Lucia: JavaScript Library for Developing Performance-Focused Web Applications

Aiden Bai

Camas High School

Abstract

The necessity of developing and deploying interactive web applications quickly has become more prevalent over the last decade, resulting in developers creating libraries and frameworks to simplify this process. These JavaScript libraries have come to dominate the web landscape, with many websites adopting them, as they allow more functionality and extensibility to normal JavaScript. However, these libraries bring overhead, reducing performance, and increasing bloat. This is why Lucia was created, to simplify web development, improve performance, while offering the core functionality of mainstream web libraries. This was done by creating an intentionally simple and straightforward core architecture, then benchmarking the performance through a suite of comprehensive and thorough tests. Using Lucia allows for not only a faster user experience for the end user, but also a better developer experience, uprooting how traditional web applications are created.

Lucia: JavaScript Library for Developing Performance-Focused Web Applications

In the past decade, the way computer programmers develop the web has changed. Previously, most web pages were static and relatively simple, primarily relying on programming languages like Hyper-Text Markup Language (HTML), Cascading Style-Sheets (CSS), JavaScript, as well as the Document Object Model (DOM). These technologies were often used to create architectures to generate and show content through statically-served markup, supplemented with simple and sparse DOM rendering.

Nowadays, the usage of the web has changed. With the advent of mobile electronics that have access to the Internet and the necessity and demand of complex functionality, it has led to the rise of Single Page Applications (SPA), dynamic web pages, and JavaScript libraries such as jQuery, React, and Vue. These innovations have drastically changed usage upon what it was originally, moving the process of development and computational logic to the JavaScript end. One of the notable examples of this is jQuery, which is a widely used library to build JavaScript logic. Along with the creation of these libraries, new paradigms of programming have emerged from this trend, notably the Virtual DOM or variants of it (Grov, 2020). These techniques have allowed for improvements in many areas, such as ease of development, customization, and performance, meaning better experience for both the user and developer.

Historical Overview

JavaScript is a programming language that has historically been an invaluable part of web development. In the days of the pioneer era of the web, web pages were statically served, meaning dynamic portions were limited to embedded animated Graphics Interchange Format (GIF) images or small CSS animations. However, coming back to today, with entire web pages

being built with primarily JavaScript, it has changed from a toy animation language to a language with a wide ecosystem of code and libraries for numerous use cases.

JavaScript's first version was created by Brendan Eich of Sun Microsystems as an effort to add a scripting language to Netscape Navigator (JavaScript, 2002). As time progressed, more and more web browsers have been created and have adopted JavaScript as their main scripting language. Some of the major web browsers include but are not limited to Internet Explorer, Google Chrome, and Mozilla Firefox, resulting in the branching of DOM-specific JavaScript implementations. As an outcome of the stratification of browser-specific JavaScript implementations, many of the companies that run these browsers banded together into the ECMA International consortium, allowing for the standardization of JavaScript as ECMAScript (*JavaScript*, 2002). Throughout the years, ECMA International has pushed out new standardizations to accommodate for new features. However, most web development standards have settled on ECMAScript 5 (ES5), even though most modern browsers support ECMAScript 6 (ES6). This is due to backward compatibility for the vast market percentage that still use older browsers.

Browser-side JavaScript has many uses. One of its main uses is to interact with "HTML and XML documents" (Persson, 2020). It does this through the DOM, which is a tree-like Application Programming Interface (API) data structure, which are nodes that branch off each other, creating a hierarchy. "The DOM tree consists of nodes and objects, originating from the root node … nodes have properties and methods which allow for manipulation of the nodes themselves. Using the DOM programmers can create documents, navigate the documents hierarchy, and modify the content of the document..." (Persson, 2020). This fundamental

innovation allowed the HTML and the JavaScript to interface with each other, creating one of today's intrinsic parts of the web.

After the creation and development of JavaScript and the DOM, web developers have created tools and libraries for simplifying development and to add functionality. One of these libraries is jQuery, a widely used DOM utility library. The goal of jQuery was to help developers solve issues and allow for cross-browser compatibility, while providing a clean and easy syntax to use (Muyldermans, 2019). jQuery was important because it helped web developers concisely manipulate the DOM, allowing for cross-browser compatibility, and the segregation of the HTML document and the JavaScript by removing the necessity of using event listener attributes (jQuery, 2007). These innovations allowed jQuery to flourish in the JavaScript community and ecosystem, and "as of Feb 2020, jQuery is used by 74.4% of the top 10 million websites" (jQuery, 2007). This continues to grow at a rapid rate, despite the declining hype around the library itself due to strong criticism (Holland, 2016).

Although jQuery had a monumental impact on web development, many web developers were still unsatisfied with the state of web development at that time. Software engineers at Facebook in 2013 started to work on a framework called React. "[React] uses a Virtual DOM, where new DOM trees can be created with JavaScript, which creates a simpler programming model. The data bindings in React are implemented with something Facebook Inc. calls diff algorithm. This algorithm causes a full re-render of the DOM of the changed data. To detect the changes in the Virtual DOM, React compares the DOM trees in every level and re-renders when a change is found" (Molin, 2016). React's usage and pioneering of the Virtual DOM allowed for a new way of performance optimization and philosophy of building web applications. React

sparked a revolution in frameworks, leading to the incorporation of the Virtual DOM in frameworks like Ember 2.0 and Vue (Grov, 2020).

One of the notable libraries inspired by React and AngularJS was Vue. Vue was created by Evan You, a former Google engineer to address some of the technical issues with React and the simplification of AngularJS. This allowed Vue to brand itself as a progressive framework, meaning that developers at any skill level would be able to use Vue in some capacity. Vue also innovated upon and focused on bundle sizes, providing a lightweight export, differing from previous mainstream frameworks in this way. As of October 2020, Vue has over 174 thousand Github stars, making it the most starred repository on Github (You, 2014).

Current Trends and Practices

JavaScript usage has evolved over the years. From using native code to create animations to powering fully functional applications, JavaScript will continue to be used and be incorporated into the web. Many developers have created and used libraries or frameworks to bootstrap the aforementioned web applications, such as jQuery, React, and Vue. These frameworks use several methodologies of development. For example, React focuses on declarative primitive components, meaning that web applications are built up, versus working down from abstractions. Another example is Vue, which focuses on being a progressive framework, allowing a wide range of developers to write declarative, view-based code regardless of experience. “Vue can be easily introduced into an app by simply including a script tag, more can be added according to one’s needs” (Muyldermans, 2019).

Most mainstream and modern libraries construct their Application Programming Interfaces (API) to be easy, declarative, scalable, and reactive. Often these libraries utilize techniques like Virtual DOM architectures to implement such functionalities or paradigms. The

Virtual DOM technique is utilized by both Vue and React, with varying differences, such as the use of JSX and Hooks in React or object and computed properties in Vue. “In addition, React has its own JavaScript language called JSX. This provides an extension to JavaScript by adding XML-like syntax that is similar to HTML” (Muyldermans, 2019). Other libraries such as jQuery take advantage of direct DOM manipulation, opting for a more imperative API but simultaneously simpler to use than native code, which allows for non-component based scalability. The variety of choice in libraries allows developers to choose a paradigm based on whatever conditions they see fit, but can also create unnecessary competition and oversaturation of certain functionalities.

Controversies and Debates

The current ecosystem of JavaScript libraries overflowed low-quality clones of popular libraries rather than useful ones. Even for the mainstream libraries, much of the codebase has received criticism for unnecessary bloat. “[Many frameworks are] over-engineered and too complicated … However, that has not stopped its adoption and legions of adoring fans” (Holland, 2016). This has created a toxic cycle of web developers using either recycled code or bloated popular packages for creating projects. This often results in a degradation of user experience and performance. In addition to this, many web developers feel that there are too many libraries to choose from, due to the oversaturation of libraries (Holland, 2016).

Another controversy in JavaScript libraries is how markup and JavaScript logic are handled. “Ember will serve as the diametric comparison for React in terms of separating logic from markup” (Holland, 2016). React encourages developers to blend and mix markup and logic together through JSX, however frameworks like Ember strictly segregate markup and logic into different partitions. These two different philosophies have been criticized either for being too

loose or opinionated, prompting different web developers on different sides to conflict. One could argue that this type of stratification of the community results in harming all parties and these issues can even propagate up to the enterprise level.

A rising issue that web developers face is the amount of abstraction in the architecture and construction of modern JavaScript libraries. The amount of abstractions ranges widely, from libraries like React focusing on a primitive developer experience with a low learning curve, with Angular being the direct opposite. Both solutions have differing use cases, but from a holistic perspective, among the general developer community, primitive construction is much more appealing, bridging the way for community modularization and widespread adoption by developers.

Oftentimes, traditionalist web developers question the necessity of jQuery and other DOM utility and compatibility libraries. With the development of more APIs and DOM methods through standardization by modern web browsers, many of the features of jQuery can also be written in native code in a similar fashion. “developers [do not] need jQuery to perform basic DOM manipulation or XHR operations, but because jQuery provides a much cleaner API than native DOM code” (Holland, 2016). Not only this, cross-browser compatibility isn’t a huge issue, as thanks to the large quantity of polyfills that have already been implemented by JavaScript version transpilers like Babel.

Materials and Methods

My overall procedure consisted of creating the core and API, performing testing, and publishing the package (see Figure 1). My specification describes the general flow of how the process was, but do note that backtracking, iteration, and modification were required. At each

point in the process, I enforced code styling and conventions with Prettier and ESLint, ensuring maximum maintainability of the codebase and minimizing barriers for external contributions.

Figure 1. Library creation procedure flowchart

Core and API

I wrote the core and API using TypeScript, using zero NPM dependencies and built the code into distribution bundles using Rollup, TypeScript compiler, Babel for legacy, and ESBuild for the development environment. The first task in the project was constructing the low-level core, consisting of the compiler, renderer, directives manager, and observer. The compiler accepts DOM nodes and outputs an optimized Abstract Syntax Tree (AST) that can be parsed. The AST contains an array of AST nodes, allowing for linear traversal, and each node is an object literal that contains the dependencies, directives, node type, and DOM node reference. Each directive contains a compute function, dependencies, and an expression. The renderer accepts an AST and modifies the structure and content of the DOM based on AST node dependencies through linear traversal. The directives manager searches and executes mutations on DOM nodes by referencing special attributes on DOM nodes, and is called on by the renderer. The observer is the reactivity engine, creating a proxy between the state and calling render functions and state watchers on change.

My implementation purposely made the core decoupled from the API, making it as API agnostic but DOM-specific as possible to ensure performance through concretization. I ensured

this by making everything a pure function, omitting all API-specific flags by default in the compiler, and allowing the render function to accept arbitrary AST. This also means that any developer can build their functionality or superset library over the core.

After this, I connected the core process (see Figure 1.) with each other to produce a cohesive and efficient product. Once I completed the core, I constructed the API to allow other programmers to be able to interface with the library. This contains the high-level abstractions that Lucia delivers: directives, state, and components. The directives mimic Vue's directives but have smaller scope for the fundamentals, constructed using a monomorphic pure function call. The state and components rely on the internal reactivity, compiler, and render core to perform, calling and passing specific parameters as necessary.

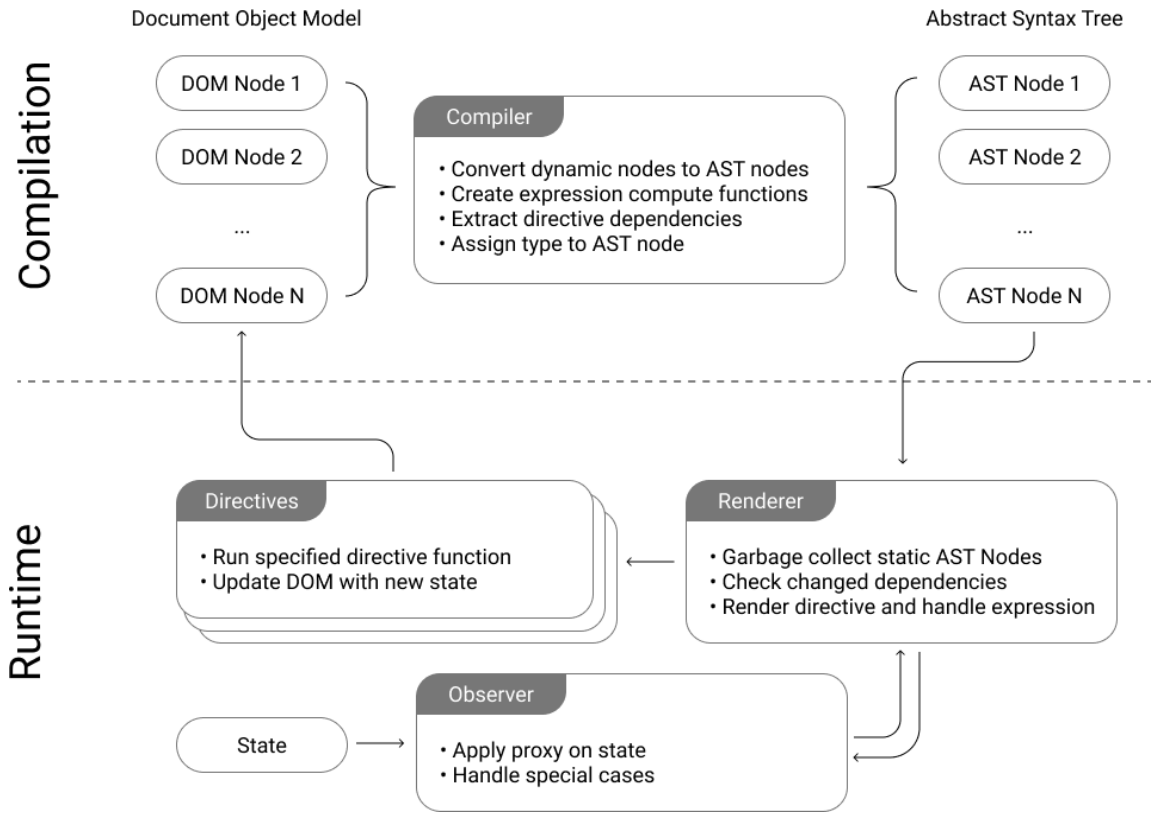
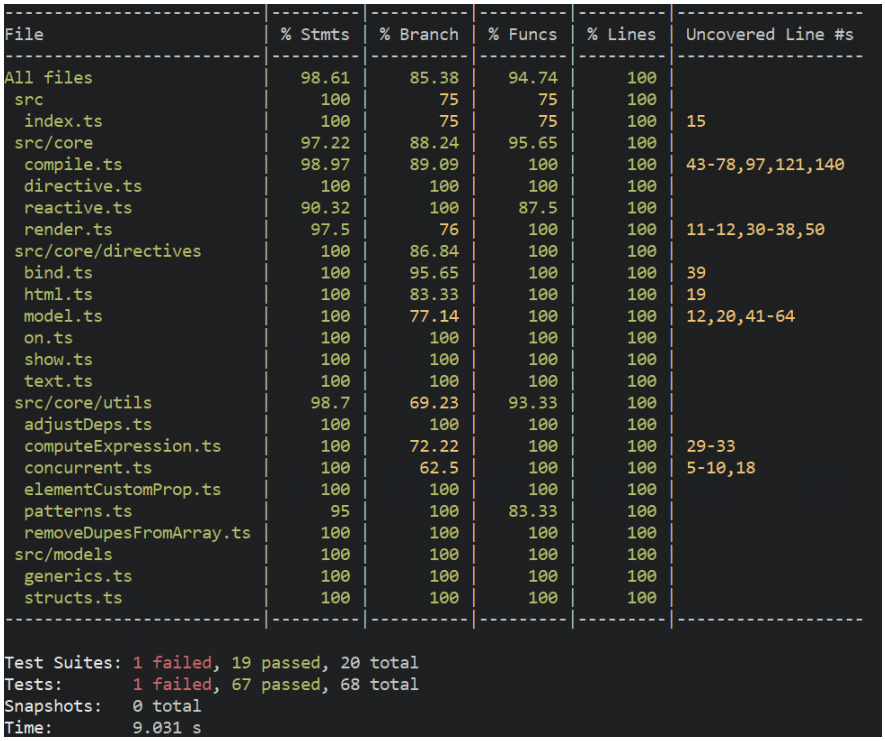


Figure 2. Lucia core process blueprint flowchart pipeline

Testing and Debugging

In conjunction to the construction of the core and API, I performed ad-hoc testing and wrote unit tests to ensure that the code is safe and bug-free. To do ad-hoc testing, I used the Chrome and VSCode debugging tools, for real-time testing of development builds. My initial commits will be pushed to the staging branch on the Github, allowing other users to provide feedback through the issue tracking system and discussion feature. I utilize this process to triage and patch bugs.

For unit tests, I used Jest to run code assertion to automate generic tests that run on lifecycle hooks to ensure that established code is working properly. During the final stages of development, I utilized the native coverage report system Jest provides (see Figure 2.) to determine which parts of the codebase need more testing, and integrated the test runner into a Github Action continuous integration script.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	98.61	85.38	94.74	100	
src	100	75	75	100	
index.ts	100	75	75	100	15
src/core	97.22	88.24	95.65	100	
compile.ts	98.97	89.09	100	100	43-78,97,121,140
directive.ts	100	100	100	100	
reactive.ts	90.32	100	87.5	100	
render.ts	97.5	76	100	100	11-12,30-38,50
src/core/directives	100	86.84	100	100	
bind.ts	100	95.65	100	100	39
html.ts	100	83.33	100	100	19
model.ts	100	77.14	100	100	12,20,41-64
on.ts	100	100	100	100	
show.ts	100	100	100	100	
text.ts	100	100	100	100	
src/core/utils	98.7	69.23	93.33	100	
adjustDeps.ts	100	100	100	100	
computeExpression.ts	100	72.22	100	100	29-33
concurrent.ts	100	62.5	100	100	5-10,18
elementCustomProp.ts	100	100	100	100	
patterns.ts	95	100	83.33	100	
removeDupesFromArray.ts	100	100	100	100	
src/models	100	100	100	100	
generics.ts	100	100	100	100	
structs.ts	100	100	100	100	

Test Suites: 1 failed, 19 passed, 20 total
Tests: 1 failed, 67 passed, 68 total
Snapshots: 0 total
Time: 9.031 s

Figure 3. Coverage report from Jest unit testing command-line interface.

Publishing and Promotion

Finally, I created a production build process with Rollup, Babel, and Terser. Then, I published the distribution bundles on NPM, and thereby Unpkg, Skypack, and JSDelivr. After that, I ensured that Dependabot does not display any security concerns, and any subsequent publishes should follow the semantic versioning policy. This is to allow for users to consistently use specific versions and update only if necessary. For promotion, I posted on platforms like Reddit, HackerNews, and EchoJS, specifically targeting developer communities for high click-through rates and usability.

Benchmarking

I benchmarked the top four mainstream libraries according to StateOfJS 2018. I chose these libraries because they characterize the technologies that developers often use to create webpages. I picked the version based on the latest, major, and non-beta release for each JavaScript library. I used the BundlePhobia diagnostics tool to measure code size, providing detailed and accurate sizes of bundled content before tree shaking. I wrote and ran benchmarks with Benchmark.js, a widely used benchmarking library, and following the official JS Framework Benchmark guidelines excluding possible variation. Afterwards, I compiled together the quantitative data to produce a final, weighted score.

Results

The purpose of my research project was not only to create a performant JavaScript library, but also to construct practical benchmarks that measure the performance capabilities of Lucia. In Table 1, I recorded quantitative measurements, performance benchmarks, and qualitative observations. These measurements are categorized into code size, benchmarks, and library attributes. I sorted and marked the scores with a color in the spectrum of green to red,

with green signifying exceptional performance and red indicating subpar performance. There are also intermediary colors, which communicate the scale of performance on a linear scale.

Some notable areas in the data (see Table 1.) include Angular 2's performance regarding script bootup time, clearing 1000 rows, and abstraction, which is quite high. Additionally, jQuery's benchmarks are high for all benchmarks, and Lucia and the baseline's benchmarks are very low.

Library	None	Lucia	React 17	Angular 2	Vue 3	jQuery
Raw size (kb)	0.00	2.29	156.00	108.00	342.00	409.00
Minified+Brotli Size (kb)	0.00	1.87	19.80	65.40	94.50	86.30
Script bootup time Non-keyed (ms)	16.0	16.0	16.0	114.8	16.0	69.1
Create 1000 Rows Non-Keyed (ms)	107.2	108.1	183.5	152.1	152.1	209.9
Update 1000 Rows Non-Keyed (ms)	35.4	36.6	45.0	39.6	45.8	205.2
Clear 1000 Rows Non-Keyed (ms)	101.6	102.5	147.6	245.1	158.7	217.5
Geometric Mean of Benchmarks	49.8	50.5	66.5	114.1	64.9	159.5
Mean of Sizes	0.00	2.08	87.90	86.70	218.25	247.65
Weighted Score	24.92	26.28	77.18	100.40	141.55	203.58
Learning Curve	Low	Low	Medium	High	Medium	Low
DOM Type	DOM	DOM	VDOM	DOM	VDOM	DOM
Abstraction	None	Weak	Weak	Strong	Weak	Weak
Rendering	Client	Client	Client/SSR	Client/SSR	Client/SRR	Client

Table 1. Comparison of mainstream JavaScript libraries benchmarks and attributes.

Using the weighted score data in Table 1, which is composed of equal parts code size measurements and performance benchmarks, I visualized the data as a column graph in Figure 4. This visualization is intended to demonstrate a trend in the weighted score data and summarize the results. Lucia and the baseline have a very low column, while the other mainstream libraries have very high columns, with higher weighted scores.

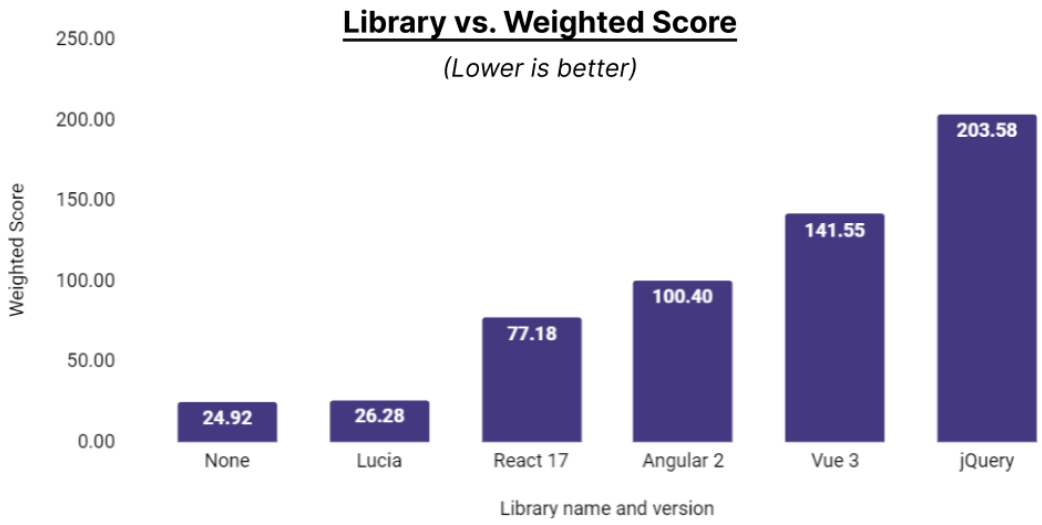


Figure 4. Column graph of weighted score for JavaScript libraries

Discussion

My hypothesis was supported by the data I collected. The goal of this project was to create a JavaScript library and benchmark it against other mainstream libraries to compare performance. These results clearly display that Lucia has a lower, significantly better weighted score when compared to other libraries. Where nuance comes in is the consideration of applying libraries into practical applications and thereby the significance of benchmark performance. For instance, creating, updating, and deleting one thousand rows is generally not prevalent among commonly used web applications, rather it may be something simple, such as opening and closing a modal. Some developers may argue that benchmarks do not accurately represent what is reality, and should be taken with a grain of salt.

Although this is true to an extent, it does not take into consideration what benchmarks actually test. At face value, it seems that benchmarks only test extremely esoteric tasks, such as bootup time and list rendering, but in reality it is also testing something much more fundamental - the compiler and rendering/interpolation engines of the libraries. It is possible it may vary

depending on tasks, but the core engines remain constant and therefore can be a valid measure of determining performance. Not only this, to ensure consistency, the well-known JS Framework Benchmark guidelines, which are well accepted amongst the JavaScript community, were used in the benchmarking process.

Additionally, a general inconsistency among all codebases alike are bugs, which could have a detrimental impact on benchmarks and performance in general. This is to be expected, as all systems are prone to this issue if programmed by a human, and this is no exception. These bugs are generally mitigated by offering issue-tracking on the Lucia Github repository, where users are able to submit issue posts that the maintainer can see and take action on. Several preventative measures are also taken, notably unit, continuous integration, and ad-hoc testing.

Lucia has received notable feedback from the web development community. Some reception of Lucia includes tech startups like Weg-li and Glitch.com using Lucia in their production environments, and publications like JavaScript Weekly and Hacker news featuring Lucia to thousands of developers. Lucia additionally has received 470+ stars on Github, and over 20k+ unique concurrent visitors to the documentation at its peak, signifying developer interest and usage.

In the future, I plan to introduce more mutability in the API, while leveraging the performance of the core. This will allow the developer to program more intuitively, avoiding hacky or unrobust syntax and higher codebase maintainability. I also want to create more developer tooling for complex use cases, to accelerate and improve developer experience, as well as more promotion to extend developer awareness of Lucia.

Conclusion

Web development is a rapidly growing field, with JavaScript's influence covering not only the web, but branching into fields like IoT, mobile, and native desktop applications. This means that the impact of web developers is immense and the way they create applications is critical in today's society. Not only this, in the past ten years, the necessity for tools and libraries web developers use have greatly increased. Web programming has shifted from native imperative code to abstracted declarative code, allowing for scalability and consistency. However, the current mainstream libraries that developers commonly use when benchmarked are often large in size and are bloated. This could have a detrimental effect on performance and hinder positive user experiences. In the current divided climate of the JavaScript ecosystem, often subpar engineering or overengineering occurs, creating confusion among web developers which could proliferate into companies and result in malusage of resources. The pursuance of functionality and one-upping other libraries has resulted in the deprioritization of performance. Lucia allows for developers to reap the benefits of performance, but also being simple, lean, and powerful.

References

Chen, R., Li, S., & Li, Z. (2017, December 6). *From Monolith to Microservices: A Dataflow-Driven Approach* [From Monolith to Microservices: A Dataflow-Driven Approach]. ResearchGate. Retrieved October 12, 2020, from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.650.1531&rep=rep1&type=pdf

Fahland, D., Lubke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., & Zugal, S. (2009, June 18). *Declarative versus Imperative Process Modeling Languages: The Issue of Understandability* [Declarative versus Imperative Process Modeling Languages: The Issue of Understandability]. Eindhoven University of Technology, The Netherlands. Retrieved October 10, 2020, from https://www.win.tue.nl/~hreijers/H.A.%20Reijers%20Bestanden/emmsad09.pdf

Grov, M. (2020, May 20). *Building User Interfaces Using Virtual DOM* [Building User Interfaces Using Virtual DOM]. University of Oslo Department of Informatics. Retrieved October 10, 2020, from https://www.duo.uio.no/bitstream/handle/10852/45209/mymaster.pdf?sequence=7&isAllowed=y

Holland, B. (2016, January 11). *What To Expect From JavaScript In 2016 - Frameworks* [What To Expect From JavaScript In 2016 - Frameworks]. Telerik. Retrieved October 18, 2020, from https://www.telerik.com/blogs/what-to-expect-from-javascript-in-2016-frameworks

JavaScript [JavaScript]. (2002, December 14). Wikipedia. Retrieved October 17, 2020, from https://en.wikipedia.org/wiki/JavaScript

Jensen, S. H., Madsen, M., & Møller, A. (2011, June 11). *Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications* [Modeling the HTML

DOM and Browser API in Static Analysis of JavaScript Web Applications]. 19th ACM SIGSOFT symposium. Retrieved October 11, 2020, from https://users-cs.au.dk/amoeller/papers/dom/paper.pdf

JQuery [JQuery]. (2007, July 25). Wikipedia. Retrieved October 18, 2020, from https://en.wikipedia.org/wiki/JQuery

Kula, R. G., Ouni, A., German, D. M., & Inoue, K. (2017, September 15). *On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem* [On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem]. arXiv. Retrieved October 11, 2020, from https://arxiv.org/pdf/1709.04638.pdf

Kumar, A., & Singh, R. K. (2020, September 9). *COMPARATIVE ANALYSIS OF ANGULARJS AND REACTJS* [COMPARATIVE ANALYSIS OF ANGULARJS AND REACTJS]. International Journal of Latest Trends in Engineering and Technology. Retrieved October 10, 2020, from https://tv-prod.s3.amazonaws.com/documents%2Fnull-AngularReact.pdf

Mariano, C. L., B.E. Hons. (2017, January). *Benchmarking JavaScript Frameworks* [Benchmarking JavaScript Frameworks]. Technological University Dublin. Retrieved October 10, 2020, from https://arrow.tudublin.ie/cgi/viewcontent.cgi?article=1100&context=scschcomdis

Molin, E. (2016, October 16). *Comparison of Single-Page Application Frameworks* [Comparison of Single-Page Application Frameworks]. Diva Portal. Retrieved October 11, 2020, from https://www.diva-portal.org/smash/get/diva2:1037481/FULLTEXT01.pdf

Muyldermans, D. (2019, May 10). *How Does the virtual DOM compare to other DOM updating mechanisms in JavaScript frameworks?* [How Does the virtual DOM compare to other

DOM updating mechanisms in JavaScript frameworks?]. University of Dublin. Retrieved October 10, 2020, from http://www.daisyms.com/THESIS.pdf

Persson, M. (2020, May 28). *JavaScript DOM Manipulation Performance* [JavaScript DOM Manipulation Performance]. Diva Portal. Retrieved October 10, 2020, from https://www.diva-portal.org/smash/get/diva2:1436661/FULLTEXT01.pdf

A Strategic Analysis of Competition Between Open Source and Proprietary Software [A Strategic Analysis of Competition Between Open Source and Proprietary Software]. (2018, July 11). Journal of Management Information Systems. Retrieved October 10, 2020, from https://econwpa.ub.uni-muenchen.de/econ-wp/io/removed/0510004.pdf

You, E. (2014, February). *Github* [Github]. Github. Retrieved October 18, 2020, from https://github.com/vuejs/vue