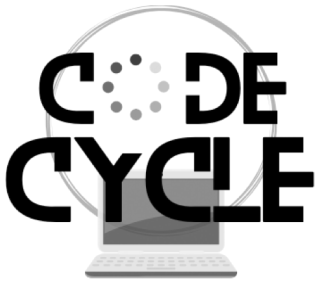


JS Runtime

Section 1

Introduction:

Here, we're gonna dive into Javascript Runtime and create a simple one of our own. Ever curious about Javascript Runtimes like NodeJS, Deno etc? Worry not!! This tutorial will feed your curiosity. Even if you're not into this and don't know what the heck these are; don't step back. This will be interesting! We'll move into the coding part only after covering some basics, theoretical concepts and debunking some common misconceptions.

This tutorial would assume that you have access to either linux or macos or WSl. (if you are in windows). Though linux or mac machines are preferable, if you're a windows user like me; you can make use of WSL.

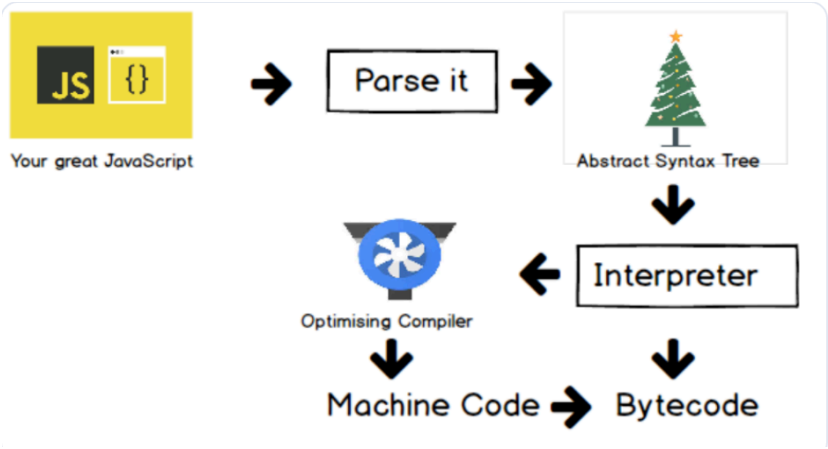
But what is a JS Runtime:

You're in the right place. Before going into JavaScript Runtime, try thinking of the language JavaScript (JS) itself. Where do you think it's being used??

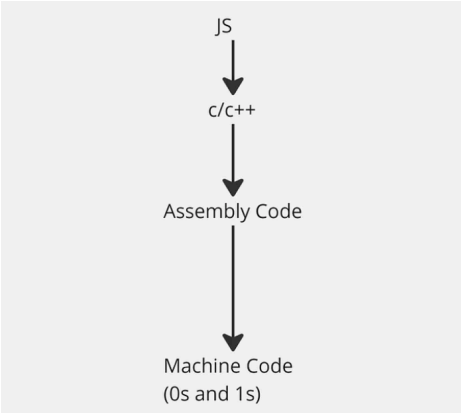
To be honest, it's everywhere now. Scripting, Frontend / Backend (web dev), Automation and even in app dev. The language itself is not used as plain in these areas; but rather in some form of a library or a framework (React, Angular, React Native). Okay enough of all this! At a very basic level, wikipedia says: "JavaScript, often abbreviated as JS, is a programming language and core technology of the Web, alongside HTML and CSS. 99% of websites use JavaScript on the client side

for webpage behavior. Web browsers have a dedicated JavaScript engine that executes the client code."

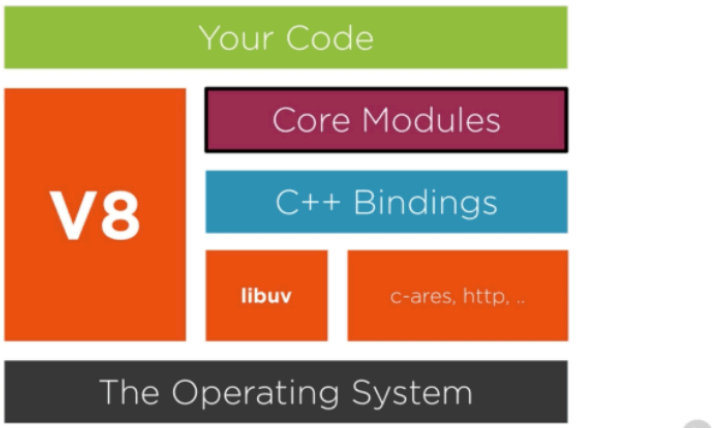
Just as any programming language, JS requires a compiler that converts it into machine code so that it could be run in a machine. The last line of the wiki says something about a JS engine that executes client code. What's that? A JS engine does the job of the compiler in addition to some specific work. The below diagram shows the job of the JS engine present in the Google Chrome, V8.



Here's a fun fact: JavaScript language itself was initially written in C and modern JavaScript (like that used in the V8 engine) is now primarily written in C++! Let's dive some more:

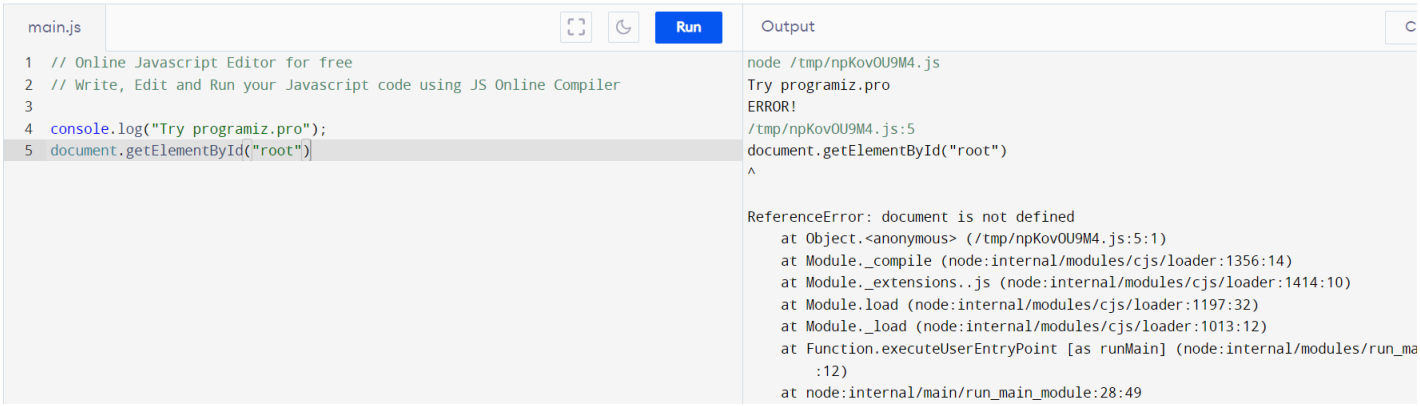


This image has the perfect explanation:



With that said, we can split Node.js into two parts: V8 and Libuv. V8 is about 70% C++ and 30% JavaScript, while Libuv is almost completely written in C.

Take a look at the above image! So by now you should have understood that JS (which itself is written in c/c++) will be compiled to machine code before being run. But that's enough for a general purpose programming language. We need something extra specific to the use case: like updating the DOM in the frontend, handling simultaneous API requests in the backend and so on. That's where the wrappers like browsers and JS runtimes kick in. The browsers like Google chrome, Firefox will provide in those additional features like web apis. For example, if you run document.getElementById("someid") outside the browser; you will get an error. Try it out in some online compiler. The reason is that; the document object (DOM) is facilitated by the browser itself when it renders the frontend as a web page.



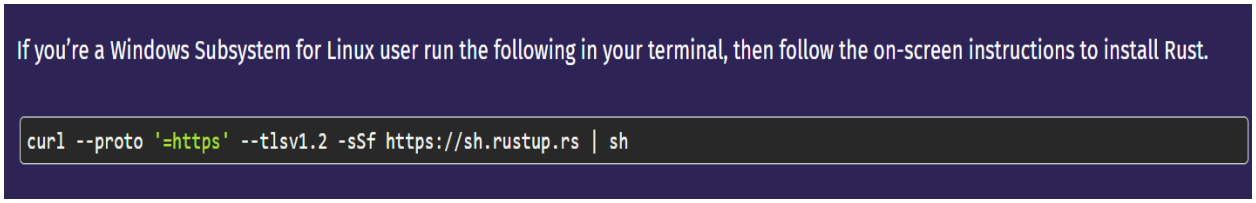
Okay! So you understand that the browser provides some specifics related to frontend. Then what are these JS runtimes and NodejS. Let's hear it from the official documentation for NodeJS itself: Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts. Wait a minute!! So NodeJS is not a programming language. There is no such thing as .nodejs files. There are only .js files! Just like browsers, NodeJS provides a runtime environment with capabilities like file system access, I/O operations, concurrency etc with the help of Google V8 engine, libuv package and so on.

That's IT! Browser and NodeJS are two different environments and that's why you can't access one's features inside the other. Both serve their own purpose. I hope now you're clear about JS as a programming language, JS engine like V8, Browser and JS runtimes like NodeJS. In no way NodeJS is the only runtime and V8 is the only JS engine. Firefox has spider monkey as an engine. And there are runtimes like Deno which come in handy as it wraps all complex apis and enables us to use it for demonstration purposes. Ready??

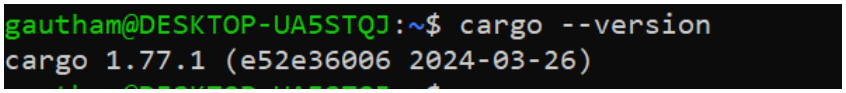
Getting started:

The first thing we want to do is to install Rust & Cargo. Rust is used for three essential purposes in programming: performance, safety, and memory management. Hence, Rust is used to develop advanced applications like gaming

engines, operating systems, and browsers that demand scalability. Visit this link to download rust: https://www.rust-lang.org/learn/get-started.



By default Cargo will be installed with Rust. Check your installation by running the version command:



If you're familiar with npm, you can easily get it. npm and cargo are both package managers used by developers to install, uninstall, create, and ship dependencies in their respective language. While npm is used for Node.js, cargo is for Rust. Without any delay, lets create a Rust project with necessary dependencies. We'll start with cargo init to create a simple template project under the name "myjs". On executing cargo run inside the directory, you can see the hello world message being printed in the terminal. That indicates that we're good to proceed to dependencies installation.

```
gautham@DESKTOP-UA5STQJ:~/codecycle$ cargo init --bin myjs
     Created binary (application) package
gautham@DESKTOP-UA5STQJ:~/codecycle$ cd myjs/
gautham@DESKTOP-UA5STQJ:~/codecycle/myjs$ ls
Cargo.toml  src
gautham@DESKTOP-UA5STQJ:~/codecycle/myjs$ cargo run
   Compiling myjs v0.1.0 (/home/gautham/codecycle/myjs)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/myjs`
Hello, world!
gautham@DESKTOP-UA5STQJ:~/codecycle/myjs$ cargo add deno_core@0.272.0
    Updating crates.io index
      Adding deno_core v0.272.0 to dependencies.
             Features:
             + deno_core_icudata
             + include_icu_data
             + v8_use_custom_libcxx
             - include_js_files_for_snapshotting
             - snapshot_flags_eager_parse
             - unsafe_runtime_options
             - unsafe_use_unprotected_platform
    Updating crates.io index
gautham@DESKTOP-UA5STQJ:~/codecycle/myjs$ cargo add tokio@1.19.2 --features=full
    Updating crates.io index
```

After doing the two cargo add, you will see the Cargo.toml file like (this file is similar to the package.json in nodejs or requirements.txt file in python) :

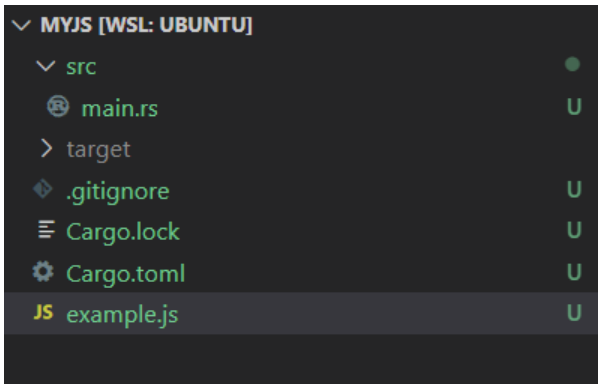
```
gautham@DESKTOP-UA5STQJ:~/codecycle/myjs$ cat Cargo.toml
[package]
name = "myjs"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo

[dependencies]
deno_core = "0.272.0"
tokio = { version = "1.19.2", features = ["full"] }
gautham@DESKTOP-UA5STQJ:~/codecycle/myjs$
```

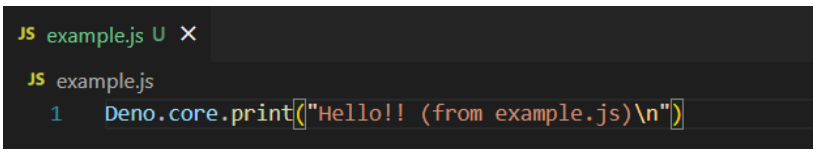
Open the directory in the code editor of your choice and go through the available files. You'll see a simple main function printing hello world in the main.rs (rust) file. Now let's change this file according to our needs: (building our own JS runtime)

```rust
// main.rs
use deno_core::{
    error::AnyError, resolve_path, FsModuleLoader, JsRuntime, PollEventLoopOptions, RuntimeOptions,
};
use std::env;
use std::rc::Rc;

async fn run_js(file_path: &str) -> Result<(), AnyError> {
    let main_module = resolve_path(file_path, env::current_dir()?.as_path())?;
    let mut js_runtime = JsRuntime::new(RuntimeOptions {
        module_loader: Some(Rc::new(FsModuleLoader)),
        ..Default::default()
    });

    let mod_id = js_runtime.load_main_es_module(&main_module).await?;
    let result = js_runtime.mod_evaluate(mod_id);
    js_runtime
        .run_event_loop(PollEventLoopOptions {
            wait_for_inspector: false,
            pump_v8_message_loop: false,
        })
        .await?;
    result.await
}

fn main() {
    let runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap();
    if let Err(error) = runtime.block_on(run_js("./example.js")) {
        eprintln!("error: {}", error);
    }
}
```

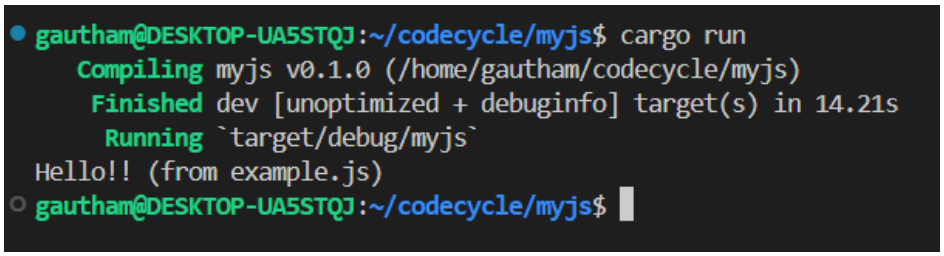
There's a lot to unpack here. At first a few lines, we're importing the necessary functions from deno_cor. This Rust crate or package contains the essential V8 bindings for Deno's command-line interface (Deno CLI). The main abstraction here is the JsRuntime which provides a way to execute JavaScript. So this run_js function will accept a JS file and its main goal is to run it. You don't have to understand each and every syntax of this file, but it's useful to know on an abstract level. The asynchronous run_js function creates a new instance of JsRuntime, which uses a file-system based module loader. After that, we load a module into js_runtime, evaluate it, and run an event loop to completion. This run_js function encapsulates the whole life-cycle that our JavaScript code will go through. So to summarize, if we pass a js file inside this function; it will execute it

as any JS runtime. But before we can do that, we need to create a single-threaded tokio runtime to be able to execute our run_js function. That's being done in the main function. After creating a tokio runtime, you could see that, we are calling the run_js function with "./example.js" as a parameter. Create that file in our working directory:





Now if you execute "cargo run"; we will be able to see the log. Just like browsers provide document object access to JS, our MYJS runtime makes use of deno_core and provides Deno.core.print access inside the JS file. That's awesome right??



Submission Link : https://forms.gle/Pja5PrMiuMAdEn8P7

Section 2

Custom logging:

Enough of that! How about making our very own console.log function. Let's make it more creative by adding a timestamp and some sarcastic message for each log. After doing so, we'll be able to access a new log function like console.sarcasm(). Are you ready? Then add the highlighted line in the run_js function:

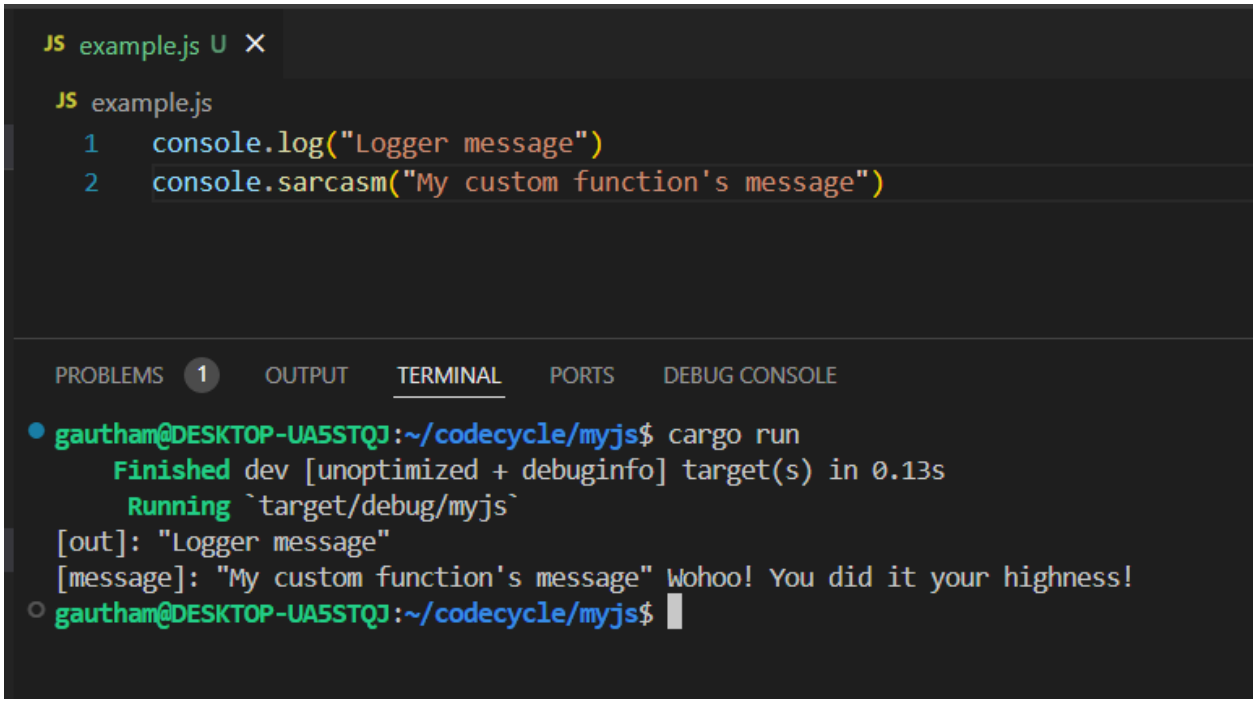
```
async fn run_js(file_path: &str) -> Result<(), AnyError> {
    let main_module = resolve_path(file_path, env::current_dir()?.as_path())?;
    let mut js_runtime = JsRuntime::new(RuntimeOptions {
        module_loader: Some(Rc::new(FsModuleLoader)),
        ..Default::default()
    });

    js_runtime
        .execute_script("[runjs:runtime.js]", include_str!("runtime.js"))
        .unwrap();

    let mod_id = js_runtime.load_main_es_module(&main_module).await?;
    let result = js_runtime.mod_evaluate(mod_id);
    js_runtime
        .run_event_loop(PollEventLoopOptions {
            wait_for_inspector: false,
            pump_v8_message_loop: false,
        })
        .await?;
    result.await
}
```

As you read from the new line, we're executing a new file under the name runtime.js and using its functionalities. So what does that file look like? We'll be using Deno.core.print to print a message and also we'll be overriding the usual console object with an additional function under the name sarcasm. Create runtime.js file inside src directory and code the following:

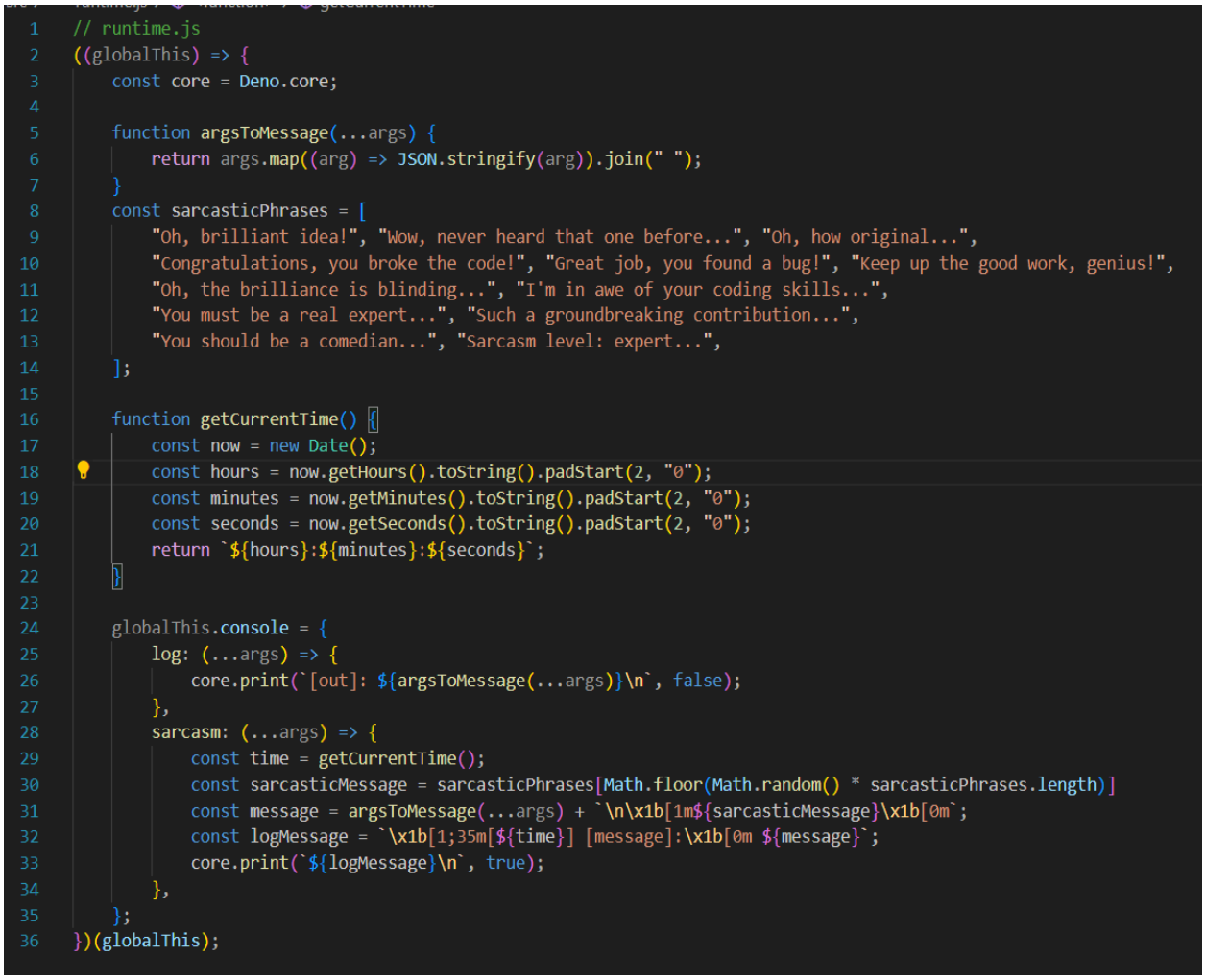
src > JS runtime.js > ...

```
// runtime.js
((globalThis) => {
    const core = Deno.core;

    function argsToMessage(...args) {
        return args.map((arg) => JSON.stringify(arg)).join(" ");
    }

    globalThis.console = {
        log: (...args) => {
            core.print(`[out]: ${argsToMessage(...args)}\n`, false);
        },
        sarcasm: (...args) => {
            const message = argsToMessage(...args) + " " + "Wohoo! You did it your highness!"
            const logMessage = `[message]: ${message}`;
            core.print(`${logMessage}\n`, true);
        },
    };
})(globalThis);
```

After this, there is only one thing left, change example.js and test it out. Let's do that:

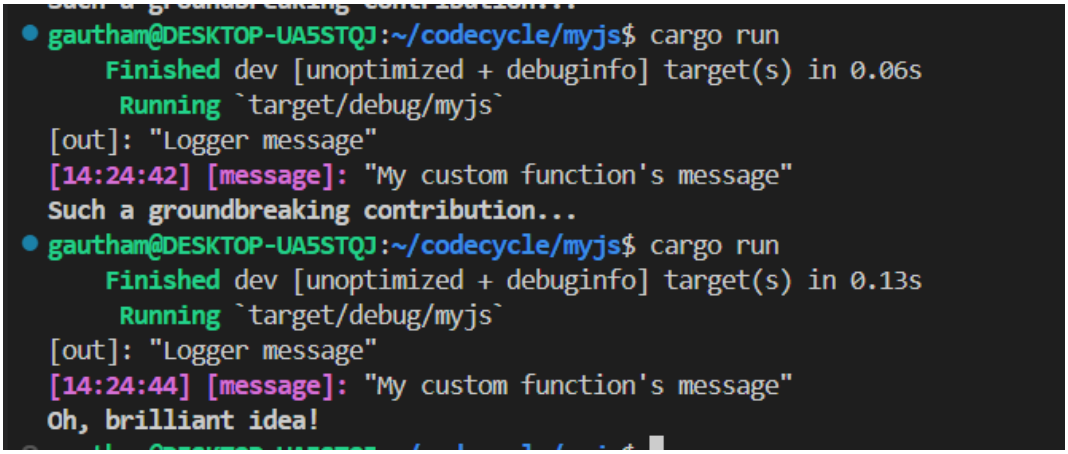


As you can see! We did it. We have our very own console log in our very own JS runtime. Come on. You can get creative with it. Let's add some timestamp, bold letters and dynamic sarcasm messages.



```
// runtime.js
((globalThis) => {
    const core = Deno.core;

    function argsToMessage(...args) {
        return args.map((arg) => JSON.stringify(arg)).join(" ");
    }
    const sarcasticPhrases = [
        "Oh, brilliant idea!", "Wow, never heard that one before...", "Oh, how original...",
        "Congratulations, you broke the code!", "Great job, you found a bug!", "Keep up the good work, genius!",
        "Oh, the brilliance is blinding...", "I'm in awe of your coding skills...",
        "You must be a real expert...", "Such a groundbreaking contribution...",
        "You should be a comedian...", "Sarcasm level: expert...",
    ];

    function getCurrentTime() {
        const now = new Date();
        const hours = now.getHours().toString().padStart(2, "0");
        const minutes = now.getMinutes().toString().padStart(2, "0");
        const seconds = now.getSeconds().toString().padStart(2, "0");
        return `${hours}:${minutes}:${seconds}`;
    }

    globalThis.console = {
        log: (...args) => {
            core.print(`[out]: ${argsToMessage(...args)}\n`, false);
        },
        sarcasm: (...args) => {
            const time = getCurrentTime();
            const sarcasticMessage = sarcasticPhrases[Math.floor(Math.random() * sarcasticPhrases.length)]
            const message = argsToMessage(...args) + `\n\x1b[1m${sarcasticMessage}\x1b[0m`;
            const logMessage = `\x1b[1;35m[${time}] [message]:\x1b[0m ${message}`;
            core.print(`${logMessage}\n`, true);
        },
    };
})(globalThis);
```

If you write this code and execute cargo run a few times, you'll notice what we've done!! Go ahead champ!



Submission Link : https://forms.gle/Pja5PrMiuMAdEn8P7

Section 3

It's now time to put in your fingers to extend our JS. Implement a new function like `console.warn()` or `console.error()` in your runtime that formats the output differently (e.g., color-coding, adding timestamps etc). Or even try to create a custom method, `console.debug()`, that logs messages only if a debug flag is set (environment variable or configuration file).

Submission has to be done in Unstop itself...

THE END