

**Software Maintenance and Evolution**

Performance Analysis of Event-Triggered ECMAScript (Javascript)

Riccardo Rigoni (21-712-393), Maximilian Huwyler (13-925-557)

University of Zurich

Supervisor: Seán Murphy (ZHAW)

17. December 2021

This project was conducted as partial requirement of the University of Zurich Software Maintenance and Evolution module taught during the Autumn Semester 2021. The module responsible is Dr. Sebastiano Panichella. Further the work was performed in collaboration with the company IoTIFY[[1]](#endnote-2) which is interested in lightweight function execution and event-based compute paradigms. The project supervisor, Sean Murphy, works with IoTIFY as well as ZHAW.

1. Introduction

Technological innovation continues to evolve at an incredible pace affecting all aspects of our lives from the way we communicate to the way we drive and walk around cities to the ways we perform medical diagnoses and how we run out factories. Much of this can be attributed to constant advances in computing and networking capabilities over the last decades. New basic technologies and new problem spaces continually give rise to new programming paradigms and new ways to express computing solutions.

One such paradigm which has quite recently been developed is serverless computing: this is primarily an event driven mechanism where small computing jobs are typically executed based on event triggers. While this has been around for some years now, there are still many open issues in this technology space.

JavaScript has been around for almost 30 years, but this does not mean that its uses have been already exhausted. On the contrary, Typescript for example is a rising language that builds on JavaScript and can be used to build client-side as well as server-side. The potential of using JavaScript as a server-side language is not fully realized. There is a growing interested in using JavaScript for server-side website programming with frameworks like Express.

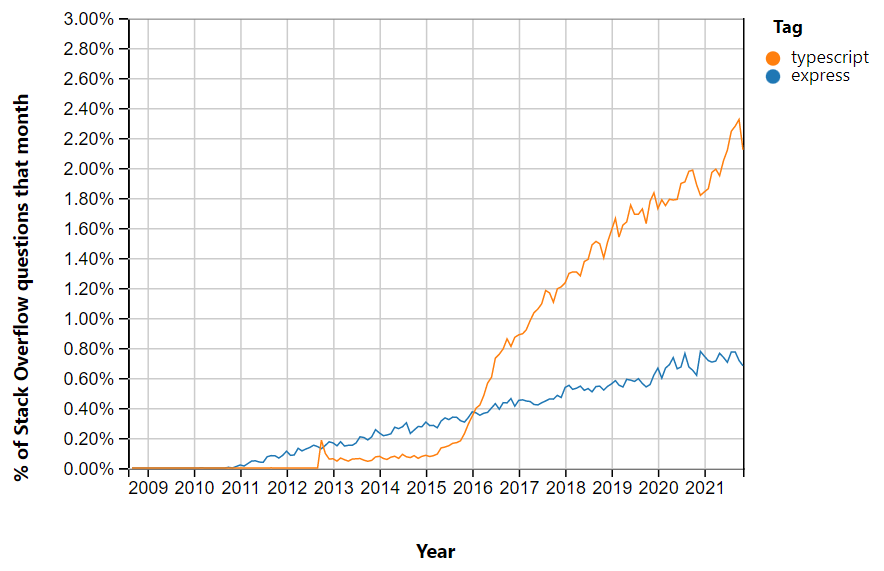


Figure 1 - Stackoverflow Trends Typescript & Express[[2]](#endnote-3)

In this project we want to go further and combine the concept of using JavaScript as a language for server-side applications with serverless computing and the paradigm of FaaS. A paradigm for developing cloud-based applications, where users upload code of their functions to the cloud, which are executed when triggered by an event[[3]](#endnote-4). Like using JavaScript as a server-side language, there is a growing interest in FaaS and serverless computing, which can be seen in the increasing relevance of one of the main providers for such services AWS-Lambda.

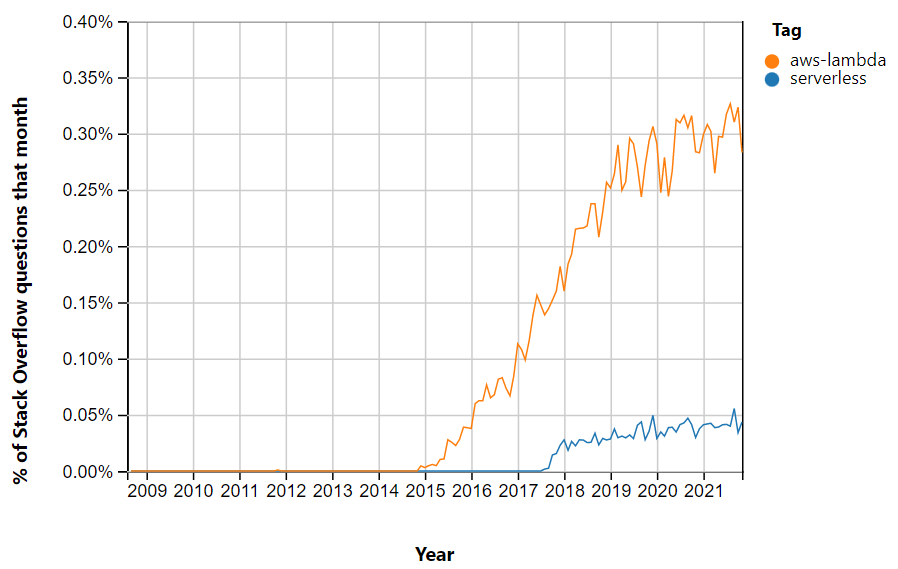


Figure 2 - Stackoverflow Trends AWS-Lambda & Serverless[[4]](#endnote-5)

The growth in IoT system also aligns with those trends. Since in IoT systems, many devices are generating a lot of small data the event-triggered aspect of the above-mentioned concepts becomes increasingly interesting. Generating data can be used as a trigger to execute lightweight function, which process the data needed by an IoT application. The serverless aspect helps the scalability of IoT systems, which can become increasingly difficult with the growing size of a project. When using a service like AWS-Lambda, the question of scalability becomes a question of cost rather than a question of whether it is possible to maintain the system. Since cost is determined on the used resources (e.g., time measured for execution), it is crucial to understand how the function execution can be performed in the best way to optimize resource usage. In this project we are interested in how this can be done using new technologies to run JavaScript for server-side application.

We started with an initial list of project objectives:

* Conducting a qualitative review of different JavaScript runtimes/engines involving evaluating published material and documentation
* Implement lightweight management systems around two reviewed runtimes, which can invoke functions based on external event triggers
* Compare the runtimes based on raw throughput for different functions triggered by the lightweight management system
* Document the process on GitHub

After the completion of the first objective, we ended up with the two promising candidates Deno and QuickJS, which should be embedded using Rust. It quickly became clear that the configuration using QuickJS was several magnitudes slower than the configuration with Deno using the JIT feature of the V8 engine (QuickJs vs. Deno). Consequently, investigating QuickJS any further was dropped, and the focus was shifted towards a closer look on Deno. The main goal of the project became to implement a lightweight function execution management system in Rust using NATS and Deno and to provide recommendations on how to properly do it. The main idea behind the new scope, is that current FaaS providers fail to provide efficient resource management in their systems, leading to costly function executions, both in term of memory and time. Identified dimensions of investigation for the recommendations are (optional written in italics):

* Security and performance comparisons between Main Worker and Web Workers
* Management of multiple v8 engines within Rust/Tokio
* *Analysis of function scheduling when functions perform IO operations*
* *Dynamic addition of web workers to running engine*
* Load testing on a single node
* Load testing on multiple nodes

To better understand the issues involved in the implementation of a serverless execution environment, a prototype of the system using Deno and NATS have been implemented. All the difficulties and the considerations have been reported and summarized to generate the set of recommendations.

The report proceeds as follows:

* Chapter 2 considers initial review on how it became apparent that we had to change the object of our project.
* Chapter 3 is about the technologies used to develop the serverless execution environment, namely NATS and Rust.
* Chapter 4 documents of the implementation process and the tests conducted.
* Chapter 5 presents the recommendations that can be done based on the implementation and the experimentation.
* Chapter 6 discusses lecture relevant learnings and suggestions.
* Chapter 7 summarizes the project.
* Chapter 8 contains the appendix.

All content of this project including project objectives, meeting notes, the qualitative reviews, the report on hand and the source code written for experimentation can be found in the GitHub Repository at <https://github.com/gavi210/UZH_ECMASCRIPT_PROJECT>.

1. ECMAScript Runtimes

For the initial qualitative review, the Wikipedia list of all ECMAScript runtimes/engines[[5]](#endnote-6) was taken as a starting point. Among all mentioned projects, all those discontinued, deprecated, with a different usage than needed for the project or an extremely poor documentation were filtered out. Remaining runtimes/candidates are mentioned in *Table 1 - Runtime Comparison*.

We then gathered more information on them using the following set of criteria:

* ECMAScript compatibility
* Size
* Ease of integration (languages to embed them)
* Whether they were open source
* WASM support
* Ability to precompile scripts
* Intelligence within runtime (runtime optimization)
* Support for isolation
* Multithreading support
* Planned support (Health of the project)

Those criterias were used to choose two candidates for the further continuation of our project.

* 1. Evaluation Process

Js-interpreter does not support an ECMAScript version late enough for our investigation; therefore, it has been discarded. Finding the exact sizes of the runtime proved to be very difficult. However, it was possible to divide the candidates into the two camps of lightweight and heavy runtimes/engines. Embedding was crucial for the implementation; therefore, ease of integration criteria was very important. We decided to go for a candidate that can be embedded using C++ or Rust. C++ was chosen since it is supported by many candidates and Rust because of its novelty and effective variable management system. This latter is composed by a strict set of rules that limits variable scope, visibility, ownership, and modifiability. Thanks to them, the compiler could statically detect improper variable manipulation, that may result in bugs or unexpected application’s behavior. The compiler provides the most support in developing a multithreaded application - which is the case in out project -, since it statically identifies all those operations that may result in racing conditions or subtle threading-specific bugs very tedious to fix. Consequently, to a-priori avoid such implementation errors statically detected is very handy since less time would be spent fixing the code rather than researching. We did not want to consider closed-source project, because of the lack of community supporting them, in case we needed help. Fortunately, only one project was ruled out by this criterion. Even though WASM support was not an important criterion for the implementation of our project, we preferred candidates supporting this promising technology. The ability to precompile scripts and runtime intelligence were some of the most important criteria, since they indicated that these candidates would have better performance. Support for isolation was not necessary, but preferred. Multithreading support also indicated that a better performance can be achieved, but because of the unfamiliarity of the students with multithreaded JavaScript execution at this time, it did not influence our decision. Planned support was difficult to measure and almost impossible to compare. Therefore, we favored candidates, which had momentum on GitHub or were declared as active projects but didn’t use this criterion to rule out any.

In the end we chose one lightweight and one heavy candidate such that the comparison would yield clear results on performance. We decided to continue with the QuickJS because it was the only lightweight candidate with optimization features and could be embedded using Rust. The second candidate was Deno. It could also be embedded with Rust and supported all the features we assessed as relevant for our project.

* 1. QuickJs vs. Deno

After the initial qualitative comparison between the runtimes and engines, quantitative analysis was conducted, to assess the trade-off between performance and memory impact between the chosen technologies. Source code for the comparison could be found in the GitHub project’s repo[[6]](#endnote-7).

Results obtained by executing the same function in both engines and measuring the execution time showed clear results*. Table 5 - Comparison QuickJs vs. Deno performance* reports the values. The results obtained were in-line with other investigations conducted by the QuickJs community themselves[[7]](#endnote-8). *Table 6 - QuickJs Benchmark* reports some key values of their investigation. Both performance comparisons proved that QuickJs performs poorly against Deno (V8 engine) therefore, it was clear that further investigation was not needed. QuickJs’ performances are too low for practical use in places, where memory for the runtime is not an extremely scarce asset.

1. Technology

During our project, we were introduced to two very interesting technologies: NATS and Deno. NATS was used for communication and Deno was used as the JavaScript runtime. In the following we give an overview of both technologies.

* 1. NATS

The NATS technology is already well documented on the NATS documentation[[8]](#endnote-9). Because of the time constraints at the end of project, the following explanations of concepts were copied from the NATS documentation. A few cifilling errors were corrected, and a few minor changes were made to fit the reading flow.

* + 1. NATS Overview

NATS is a connective technology responsible for addressing, discovery and exchanging of messages. It aims to support distributed systems communications by providing a consistent and efficient solution to the most common communication patterns. These are:

* Asking and answering questions, aka services/microservices
* Making and processing statements
* Data-streams processing.

By means of NATS technology, developers could:

* Effortlessly build distributed and scalable client-server applications
* Store and distribute data in real-time in a general manner. This can flexibly be achieved across various environments, languages, cloud providers and on-premises systems.[[9]](#endnote-10)

Modern distributed systems are composed by an ever-increasing number of devices hyper-connected with each other. Therefore, they require a connective technology that guarantees location independence and mobility. With its unique and revolutionary architecture, NATS performs very well in this context and achieved to ensure high-level security without affecting performances.

Thanks to its properties, NATS fits very well in these contexts:

* Cloud messaging
* Command and control: IoT and Edge, Telemetry / Sensor Data / Command and Control
* Augmenting and replacing legacy systems[[10]](#endnote-11)

Further on follows a more technical overview of the NATS technology.

* + 1. Data Exchange Process

NATS implements the so-called "message-oriented middleware", meaning that it is a software infrastructure powering efficient data sharing with an exchange of messages among computer applications and services. Data transmission is achieved by mean of a NATS client library, through which applications’ code could perform publish, subscribe, request, and reply operations7.

Communicating NATS Clients are timely and physically decoupled from each other and could belong to separate instances of the same application or to different applications. Time independence is achieved by making the communication asynchronous. A publisher does not have to lock and wait for a response from a receiver and could proceed its own execution after the publish operation. Location independence is achieved through subjects. Subjects are string-identified NATS entities, that are used to identify resources over the network without knowing their IP address. Close similarity in approach could be found in TIBCO **Rendezvous**. *Figure 3 - Subject-Based Messaging* provides an example of communication flow using such Subjects happens.

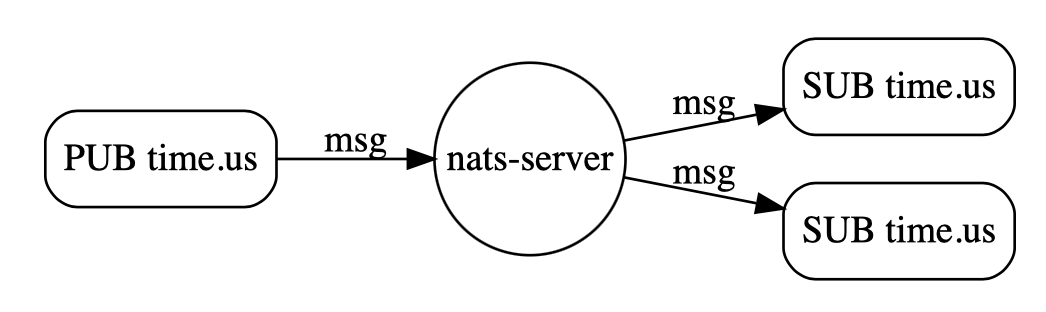


Figure 3 - Subject-Based Messaging[[11]](#endnote-12)

Within the NATS architecture, subjects are managed by the central NATS server. Clients could subscribe to a subject, stating their interest for all messages directed to it. The server will then broadcast all messages sent to a subject to all interested clients. *Figure 6 – NATS Subscription* and *Figure 7 – NATS Routing* show such process.

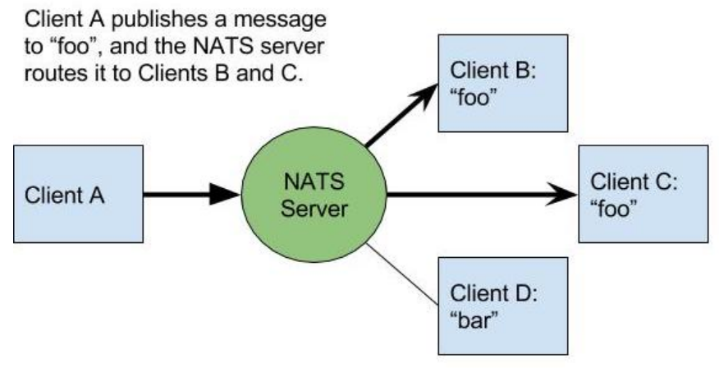


Figure 4 - NATS Subscriptions10

Diagram

Description automatically generated

Figure 5 - NATS Routing10

By mean of the above-described message flow, senders and receivers don’t have to know their respective IP addresses. The receivers themselves inform the server about their current IP address during subscription phase and with the IP known, the server then broadcasts messages on behalf the senders.

* + 1. Cluster of Servers

Sharing of messages are provided by one or more NATS server processes that are configured to interconnect with each other and provide a fault-tolerant NATS service infrastructure. Server instances route messages to NATS clients - applications that use the NATS protocol (usually via a NATS client library) to connect to the NATS server. Logically, applications communicate over a message bus, but the network configuration is the standard TCP client-server model. *Figure 6 - NATS Middleware7* provides a representation of the logical bus.

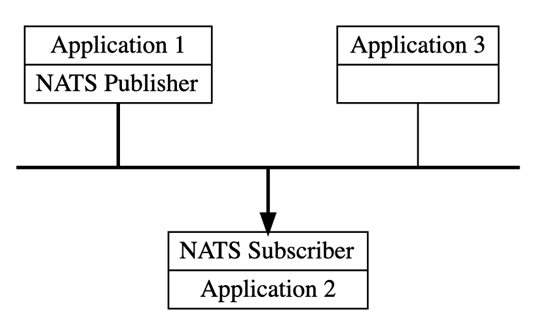


Figure 6 - NATS Middleware7

Running a single NATS server introduces a single point of failure. To provide high availability, scalability and reliability, server instances are connected in a full mesh. *Figure 8 - NATS Cluster* provides an illustration of such a mesh. With such a strict servers’ connection, crucial properties are ensured: there is a one-hop message routing maximum and messages will never loop throughout a cluster. The servers communicate with each other using a server-to-server clustering protocol over a TCP connection. The protocol supports "discovery" to propagate topology information and changes in real-time with other members of the cluster and clients. Thus, servers can be dynamically allocated or removed with zero downtime10.

Diagram

Description automatically generated

Figure 7 - NATS Cluster10

It is important to note that from a client perspective, a NATS cluster is considered one entity. An officially supported NATS client only requires addressing one server in the cluster to have full connectivity. At subscription phase, the servers share the full topology of the system, so to make aware each client about the entire backend system. In case of server failure or network partition, the client can fail over by interacting with the other servers in the cluster[[12]](#endnote-13).

* + 1. JetStream

A remarkable limitation simple publish/subscribe communication is time coupling of the clients – both sender and receiver must exist at the same time, otherwise, messages are lost. To realize time decoupling, NATS introduced a built-in distributed persistence system called JetStream. The new technology provides a mean to store and maintain at server side all published messages for one or more subjects. Subscribers (JetStream consumers) can access the stream at any time and to replay/consume messages on one ore more subjects. The different options are

* All the messages stored in the stream, either as fast as possible or in the original rate
* Only the last message stored in the stream or last message for each subject
* Starting from a specific sequence number
* Starting from a specific start time[[13]](#endnote-14)

To scale, streaming servers can be partitioned such that multiple streaming servers in the same cluster distribute work based on assigned channels.

Diagram

Description automatically generated

Figure 8 - Streaming Partitioning10

Streaming uses a different protocol, which is more complex. The typical flow of a NATS streaming client is to first establish a connection to a streaming server. Optionally subscribe to subjects and setup handlers to process messages. Acknowledge messages. Optionally publish messages and handle publish acknowledgements from the server. At last, close the connection with the NATS streaming server10.

* + 1. Key/Value Store

JetStream also enables client applications to create key/value ‘buckets’, which are used as consistent and persistent associative array. The available operations that can be performed with key/value buckets are put, get, delete, purge (clear), create (same as put but currently no value associated with key), update, and keys (get copy of all keys). To control the buckets, they can be limited by size, size of a single value, or time-to-live of values. Key/value buckets have even functionalities that a typical key/value store does not have: You can watch for changes happening for key like subscribing to the key or retrieve a history of the values associated with each key over time[[14]](#endnote-15).

* 1. Deno

The internals of Deno are already well documented in the gitbook *Internals of Deno[[15]](#endnote-16)*. Because of the time constraints at the end of project the following explanations of concepts were copied from the above mentioned gitbook. A few spelling errors were corrected, and a few minor changes were made to fit the reading flow.

Deno is a simple, modern, and secure runtime for JavaScript and TypeScript that uses V8 and implemented in Rust.

Its main features are:

* Security: The runtime is secure by default. File, network, and environment access are not enabled if not explicitly specified through permissions
* Sandboxing: Untrusted JavaScript and TypeScript code could be evaluated in an isolated environment, to increment system security
* Out of the box TypeScript support
* Deno ships a single executable file, to make its deployment easier
* The runtime comes with a set of audited standard modules that can be found and imported from [deno.land/std](https://deno.land/std)[[16]](#endnote-17),[[17]](#endnote-18) and general ES Modules could be loaded from the file system

This technology is a perfect candidate for our project, since it comes with an additional set of handy properties:

* It is modern, open source and has an active community supporting the development
* To embed Deno using Rust is very simple and straightforward (deno\_core, deno\_runtime Rust crates)
* The V8 technology succeeds in executing JS and TypeScript code efficiently

To understand how Deno succeeds in providing the above-mentioned features, it follows a technical description of its architecture, with further focus on the most relevant components heavily used during the project.

* + 1. Architecture

Deno has eight major components as shown in *Figure 8 - Deno Architecture* of which five are from Deno itself and three are from third parties. Their roles in the system could be summarized as follows:

* CLI: It glues together all other components and orchestrate them to execute code. It is the main entry point, since implements major functionalities provided by the system. It is in charge to execute TypeScript code
* Runtime: Implemented both in Rust and JavaScript, it provides JavaScript evaluation capabilities. It comes with a set of handy components, such as low-level ops, inspector, metrics, main worker, web worker and permissions
* Core: It implements the bindings between Rust-implemented and JavaScript-implemented components of the system, and provides a set of low-level functionalities, such as v8 bindings, flags, modules, JsRuntime, zero, shared queue, etc
* STD: It is the Standard Library for the additional utilities and functionalities provided by the system. It supports fully asynchronous interaction with the runtime
* Tokio: It manages and orchestrates all the internal operations and makes them asynchronous.
* Rusty\_v8: It provides high-quality v8 C++ bindings
* v8: It efficiently evaluates JavaScript code

By mean of *Figure 9 - Deno Architecture*, the two sets of system’s components could be easily identified. On one hand, those highlighted in yellow are Deno-specific elements and they compose together the so-called *Deno Core*. Attention must be paid on terminology, since this latter name has not to be confused with the *Core* component itself. The third-part libraries are highlighted in green, and they are: Tokio, Rusty\_v8 and v8 engine. These libraries have been implemented independently from the runtime itself, and they are included in the system as *self-contained* *black boxes*.

These two sets of components interact together to provide TS/JS execution capabilities. Bindings between them are high-level and operate in sync, extending each other’s capabilities.

On its side, the *Deno Core* provides high-quality engine management by allowing runtime’s inspection, linter, files fetching, modules resolution and standard libraries. Furthermore, it extends the v8 with file system interaction, networking and io capabilities. On its side, the v8 efficiently evaluates JavaScript code, by achieving goal of the engine[[18]](#endnote-19).

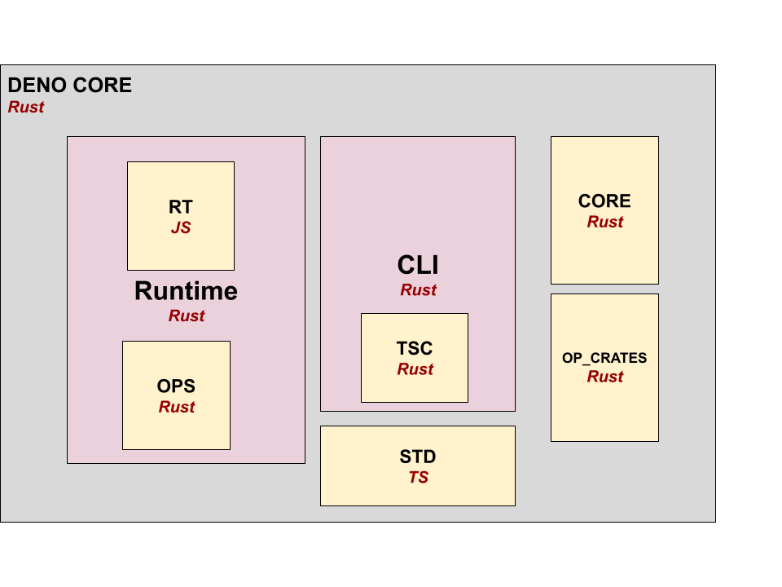
Diagram

Description automatically generated

Figure 9 - Deno Architecture16

3.2.2 Deno Core

*Figure 10 – Deno Core* summarizes the *Deno Core*’s internal organization.



*Figure 10 – Deno Core17*

Mostly relevant here are the CLI, the Runtime and the OPS[[19]](#endnote-20).

3.2.2.1 CLI

CLI is the main entry point to trigger JS/TS execution. It manages the Deno command-line interaction and provides useful commands such as *deno run* and *deno test.* To process these commands, it executes the provided Main Program by orchestrating and dispatching tasks to the other support components. Additional to command-line interaction, the CLI takes care of Module Fetching and Import. Deno doesn’t rely on a package manager (i.e. npm) to automatically download and organize support modules. Consequently, the CLI must implement such functionalities.

To support TypeScript support, Deno translates the code into pure JavaScript, that will be dispatched and evaluated within the v8. The TSC subcomponent provides such translation functionality[[20]](#endnote-21).

**3.2.2.2 Runtime**

The components are split into two parts: RT and OPS. RT is implemented in JavaScript, and it encapsulates a handy set of services: Workers, Permissions, Metrics, Ops, JS Runtime. Workers are an integral part to each runtime and Deno supports both Main and Web Workers. Permissions are used to customize Workers’ capabilities, by explicitly specifying which kind of operations they are allowed to perform. Such permissions involve i.e., Net Capabilities or File System interaction. Metrics are very handy to measure system performances and to conduct benchmarking analysis. Js Runtime encapsulates all the code available in the user space or the JS space.

This latter provides core functionality in pure JavaScript, and these are: Build Info (system architecture, vendor, os), Errors (that are not part of the ECMAScript specification), Console Utils (logging), Dispatch, Timers, IO, Buffer, WebSocket, FS and many more. Such functionalities are directly or indirectly accessible in the JS space. An example of direct interaction is *Deno.console.log(“message”).* Through this function, the JS program could log information to the standard output. Example of indirect invocation relates to new workers’ instantiation. Within JavaScript, to declare a new worker, the *new Worker()* constructor is used. Implementation of such constructor is provided by RT, which is indirectly triggered to deal with the worker’s instantiation process.

As opposite to RT, OPS is written in Rust. Such component implements a set of Rust functions available in the JS space, that further extend JS capabilities. Examples of them are: Crypto, Fetch, HTTP Client and Worker. OPS allows to declare custom functions too. Developers could define their own set of Rust methods that could be at runtime into the execution environment and triggered by the JS code itself. This technology is very interesting and dramatically expand JS runtime capabilities, since it provides a mean to include the Rust environment (and all its modules) into the JS space[[21]](#endnote-22).

3.2.3 Third-party Components

Deno relies on the powerful v8 engine to efficiently evaluate JavaScript code. It’s written in C++ and bindings are needed to communicate and interact with it from Rust. Rusty\_v8 fits in this scope and it is thought to offer the best match possible. It introduces no call overhead and matches most of the v8 functionalities16.

Deno is heavily asynchronous and to simplify the overall architecture implementation, it tries to get rid of callback functions. To properly and efficiently manage the async environment, Tokio runtime is used. This latter allows writing reliable, asynchronous, and slim applications with the Rust programming language. It is event-driven and non-blocking when I/O operations are performed. Among the Tokio functionalities, the most relevant are:

* fs: Methods and adapter type for input/output operations and standard streams
* io: Non-blocking I/O operations
* net: Implements networking protocols
* runtime: An I/O event loop that manages the v8 runtime operations
* task: Asynchronous independent thread[[22]](#endnote-23)

1. Implementation

To investigate the above-mentioned dimensions of a serverless execution environment, a prototype of the system has been implemented. Implementation has been done incrementally, starting from the core functionality (JavaScript function evaluation). With each increment, new features have been added, and the overall system’s complexity increased. Prior to moving onto the next step, strength and weaknesses of the partial architecture are evaluated and reported. Four main increments could be identified in the development process:

* Get a simple function running in Deno
* Comparing Main Worker and Web Workers
* Triggering Deno from NATS on a single thread
* Triggering Deno from NATS on multiple threads

It follows a brief description of each increment, the conducted tests, and their results.

* 1. Get a Simple Function Running in Deno

The first step to create an execution environment is to extend Rust capabilities to support JavaScript evaluation. Rust does not natively support JavaScript; therefore, a JavaScript execution environment must be embedded. Embedding Deno is very simple and straightforward, and the *deno\_core* developers provided a clarifying example[[23]](#endnote-24).

In the code, a *JsRuntime**[[24]](#endnote-25)* instance is created. It contains all the machinery to evaluate JavaScript code and the *execute\_script()[[25]](#endnote-26)* method is invoked for this purpose.

* 1. Comparing Main Worker and Web Workers

Workers are JavaScript execution environments encapsulating and extending the *JsRuntime*9 functionalities. There are of two different types: Main[[26]](#endnote-27) and Web[[27]](#endnote-28),[[28]](#endnote-29) Workers. Main workers manage the workflow of a JavaScript program and Web Workers support main execution. Workers play an important role within the serverless paradigm since they are a mean to evaluate functions. To ensure highest throughput possible, the most performant worker type must be used. Since Web Workers are intended to be only support-environments for a Main Worker, the question came up, whether it is possible that less resources are allocated for them. Consequently, it might be possible that the same function could take longer to be evaluated in the support environment.

To answer this question, the initial system has been extended to execute functions within Workers and performance analysis is conducted. The system architecture developed to conduct performance analysis is composed by two independent Rust programs. The first one dispatches massages via NATS server to a NATS subject and each message corresponds to a test execution request. Published NATS messages are processed by the other program, the receiver, which executes the test and stores the execution time. The run test case is constructed as follows: A simple JavaScript for loop is encapsulated within an ES module used as main module to boot-up a new worker. When running the test, the worker is instantiated, and the time needed for it to fully boot is measured. The booting-up phase depends on the main module’s evaluation, which itself executes the specified for loop. Tests are conducted for both worker’s types.

Outcomes of the experiment were against the initial hypothesis and no significant difference were noticed in execution times when using Main or Web Workers. *Table 2- Execution Time with Worker’s instantiation* summarizes the execution times. After an in-depth investigation of the *deno\_core* craft and the Workers’ source code[[29]](#endnote-30),[[30]](#endnote-31), emerged that the two worker types rely on the same java script runtime. Therefore, the same number of resources and capabilities are given to each of them to evaluate code and the same version of the v8[[31]](#endnote-32) engine is used. Furthermore, no significant difference was noticed between the two booting-up sequences and results are in-line with these new investigation outcomes.

The previous experiment showed no gain in performances if the Main and Web Workers are instantiated from scratch to execute a test function. The question arose, whether the situation would change when pre-instantiated workers are used instead of creating a new one each time. The idea behind it is that there could be a gain in performances if a pool of Web Workers is kept alive and functions are executed within such running workers, since booting-up time could be avoided. To make an analogy with a cache memory, data is retrieved faster when a cache-hit happens, and data doesn’t need to be moved from the lower main memory.

To answer this question, the previously developed testing environment was revised. In the new setting, a Web Worker instance is created by a Main Worker and kept alive ready to execute functions. As opposed to the previous setting, in which Web Workers were manually instantiated by the Rust management system, it is the Main Worker itself that creates its own child Web Worker. To allow this automation, lots of effort has been put to understand and properly implement the dynamic create\_web\_worker\_cb()[[32]](#endnote-33) callback. Implementation of the function can be found in project’s GitHub repo[[33]](#endnote-34). To let the Main Worker dispatch function execution to the Web Worker, JavaScript-provided message passing is used[[34]](#endnote-35).

The new experiment shows a significant decrease in execution time by an order of two magnitudes when executing the function in a running worker. *Table 3 - Execution Time without Worker’s instantiation* summarizes the execution times. By merging, the experiments’ evidence, two main conclusions could be drawn:

* The two-order-magnitude difference in execution time depends on the initial worker booting-up phase, which takes significantly longer than the function execution itself
* Executing functions within an already running worker increases overall performance reducing execution time
  1. Triggering Deno from NATS on a Single Thread

The execution environment to conduct the previous tests relies on function evaluation within a single worker. Nevertheless, a serverless system with such architecture happens to be very limited since it does not scale well as the workload increases. To increase execution throughput, more workers are needed. Furthermore, with this architecture there is no mean from the rust management system to directly interact with the workers to request function invocation. In the above test case, function evaluations have been hard coded in the Main Worker code, but a more consistent communication is needed. Therefore, further research has been conducted to provide a more consistent and reliable architecture that allows the following features:

* Lightweight management system and Main Worker communication
* Workload management, in the sense of automatic dispatching of execution to an available worker

To extend the initial architecture with the above-mentioned properties, a pool of running workers that could be triggered by the NATS client is used. The pool is based on independent workers running on different threads and thread-safe communication and synchronization is used to share function invocation requests with the first available thread. As initial step to implement the pool, the worker is moved in a new thread independent from the NATS’ client one. To the worker, a new event loop is associated, so that its execution doesn’t depend on the NATS client.

To make the thread communicate, using native Rust thread-safe communication channels has been tested and worked perfectly. One channel has been used to send invocation requests and another to send back execution results.

* 1. Triggering Deno from NATS with Multiple Threads

As last step of the development process, multiple Workers in multiple threads are created and workload management is introduced. To have a set of communication channels associated with each thread in the workers’ pool is straightforward in Rust. Nevertheless, a major difficulty has been faced when trying to implement workload management in the previous architecture. It turned out that the NATS client monitoring the pool’s threads is not able to figure out whether a thread is halted cause the worker is waiting for an incoming message to arrive or whether it is still executing the previous code. Therefore, a workload management policy going from the NATS client to the worker’s threads is not feasible.

However, a simple change of perspective in the management direction allows to easily work-around the problem. By letting the workers, themselves query the client for a new function invocation whenever they complete the previous execution, no external thread-state monitoring from the client itself is required.

To implement this new workload management policy, a simple shared object could be used. Both the NATS client and the workers share a common queue[[35]](#endnote-36) and they compute respectively push and pull operations on it. The client pushes new incoming messages, and the workers pull them. Implementation of the system using a share queue could be found in the GitHub project’s repo[[36]](#endnote-37).

1. Recommendations

Due to lack of time, not all the dimensions of investigation have been properly explored. Nevertheless, important recommendations could still be mentioned. It follows for each above-mentioned investigated dimension, the corresponding guidelines.

* 1. Keeping State Among JavaScript Code Evaluation

As mentioned in the *execute\_script()* function’s documentation, the input string is treated as native JavaScript code from the *JSRuntime* and global state within the execution context in kept among multiple invocations. This means that subsequent code execution could access all variables and methods declared before.

However, *execute\_script()* method is not the only way to trigger function execution in *deno\_core*. JavaScript programs could be declared and evaluated as ES Modules. Both *JsRuntimes, MainWorkers* and *WebWorkers* do support module compilation and evaluation through *load\_main\_module()[[37]](#endnote-38)* and *load\_side\_module()[[38]](#endnote-39)*. As opposite to *execute\_script()*, since ES Modules are supposed to be executed once, no state is kept by *deno\_core* among different module evaluation. Variables and functions declared in one module are consequently not accessible by the others.

To make variables and function persistent and accessible outside the ES Module itself, a special *deno\_core* object could be used. Within each *JSRuntime*, a *globalThis* object instance is kept and it is made accessible by default to every JavaScript piece of code being executed within the runtime (either as native code or module). Therefore, by manually storing variables in this shared object, subsequent function executions could rely on previously computed values of declared functions, simulating the keep of state. Particular attention must be kept in accessing the *globalThis* properties, since redeclaration of function and variables is not allowed and consistency for the values stored has to be manually ensured.

* 1. Security within Workers

As mentioned17, workers executes under a different global scope. Therefore, data in the parent workers are not accessible from the children Web Workers. The only way to share data is through message passing technique and no global objects could be directly shared.

Nevertheless, *deno\_runtime* crate allows specifying which permissions each Web Worker has. The list of all permissions is *read, write, net, env, run, ffi, hrtime[[39]](#endnote-40)*. Allowing Web Workers to access the current file system with the read and write properties make file sharing between workers possible, introducing a new information exchange technique. However, attention should be put in preserving security within the system, since Web Workers executing untrusted code could maliciously access Main Worker’s files and corrupt data.

* 1. Performance comparisons between Main Worker and Web Workers

Out of the performance experiments conducted, two major conclusions could be drawn:

* booting-up phase takes significant time: around 200 ms (see *Table 4 - Evaluation of Booting-Up time*). A serverless workload management system must consider this initial delay prior to instantiating a new worker. It could be more profitable for the management system to wait for an already running worker finishing its current function execution and dispatch to it new request than instantiating a new ad-hoc worker for the incoming function.
* Performance is not affected by the worker type.
  1. Management of multiple v8 engines within Rust/Tokio

Management of multiple v8 engines could be very tricky and must satisfy the following properties:

* Independence of work among instances
* Communication and synchronization

To ensure independence of work among engines, they cannot run within the same thread. Each *JsRuntime* needs a Tokio *event loop[[40]](#endnote-41)* to execute, but only one loop exists per thread. If engines are associated to the same loop and one other locks, all the others will be locked as well. Therefore, to ensure independence of work, they must be moved to separate threads.

Now that engines are in separate threads, communication and synchronization is not trivial, since they are independent from each other, and thread-safe communication must be used. An efficient way of communication should not rely on native Rust channels[[41]](#endnote-42), since this approach doesn’t scale as the system workload increases (new workers are created). Therefore, the use of a share object seems a better option. Rust natively allows to share object between threads by using thread-safe ARC pointers[[42]](#endnote-43). By protecting a shared object behind such pointer, it could be used by all engines to send and receive messages from the others.

* 1. Workload management policy

An efficient workload management policy aims to dispatch a function execution request to the first available worker. To define this policy, two different approaches could be used. On one hand, it could be the NATS client that looks for an available worker. On the other hand, it could be the workers themselves that asks for new requests whenever they are available.

As stated above, the first approach is not feasible, since the NATS client cannot distinguish between the worker state. The second approach could be easily implemented by using the above-mentioned shared object as a mean of communication. The NATS client receiving incoming execution requests from the users pushes invocations in the queue and workers pull them as soon as they are available.

Even though the shared queue drastically reduces the management policy’s complexity, it introduces another subtle problem, that may reduce performances of the system. Overall performance depends on how many messages are injected by the NATS client into the queue. The higher the number of messages, the higher the workload on the workers and the higher the throughput. For the NATS client to access the queue, its lock request must be granted. Long waiting time (or even starvation) may arise for the client if the number of workers is significantly big, since they all try to access the queue as the NATS client. Due to the higher number, a worker has a higher probability of getting its request for the lock being approved than the single NATS client. This latter could therefore wait long time before being able to push messages.

A simple policy has been introduced in the prototype developed architecture to avoid long client waiting time, and it could be summarized as follows: the NATS client never waits before requesting the lock for the shared queue; workers wait a random time before sending their lock request. The workers’ waiting time tries to leverage their higher probability to get the lock for the queue. Further investigation is needed to assess and measure overall system performance applying this policy, to check whether the waiting-client problem persists or not.

1. Lecture Relevant Learnings

During the development of the project, we got to know Rust, NATS and Deno. During the early implementation phase, the Rust documentation and tutorials helped us a lot to climb the steep learning curve of the language and the NATS examples were extremely clarifying on how to embed such technology. Working with Deno was more difficult in the beginning, due to high system complexity. As a starting point to understand the Deno architecture, we heavily relied on docs.rs, an open-source documentation tool for Rust crates. Providing as input a crate or a module, the tool automatically produces structured documentation for it. This is based on links, that allows to easily navigate the code to understand the components’ organization and their respective interaction. Additionally, for each structure, variable and function, a pointer to their respective definition is provided within the raw source code.

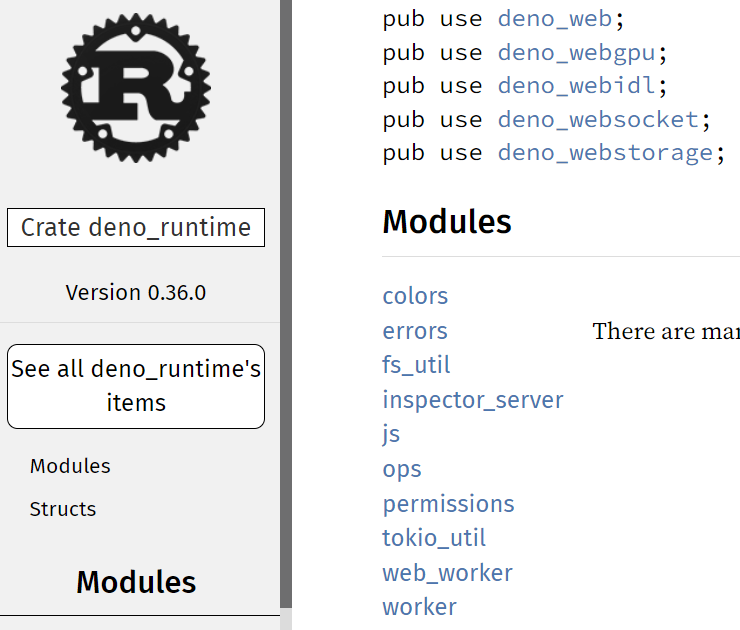


Figure 10 - Docs.rs[[43]](#endnote-44)

Even though docs.rs provides a handy solution to move around the code, the amount of information gathered from the documentation was only marginal. Comments for structures, variables and functions could have been more detailed for beginners, therefore, to understand their precise role within the system was very tedious for us. To work around this problem and gather a deeper knowledge of the system, the project supervisor suggested us to join the Deno community and ask them questions in their public Discord server. Our experience with the community was very profitable and sped-up our learning phase. It was possible to ask questions not only about bug solutions and implementation difficulties, but also about the system itself, about its structure and the roles of each component. Our questions were always answered quickly (within the span of a few minutes) and in a professional manner. With the help of the community, we gathered a decent knowledge of the system, that allowed us to conduct the analysis explained above. Starting from our experience, we highly recommend joining communities for open-source project, especially as a complete beginner, to get help to understand complex architectures.

The Deno community provided us crucial information about the internal system organization, by clearly explaining internal mechanisms and components interaction. Additionally, they often suggested best practices to implement the Rust engine embedder. To include these kinds of explanations and tips within the automatically generated documentation would be very helpful for developers, so that there would be a unique access point to gather system information. For each Rust module, struct of function, a summarization tool like CODES could collect and merge together documentation of submodules, to avoid the need of moving around the code. Additionally, a Web Crawler/Scraper could mine useful information from the Internet, by analyzing StackOverflow posts, exchange of messages within the community and commit messages on GitHub. By merging the two sources of information, a handy description for each system component, module, structure, and function could be provided.

1. Summary

The initial goal of the project was to compare ECMAScript engines and runtimes, to assess their performances both in term of raw throughput and memory consumption. An initial qualitative comparison was conducted, and a table summarizing the result constructed. By mean of such table, the most interesting and promising engines were chosen to conduct the performance analysis. Deno and QuickJs were perfect candidates. Quantitative comparison was conducted by executing the same function within the two engines and measuring the total evaluation time. Deno was proven to be way faster than QuickJs (2 order of magnitude). Independent researchers conducted the same performance comparison and same conclusion derived. Since Deno would have had outperformed QuickJs in most of the test cases, no further research was conducted in this direction, because it would not have led to interesting results. Therefore, a shift of goal for the project was necessary and it was agreed with the coordinator to investigate the implementation of a possible JavaScript serverless execution environment based on NATS technology (for communication) and Deno. The goal of the project became to provide recommendations on how to properly develop such system.

Implementation were conducted out of little increments, and at each step, experiments and tests were conducted on the partial system, to assess the best way to proceed. Reflections on possible strengths and weaknesses of the architecture were also made.

With the information obtained, we were able to make recommendations to develop a safe, reliable, and efficient system. The recommendations entailed keeping state among JavaScript execution, Security within workers, performance comparison between worker types, management of multiple V8 engines and workload management policy.

1. Appendix

Table 1 - Runtime Comparison





Table 2- Execution Time with Worker’s instantiation

|  |  |  |
| --- | --- | --- |
| Worker Type | Number of Tests | Execution Times |
| Main Worker | 10 | 459.543518ms, 456.006672ms, 448.712379ms, 439.332617ms, 447.088314ms, 445.702962ms, 458.252155ms, 447.97186ms, 443.961152ms, 444.470175ms |
| Web Worker | 10 | 459.740987, 462.12933, 454.1517849999999, 445.3357199999998, 453.29438099999993, 450.8542970000003, 470.313803, 448.2627430000002, 451.08487200000036, 452.7584130000005 |

Table 3 - Execution Time without Worker’s instantiation

|  |  |  |
| --- | --- | --- |
| Worker Type | Number of Tests | Execution Times |
| Main Worker | 10 | 468.768474ms, 465.836208ms, 5.259826ms, 4.900947ms, 4.783687ms, 4.802829ms, 5.697033ms, 4.846011ms, 4.693492ms, 4.692548ms |

Table 4 - Evaluation of Booting-Up time

|  |  |  |
| --- | --- | --- |
|  | Number of Tests | Execution Times |
| Main Worker | 10 | 468.768474ms, 465.836208ms, 5.259826ms, 4.900947ms, 4.783687ms, 4.802829ms, 5.697033ms, 4.846011ms, 4.693492ms, 4.692548ms |

By looking at the Main Worker’s execution times, a decrease in time of 460 ms could be noticed. This is when JIT compilation optimizes the function execution. By taking as a mean the execution time of the compiled function, an upper bound of the booting-up execution time could be derived. Half of the time is taken, to avoid overestimation.

Table 5 - Comparison QuickJs vs. Deno performance

|  |  |
| --- | --- |
| Runtime | Execution Time |
| Deno | 43.372215 ms |
| QuickJs | 9667.967495 ms |

Table 6 - QuickJs Benchmark6

*Immagine che contiene tavolo

Descrizione generata automaticamente*

1. References

   <https://iotify.io/> [↑](#endnote-ref-2)
2. <https://insights.stackoverflow.com/trends?tags=express%2Ctypescript> [↑](#endnote-ref-3)
3. <https://www.usenix.org/system/files/atc20-shahrad.pdf> [↑](#endnote-ref-4)
4. <https://insights.stackoverflow.com/trends?tags=serverless%2Caws-lambda> [↑](#endnote-ref-5)
5. <https://en.wikipedia.org/wiki/List_of_ECMAScript_engines> [↑](#endnote-ref-6)
6. <https://github.com/gavi210/UZH_ECMASCRIPT_PROJECT/tree/simple_performance_comparison/perf_comparison> [↑](#endnote-ref-7)
7. <https://bellard.org/quickjs/bench.html> [↑](#endnote-ref-8)
8. <https://docs.nats.io/> [↑](#endnote-ref-9)
9. <https://docs.nats.io/nats-concepts/what-is-nats> [↑](#endnote-ref-10)
10. <https://docs.nats.io/nats-concepts/overview> [↑](#endnote-ref-11)
11. <https://docs.nats.io/nats-concepts/subjects> [↑](#endnote-ref-12)
12. <https://github.com/nats-io/nats-general/blob/master/architecture/ARCHITECTURE.md> [↑](#endnote-ref-13)
13. <https://docs.nats.io/nats-concepts/jetstream> [↑](#endnote-ref-14)
14. <https://docs.nats.io/nats-concepts/jetstream/key-value-store> [↑](#endnote-ref-15)
15. <https://choubey.gitbook.io/internals-of-deno/> [↑](#endnote-ref-16)
16. <https://deno.land/> [↑](#endnote-ref-17)
17. <https://choubey.gitbook.io/internals-of-deno/introduction/about> [↑](#endnote-ref-18)
18. <https://choubey.gitbook.io/internals-of-deno/architecture/overall-architecture> [↑](#endnote-ref-19)
19. <https://choubey.gitbook.io/internals-of-deno/architecture/core> [↑](#endnote-ref-20)
20. <https://choubey.gitbook.io/internals-of-deno/architecture/core/cli> [↑](#endnote-ref-21)
21. <https://choubey.gitbook.io/internals-of-deno/architecture/core/rt> [↑](#endnote-ref-22)
22. <https://choubey.gitbook.io/internals-of-deno/architecture/tokio> [↑](#endnote-ref-23)
23. <https://github.com/denoland/deno/blob/main/core/examples/hello_world.rs> [↑](#endnote-ref-24)
24. <https://docs.rs/deno_core/latest/deno_core/struct.JsRuntime.html> [↑](#endnote-ref-25)
25. <https://docs.rs/deno_core/latest/deno_core/struct.JsRuntime.html#method.execute_script> [↑](#endnote-ref-26)
26. <https://docs.rs/deno_runtime/0.36.0/deno_runtime/worker/struct.MainWorker.html> [↑](#endnote-ref-27)
27. <https://docs.rs/deno_runtime/0.36.0/deno_runtime/web_worker/struct.WebWorker.html> [↑](#endnote-ref-28)
28. <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API> [↑](#endnote-ref-29)
29. <https://docs.rs/deno_runtime/latest/src/deno_runtime/worker.rs.html#39-42> [↑](#endnote-ref-30)
30. <https://docs.rs/deno_runtime/latest/deno_runtime/web_worker/struct.WebWorker.html> [↑](#endnote-ref-31)
31. <https://v8.dev> [↑](#endnote-ref-32)
32. <https://docs.rs/deno_runtime/latest/deno_runtime/web_worker/struct.WebWorkerOptions.html#structfield.create_web_worker_cb> [↑](#endnote-ref-33)
33. <https://github.com/gavi210/UZH_ECMASCRIPT_PROJECT/blob/stage-2-reuse-main-workers/stage-2/nats-receiver/src/web_worker_manager.rs> [↑](#endnote-ref-34)
34. <https://developer.mozilla.org/en-US/docs/Web/API/Worker/postMessage> [↑](#endnote-ref-35)
35. <https://docs.rs/queue/0.3.1/queue/> [↑](#endnote-ref-36)
36. <https://github.com/gavi210/UZH_ECMASCRIPT_PROJECT/tree/stage-2-dispatch-invocation/stage-2/nats-receiver> [↑](#endnote-ref-37)
37. <https://docs.rs/deno_core/0.110.0/deno_core/struct.JsRuntime.html#method.load_main_module> [↑](#endnote-ref-38)
38. <https://docs.rs/deno_core/0.110.0/deno_core/struct.JsRuntime.html#method.load_side_module> [↑](#endnote-ref-39)
39. <https://docs.rs/deno_runtime/0.36.0/deno_runtime/permissions/index.html> [↑](#endnote-ref-40)
40. <https://docs.rs/tokio/1.14.0/tokio/index.html> [↑](#endnote-ref-41)
41. <https://doc.rust-lang.org/std/sync/mpsc/fn.channel.html> [↑](#endnote-ref-42)
42. <https://doc.rust-lang.org/std/sync/struct.Arc.html> [↑](#endnote-ref-43)
43. <https://docs.rs/deno_runtime/latest/deno_runtime/> [↑](#endnote-ref-44)