Deno: A New JavaScript Runtime Environment

Samir Sherlekar, Avery Peiffer

*Department of Computer Engineering, University of Pittsburgh*

*3700 O’Hara Street, Pittsburgh, PA 15213, United States*

sas450@pitt.edu

aep65@pitt.edu

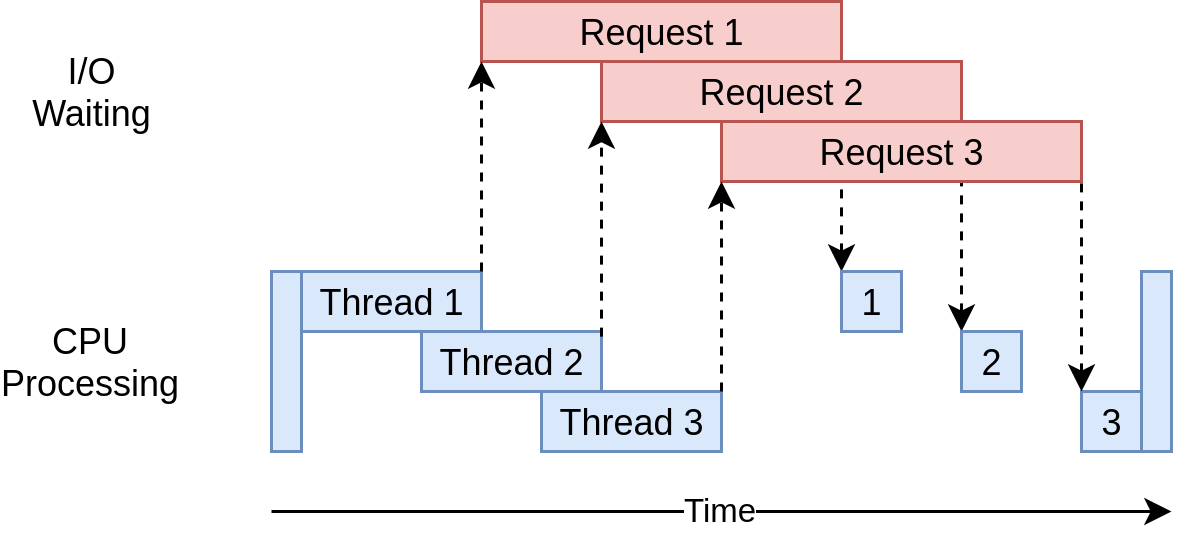
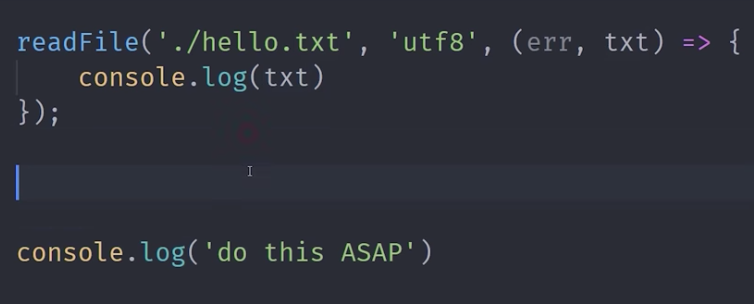
***Abstract*— JavaScript runtime environments are widely used among developers to improve efficiency in web servers. This paper examines the importance of runtime environments to modern application development and discusses the current foremost environment, Node.js, as well as Deno, a new framework designed to address its shortcomings.**

***Keywords***—**Deno, Node.js, JavaScript, Node Package Manager, runtime environment**

1. INTRODUCTION

Web servers are collections of software and/or hardware that are able to process requests from the internet using HTTP protocols. With between 20 and 30 billion devices connected to the internet currently, the challenge of scalability is a significant one for many serious web applications [1]. The two key metrics by which web applications are measured are latency, how long requests take to be completed, and throughput, the amount of requests that can be completed in a given time [2].

Traditionally, web servers manage an increase in requests through the practice of multithreading, in which new threads are spawned to execute operations in parallel. Theoretically, this practice should improve throughput significantly, as multiple requests can be processed simultaneously. However, this process of creating new threads has a high overhead cost and can become infeasible as servers have to deal with progressively more requests. As a result, JavaScript runtime environments have had to innovate their architecture to keep pace with the amount of requests created by 20-30 billion connected devices.

Fig 1: Diagram highlighting the use of multithreading to process requests [3]

1. METHODOLOGY

This section discusses the most popular JavaScript runtime environment, Node.js, its shortcomings, and how the desire for

innovation upon Node.js has led to the creation of the Deno environment.

*A.* *Node.js*

Node.js is a single threaded, asynchronous event-driven JavaScript runtime. The asynchronous event driven architecture means that function calls are typically non blocking. This is primarily achieved using the event loop.

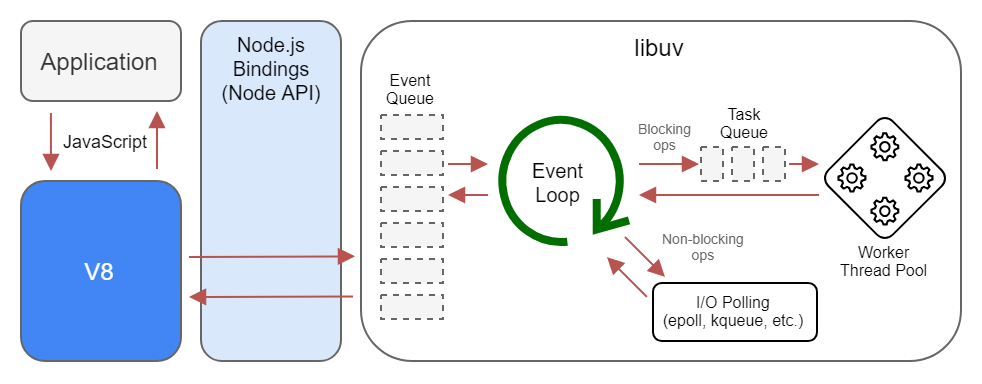


Fig. 2: Example of Node.js Architecture [4]

In the event loop, functions are queued up and then scheduled according to their priority and the time they will take to finish. This method of scheduling maximizes throughput while incurring minimal overhead, unlike that seen in multithreading. A concrete example is shown below in Figure 3. The sample program will output ‘do this ASAP’ before it outputs the contents of the text file in the readFile function. This occurs because the event loop has scheduled the readFile function to be executed after the rest of the program.

Fig. 3: Code snippet that illustrates the event loop used by Node.js.

The actual program that runs JavaScript code is called an engine. Early JavaScript engines only served as interpreters, similar to the interpreter that runs Python code. Interpreters are incredibly inefficient; they must analyze each statement each time it is executed before performing the action. In comparison, compilers, used for languages such as Java and C++, analyze statements beforehand and then can perform actions within a fixed context [5]. This means that JavaScript, as an interpreted language, saw major efficiency issues when run on the earliest engines.

Google’s development of its aptly-named V8 JavaScript engine provided the opportunity for Node.js to radically improve in efficiency. Developed for use in Chromium-based browsers, the V8 engine makes use of just-in-time compilation to translate JavaScript instructions to machine code before executing them, shown below in Figure 4. The instructions are then cached for reduced compilation time in future executions [5]. Translating instructions to machine code means that they can be executed directly by a CPU, as opposed to the interpreter. This incurs less memory and CPU usage and can therefore increase throughput when handling requests.Node.js is built to make full use of the V8 engine, meaning its JavaScript code can often execute faster than that of compiled languages [5].

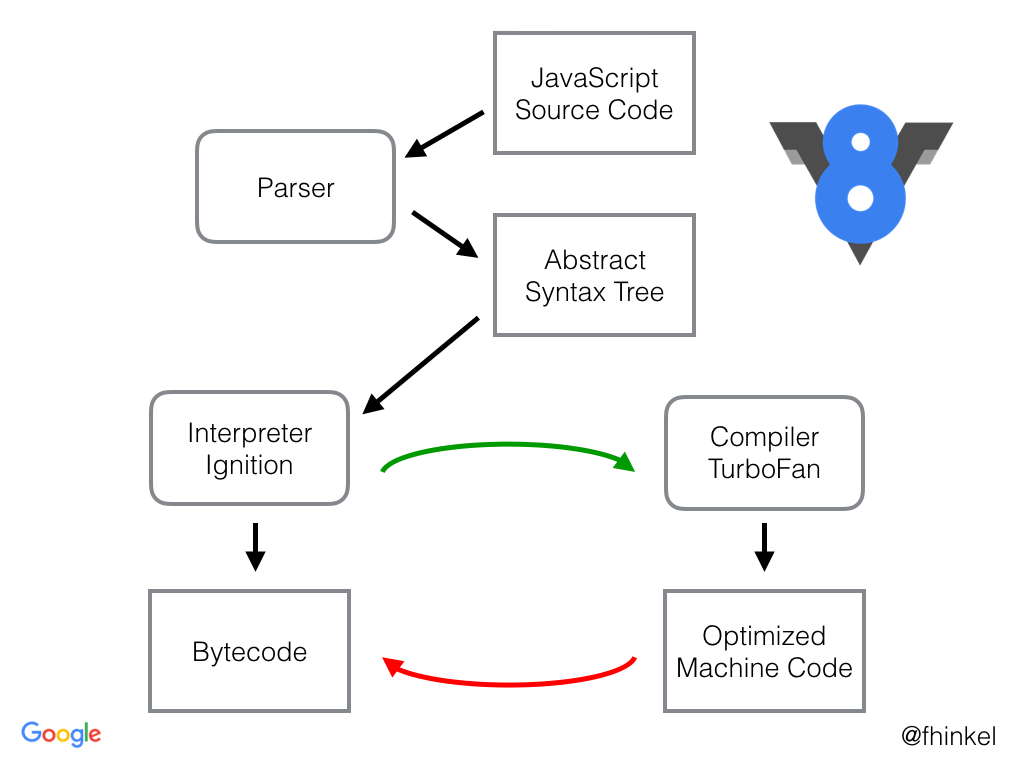


Fig. 4: Architecture of Google Chrome’s V8 JavaScript engine [6]

*B. The Shortcomings of Node.js*

It is not surprising that Node.js, originally released in 2009, lacks the infrastructure necessary to match the ever-evolving needs of the current software development industry [7]. The creator of Node.js, Ryan Dahl, gave a talk titled *10 Things I Regret About Node.js* at JSConf EU in 2018 [8]. One of his key issues with the environment revolves around security. The modules and packages installed with Node.js are not sandboxed, which means they are not run separately and can affect each other. As a result, a project’s linter (a tool to check and maintain code style) can theoretically access the machine and network on which the program is run. This presents a very realistic security risk when using Node.js.

Additionally, Dahl regrets that the packages and modules imported by Node.js are managed and maintained by Node Package Manager (NPM). This is a centralized authority for Node.js modules and has recently been acquired by Github, itself recently acquired by Microsoft. Because all packages used by Node.js are maintained by NPM, an entire web application can depend on NPM’s integrity. If NPM itself has issues or is compromised by a malicious entity, web applications that use it can be severely punished. Further shortcomings of Node.js will be discussed in the next section, though placed in comparison to its successor. Though many of Dahl’s regrets about Node.js are somewhat low-priority, it has become obvious that it is a relic of the past decade; an opportunity exists to take the foundation of Node.js and modernize it for the future of web application development.

*C. The Creation of Deno*

*1) Overview and security:* In that same 2018 talk, Ryan Dahl announced that work had begun on a new runtime environment that would address his mentioned problems with Node.js. Deno, the name for this environment, is written in Rust as opposed to C++, which provides Deno with increased speed and memory safety [9]. Currently, Node.js requires the use of a TypeScript configuration file to transpile code to executable JavaScript. However, Deno supports both TypeScript and JavaScript immediately out of the box, eliminating the need for this unnecessary file [8]. Deno is also built on top of the Google V8 JavaScript engine, providing it with the same CPU and memory benefits that are reaped by Node.js.

Despite the traits shared by both Node.js and Deno, there are key differences between the two that cement the latter’s status as an innovation upon the former. The foremost issue Deno attempts to solve concerns security permissions. Deno requires explicit permissions, meaning that the runtime has no access to “the file system, the network, execution of other scripts, and the environment variables” [8]. To gain access to other parts of the environment, the programmer has to explicitly grant permission, typically with the usage of a flag such as --allow-write, --allow-env, or --allow-run. Otherwise, Deno will prompt the developer each time access to an external entity is needed. The developer can then choose to grant access once, always, or deny access entirely. This principle of requiring explicit permissions from the programmer promotes the principle of least permissions for modules within server side JavaScript, and directly addresses the security concerns with Node.js.

*2) Modules and NPM:* There is a distinct difference between Deno and Node in the way that each loads third-party modules. In Node’s setup, the developer has to download a package from NPM, which gets added to the project’s package.json file. Then, every time the program is executed, a package resolution algorithm finds the correct package, resolves it within the package.json setup file, and then find the package within the NPM registry [8]. Deno aims to import modules similarly to the manner by which browsers fetch content in a script tag; that is, Deno allows the developers to import modules directly from a URL. This eliminates the need for the confusing and bloated package.json file and node\_modules directory.

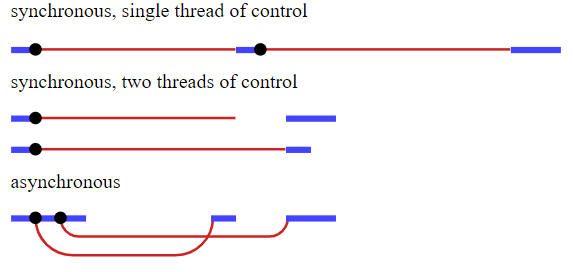
Theoretically, directly importing modules from a URL can lead to issues. If the website from which the developer is importing goes offline for whatever reason, that package would then be useless and cause serious ramifications throughout the program. Thankfully, Deno takes steps to preventing this from happening by caching the downloaded modules. This leaves the developer with the last working copy of the imported module, preventing such errors with websites crashing [8].

Directly importing modules has the extra benefit of breaking free from the centralized repository of modules, NPM. Though NPM can serve as a convenient source for finding all of these third-party modules, its reach makes it a single point of failure that can seriously affect applications that depend on it. Although NPM is maintained by Github and Microsoft, it has gone offline for hours multiple times in the past ninety days, with serious consequences for every app that uses NPM to import modules [10].

This dependency issue with NPM was made clear on March 23, 2016, when a developer named Azer Koçulu unpublished over 250 of his open-source modules from NPM [11]. He removed his modules in anger after NPM forcibly unpublished one of his modules following a legal dispute with an instant messaging service about its name. Unfortunately, one of the modules removed by Koçulu in protest was an incredibly popular module called left-pad. From February to March of 2016, left-pad had been fetched nearly 2.5 million times, and was used by thousands of its projects [11].. Its absence caused all of these projects to fail, creating a confused fervor among developers. NPM eventually forcibly re-published that module three hours later to remedy the issue, though that brought up concerns about its role as a centralized repository [11].

Overall, Deno’s caching of downloaded modules would not work with the NPM repository, as there are too many modules within the repository that would need to be cached. This allows Deno to distance itself from the competing forces of NPM’s corporate ownership and the developers who contribute to its open-source packages. Therefore, Deno’s independence from NPM allows it to remove a single point of failure from applications, while presenting solutions for the smaller amount of module issues that do arise.

*3) Asynchronous code, callbacks, and promises:* At the beginning of one’s study of software development, programming is typically modeled synchronously; this means that the program must wait for one event to finish executing before the next event can be executed. Synchronous programs can be bogged down by one or more functions that take a considerable amount of time to execute, slowing down the overall program’s execution time [12].

Fig. 5: Visualization of synchronous and asynchronous programming [12]. The blue lines represent the time spent executing actions, while the red lines represent time spent waiting for actions to complete.

By contrast, asynchronous programming allows multiple events to happen at the same time [12]. When an event is started, the main program continues to run and gains access to the results of that event after it has completed. While this programming model can result in a significant decrease in execution time, its increased complexity requires some additional infrastructure to be added to a programming language’s architecture.

The stock version of Node.js uses callbacks to accomplish asynchronous operations. Callbacks are functions called after a given task has been completed; this prevents the program from waiting and allows other code to be run in the meantime [13]. Though somewhat intuitive, callbacks can quickly lose their usefulness as more are added. *Callback hell*, shown in Figure 6, is an affectionate way of referring to the visually unappealing code that results from too many callbacks [14]. Ironically, this occurs when writing code for operations that are optimal for synchronous programming; since the individual operations *must* be completed in succession, they are forced to be nested inside one another.

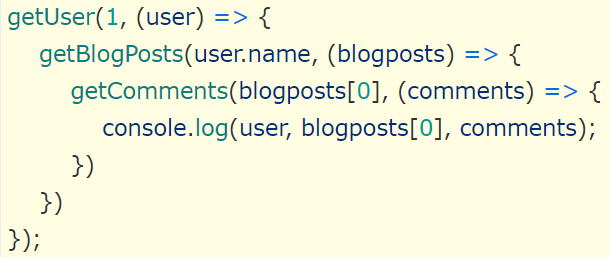


Fig. 6: A small example of callback hell [14].

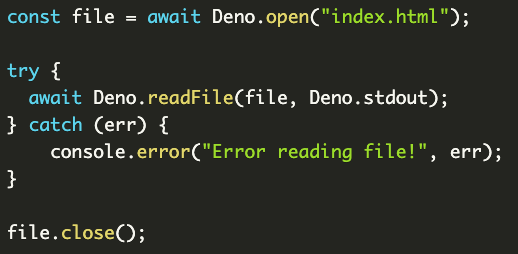
Though not native to Node.js, general JavaScript makes use of promises to alleviate the problems caused by callbacks. A promise is essentially “a proxy not necessarily known when the promise is created” [15]. This allows asynchronous methods to act similar to synchronous methods; “instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future” [15]. All promises have access to the then() and catch() methods, which are used for further action after the promise has succeeded or failed. The then() method specifically allows promises to be chained together, further allowing the asynchronous code to appear synchronous.



Fig. 7: A collection of chained promises that resemble the general structure of synchronous code [15].

Node.js does not support promises out of the box; rather, the use of promises requires a specific library which is not identical to promises in ordinary JavaScript [16]. This required use of an external library means that Node.js is not an extension of regular JavaScript, as it should be. Node.js creator Ryan Dahl mentioned this issue in his aforementioned talk, stating that he added promises to Node.js in June 2009 but “foolishly” removed them in February 2010 [16]. As a result, several Node.js APIs dealing with asynchronous code have aged badly because there is no unified standard by which Node.js is to work with promises. .

Dahl has made no such mistakes with Deno, which supports promises out of the box in a simple, intuitive manner. These promises are implemented using the syntax *async* and *await*. When placed before a function, the *async* keyword indicates that the function will always return a promise [17]. Likewise, the *await* keyword, which can only be used inside functions labeled with *async*, makes the program wait until a promise settles and returns its result. Shown in Figure 8, this implementation bears a striking resemblance to synchronous code, making it intuitive to read and understand. Natively implementing asynchronous programming while not requiring the use of callbacks means that Deno can remain standardized with the rest of the JavaScript community.

Fig. 8: A brief implementation of *async* and *await* in the Deno environment [18].

*4) TypeScript:* TypeScript, a programming language that is a superset of JavaScript, has seen a significant rise in popularity among the web development community. The language, released in 2012, is intended to address JavaScript’s issues with creating scalable applications [19]. Most notably, TypeScript is a statically typed language, making it easier to find bugs related to variable typing than in the dynamically-typed JavaScript [19]. Since TypeScript is a superset of JavaScript, existing projects in JavaScript can adopt the usage of TypeScript with little additional effort. The language is well-suited for widely-used web applications and is used by organizations such as Slack, Angular, and Accenture [19].

As mentioned earlier, Node.js does not natively support TypeScript, instead requiring the use of a separate configuration file for the language. Deno, on the other hand, natively supports both JavaScript and TypeScript by default, requiring no further actions from developers [20]. Deno will automatically transpile TypeScript code to JavaScript, caching the most recent versions of the program to reduce the time spent transpiling code and decrease execution time [21]. With its policy on TypeScript, Deno demonstrates the strength of its flexibility; as TypeScript becomes a more appealing language for writing web applications, Deno will, in turn, become a more appealing runtime environment, while still retaining support for regular JavaScript.

IV. CONCLUSIONS

In conclusion, Deno is a very promising runtime environment for web applications that aims to address several issues of Node.js. However, it remains to be seen whether it provides enough of a benefit over Node.js to incentivize developers to switch to a new runtime environment, a process that is likely mired with inconveniences in learning Deno’s rules and conventions. It would not be surprising to see Deno be adopted gradually, as developers start new projects and wish to branch out. It could very well overtake Node.js, but that is unlikely to happen in the near future, simply due to Node.js’ status as an established presence in the field of JavaScript runtime environments.

REFERENCES

1. *Number of connected devices reached 22 billion, where is the revenue?,* May 2019. Accessed on Jul. 15, 2020. [Online]. Available: <https://www.helpnetsecurity.com/2019/05/23/connected-devices-growth/>
2. T. Keary, *Latency vs Throughput - Understanding the Difference,* June 2020. Accessed on Jul. 15, 2020. [Online]. Available: <https://www.comparitech.com/net-admin/latency-vs-throughput/#:~:text=Latency%20indicates%20how%20long%20it,they%20work%20within%20a%20network.>
3. J. Anderson, *Speed Up Your Python Program With Concurrency.* Accessed on Jul. 15, 2020. [Online]. Available: <https://realpython.com/python-concurrency/>
4. A. Keshishyan, *Node.js Event loop architecture*, Mar. 2019. Accessed on Jul. 20, 2020. [Online]. Available: <https://medium.com/preezma/node-js-event-loop-architecture-go-deeper-node-core-c96b4cec7aa4>
5. A. Zlatkob, *How JavaScript works: Inside the V8 engine + 5 tips on how to write optimized code*, Aug. 2017. Accessed on Jul. 20, 2020. [Online]. Available: <https://blog.sessionstack.com/how-javascript-works-inside-the-v8-engine-5-tips-on-how-to-write-optimized-code-ac089e62b12e>
6. K. Duarte, *Basics of understanding Chrome’s V8 Engine*, Sep. 2018. Accessed on Jul. 20, 2020. [Online]. Available: <https://medium.com/@duartekevin91/basics-of-understanding-chromes-v8-engine-c5c8ec61fa6b>
7. Node initial release. Accessed on Jul. 20, 2020. [Online]. Available: <https://github.com/nodejs/node-v0.x-archive/tags?after=v0.0.4>
8. R. Dahl, *10 Things I Regret About Node.js*, Jun. 2018. Accessed on Jul. 20, 2020. [Online]. Available: <https://www.youtube.com/watch?v=M3BM9TB-8yA>
9. A. Lozovsky, *Rust vs C++ Comparison*, Feb. 2018. Accessed on Jul. 20, 2020. [Online]. Available: <https://www.apriorit.com/dev-blog/520-rust-vs-c-comparison>
10. NPM status. Accessed on Jul. 20, 2020. [Online]. Available: <https://status.npmjs.org/>
11. C. Williams, *How one developer just broke Node, Babel, and thousands of projects in 11 lines of JavaScript*, Mar. 2016. Accessed Jul. 20, 2020. [Online]. Available: <https://www.theregister.com/2016/03/23/npm_left_pad_chaos/>
12. *Asynchronous programming.* Accessed Jul. 25, 2020. [Online]. Available: <https://eloquentjavascript.net/11_async.html>
13. *What are callbacks?* Aug. 2011. Accessed Jul. 25, 2020. [Online]. Available: <https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/>
14. *JavaScript Callbacks vs Promises vs Async Await.* Accessed Jul. 25, 2020. [Online]. Available: <https://vegibit.com/javascript-callbacks-vs-promises-vs-async-await/#:~:text=Callbacks%20are%20used%20when%20you,These%20are%20asynchronous%20operations.>
15. *Promise*. Accessed Jul. 25, 2020. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>
16. *Native Support for Promises in Node.js*, Feb. 2014. Accessed Jul. 25, 2020. [Online]. Available: <https://stackoverflow.com/questions/21564993/native-support-for-promises-in-node-js>
17. *Async/await*, Feb. 2020. Accessed Jul. 27, 2020. [Online]. Available: <https://javascript.info/async-await>
18. M. McKeaveney, *Deno: A Simple Guide*, May 2020. Accessed Jul. 27, 2020. [Online]. Available: <https://www.martinmck.com/posts/deno-a-simple-guide/>
19. M. Nowak, *Exploring TypeScript from a Business and Software Development Perspective*, Apr. 2020. Accessed Jul. 30. 2020. [Online]. Available: <https://www.monterail.com/blog/typescript-business-development#:~:text=Because%20TypeScript%20calls%20for%20assigning,familiarizing%20themselves%20with%20a%20project.>
20. D. Sasidharan, *Forget NodeJS! Build native TypeScript applications with Deno*, Feb. 2020. Accessed Jul. 30, 2020. [Online]. Available: <https://deepu.tech/deno-runtime-for-typescript/>
21. Deno Manual, *Using TypeScript*. Accessed Jul. 30, 2020. [Online]. Available: <https://deno.land/manual/getting_started/typescript>