# programming language using Unity game engine, and then compile it to the language of web assembly. After that, it can be automatically delivered in any existing browser.

### **4-3 Comparison and Results**

This experiment is meant to compare between a dominant framework by the name of React, against a newly born way of building web applications with web assembly using statically typed languages. For React, TypeScript is the language used for building the app, and for web assembly, Rust is the language of choice. First, a server is created with these two frameworks, and it's going to scale based on its Uniform Resource Locator (URL). The way this is going to happen is by calling into these apps via link by rendering a depth, and girth:  $\frac{\text{depth}}{\text{girth}}$



This allows us to test for parallel numbers of requests. Making every requests to either apps able to measure the amount of time each one is server side rendering. Along with keeping track of how long it will take each application to take as an example 10,000 renders of a page, and then measure how many requests per second we are going to reach with different parallel connections. The aim is to compare for 50, 100, 200 and 500 parallel connections, with depths of 1, 3, 5, and 10, meaning how many elements will be rendered for each block.



For example, at 50 parallel connections, and at depth of 10, about 500 elements will be rendered, while at depth of 1, only 50 will be. All this in the sake of comparing these framework's overall

performance with each other when it comes to the work-load they can handle based on server side renders, which is a web app's capacity to transform all HTML files on the server into a completely rendered HTML page for the user, and requests per second (RPS) which is one of the most important metrics when it comes to determining the ability of an application that measures its throughput.

### **Server Side Rendering (SSR): React Vs Web Assembly**

Parallel Connections \ Depth	Depth 1	Depth 3	Depth 5	Depth 10
50	462 $\mu$ s	1041 $\mu$ s	1686 $\mu$ s	3874 $\mu$ s
100	581 $\mu$ s	1484 $\mu$ s	2219 $\mu$ s	3869 $\mu$ s
200	622 $\mu$ s	1597 $\mu$ s	2213 $\mu$ s	3906 $\mu$ s
500	616 $\mu$ s	1536 $\mu$ s	2267 $\mu$ s	3865 $\mu$ s

**Table 1: Web Assembly Server Side Rendering (SSR) Time (In microseconds)**

Parallel Connections \ Depth	Depth 1	Depth 3	Depth 5	Depth 10
50	1345 $\mu$ s	2549 $\mu$ s	3770 $\mu$ s	6630 $\mu$ s
100	1387 $\mu$ s	2597 $\mu$ s	3618 $\mu$ s	6571 $\mu$ s
200	1361 $\mu$ s	2512 $\mu$ s	3641 $\mu$ s	6807 $\mu$ s
500	1362 $\mu$ s	2538 $\mu$ s	3743 $\mu$ s	7008 $\mu$ s
	239.15%	180.21%	176.17%	174.14%

**Table 2: React Server Side Rendering (SSR) Time (In microseconds)**



**(a) Server Side Redering with depths 1, 3, 5, and 10 – React Vs Web Assembly**

(WASM is short for Web Assembly)

The first two tables are a comparitions between React and Web Assembly when it comes to server side redering (SSR). All the result data shown in these tables are in microseconds. As we can notice, web assembly is faster when it comes to server side rendring. Between 174% to 239% faster, which is on average double the performace. When you think about it, web assembly is not that much faster then React in the grand scheme of things. After deth 2, the percentage of difference goes from 180.21% to 176.17% with depth of 5, and finally 174.14% for depth 10. This means this number is going to relatively aproach a number close to this range, and make web assembly not even double the speed of React. However, there is a catch. For example, the 6639 microsedonds from the React SSR table with 50 parallel connection and a depth of 10 is almost missleading. This is due to the fact that this is a contiguous time that's runing the React

run time. This indicates that no other JavaScript will execute at the same moment. Whereas with web assembly, the 3874 microseconds with 50 parallel connections and depth of 10 is in fact is actually interleaving many requests at the same time. This leaves it disproportionately affecting requests per second. Meaning that the percentage difference between React and web assembly when it comes to server side renders is not going to be the same when it comes to present different for the number of requests per second.

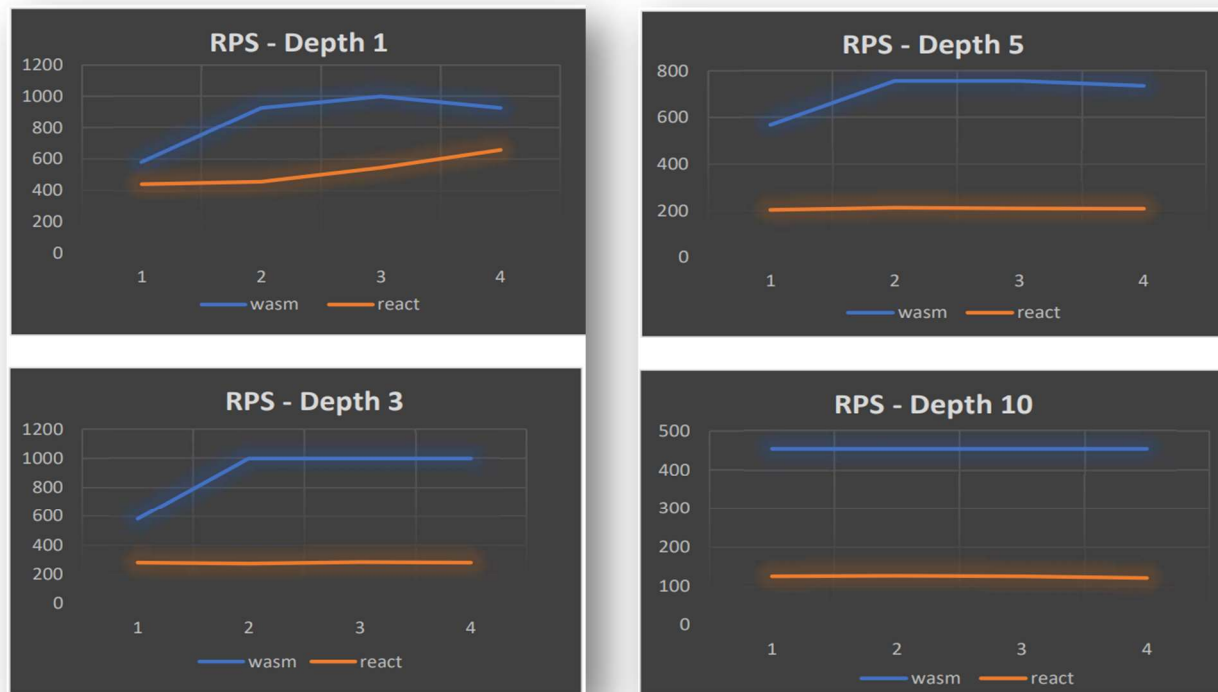
### **Requests per second (RPS): React Vs Web Assembly**

Parallel Connections \ Depth	Depth 1	Depth 3	Depth 5	Depth 10
50	581	581	568	454
100	925	1000	757	454
200	1000	1000	757	454
500	925	1000	735	454

**Table 3: Web Assembly Requests per second (RPS)**

Parallel Connections \ Depth	Depth 1	Depth 3	Depth 5	Depth 10
50	438	280	204	125
100	454	274	213	127
200	543	284	210	125
500	657	280	208	121
	164.01%	320.30%	337.37%	364.66%

**Table 4: React Requests per second (RPS)**



### (b) Requests per second with depths 1, 3, 5, and 10 – React Vs Web Assembly

(WASM is short for Web Assembly)

We can notice that React did not lose as hard for depth of 1, indicating that web assembly is only 164% faster. However, this is not a real world scenario for big applications. When we compare web assembly's performance at depths of 3, 5, and 10, we notice that it has enormously increased all the way to 364.66% at depth of 10. Making web assembly doing more than triple the amount of requests per second React is doing.

This goes to show the huge performance difference is only going to increase with more depth and with more upgrades in the future. This is because React has been out for more than 8 years and its already losing to a compitor who's only been out for 3 years. The core of this

performance difference lies on the language of choice when it comes to developing web applications using these two frameworks. JavaScript is inherently a slow language, and so is its sister TypeScript. Whereas Rust is a competitor in speed and performance against the C or C++ programming language, which are 4 times faster than JavaScript as a dozen studies and comparisons indicate.

## **5- Future of web development technologies**

### **5-1 Development experience & tools**

The future of Web developments depends on the architecture of microservices. Massive scale applications won't be able to rely on a singular technology stack to implement every piece of their software. Of course, this is already adopted by most big companies. However, they are now debating whether these different stacks should make the transition from classic programming languages when it comes to building for the web such as JavaScript, HTML, and CSS, to the recently born new way of building applications using web assembly with much faster languages such C++, Go, and Rust. Plus, as we established earlier, Web Assembly is much performant than JavaScript and its frameworks. So, this begs the question, will companies adopt web assembly as their main stack for building their software and web applications? The answer is no in my opinion, and I'm going to tell you why.

Web assembly is in fact a better tool for building future apps. However, modern companies can't just replace their entire code from something that already works relatively good. Plus, the ecosystem of JavaScript, HTML, and CSS frameworks is huge and not easily replaceable. This takes us to the question of money, and how companies won't approve a very expensive switch of

technology and professional developers instead of better optimizing what they already have and use in their stacks. Web assembly is a wonderful improvement to the way we are used to engineering software and web applications. It might not entirely replace the current stacks that companies use, however, it will definitely enlarge the nature of web applications by accomplishing additional goals that were not super optimized or even possible using current stacks. For example, if an engineer wants to render a scene from a 3D game where users can explore using CSS. It is pretty much impossible to achieve. A high-performance language is needed to perform such a task, which has only been feasible with the use of the low-level language of Web Assembly via distributing games initially through browsers.

## **6- Conclusion**

To conclude, this paper removes the mystery of how web development technologies and frameworks came to be, and the pattern of choices made by their mother companies to ensure their superiority in the field. From classical tools we rarely use any more like jQuery, to dominating tools we currently use such as React, and to the future of web development with Web Assembly. The experiment presented in this paper highlights the defeat in performance of React - the leading framework in our current era-, against web assembly - a newly invented technology of developing web apps – when it comes to performance, and specifically, server-side rendering and requests per second. Though this might seem as a clear indication that adopting web assembly as the main stack for any start up, the ecosystem around it is still at a very early stage of maturity. Whereas tools such as React and Angular have engraved their places deep in the market of software development technologies with their massive ecosystems that either come with the framework straight out from the box, or from third party libraries created by open-



source communities of web developers and software engineers to achieve a specific goal with maximum possible optimization. I argue in this paper that though Web Assembly has strong future, but upcoming developers should not ignore the fact that it can never be as massive as its current competition and should make their choice when picking a new tool for their projects having this in mind.

## References

- Delcev, S. (2018, May 30). *Modern JavaScript frameworks: A Survey Study*. Retrieved from IEEEExplore: <https://ieeexplore.ieee.org/abstract/document/8448444>
- Gizas, A. (2012, April 16). *Comparative evaluation of javascript frameworks*. Retrieved from Digital Library: [https://dl.acm.org/doi/abs/10.1145/2187980.2188103?casa\\_token=RzZrTKwq88cAAAAA:jbtUp6L9tUpnMZr7xvAwZ84DQU1z\\_8ABfHm1JealUdTdE\\_zE6rU9G-hbJQki66MzJN-WyQthNG5Cgw](https://dl.acm.org/doi/abs/10.1145/2187980.2188103?casa_token=RzZrTKwq88cAAAAA:jbtUp6L9tUpnMZr7xvAwZ84DQU1z_8ABfHm1JealUdTdE_zE6rU9G-hbJQki66MzJN-WyQthNG5Cgw)
- Graziotin, D. (2013). *Making Sense Out of a Jungle of JavaScript Frameworks*. Retrieved from Springer: [https://link.springer.com/chapter/10.1007/978-3-642-39259-7\\_28](https://link.springer.com/chapter/10.1007/978-3-642-39259-7_28)
- Hoffman, K. (2019). *Programming WebAssembly with Rust : Unified Development for Web, Mobile, and Embedded Applications*. Retrieved from Torrossa: <https://www.torrossa.com/it/resources/an/5241301>
- Ollila, R. (2022). *Modern Web Frameworks : A Comparison of Rendering Performance*. Retrieved from Helda: <https://helda.helsinki.fi/handle/10138/343674>Modern Web Frameworks : A Comparison of Rendering Performance
- Uzayr, S. b. (2019). *JavaScript Frameworks for Modern Web Development*. Retrieved from Springer: <https://link.springer.com/content/pdf/bfm%253A978-1-4842-4995-6%252F1.pdf>