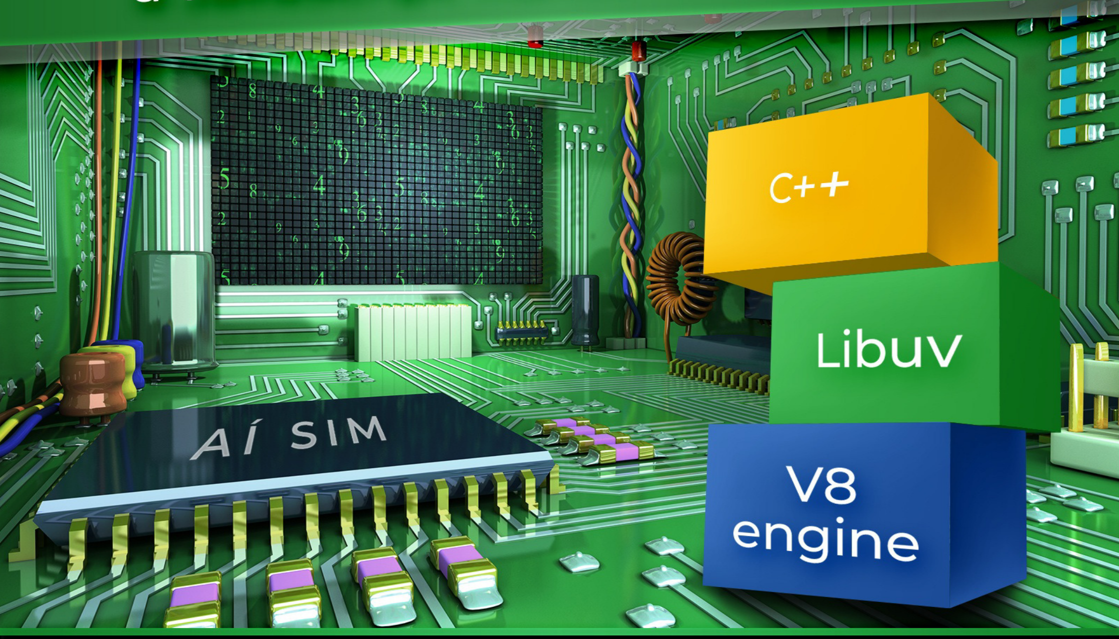
Recreating Node.js from Scratch using V8, Libuv, and C++

Learn how to create a JavaScript Runtime in practice



ABOUT THE AUTHOR

Erick Wendel is an active Node.js core team developer, Keynote Speaker, and professional educator. He has given over 100 tech talks in more than 10 different countries worldwide. He was awarded as a Node.js Specialist with the Google Developer Expert, Microsoft MVP, and GitHub Stars awards. Erick Wendel has trained more than 100K people around the world in his own company https://erickwendel.com

He's already done other interesting subjects such as creating Web APIs from scratch and with no frameworks using Node.js, recreating the Web Socket Protocol using raw JavaScript, and the most interesting one, a video about Recreating Node.js from scratch on which this book was based.

This is the **first and only content on the Internet** (as of the time of this writing) that shows how to create a JavaScript Runtime from scratch, using V8, libuv, and more.

If you find any issue consider reaching out the author on @erickwende_

We hope you enjoy it.

TL;DR

If you want to skip the tutorial and go straight to the code, you can find it here, and you can also check out the same content in my video here.

Table of Contents

1 Introduction **1**

1.1 The sea of runtimes 2

1.2 JavaScript runtimes? Interpreters? 3

2 How was Node built? **5**

3 Understanding the main components used in Node.js **9**

3.1 Node.js main components: 1/3 - V8 engine 10

3.2 Node.js main components: 2/3 - Libuv 15

3.3 Node.js main components: 3/3 - C++ Layer 17

4 Creating your own JS engine **18**

4.1 Setting up the dev environment 19

4.2 Diving into the template 20

4.3 Diving into the Capivara initial code 27

4.4 Adding life to the project: Implementing libuv timers 32

5 Going further - Reading files using libuv (challenge) **50**

6 Conclusion **52**

Version: 27.9.18

Chapter 1

INTRODUCTION

This book will walk you throughout the whole process of creating a JavaScript runtime using the same components as Node.js. You'll be able to understand what is a runtime, what's the responsbility of JavaScript and relashionship with C++ extensions.

In this book you'll create a JavaScript runtime that:

- Evaluates JavaScript code
- Uses Libuv as an event loop
- Create timers such as setTimeout and setInterval from C++ code
- Recreate the console.log function
- Synchronize operations between async operations, JavaScript code and C++ land

1.1 THE SEA OF RUNTIMES

You have probably noticed that there are a lot of JavaScript runtimes being created and launched to the world nowadays. Tools such as Deno, Bun, and Cloudflare Workers all came up in a short period of time, this wasn't like this in the past.

Creating new JavaScript runtimes is getting so common that a community group called WinterCG was created so people could talk to each other and exchange experiences and design decisions focusing on documenting and improving interoperability of web platform APIs across runtimes (especially non-browser ones).

This got me wondering: "Is it so hard to build a JavaScript runtime from scratch?", "Do I need to create a compiler?", "How can I create my own JS runtime?". And these sent me to other questions to understand if people built their own JavaScript interpreters from the ground up, or if they used popular JavaScript runtimes such as SpiderMonkey used by Firefox, V8 on Chrome, or SquirrelFish used by Safari.

1.2 JAVASCRIPT RUNTIMES? INTERPRETERS?

When I was starting this project I was a bit confused with basic concepts like **What is a runtime? is it the same thing as a interpreter?**

In the context of JavaScript, a **JavaScript runtime** is where a JavaScript code string is interpreted and evaluated. Therefore, if you open the browser inspector in any of the browsers listed below and type a JavaScript code, you will see that it is executed and the result is printed out on the console. This would only happen because of a JavaScript runtime being executed behind the scenes.

JS runtimes are commonly seen in browsers but they don't stick only to Web environments. In fact you can create a program that uses, let's say Chrome's V8 engine, and make calls back and forth to it.

> Spoiler alert: that's how Node.js executes JavaScript on server environments

Even though you haven't created the runtime yourself this program is also a *JavaScript runtime* because it receives a string, evaluates it, and sends the result back to the client.

On other hand, an interpreter is the part of the runtime that will interpret that code and convert it into something "runnable" (which you usually don't have access to).

Often we hear the words runtime and engine as synonyms. The engine is the complete system that interprets and validates code and then the runtime executes it. For the end-user, the runtime is embedded in the engine and that's why we've seen people referring to them as the same thing.

CHAPTER 2

HOW WAS NODE BUILT?

To understand how everything began, I jumped to the Node.js source code on the V0.0.1 branch, I was looking for commits that Ryan Dahl wrote to implement the first proof of concept of an application that could run JavaScript on the server.

One interesting thing I noticed is that Node is, in fact, "just" a proxy to Chrome's V8 engine. In essence what it does is to send a JS string to V8 and it executes that string, returning a result like you can see in this bit of code from the node repository:

```
Handle<Value> ExecuteString(
  v8::Handle<v8::String> source,
  v8::Handle<v8::Value> filename
  ) {
    HandleScope scope;
    TryCatch try_catch;

    Handle<Script> script = Script::Compile(source, filename);
    if (script.IsEmpty()) {
      ReportException(&try_catch);
      ::exit(1);
    }

    Handle<Value> result = script->Run();
    if (result.IsEmpty()) {
      ReportException(&try_catch);
      ::exit(1);
    }
    return scope.Close(result);
  }
```

A few years ago Ryan Dahl presented a talk where he speaks of things he regrets about Node.js and presents **Deno**. A new JS runtime meant to fix all problems Node.js had.

Deno is written in Rust built also on top of the V8 engine. The interesting part for me is that both of them follow the same paradigm.

This code snippet that I got from their public repo on GitHub is a variation of what I just showed earlier, a program that extends the V8 engine to execute JavaScript code:

```
deno_core::v8_set_flags(env::args().collect());

let mut js_runtime = create_js_runtime();
let runtime = tokio::runtime::Builder::new_current_thread()
  .enable_all()
  .build()
  .unwrap();

let future = async move {
  js_runtime
    .execute_script(
      "http_bench_json_ops.js",
      include_str!("http_bench_json_ops.js"),
    )
    .unwrap();
  js_runtime.run_event_loop(false).await
};
```

What about our next competitor? **Bun** is yet another JS runtime written in Zig that promises to be faster than Node and Deno.

But in the end, can you guess what it does? Sends JS code to a JS runtime, this time it's the **JavaScriptCore** engine, also known as SquirrelFish. This is pretty evident on the runtime calls to import JSCore modules:

```
#include "JavaScriptCore/JavaScript.h"
#include "wtf/FileSystem.h"
#include "wtf/MemoryFootprint.h"
#include "wtf/text/WTFString.h"
#include "JavaScriptCore/CodeBlock.h"
#include "JavaScriptCore/JSCInlines.h"
#include "JavaScriptCore/TestRunnerUtils.h"
#include "JavaScriptCore/JIT.h"
#include "JavaScriptCore/APICast.h"
#include "JavaScriptCore/JSBasePrivate.h"
#include "JavaScriptCore/ObjectConstructor.h"
#include "JavaScriptCore/AggregateError.h"
#include "JavaScriptCore/BytecodeIndex.h"
#include "JavaScriptCore/CallFrameInlines.h"
#include "JavaScriptCore/ClassInfo.h"
#include "JavaScriptCore/CodeBlock.h"
#include "JavaScriptCore/CodeCache.h"
#include "JavaScriptCore/Completion.h"
#include "JavaScriptCore/Error.h"
#include "JavaScriptCore/ErrorInstance.h"
#include "JavaScriptCore/HeapSnapshotBuilder.h"
#include "JavaScriptCore/JSONObject.h"
#include "JavaScriptCore/DeferTermination.h"
#include "JavaScriptCore/SamplingProfiler.h"
#include "JavaScriptCore/VMTrapsInlines.h"
```

So what's the real difference between them? If they're "just" proxies to runtimes that execute code, how can they be better than each other? The answer relies not upon the JS itself, but on the way they control the data flow back and forth from the engine, and how they handle OS tasks.

Let's imagine a use case. You sent a quick JS code that prints a huge string. Deno uses Rust, Node uses C++, and each of them will parse the string in a way that they'll need to cut it into pieces so they can call back a JavaScript function via the engine's API.

The algorithm they use to both import the modules you're using, as well as parse and process that huge string is what makes one faster than the other.

Imagine that there are two different cars with the same engine. The way they're designed to resist the air, the weight, and the shape is what makes one faster than another.

Bun uses two key components that make them claim they're faster than any other runtime currently in the market.

They use Zig as a programming language and Apple's JavaScript engine JavaScriptCore (also known as SquirrelFish).

This leads to what I've said before. Even though they're interfacing with pre-built runtimes, Node, Deno, and Bun are also considered runtimes themselves because they not only interpret JavaScript code but also extend engines adding special features. Some examples of that are **file handling, OS processes, threads, timers**, and much more.

CHAPTER 3

UNDERSTANDING THE MAIN COMPONENTS USED IN NODE.JS

Going back to the idea of recreating the Node.js project. **We should ask ourselves: What is Node.js?**

Node.js is a system built into three main components:

- Chrome's V8 JavaScript engine
- Libuv - async operations
- C++ Layer - functions to help control the data flow

In this section, you're gonna learn the main components and why they're important.

3.1 NODE.JS MAIN COMPONENTS: 1/3 - V8 ENGINE

> Before writing your own JavaScript runtime I'll explain how the V8's engine works but you can jump right to Creating your own JS engine section if you want to write code right away.

As I told you the V8 engine is responsible for interpreting your JavaScript code and execute it. Although, it does much more than that. It translates your JavaScript code to C++ object instances.

The most interesting thing for me is that `console.log` doesn't exist on V8. If you call this function you'll get the error **"console is undefined"**.

console.log is not JavaScript... It's a C++ function that calls a function that you might have heard of, the C++ `printf` .

That's because V8 run only what's on the ECMAScript's specification such as **promises, classes, functions, and variables** but other commonly used functions like **console, setTimeout, and setInterval** are not part of this specification, so V8 doesn't even know what they are.

Below is a piece of code I used to create my own `console.log` implementation. I called it `Print` as I didn't want to create an object `console` and add a function `log`

```
void Print(const v8::FunctionCallbackInfo<v8::Value> &args) {
  bool first = true;
  for (int i = 0; i < args.Length(); i++) {
    v8::HandleScope handle_scope(args.GetIsolate());
    if (first) {
      first = false;
    }
    else {
      printf(" ");
    }
    v8::String::Utf8Value str(args.GetIsolate(), args[i]);

    printf("%s", *str);
  }
  printf("\n");
  fflush(stdout);
}
```

checkout the complete code here

Don't get too trapped in the C++ implementation, in the end, the answer is the `printf` method, one of the most primitive C++ functions is also used on Node.js to print the code to the standard output.

Mindblowing, isn't it?

After I create the `Print` function, I'll need to inject it into the global JavaScript context. And here things get interesting.

```
v8::Local<v8::Context> CreateContext(v8::Isolate *isolate) {
  // Create a template for the global object.
  v8::Local<v8::ObjectTemplate> global =
    v8::ObjectTemplate::New(isolate);

  // Bind the global 'print' function
  // to the C++ Print callback.
  global->Set(
    isolate,
    "print",
    v8::FunctionTemplate::New(isolate, Print)
  );

  // Create a new context.
  return v8::Context::New(isolate, NULL, global);
}
```

checkout the complete code here

In `global->Set` is where I'm mapping the C++ function `Print` I just wrote to become a `print` function available in the global JavaScript context.

Then I can use it like this in the `index.js` file:

```
print('Hello World')
```

While using a JavaScript file calling the `print` function I just implemented on C++ land, the engine will understand it as a JavaScript function and call my function built from C++.

That's why I said Node.js is an extension of the V8 engine because you can implement whatever functions JavaScript doesn't have, the same way we did up there.

Don't believe in me? look at this code from Node v0.12 which implements the `fs` module:

```
NODE_SET_METHOD(target, "access", Access);
NODE_SET_METHOD(target, "close", Close);
NODE_SET_METHOD(target, "open", Open);
NODE_SET_METHOD(target, "read", Read);
NODE_SET_METHOD(target, "fdatasync", Fdatasync);
NODE_SET_METHOD(target, "fsync", Fsync);
NODE_SET_METHOD(target, "rename", Rename);
NODE_SET_METHOD(target, "ftruncate", FTruncate);
NODE_SET_METHOD(target, "rmdir", RMDir);
NODE_SET_METHOD(target, "mkdir", MKDir);
NODE_SET_METHOD(target, "readdir", ReadDir);
NODE_SET_METHOD(target, "stat", Stat);
NODE_SET_METHOD(target, "lstat", LStat);
NODE_SET_METHOD(target, "fstat", FStat);
NODE_SET_METHOD(target, "link", Link);
NODE_SET_METHOD(target, "symlink", Symlink);
NODE_SET_METHOD(target, "readlink", ReadLink);
NODE_SET_METHOD(target, "unlink", Unlink);
NODE_SET_METHOD(target, "writeBuffer", WriteBuffer);
NODE_SET_METHOD(target, "writeString", WriteString);
```

This is how other modules like **crypto, HTTP, net, and child process** are also built. They're just C++ functions that extend V8's default behavior. And this is also how *Bun* and *Deno* can implement those functions differently and claim to be faster than *Node.js*.

JavaScript Promises

Another very important part of our runtime is **Promises**. As you may not know, promises are, in fact, a design pattern, they're not async functions. What they do is wrap async functions and return an object that handles whether a function has finished successfully or not.

As Promises **are** part of the ECMAScript spec, you can run them directly to V8.

The code shown bellow, we're gonna build it later. For now, just notice that the `timeout` .

I'm just wrapping it into a **Promise** object. When the *C++* layer calls the callback function from the `timeout` function, it'll also call the `resolve` function from the *Promise* object which tells our `await` keyword that the function has succesfully finished.

```
const setTimeout = (ms, cb) => timeout(ms, 0, cb)
const setTimeoutAsync = (ms) =>
  new Promise((resolve) => setTimeout(ms, resolve))

;(async function asyncFn() {
  print(new Date().toISOString(), 'waiting a sec...')
  await setTimeoutAsync(1000)
  print(new Date().toISOString(), 'waiting a sec...')
  await setTimeoutAsync(1000)
  print(new Date().toISOString(), 'finished at')
})()
```

In the end, a *Promise* object is just a pretty way of handling callback functions inline.

3.2 NODE.JS MAIN COMPONENTS: 2/3 - LIBUV

As I said, neither `setTimeout` , `setInterval` , nor `setImmediate` exists on JavaScript. Those are timer functions and they're async functions handled by a well-known C library called **libuv**.

Libuv is a C library that empowers Node.js to create:

- Async Functions
- Threads
- Timers
- Child Processes
- Event Loops

And much more.

It was originally made to help Node.js but it's extensible and can be used on other languages as well. And the result it has with its **event loop** is what empowers Node.js core and transformed it into one of the biggest and most used runtimes in the world.

How? By allowing it to run async operations. Let's take, for example, `setTimeout` .

Timers are async functions that run in the background and call back the main context when they're done. Every time we run `setTimeout` in JavaScript, it's Libuv executing that code in the background and calling the provided callback. This is similar to how a game loop functions in games, an endless `while` loop that keeps asking if there are new events in a game and calls the provided functions back when finishing executing.

This is what we all know as the **single-threaded** part of JavaScript. And, even though you can create and manipulate threads with Libuv, every task will eventually send a message back to the event loop to keep consistency and order, this will keep tasks from conflicting with each other and causing **deadlocks**.

In summary that's how Node.js can work with multi-threads using the Worker Threads module.

Every time a task has finished it'll send a message back to the event loop and the event loop will call its callback and remove the function from the queue.

3.3 NODE.JS MAIN COMPONENTS: 3/3 - C++ LAYER

V8 is the engine that interprets JavaScript and can call custom C++ functions and libuv is the library that provides the event loop and other capabilities such as threading and performing async tasks on the operating system.

The last part of the Node.js system is what I call the **C++ layer**. The C++ Layer is the mediator between the JavaScript code you write, the V8 engine, and Libuv. It handles responses from V8 and Libuv and replies to the JavaScript layer.

Imagine the following pipeline:

1. **Execute a C++ program and send a JS file as the argument**
    - Executed by the C++ layer, which means, your program.
2. **Read the contents of the file**
    - Also executed by the C++ layer.
3. **Send the string to the V8 engine and it transforms the code into a C++ object**
    - V8 evaluating the string you sent.
4. **Wait for events, timers, processes, and other async calls to finish processing**
    - This is the Libuv event loop running as an endless loop.
5. **Libuv finishes the task and calls your given C++ functions**
    - The C++ layer receives the response.
6. **The C++ layer calls V8 API to reply to the JS function**
    - The C++ layer invokes the callback function provided and finishes the request.

CHAPTER 4

CREATING YOUR OWN JS ENGINE

In this section, you're gonna implement your JS Engine using V8, Libuv and C++. It won't be needed to compile binaries as I set all the environment on a GitPod.

I put a challenge for you go to further and extend this experiment after this section

4.1 SETTING UP THE DEV ENVIRONMENT

Installing the C++ libs on your own is demanding as you'd need to install and compile V8, Libuv, and C++ dependencies targeting your machine's hardware architecture (such as arm64, x86_64, and x86).

To solve this issue for you, I've prepared a GitPod environment so you can run everything pre-compiled and not have to go through this hassle. This will allow you to get up to speed very quickly and start playing with the code in a matter of seconds.

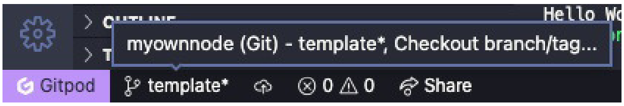
Click the link, log in to your account, and you are ready to go!

> In case you want to compile it yourself, I gathered the instalation process on the scripts directory.

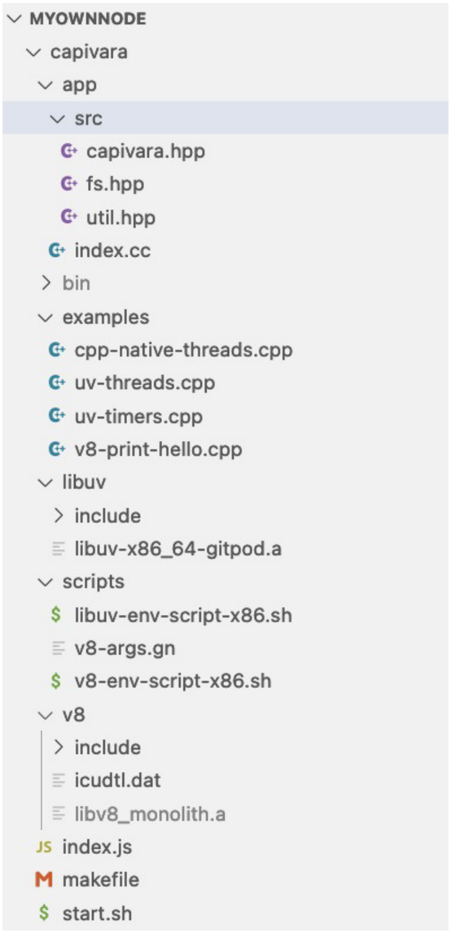
4.2 DIVING INTO THE TEMPLATE

I called this JavaScript runtime **Capivara BR**. A Capivara (Capybara) is a very common animal in South America and I thought I could share a bit of Brazilian culture with you all! Although you can rename it as you like.

Let's start by opening the GitPod and doing some code. When you get to your GitPod, choose the `template` branch and open the `capivara` directory.



This should be the file tree:



```
MYOWNNODE
  capivara
    app
      src
        capivara.hpp
        fs.hpp
        util.hpp
      index.cc
    bin
    examples
      cpp-native-threads.cpp
      uv-threads.cpp
      uv-timers.cpp
      v8-print-hello.cpp
    libuv
      include
      libuv-x86_64-gitpod.a
    scripts
      libuv-env-script-x86.sh
      v8-args.gn
      v8-env-script-x86.sh
    v8
      include
      icudtl.dat
      libv8_monolith.a
    index.js
    makefile
    start.sh
```

Looking at the `capivara/start.sh` file you'll find the following code:

```
if ! command -v "nodemon" /dev/null
then
  npm i -g nodemon --silent
fi

if ! command -v "ccache" /dev/null
then
  sudo apt-get install -y ccache
fi

nodemon -e cc,h,js,cpp,hpp
  --exec "make && ./bin/capivara index.js"
```

What I'm doing here is installing `nodemon`, which is widely used in the Node ecosystem, but it can also watch for other extensions, so I'm using it to build and watch our C++ code.

Let's run this `start.sh`. First, go to the directory using `cd capivara` and run `./start.sh`.

```
[nodemon] 2.0.21
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: cc,h,js,cpp,hpp
[nodemon] starting `make && ./bin/capivara index.js`

mkdir -p bin

ccache g++ $APP -I $INCLUDE -I $INCLUDEUV  \
-std=c++17 -pthread -o $OUTPUT_FILE -DV8_COMPRESS_POINTERS \
$OBJ -Wl,--no-as-needed -ldl

Hello World
[nodemon] clean exit - waiting for changes before restart
```

You should see a `Hello World` text on the terminal as above. This means that the V8 engine is set up as well as other dependencies so we can move forward.

If you run `make` on the `capivara` directory, we'll see that a `bin` folder was created in the directory and inside it, there is a binary file called `capivara`.

Then run `./bin/capivara` so you'll get an error stating there's no file to be read. Notice that this program needs a JavaScript file to execute.

In the `capivara` directory, there's an `index.js` file that we need to send to our program.

Run it as `./bin/capivara index.js` and you'll see an output with a `Hello World` text as follows:

```
./bin/capivara index.js
Hello World
```

I let in the examples directory how each piece of our puzzle works itself.

In the uv-timers.cpp file I let the example of how to use libuv timers to reproduce the `setTimeout` function as follows:

```cpp
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>

#include <uv.h>

uv_loop_t *loop;
uv_timer_t gc_req;
uv_timer_t fake_job_req;

void callbackFn() {
    printf("callback executed!\n");
}

typedef void (*functionTemplate)(void);

struct timer {
    uv_timer_t req;
    std::string text;
    functionTemplate *callback;
};


void work(uv_timer_t *handle) {
    timer *timerWrap = (timer *)handle->data;
    ((functionTemplate)timerWrap->callback)();

    printf("%s", timerWrap->text.c_str());
}

int main() {
    loop = uv_default_loop();
    for (size_t i = 0; i < 10; i++) {
        timer *timerWrap = new timer();
        timerWrap->callback = (functionTemplate *)callbackFn;
        timerWrap->text = "hello\n";

        timerWrap->req.data = (void *)timerWrap;

        uv_timer_init(loop, &timerWrap->req);
        int delay = 500 + i;
        int interval = 0;
        uv_timer_start(&timerWrap->req, work, delay, interval);
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

The `main` function will set up the libuv event loop and schedule 10 functions that will be executed in the future.

The `delay` variable there defines when the `work` will be executed. Notice that, this is exactly how a `setTimeout` function works in Node.js.

Run `make uv-timers` you'll see the code with the output.

```
callback executed!
hello
callback executed!
hello
callback executed!
```

In case you wanna check out all the other examples in the directory, run `make <example-name>` and you'll see the output.

To make things as simple as possible I included all the **CPP headers** for V8 and libuv in the `capivara/v8/include` and `capivara/libuv/include` directories.

Those headers are mappings for functions that are on the `v8` and `libuv` binaries. For example, in the `uv_timers` example, you'd only run the `uv_run` function because you've imported the `<uv.h>` header first.

You can also see where headers are being imported on the `makefile` file:

```
define INCLUDE
    v8/include/
endef

define INCLUDEUV
    libuv/include/
endef
```

And how I use them for compiling the code:

```
build:
    mkdir -p bin
    $(CXX) $$APP -I $$INCLUDE -I $$INCLUDEUV  -std=c++17  \
    -pthread -o $$OUTPUT_FILE \
    -DV8_COMPRESS_POINTERS $$OBJ \
    -Wl,--no-as-needed -ldl
```

4.3 DIVING INTO THE CAPIVARA INITIAL CODE

Notice that I'm not a C++ developer. So you may find C++ bad practices in the following sections.

In the `capivara/app/src` directory, you'll see there are several `.hpp` files. The `utils.hpp` file has a bunch of utility functions like `v8_str` which converts a C++ string into a V8 string:

```
static inline v8::Local<v8::String> v8_str(const char *x) {
    return v8::String::NewFromUtf8(
      v8::Isolate::GetCurrent(), x
    ).ToLocalChecked();
}
```

I put those functions there just to make our life easier and avoid replicating code everywhere.

Going on the `capivara/app` directory let's check the `index.cc` file as follows:

```
#include "v8.h"
#include "./src/capivara.hpp"
int main(int argc, char *argv[]) {
    char *filename = argv[1];
    auto *capivara = new Capivara();
    std::unique_ptr<v8::Platform> platform =
        capivara->initializeV8(argc, argv);

    capivara->initializeVM();
    capivara->InitializeProgram(filename);
    capivara->Shutdown();
    return 0;
}
```

This code will grab the filename passed by the CLI (eg. `./bin/capivara index.js`) and initialize the project.

On `capivara/app/src/capivara.hpp` , there's a function called `ExecuteScriptAndWaitForEvents` , which takes as an argument the file name:

```
void ExecuteScriptAndWaitForEvents(char *filename)
{
  // Enter the context for compiling and
  // running the hello world script.
  v8::Context::Scope context_scope(this->context);
  {
    // Read file from args
    v8::Local<v8::String> source;
    if (!Fs::ReadFile(isolate, filename).ToLocal(&source))
    {
      fprintf(stderr, "Error reading file\n");
      return;
    }

    // Create a string containing the JavaScript source code.
    v8::ScriptOrigin origin(isolate, v8_str(filename));

    // Compile the source code.
    v8::Local<v8::Script> script =
      v8::Script::Compile(
        this->context, source, &origin
      ).ToLocalChecked();

    // Run the script to get the result.
    v8::Local<v8::Value> result = script->Run(
      this->context
    ).ToLocalChecked();

    // Convert the result to an UTF8 string and print it.
    v8::String::Utf8Value utf8(this->isolate, result);

    WaitForEvents();
  }
}
```

Let's take a moment to analyze this code. We set a context scope, which means that all the following code will be executed inside a specific box.

It reads the content and maps all JavaScript code to V8 instances (using the `::Compile` function). Then V8 executes the code and put the result in a variable to be printed later.

Right after executing the code, I call the `WaitForEvents` function to start listening for async functions and events that may have been scheduled during the execution.

Then going on the `WaitForEvents` function you'll notice that it's starting the libuv's event loop with `uv_run`.

```
uv_loop_t *DEFAULT_LOOP = uv_default_loop();

class Capivara {
private:
    v8::Isolate *isolate;
    v8::Local<v8::Context> context;
    std::unique_ptr<v8::Platform> *platform;
    v8::Isolate::CreateParams create_params;

    void WaitForEvents() {
        uv_run(DEFAULT_LOOP, UV_RUN_DEFAULT);
    }
```

Initializing the V8 engine

If you go back to the `index.cc` file, you'll notice some functions I call to initialize the V8 engine such as `initializeV8`, `initializeVM`, and others as follows:

```
int main(int argc, char *argv[]) {
    char *filename = argv[1];
    auto *capivara = new Capivara();
    std::unique_ptr<v8::Platform> platform =
      capivara->initializeV8(argc, argv);

    capivara->initializeVM();
    capivara->InitializeProgram(filename);
    capivara->Shutdown();

    return 0;
}
```

Let's have a look at the `initializeV8` function in the `capivara.hpp` file. It's responsible for initializing the V8 virtual machine as follows:

```
public:
    std::unique_ptr<v8::Platform> initializeV8(
        int argc,
        char *argv[]
      ) {
        std::unique_ptr<v8::Platform> platform =
          v8::platform::NewDefaultPlatform();

        v8::V8::InitializePlatform(platform.get());
        v8::V8::Initialize();

        this->platform = &platform;
        return platform;
    }
```

The next function is called is the `initializeVM` . It creates a context, like I said, a little box where our code will run and not leak any data outside of its defined boundaries:

```
void initializeVM() {
  // Create a new Isolate and make it the current one.
  v8::Isolate::CreateParams create_params;
  create_params.array_buffer_allocator =
    v8::ArrayBuffer::Allocator::NewDefaultAllocator();
  this->isolate = v8::Isolate::New(create_params);
  this->create_params = create_params;
}
```

Then `InitializeProgram` is where we inject C++ custom functions into the global V8 context on a specific scope and call `ExecuteScriptAndWaitForEvents` to start the actual JavaScript program.

```
void InitializeProgram(char *filename) {
  v8::Isolate::Scope isolate_scope(this->isolate);

  // Create a stack-allocated handle scope.
  v8::HandleScope handle_scope(this->isolate);

  // Create a template for the global object.
  v8::Local<v8::ObjectTemplate> global =
    v8::ObjectTemplate::New(isolate);

  // Bind the global 'print' function to the C++ Print callback.
  global->Set(
      isolate,
      "print",
      v8::FunctionTemplate::New(isolate, Print)
  );

  // Create a new context.
  this->context = v8::Context::New(this->isolate, NULL, global);

  ExecuteScriptAndWaitForEvents(filename);
}
```

4.4 ADDING LIFE TO THE PROJECT: IMPLEMENTING LIBUV TIMERS

If we go to the `index.js` file and write some more JS on it, we can better understand how things work. If you haven't executed the `./start.sh` script, execute it now so we can have it running for our code to be compiled in real-time.

Let's try to write something like this:

```
// index.js
setTimeout(() => {
  print('hello world')
}, 1000)
```

If you check your console after the execution, you'll see that `setTimeout` is not a defined function, which means it's not a part of JavaScript and we should implement it.

```
index.js:0: Uncaught ReferenceError: setTimeout is not defined

#
# Fatal error in v8::ToLocalChecked
# Empty MaybeLocal
#

Trace/breakpoint trap
[nodemon] app crashed - waiting for file changes before starting.
..
```

So let's remove the code that uses `setTimeout` from `index.js` and create a `timeout` function instead. First, let's think about the signature, I want it to look a bit like this:

```
// index.js
let interval = 0
let sleep = 200

timeout(sleep, interval, () => {
  print(`1 ${new Date().toISOString()}`)
})
```

Then we'll jump back to the `capivara/app/src` directory and create a new `timer.hpp` file where we'll define the content of the function. First, let's import both V8 and libuv and create the `Timer` class:

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
      const v8::FunctionCallbackInfo<v8::Value> &args
    ) { }
};
```

Then go back to the `capivara.hpp` file. Import the `./timer.hpp` module at the beginning of the file.

After it, on the `InitializeProgram` function initialize the timer as the comments show below:

> Notice that I'll be using the pattern `// ...omitted` to refer that all remaining code of that section were omitted to avoid replicating code on the tutorial

```cpp
// capivara.hpp
#include <libplatform/libplatform.h>
#include <uv.h>
#include "v8.h"

#include "./fs.hpp"
#include "./util.hpp"

// 1 - Import our timer
#include "./timer.hpp"

// ...omitted

void InitializeProgram(char *filename)
{
  // ...omitted
  global->Set(
      isolate,
      "print",
      v8::FunctionTemplate::New(isolate, Print)
  );

  // 2 - Initialize timer
  Timer timer;
  timer.Initialize(DEFAULT_LOOP);

  // 3 - inject Timeout function in the context
  global->Set(
    isolate,
    "timeout",
    v8::FunctionTemplate::New(isolate, timer.Timeout)
  );

  this->context = v8::Context::New(
    this->isolate,
    NULL,
    global
  );

  ExecuteScriptAndWaitForEvents(filename);
}
```

Now, if you execute our file again, you won't get any other errors, as we already have defined the `timeout` function. What's left is to implement the function. Let's start by defining isolates and the runtime context:

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
      const v8::FunctionCallbackInfo<v8::Value> &args
    ) {
      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
    }
};
```

Then you need to get the parameters. The first two parameters will be integers, so let's print them:

```
// timer.hpp
#include <v8.h>
#include <uv.h>
```

```
uv_loop_t *loop;

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
      const v8::FunctionCallbackInfo<v8::Value> &args
    ) {
      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();

      // 1 - sleep param
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();

      // 2 - interval param
      int64_t interval = args[1]->IntegerValue(context)
        .ToChecked();

      // Just printing out params
      printf("sleep %ld, interval %ld\n", sleep, interval);
    }
};
```

I won't get into details about the V8 API we're using on `ToChecked` and other methods, but you can find everything in the project's official repo.

If you look at your console now, it should be printing the first two parameters as:

```
sleep 200, interval 0
```

For callbacks, this is a bit harder. You'll deal with a JS function that needs to be parsed into a V8 function in the C++ world.

To check what's in the third param, let's print it out as a string:

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
      const v8::FunctionCallbackInfo<v8::Value> &args
    ) {

      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();
      int64_t interval = args[1]->IntegerValue(context)
        .ToChecked();
      // checking the receiving params
      printf("sleep %ld, interval %ld\n", sleep, interval);

      v8::String::Utf8Value callbackStr(isolate, args[2]);
      printf("%s", *callbackStr);
    }
};
```

If you run that, you'll see that the code is printing the string of the function, which means this is working.

```
sleep 200, interval 0
() => {
  print(`1 ${new Date().toISOString()}`)
}
```

Now you're gonna implement the UV timers as you've already done on `uv_timers.cpp` file in the `examples` directory.

Copy the `work` function signature and paste it into the `timer.hpp` file. Rename it to `onTimerCallback` and annotate it with the `static` modifier as the other functions already are.

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
      const v8::FunctionCallbackInfo<v8::Value> &args
    ) {

      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();

      int64_t interval = args[1]->IntegerValue(context)
        .ToChecked();

      printf("sleep %ld, interval %ld\n", sleep, interval);

      v8::String::Utf8Value callbackStr(isolate, args[2]);
      printf("%s", *callbackStr);
    }

    // 1 - here
      static void onTimerCallback(uv_timer_t *handle) {
      printf("Hey I was called!");
    }
};
```

We also need to use two other functions to make it work. First, you need to start the UV loop during the `Timeout` function. You can also remove the print functions as you don't need them anymore:

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

class Timer {

  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
          const v8::FunctionCallbackInfo<v8::Value> &args
        ) {

      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();

      int64_t interval = args[1]->IntegerValue(context)
        .ToChecked();

      v8::String::Utf8Value callbackStr(isolate, args[2]);

      // 1 - added here
      uv_timer_init(loop);
    }

      static void onTimerCallback(uv_timer_t *handle) {
      printf("Hey I was called!");
    }
};
```

The function also needs a `uv_timer` . Let's create a C++ `struct` at the beginning of the file and finally, set up the `uv_timer_init` and `uv_timer_start` with the proper parameters as follows:

```cpp
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

// 1 - add the struct
struct timer {
  uv_timer_t uvTimer;
};

class Timer {

  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
        const v8::FunctionCallbackInfo<v8::Value> &args
      ) {
      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();

      int64_t interval = args[1]->IntegerValue(context)
        .ToChecked();


      v8::String::Utf8Value callbackStr(isolate, args[2]);

      // 2 - initialize the struct
      timer *timerWrap = new timer();

      // 3 - initialize the uv timer
      uv_timer_init(loop, &timerWrap->uvTimer);

      // 4 - configure the timer with the onTimerCallback
      //    and params
      uv_timer_start(
        &timerWrap->uvTimer,
        onTimerCallback,
        sleep,
        interval
      );

    }

      static void onTimerCallback(uv_timer_t *handle) {
      printf("Hey I was called!");
    }
};
```

Going on to the terminal, you should see the output as below:

```
Hey I was called!
```

Notice that, this is not the function defined on the JavaScript side, it's just the `print` defined in the `onTimerCallback` implementation.

To invoke the callback function defined from the JavaScript file, you'll need to modify the struct to include an `isolate` .

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

struct timer {
  uv_timer_t uvTimer;

  // 1 - add the property below
  v8::Isolate *isolate;
};

class Timer {
  // omitted
};
```

You'll also need to store the `callback` in the global state to be able to use it after the timeout has ended.

The `v8::Global<type>` data type is the V8's generic global state manager that will help us to store the `callback` in memory until we'll be able to use it.

You'll now

1. add the callback to the `timer` struct,
2. get callback from the arguments
3. store the current `callback` value, and
4. update the `uvTimer` variable with the `timerWrap` content as the following comments show:

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

struct timer {
  uv_timer_t uvTimer;
  v8::Isolate *isolate;

  // 1 - add the callback to the struct
  v8::Global<v8::Function> callback;
};

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }

    static void Timeout (
        const v8::FunctionCallbackInfo<v8::Value> &args
      ) {

      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();

      int64_t interval = args[1]->IntegerValue(context)
```

```
        .ToChecked();


      // 2 - get callback from the arguments
      v8::Local<v8::Value> callback = args[2];

      if(!callback->IsFunction()) {
        printf("Callback is not a function");
        return;
      }

      timer *timerWrap = new timer();

      // 3 - reset the instance with the current callback value
      timerWrap->callback.Reset(
        isolate,
        callback.As<v8::Function>()
      );

      // 4 - save the timerWrap data on the uv_timer prop
      timerWrap->uvTimer.data = (void *)timerWrap;
      timerWrap->isolate = isolate;

      uv_timer_init(loop, &timerWrap->uvTimer);
      uv_timer_start(
        &timerWrap->uvTimer,
        onTimerCallback,
        sleep,
        interval
      );

    }

      static void onTimerCallback(uv_timer_t *handle) {
      printf("Hey I was called!");
    }
};
```

When the timeout finishes you'll need to access data from that specific request that was stored on the struct within the `onTimerCallback` function.

Change the `onTimerCallback` to grab the `timerWrap` instance from the uv handle, check if the V8's isolate is still running, and then bind the callback function stored on the `timerWrap` object and put it on a local V8 function as follows:

```
static void onTimerCallback(uv_timer_t *handle) {
    timer *timerWrap = (timer *)handle->data;
  v8::Isolate *isolate = timerWrap->isolate;
  v8::Local<v8::Context> context =
    isolate->GetCurrentContext();

  if (isolate->IsDead()) {
    printf("isolate is dead");
    return;
  }

  v8::Local<v8::Function> callback =
      v8::Local<v8::Function>::New(
        isolate,
        timerWrap->callback
      );
}
```

Now that we have the callback instance carried from the JavaScript side, we can invoke it as follows:

```
static void onTimerCallback(uv_timer_t *handle) {
    timer *timerWrap = (timer *)handle->data;
  v8::Isolate *isolate = timerWrap->isolate;
  v8::Local<v8::Context> context =
    isolate->GetCurrentContext();

  if (isolate->IsDead()) {
    printf("isolate is dead");
    return;
  }

  v8::Local<v8::Function> callback =
    v8::Local<v8::Function>::New(
      isolate,
      timerWrap->callback
    );

  // set up what will be used on the callback call
  v8::Local<v8::Value> result;
  v8::Handle<v8::Value> resultr [] = {
    v8::Undefined(isolate),
    v8_str("hello world")
  };

  // calling the JS function as callback(undefined, 'hello world'

  if (callback->Call(
    context,
    v8::Undefined(isolate),
    2,
    resultr
  ).ToLocal(&result)) {
      // callback is a success
    } else {
      // failed
    }
}
```

Your output should look like as below:

```
1 2023-03-14T19:50:07.602Z
```

In case you missed something, below you'll find the full code for the `timer.hpp` class.

```
// timer.hpp
#include <v8.h>
#include <uv.h>

uv_loop_t *loop;

struct timer {
  uv_timer_t uvTimer;
  v8::Isolate *isolate;
  v8::Global<v8::Function> callback;
};

class Timer {
  public:
    static void Initialize(uv_loop_t *evloop) {
      loop = evloop;
    }
    static void Timeout (
        const v8::FunctionCallbackInfo<v8::Value> &args
      ) {

      auto isolate = args.GetIsolate();
      auto context = isolate->GetCurrentContext();
      int64_t sleep = args[0]->IntegerValue(context)
        .ToChecked();

      int64_t interval = args[1]->IntegerValue(context)
        .ToChecked();


      v8::Local<v8::Value> callback = args[2];

      if(!callback->IsFunction()) {
        printf("Callback is not a function");
        return;
      }

      timer *timerWrap = new timer();
      timerWrap->callback.Reset(
        isolate,
        callback.As<v8::Function>()
      );
      timerWrap->uvTimer.data = (void *)timerWrap;
      timerWrap->isolate = isolate;

      uv_timer_init(loop, &timerWrap->uvTimer);
      uv_timer_start(
        &timerWrap->uvTimer,
        onTimerCallback,
        sleep,
        interval
      );

    }

  static void onTimerCallback(uv_timer_t *handle) {
    timer *timerWrap = (timer *)handle->data;
    v8::Isolate *isolate = timerWrap->isolate;
    v8::Local<v8::Context> context =
      isolate->GetCurrentContext();
    if (isolate->IsDead()) {
      printf("isolate is dead");
      return;
    }

    v8::Local<v8::Function> callback =
      v8::Local<v8::Function>::New(
        isolate,
        timerWrap->callback
      );

    v8::Local<v8::Value> result;
    v8::Handle<v8::Value> resultr [] = {
      v8::Undefined(isolate),
      v8_str("hello world")
    };

    if (callback->Call(
        context,
        v8::Undefined(isolate),
        2,
        resultr).ToLocal(&result)) {
          // callback is a success
        } else {
          // failed
        }
  }
};
```

That's breathtaking, isn't it? I've spent weeks just to make the `onTimerCallback` get the callback instance and properly call it.

I haven't found documentation or any examples on the internet showing how to callback a JavaScript function from C++.

Hopefully, my friend Santi Gimeno, who is a Node.js and Libuv core team dev, was able to join me on a call to show me what was I doing wrong there. I'm very grateful for this!

Now, going back to the timer example in the `index.js` file, we can check whether JavaScript is being called in order with the example below:

```
//index.js
let interval = 0
let sleep = 200

timeout(sleep, interval, () => {
  print(`1 ${new Date().toISOString()}`)
  timeout(sleep, interval, () => {
    print(`2 ${new Date().toISOString()}`)
  })
})
```

Your terminal should have the output as the print below:

```
1 2023-03-14T19:50:56.791Z
2 2023-03-14T19:50:56.992Z
```

Remember I told you that Promises are just wrappers around the same callbacks? In the example below you're gonna wrap the `timeout` function into Promise objects and use `async` and `await` keywords to wait for results as shown below:

```
// index.js
let interval = 0
let sleep = 200

const setTimeout = (ms, cb) => timeout(ms, 0, cb)
const setInterval = (ms, cb) => timeout(0, ms, cb)

const setTimeoutAsync = (ms) =>
  new Promise((resolve) => setTimeout(ms, resolve))

;(async function () {
  print(new Date().toISOString(), 'waiting a bit')
  await setTimeoutAsync(1000)
  print(new Date().toISOString(), 'waiting a bit')
  await setTimeoutAsync(1000)
  print(new Date().toISOString(), 'finished')
})()
```

It'll ensure that even though we're handling the callback function from the C++ side we still can use Promises.

```
2023-03-14T19:52:20.162Z waiting a bit
2023-03-14T19:52:21.159Z waiting a bit
2023-03-14T19:52:22.160Z finished
```

That's how the bridge between C++ and JavaScript works on V8. In the end, they're "translated" into the same language and interpreted as one.

CHAPTER 5

GOING FURTHER - READING FILES USING LIBUV (CHALLENGE)

Now you understood behind the scenes of a JavaScript runtime I have a task for you to practice what you've learned here.

Extend this project by adding new features that Node.js and other JavaScript runtimes have.

My suggestion to get started is to implement a function that reads the contents of a file (e.g. `fs.readFile`).

I put an example here about how to read a file using libuv. You'd start by copying the content and pasting it to an existing example in the project.

Let's say that you paste the content into the `uv-threads.cpp` . Then run `make uv-threads` and you'll see an error of **invalid argument** but don't worry.

Stop the terminal and run `bin/uv-threads index.js` so it should run the contents of the `index.js` file and print out the result.

Now that the example is working is time to implement it into your very own JavaScript runtime.

To do so, create a new class named `fs` , paste the example, and adapt it to be called from JavaScript exactly how you did when implementing the `timer` function.

The final expected result is that in the `index.js` file, you should be able to run the `readFile` function and print out the contents of a given file.

```
readFile('file.txt', function (error, result) {
  if(error) {
    print(error);
    return;
  }
  print(result);
})
```

Are you ready? After you finish this challenge, make a post on your social media mentioning that you were able to do it and mention my profile there (I'm @ErickWendel or @ErickWendel_) in the networks so I can get to know that you're done.

CHAPTER 6

CONCLUSION

What a ride! We used **V8** to interpret **JavaScript**, **Libuv** to handle async operations, and **C++** to extend the JavaScript engine.

With the knowledge shown to you here, you'd be able to understand how **Bun**, **Deno**, and other **Node.js** competitors work behind the scenes to make the Web better.

Check out the full video that was used as a reference to build this content. There I show even more concepts and examples for you to go deeper into the JavaScript runtimes world.

Congratulations on reading this extensive content I hope it has exceeded your expectations. Don't forget to follow me on social media networks such as twitter, instagram, linkedin and reach out at any time.

Consider joining our Telegram Channel and don't forget to tell other friends about this content, I think they will love it.

Thank you for your time and for learning something new with me. I'm Erick Wendel and until next time!