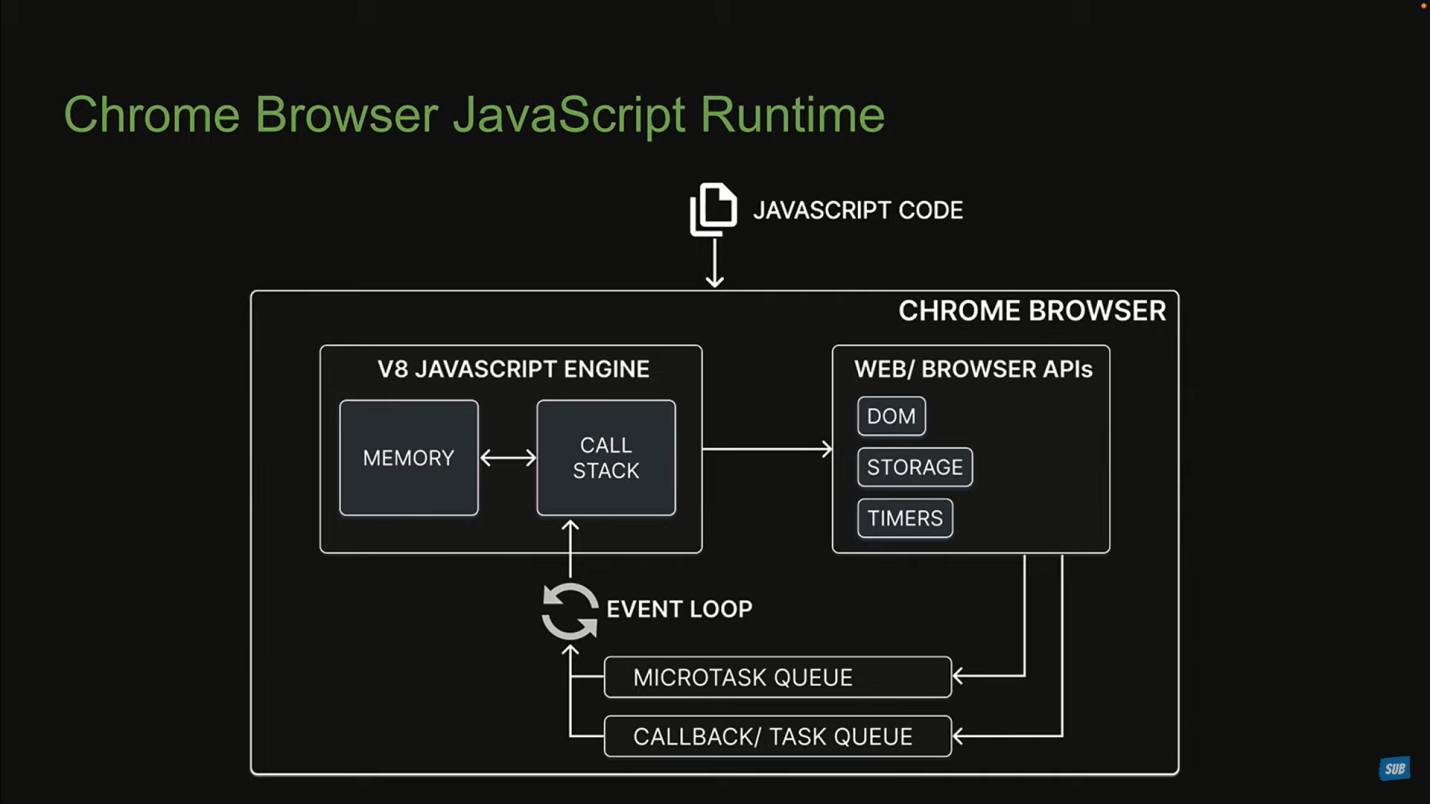
**Node JS**

JavaScript Engine.  
It Converts JS code into Machine code

**JS Run Time**  
It is the environment in which JS program runs.  
Chrome Browser Run Time includes

1. V8 Engine (consists of heap memory, Call Stack)  
2. Web/Browser API’s (DOM, Storage, Timers (setTimeOut, setInterval, Promises)). They are provided by browser.  
3. Queues (Micro Task Queues, Callback Queues, etc.)  
4. Event Loop



**Executing JS with Node**

1. Using Node REPL (Read, Evaluate, Print, Loop). Just type Node in cmd prompt and then perform appropriate actions. (Less Likely to be used.)  
2. Executing code in JS file in cmd prompt. Type node <filename>.js. (More Likely to be used)

**Module**

Module is re-useable chunk of code. In Node JS, every file is a module.  
Types Of Modules: -

1. Local Modules: Modules we create in our application.  
2. Build In Modules: Modules that Node.js ships with out of the box.  
3. Third party Modules: Modules written by other developers that we can use in our application.

**Local Modules**

To Load another module we need ‘require()’ function.  
We can use module.exports to return a module. This returned module can be captured using require() keyword.

**Module Wrapper**

Under the hood, NodeJS does not run our code directly, it wraps the entire code inside a function before execution. This function is termed as Module Wrapper Function or simply Module Wrapper.

Module Wrapper has 5 arguments:

1. \_\_dirname: - Specifies path to folder name of current module.  
2. \_\_filename: - Specifies path of file name of current module.  
3. module: - provides reference to current module.  
4. require: - used to import a module via path.  
5. Exports: -

In any Module we get access to these global variables, but they are specific to the Module in which they are used.

If we export using ‘module.exports’ then we should import using require function  
If we export using ‘export default’ then we should import using ‘import’ keyword

**Asychronous JavaScript**

JavaScript is synchronous, blocking, single threaded language. This nature is however not beneficial for writing apps.  
We want non-blocking asynchronous behaviour which is made possible by browser for frontend and Node.js for backend.

**Build In Modules**

Sometimes referred as Core Modules

1. Path  
2. Events  
3. Fs  
4. Stream  
5. http

**Path Module**

Used while we work for file & directory paths.  
Importing path module.  
const path = require(‘node:path’); // prefixing it with node to indicate that it is built in module.

Or we can write const path = require(‘path’);  
Path Module has about 14 different properties and method exposed. Some of them are as follows:

1. path.basename(\_\_filename or \_\_dirname)  
2. path.extname(\_\_filename or \_\_dirname)  
3. path.parse(\_\_filename)  
4. path.isAbsolute(\_\_filename)  
5. path.join(“folder1”, “folder2”, “index.html” ) --> gives folder1/folder2/index.html  
6. path.resolve(“folder1”, “folder2”, “index.html”) --> will give absolute/complete path

**Events Module**

Const eventEmitter = require(“node:events”); or const eventEmitter = require(“events”);  
const emitter = new eventEmitter();

We can register an event using emitter.on() & emit a method using emitter.emit() method.

**FS Module**

Reading from file  
Const fs = require(“node:fs”);  
const fileContent = fs.readFileSync(<path of file that we want to read>, ‘utf-8’);

fs.readFIle(<path of file that we want to read>, ‘utf-8’,(error,data)=>{

if(error){ console.log(error)}  
 else {console.log(data)}

});

In fs.readFile() method, we can read a file in a non-blocking asynchronous way, but in fs.readFileSync() method, we can read files in a synchronous way, i.e. we are telling node.js to block other parallel process and do the current file reading process.

Writing to file

fs.writeFileSync(<file name>, “<file content>”);  
fs.writeFile(<file name>, “<file content>”, {flag: ‘a’}, (error) => {  
 if(error){console.log(error)}  
 else {console.log(“content has been written”)}

});

If a file to which we are adding data is not present then both these methods will create that file.  
Flag: ‘a’ will append the data with existing file content. Absence of this flag will override the content.

fs.writeFileSync is blocking in nature. fs.writeFile is non-blocking in nature.

Fs.readdir(<directory address>) 🡪 The fs.readdir() method is used to asynchronously read the contents of a given directory.

**createReadStream()**🡪 allow us to open up a file/stream and read the data present in it

**createWriteStream()**🡪 allows to quickly make a writable stream for the purpose of writing data to a file. This method may be a smarter option compared to methods like fs.writeFile when it comes to very large amounts of data.

**Fs Promise Module**

Const fs = require(‘node:fs/promises’);  
fs.readFIle(‘path of file’, ‘utf-8’)  
.then((data)=> {console.log(data)})  
.catch((error)=>{console.log(error)});  
It is non blocking in nature.

For reading from file or writing to file Fs Module has edge over Fs Promise Module in terms of execution time and memory allocation.

**ReadLine Module**

Readline Module in Node.js allows the reading of input stream line by line. For the interaction, we will first create an interface for the input and output.

let rl = readline.createInterface( process.stdin, process.stdout);

Here, the createInterface() method takes two arguments. The first argument will be for the standard input and the second one will be for reading the standard output.

rl.question() method is used for asking questions from the user and reading their reply (output).

**Stream Module**There are 4 types of streams

1. Readable Streams (from which data can be written. Ex- Reading from file as readable stream)
2. Writeable Streams (to which we can write data. Ex- Writing from file as writeable stream)
3. Duplex Streams (that are both Readable & Writeable. Ex- Sockets as duplex stream)
4. Transform Streams (to transform or modify data. Ex- File Compression where you write compressed data and read de-compressed data.)

const readableStream = fs.createReadStream(‘<filepath>’,{encoding: “utf-8”}); // for reading  
const writeableStream =fs.createWriteStream(‘<filepath>’);  
readableStream.on(“data”,(dataChunk)=>{writeableStream.write(dataChunk)});  
in above code “data” is type of eventEmitter & (dataChunk) => {writeableStream.write(dataChunk)} is callback function.

**Pipes**The readable.pipe() method in a Readable Stream is used to attach a Writeable stream to the readable stream.  
Above example can be written as  
readableStream.pipe(writeableStream);  
We can also do chaining using zlib module.

**Libuv**It is a library which handles asynchronous non-blocking operations in node.js.  
It does that using Thread Pool(a.k.a Worker Pool) & Event loop(a.k.a main loop, main thread).

**Http Module**const http = require(node:http);  
const server = http.createServer((req,res)=>{

res.writeHead(200, “Content-Type”: “text/plain”) // status code & content type  
 res.end(“Hello World”); // response

});

server.listen(<port-number>, () => {console.log(“server running on port 3000”)});

In res.end() we can’t send object. We need to convert object into string using JSOn.stringify() & Content-Type should be “application/json”.  
If we want to send html content in res.end() then we need to give Content-Type as “text/html”.  
const htmlFile = fs.readFileSync(“./index.html”, “utf-8”);  
res.send(htmlFile);  
// above 2 lines of code can be written as   
fs.createReadStream(\_\_dirname+ “./index.html”).pipe(res);

**Thread Pool**

Libuv maintains a pool of threads to perform long-running operations in the background, without blocking its main thread.  
Node uses the Worker Pool to handle “expensive” tasks.  
This include I/O for which an operation system does not provide a non-blocking version, as well as CPU-intensive tasks.  
I/O intensive

1. DNS: dns.lookup(), dns.lookupService().
2. File System: All file system API’s except fs.FSWatcher() and those that are explicitly synchronous use libuv’s threadpool.

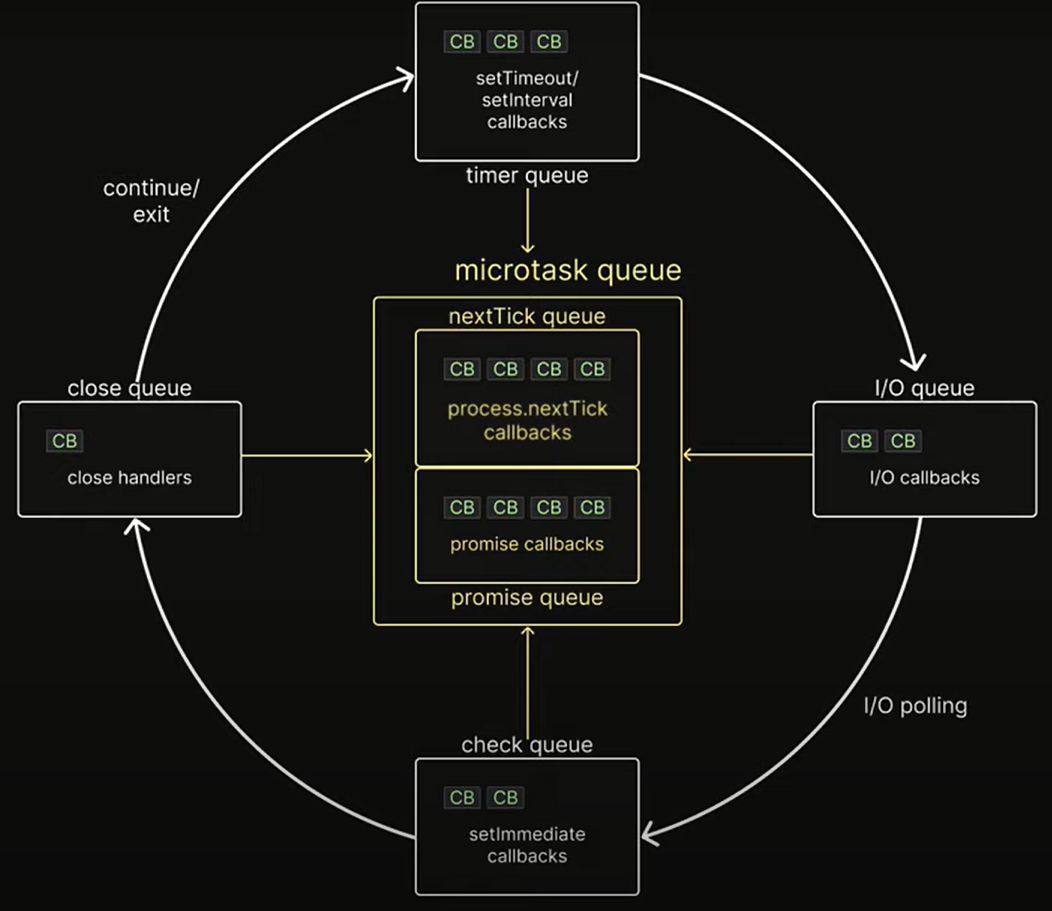
CPU-intensive

1. Crypto: crypto.pbkdf2(), crypt.scrypt(), crypto.randomBytes(), crypto.randomFill(), crypto.generateKeypair()
2. Zlib: All zlib API’s except those that are explicitly synchronous use libuv’s threadpool.

Note: Every function with “sync” suffix runs on main thread and it is blocking in nature.  
By default Thread Pool size is 4 threads, we can increase the thread pool size by setting process.env.UV\_THREADPOOL\_SIZE. For optimal utilization we should limit the thread pool size upto the number of cores of our processor, otherwise CPU will be juggling between threads and will cost the performance.  
Example given in :   
<https://nodejs.org/en/docs/guides/dont-block-the-event-loop/#why-should-i-avoid-blocking-the-event-loop-and-the-worker-pool>

**Event Loop**It is a loop which is alive as long as our application is up & running.  
In every iteration of loop, we come across 6 different types of queues. Each queue hold one or more callback functions, that need to be eventually executed on call stack.

1. Timer Queue: contains callbacks related to setTimeOut and setInterval.
2. I/O Queue: contains callback associated with all the sync methods such as methods associated with fs and http module.
3. Check Queue: contains callback associated with function called set immediate. This function is specific to Node.
4. Close Queue: contains callback associated with close event of async task.
5. Microtask Queue: It contains 2 queues. Next-Tick Queue & Promise Queue.  
   a. Next-Tick Queue: contains callback associated with a function process.nextTick. It is specific to Node js.  
   b. Promise Queue: contains callback that are associated with native promise in javascript.

  
Execution Order in Event Loop  
Synchronous js code always take priority over async code for execution.  
Synchronous code is placed in call stack. Event loop comes into picture once call stack is empty.

In Event Loop we have the following steps:

1. Any callbacks in micro task queue are executed. First, task in nextTick Queue and only then task in promise queue.
2. All callbacks within timer queue are executed.
3. Callbacks in micro task queues if present are executed. Again first, task in nextTick Queue and only then task in promise queue.
4. All call backs within I/O queue are executed.
5. Callbacks in micro task queues if present are executed. Again first, task in nextTick Queue and only then task in promise queue.
6. All callbacks within check queue are executed.
7. Callbacks in micro task queues if present are executed. Again first, task in nextTick Queue and only then task in promise queue.
8. All callbacks with in close queue are executed.
9. For one final time in the same loop, the micro task queues are executed. nextTick Queue followed by promise queue.