

Episode 5: Diving into the NodeJS Github repo.

In this episode, we'll explore how modules actually work behind the scenes. We'll dive into how modules load into a page and how Node.js handles multiple modules, focusing on a deep dive into the

`module.exports` and `require` functions

Behind the scenes

In JavaScript, when you create a function...

```
function x () {  
  const a = 10;  
  function b () {  
    console.log("b");  
  }  
}
```

```
}
```

Will you be able to access `this` value? no
`console.log(a);`

//op - a is not defined

Q: if u execute this code, will you be able to access it outside the function?

A:

You cannot access `a` value outside the function `x` because it is defined within the function's scope. Each function creates its own scope, so variables inside a function are not accessible from outside that function.

To learn more about scope, check out this video: [Understanding Scope in JavaScript](#).

- **imp concept** 😊
- Modules in Node.js work similarly to function scopes. When you require a file, Node.js wraps the code from that file inside a function. This means that all variables and functions in the module are contained within that function's scope and cannot be accessed from outside the module unless explicitly exported.
- To expose variables or functions to other modules, you use `module.exports`. This allows you to export specific elements from the module, making them accessible when required elsewhere in your application.
- All the code of a module is wrapped inside a function when you call `require`. This function is not a regular function; it's a special type known as an IIFE (Immediately Invoked Function Expression). Here's how it works:

```
(function () {
    // All the code of the module runs inside here
})();
```

In this pattern, you create a function and then immediately invoke it. This is different from a normal function in JavaScript, which is defined and then called separately:

```
function x() {}  
x();
```

In Node.js, before passing the code to the V8 engine, it wraps the module code inside an IIFE. The purpose of IIFE is to:

- 1. Immediately Invoke Code:** The function runs as soon as it is defined.
- 2. Keep Variables and Functions Private:** By encapsulating the code within the IIFE, it prevents variables and functions from interfering with other parts of the code. This ensures that the code within the IIFE remains independent and private.

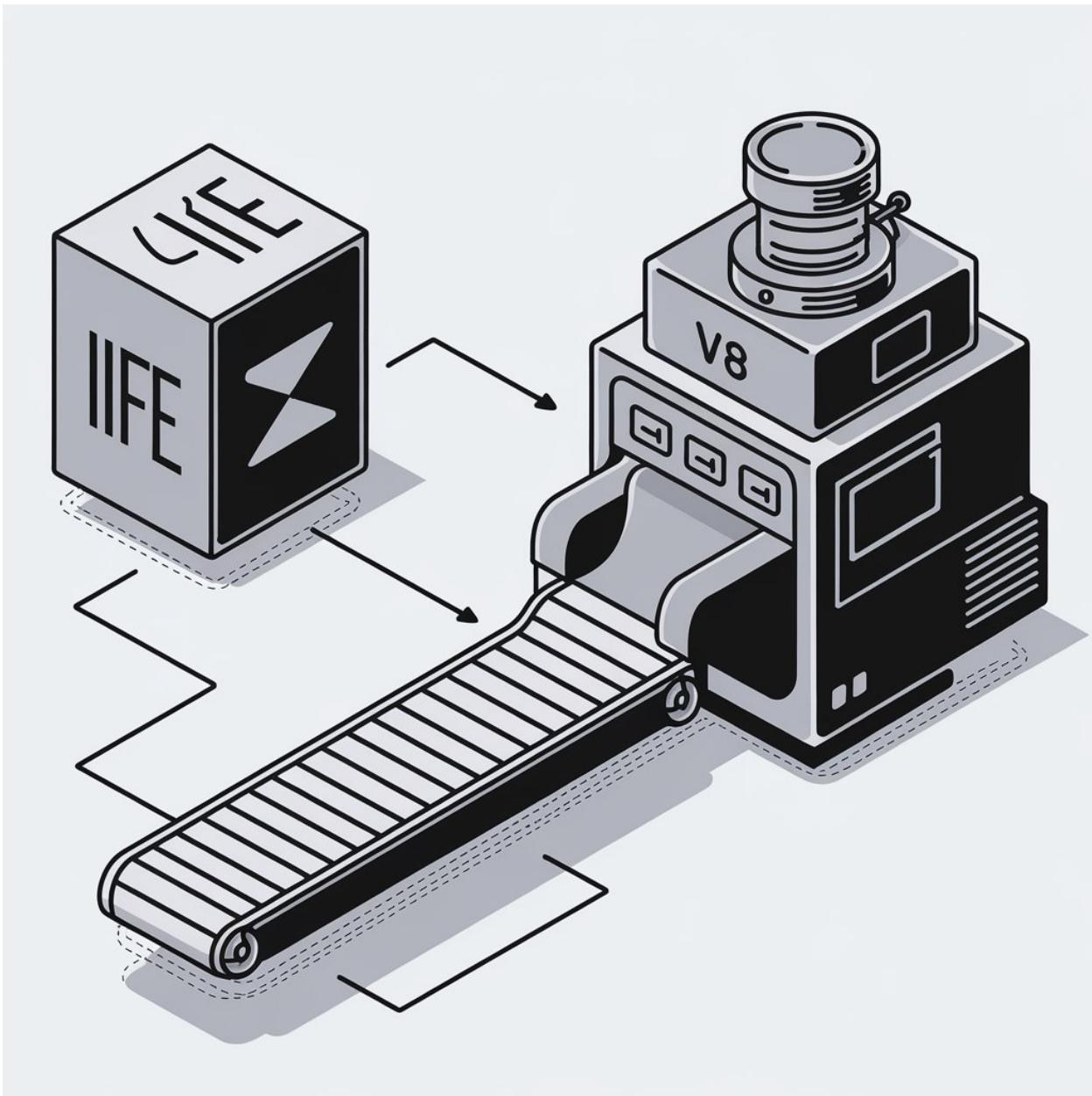
Using IIFE solves multiple problems by providing scope isolation and immediate execution.

very Imp:

Q1: How are variables and functions private in different modules?

A:

because of IIFE and the requirement (statement) wrapping code inside IFE.



Q2: How do you get access to `module.exports`? Where does this `module` come from?

A:

In Node.js, when your code is wrapped inside a function, this function has a parameter named

`module`. This parameter is an object provided by Node.js that includes `module.exports`.

The screenshot shows a code editor with a sidebar containing files: calculate, index.js, multiply.js (selected), sum.js, app.js, data.json, and xyz.js. The multiply.js file contains the following code:

```
function calculateMultiply(a, b) {
  const result = a * b;
  console.log(result);
}

//Follow one pattern
module.exports = { calculateMultiply };
```

A red arrow points from the text "where is this module coming from?" to the line `module.exports = { calculateMultiply };`. Below the arrow, the text "A:Node js is adding module ." is displayed.

The screenshot shows a code editor with a sidebar containing files: calculate, index.js, multiply.js (selected), sum.js, app.js, data.json, and xyz.js. The xyz.js file contains the following code:

```
function(module) {
  //All code of module runs inside here
  function calculateMultiply(a, b) {
    const result = a * b;
    console.log(result);
  }
  module.export = { calculateMultiply };
}(module);
```

Two parts of the code are circled in red: the parameter `(module)` in the first line and the assignment `module.export = { calculateMultiply };` in the eighth line.

To the right of the code, a explanatory text box states: "In Node.js, when your code is wrapped inside a function, this function has a parameter named module. This parameter is an object provided by Node.js that includes module.exports."

When you use `module.exports`, you're modifying the `exports` object of the current module. Node.js relies on this object to determine what will be exported from the module when it's required in another file.

The `module` object is automatically provided by Node.js and is passed as a parameter to the function that wraps your code. This mechanism allows you to define which parts of your module are accessible externally.

suppose you want to include one module inside it.

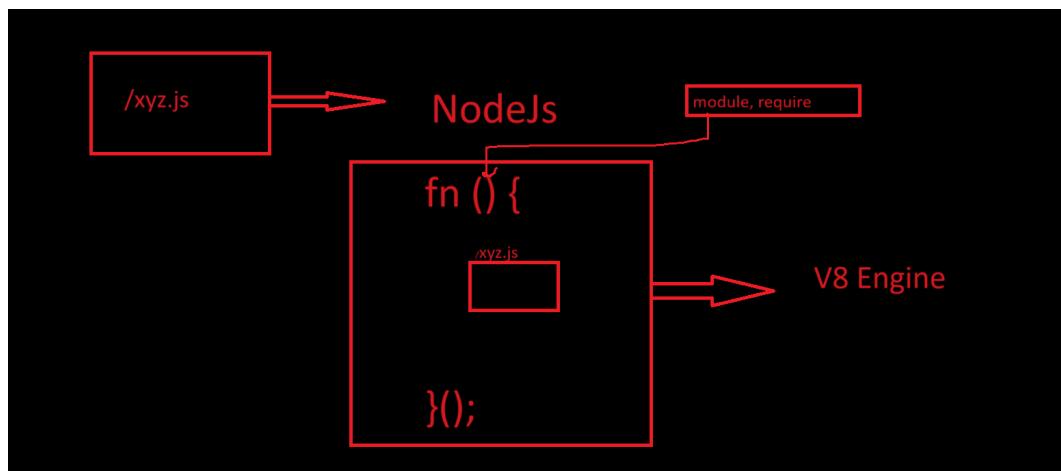
```

File Edit Selection View Go ...
JS xyz.js 2 JS multiply.js ...
calculate > JS multiply.js > ...
1 require "/path"; //can u write this? Yes u can give any path u want to
2 //node.js will not complain
3 function calculateMultiply(a, b) {
4   const result = a * b;
5   console.log(result);
6 }
7
8 //follow one pattern
9 module.exports = { calculateMultiply };
10
11 where is this require coming from? this require is also pass over here.

...
JS xyz.js 2 ...
1 (function (module, require) {
2   require("/path");
3   //All code of module runs inside here
4
5   function calculateMultiply(a, b) {
6     const result = a * b;
7     console.log(result);
8   }
9   module.exports = { calculateMultiply };
10
11 })(this);
Module exports = {};

```

Summary



How `require()` Works Behind the Scenes



1. Resolving the Module

- Node.js first determines the path of the module. It checks whether the path is a local file (`./local`), a JSON file (`.json`), or a module from the `node_modules` directory, among other possibilities.

2. Loading the Module

- Once the path is resolved, Node.js loads the file content based on its type. The loading process varies depending on whether the file is JavaScript,

JSON, or another type.

3. Wrapping Inside an IIFE

- The module code is wrapped in an Immediately Invoked Function Expression (IIFE). This wrapping helps encapsulate the module's scope, keeping variables and functions private to the module.

4. Code Evaluation and Module Exports

- After wrapping, Node.js evaluates the module's code. During this evaluation, `module.exports` is set to export the module's functionality or data. This step essentially makes the module's exports available to other files.

5. Caching(very imp)

- **Importance:** Caching is crucial for performance. Node.js caches the result of the `require()` call so that the module is only loaded and executed once.

Example



- **Scenario:** Suppose you have three files: `sum.js`, `app.js`, and `multiply.js`. All three files require a common module named `xyz`.
- **Initial Require:**
 - When `sum.js` first requires `xyz` with `require('./xyz')`, Node.js performs the full `require()` process for `xyz`:
 - 1. Resolving** the path to `xyz`.
 - 2. Loading** the content of `xyz`.

3. Wrapping the code in an IIFE.

4. Evaluating the code and setting `module.exports`.

5. Caching is the result.

- Node.js creates a cached entry for `xyz` that includes the evaluated module exports.

Subsequent Requires:

- When `app.js` and `multiply.js` later require `xyz` using `require('./xyz')`, Node.js skips the initial loading and evaluation steps. Instead, it retrieves the module from the cache.
- This means that for `app.js` and `multiply.js`, Node.js just returns the cached `module.exports` without going through the resolution, loading, and wrapping steps again.

```
JS xyz.js JS multiply.js JS app.js JS sum.js
calculate > JS multiply.js > ...
1. require("./xyz");
2. const { calculateSum, calculateMultiply } = require("./xyz");
3. const data = require("./data.json");
4. console.log(data);
5. // import { calculateSum, x } from "./sum.js";
6. let name = "Node JS 03";
7. let a = 5;
8. let b = 10;
9.
10. calculateSum(a, b);
11. calculateMultiply(a, b);
12. // console.log(x);
13. console.log(z);
14.

JS app.js > ...
1. require("./xyz");
2. const { calculateSum, calculateMultiply } = require("./xyz");
3. const data = require("./data.json");
4. console.log(data);
5. // import { calculateSum, x } from "./sum.js";
6. let name = "Node JS 03";
7. let a = 5;
8. let b = 10;
9.
10. calculateSum(a, b);
11. calculateMultiply(a, b);
12. // console.log(x);
13. console.log(z);
14.

JS sum.js > ...
1. require("./xyz");
2. let x = "export in React exports in Node";
3. z = "Non strict Mode Demo";
4. function calculateSum(a, b) {
5.   let sum = a + b;
6.   console.log(sum);
7. }
8.
9.
10. //what is module.exports?
11. console.log(module.exports);
12.
13. module.exports = {
14.   x,
15.   calculateSum,
16. };
17.
```

Impact on Performance:

- If caching did not exist, each `require('./xyz')` call would repeat the full module loading and evaluation process. This would result in a performance overhead, especially if `xyz` is a large or complex module and is required by many files.
- With caching, Node.js efficiently reuses the module's loaded and evaluated code, significantly speeding up module resolution and reducing overhead.



Welcome back! Now, I will go to the Node.js GitHub repo to show you what's happening.

<https://github.com/nodejs>

- 1. NodeJs is an open-source Project**
- 2. I will now show you how the V8 JavaScript engine is integrated within the Node.js GitHub repository to illustrate its role and interaction with Node.js.**

V8 JavaScript Engine

V8 is Google's open source JavaScript engine.

V8 implements ECMAScript as specified in ECMA-262.

V8 is written in C++ and is used in Google Chrome, the open source browser from Google.

V8 can run standalone, or can be embedded into any C++ application.

V8 Project page: <https://v8.dev/docs>

Getting the Code

Checkout [depot tools](#), and run

```
fetch v8
```

This will checkout V8 into the directory `v8` and fetch all of its dependencies. To stay up to date, run

3. when i say there are superpowers, what are this superpowers? This is all the code for the superpowers

V8 JavaScript Engine

V8 is Google's open source JavaScript engine.

V8 implements ECMAScript as specified in ECMA-262.

V8 is written in C++ and is used in Google Chrome, the open source browser from Google.

V8 can run standalone, or can be embedded into any C++ application.

V8 Project page: <https://v8.dev/docs>

Getting the Code

Checkout [depot tools](#), and run

```
fetch v8
```

This will checkout V8 into the directory `v8` and fetch all of its dependencies. To stay up to date, run

4. Libuv is the most amazing superpower



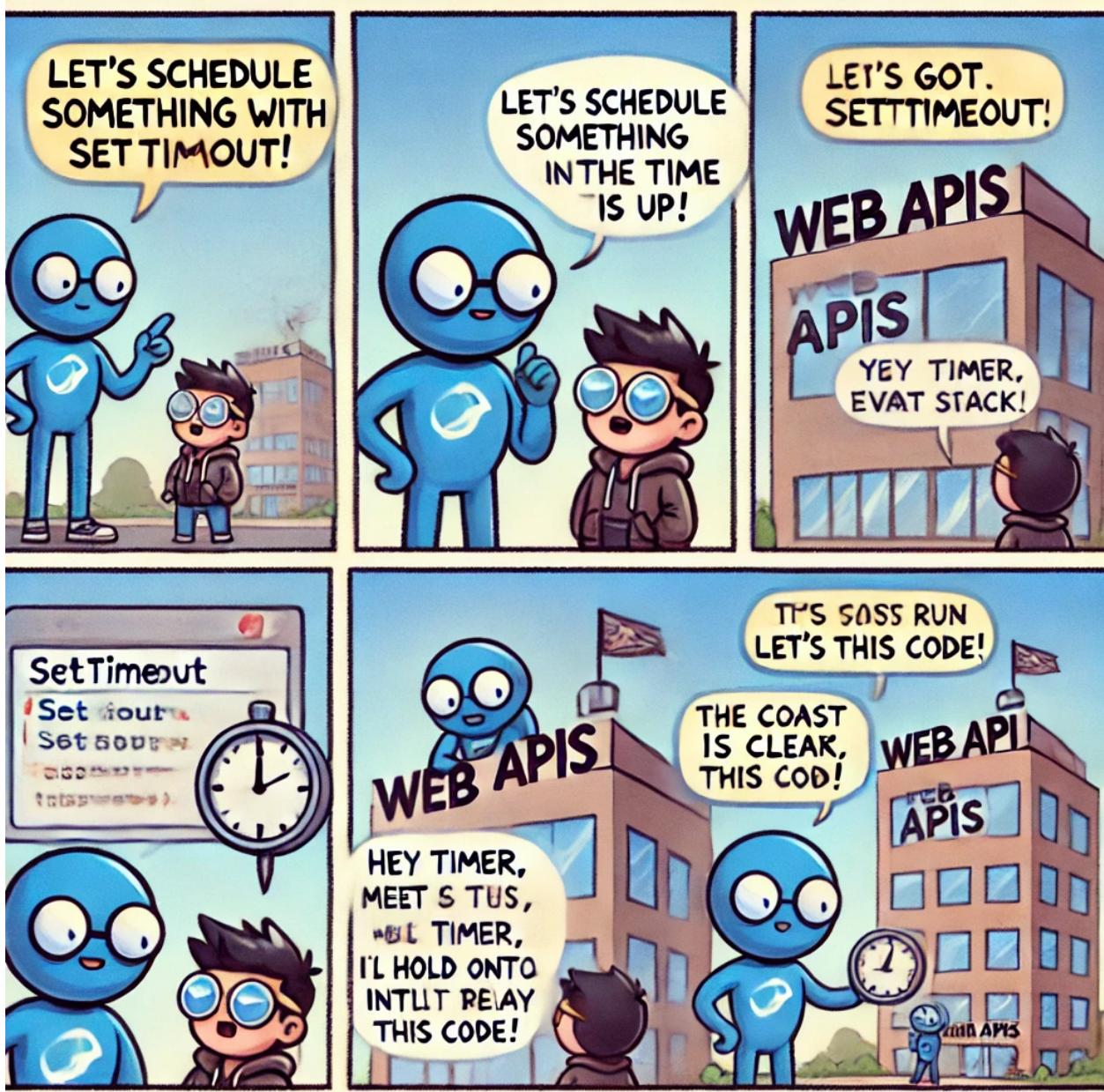
libuv

- **Node.js is popular just because of libuv**
- **libuv plays a critical role in enabling Node.js's high performance and scalability. It provides the underlying infrastructure for asynchronous I/O, event handling, and cross-platform compatibility.**

In the Node.js repository, if you navigate to the `lib` directory, you'll find the core JavaScript code for Node.js. This `lib` folder contains the source code for various built-in modules like `http`, `fs`, `path`, and more. Each module is implemented as a JavaScript file within this directory.

Q: Where is setTimeout coming from and how it work behind scenes ?

<https://github.com/nodejs/node/blob/main/lib/timers/promises.js>



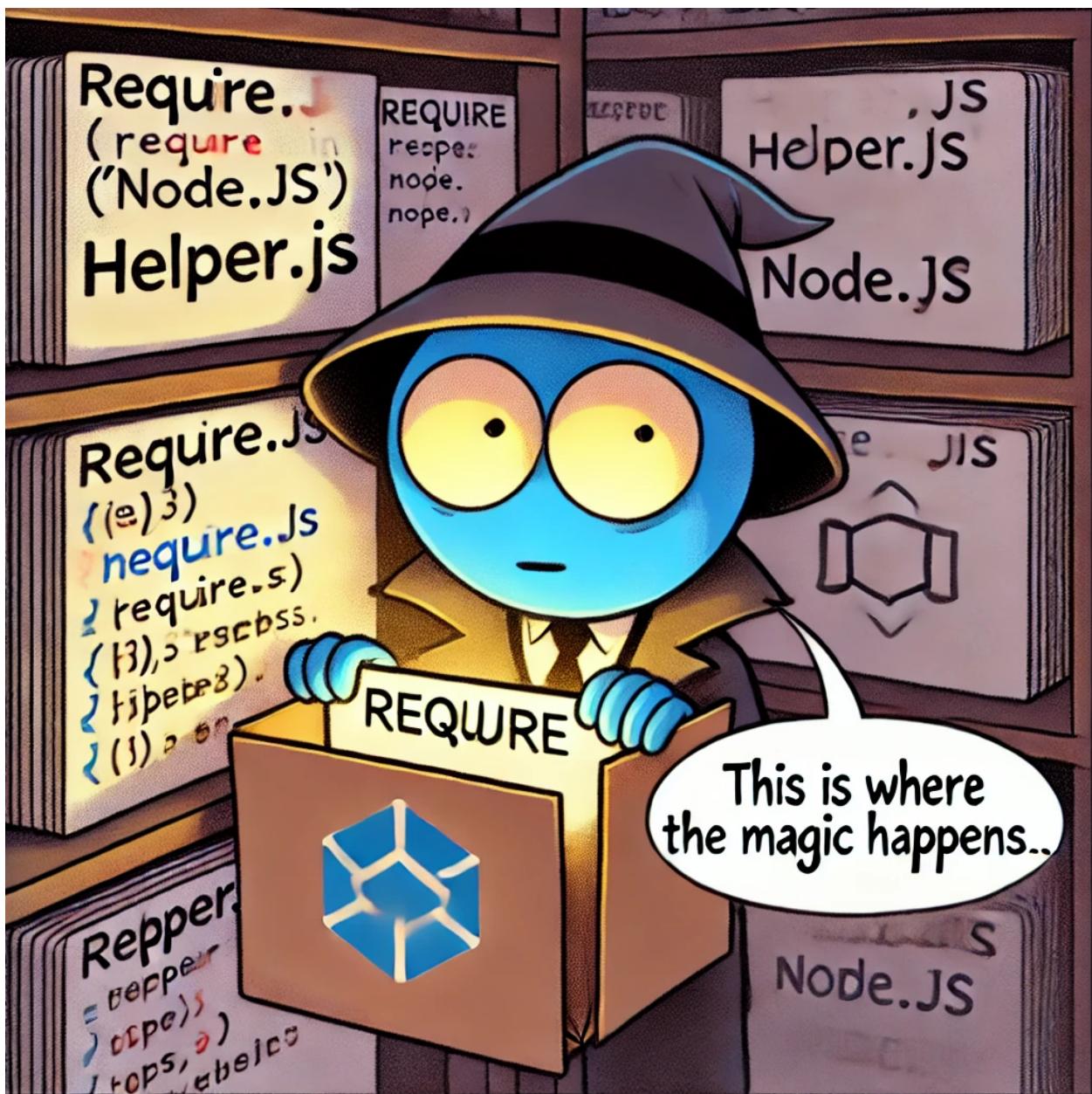
github.com/nodejs/node/blob/main/lib/timers/promises.js

node / lib / timers / promises.js

Code Blame 230 lines (211 loc) · 6.23 KB

```
45     function cancellerHandler(clear, reject, signal) {
46       ...
47   }
48   ...
49 }
50 }
51 <-- setTimeout comes from here
52 <-- behind the scenes
53 function setTimeout(after, value, options = kEmptyObject) {
54   const args = value !== undefined ? [value] : value;
55   if (options == null || typeof options !== 'object') {
56     return PromiseReject(
57       new ERR_INVALID_ARG_TYPE(
58         'options',
59         'Object',
60         options));
61   }
62   const { signal, ref = true } = options;
63   try {
64     validateAbortSignal(signal, 'options.signal');
65   } catch (err) {
66     return PromiseReject(err);
67   }
68   if (typeof ref !== 'boolean') {
69     return PromiseReject(
70       new ERR_INVALID_ARG_TYPE(
71         'options.ref',
72         'boolean',
73         ref));
74   }
75   if (signal?.aborted) {
76     return PromiseReject(new AbortError(undefined, { cause: signal.reason }));
77   }
78   let oncancel;
79   ...
80 }
```

require in nodejs repo



In helper.js file, you can find the actual implementation of require method Here is where the required function is formed

go to this path node/lib/internal/modules/helper.js

<https://github.com/nodejs/node/blob/main/lib/internal/modules/helpers.js>

```

node / lib / internal / modules / helpers.js
Code Blame 409 lines (365 loc) · 12.1 KB
115 * @param {Module} mod - The module to create the 'require' function for.
116 * @typedef {(specifier: string) => unknown} RequireFunction
117 */
118 function makeRequireFunction(mod) {
  // lazy due to cycle
  const Module = lazyModule();
  if (mod instanceof Module !== true) {
    throw new ERR_INVALID_ARG_TYPE('mod', 'Module', mod);
  }
124
125 function require(path) {
  return mod.require(path);
}
128
129 /**
130 * The 'resolve' method that gets attached to module-scope 'require'.
131 * @param {string} request
132 * @param {Parameters<Module['_resolveFilename']>[3]} options
133 */
134 function resolve(request, options) {
  validateString(request, 'request');
  return Module._resolveFilename(request, mod, false, options);
}
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160

```

```

node / lib / main.js
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160

```

the require you are using in your code is returned from here

the `makeRequireFunction` creates a custom `require` function for a given module `mod`. This function:

- **Validates** that `mod` is an instance of `Module`.
- **Defines** a `require` function that uses `mod.require()` to load modules.
- **Implements** a `resolve` method for resolving module paths using `Module._resolveFilename()`.
- **Implements** a `paths` method for finding module lookup paths using `Module._resolveLookupPaths()`.
- **Sets** additional properties on the `require` function, such as `main`, `extensions`, and `cache`.

```
27  */
28 W Function nonRequireFunction(mod) {
29   // lazy load to cycle
30   const Module = lazyModule();
31   if (!mod instanceof Module || !mod) {
32     throw new Error(`Module ${mod} must be a 'Module' object`);
33   }
34 
35   function require(path) {
36     return mod.require(path);
37   }
38 
39   /**
40    * The 'require' method that gets attached to module-scope 'require'.
41    * @param {string} request
42    * @param {Object} [options]
43    */
44   function resolve(request, options) {
45     validateString(request, 'request');
46     return Module._resolveFilename(request, mod, false, options);
47   }
48 
49   require.resolve = resolve;
50 
51   /**
52    * The 'paths' method that gets attached to module-scope 'require'.
53    * @param {string} request
54    */
55   function paths(request) {
56     validateString(request, 'request');
57     return Module._resolveLookupPaths(request, mod);
58   }
59 
60   resolve.paths = paths;
61 
62   setDefaultProperty(require, 'main', process.mainModule);
63 
64   // Enable support to add extra extension types.
65   require.extensions = Module._extensions;
66 
67   require.cache = Module._cache;
68 
69   return require;
70 }
```

Creates a custom require function for a specific module (mod).

Module class is loaded lazily to avoid circular dependencies.

Verifies mod is an instance of Module; throws error if not.

Uses mod.require(path) to load modules within mod's context.

require.resolve uses Module._resolveFilename to find the module's full path

require.paths provides lookup paths using Module._resolveLookupPaths.

require.main is set to process.mainModule.

require.extensions links to Module._extensions for additional file types.

Caching:
require.cache utilizes Module._cache for efficient module reuse.

LazyModule()

<https://github.com/nodejs/node/blob/main/lib/internal/modules/cjs/loader.js>

The diagram illustrates the execution flow and error handling in a Node.js application:

- loader.js** (highlighted in red) contains code to resolve imports for `app.js` and `helper.js`.
- helper.js** (highlighted in green) is a common module used by both `app.js` and `multiplier.js`.
- app.js** (highlighted in blue) contains logic to handle file paths and require statements.
- multiplier.js** (highlighted in orange) is a module that requires `helper.js`.
- Terminal Output** shows an error message indicating that the `id` argument must be of type string, received `undefined`, which corresponds to the error in `multiplier.js`.
- Annotations** provide context:
 - `this id is the path u pass` points to the `path` parameter in `app.js`.
 - `this module is coming from lazy module` points to the `Module` object in `loader.js`.
 - `this require means u are importing the code of loader.js file` points to the `require('loader')` statement in `app.js`.
 - `this helper.js file is common for esm and cjs but this cjs loader is only for commonjs modules` points to the `helper` import in `multiplier.js`.

- If the `id` argument provided to the `require()` function is empty or undefined, Node.js will throw an exception. This is because the `require()` function expects

a string representing the path or identifier of the module to load. When it receives `undefined` instead, it results in a `TypeError`, indicating that an invalid argument value was provided.

- **Node.js documentation** and **GitHub repository** provide insights into how `require()` handles module loading. Reviewing these resources can help you understand how to properly use `require()` and handle potential errors.

**Reading
documentation
in 10 min and
then coding
the solution**



**Coding first
and then
debugging it
for 10 hours**



ProgrammerHumor.io

