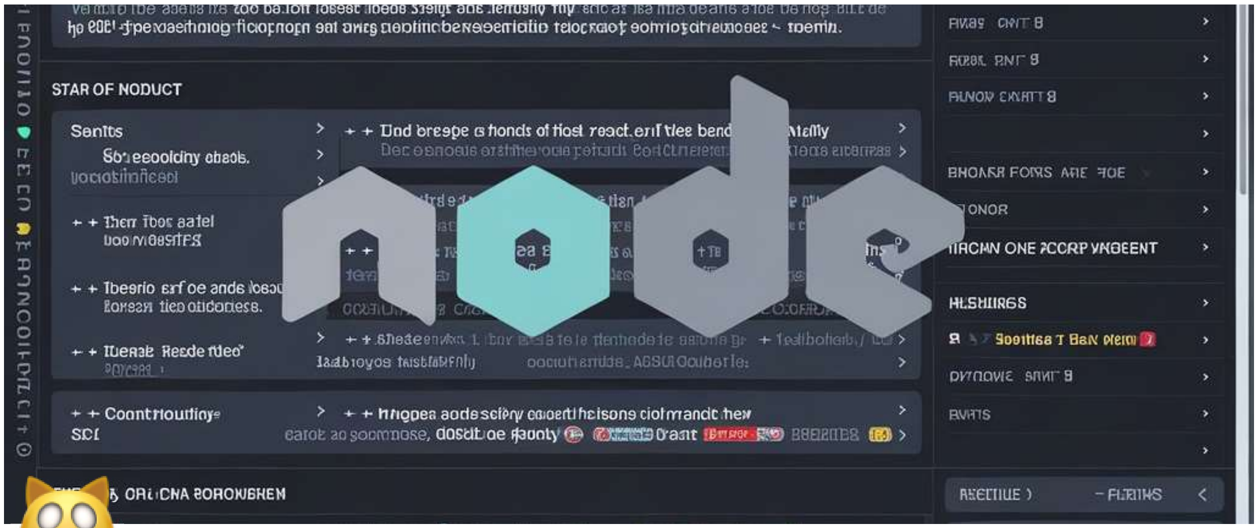


😱

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:

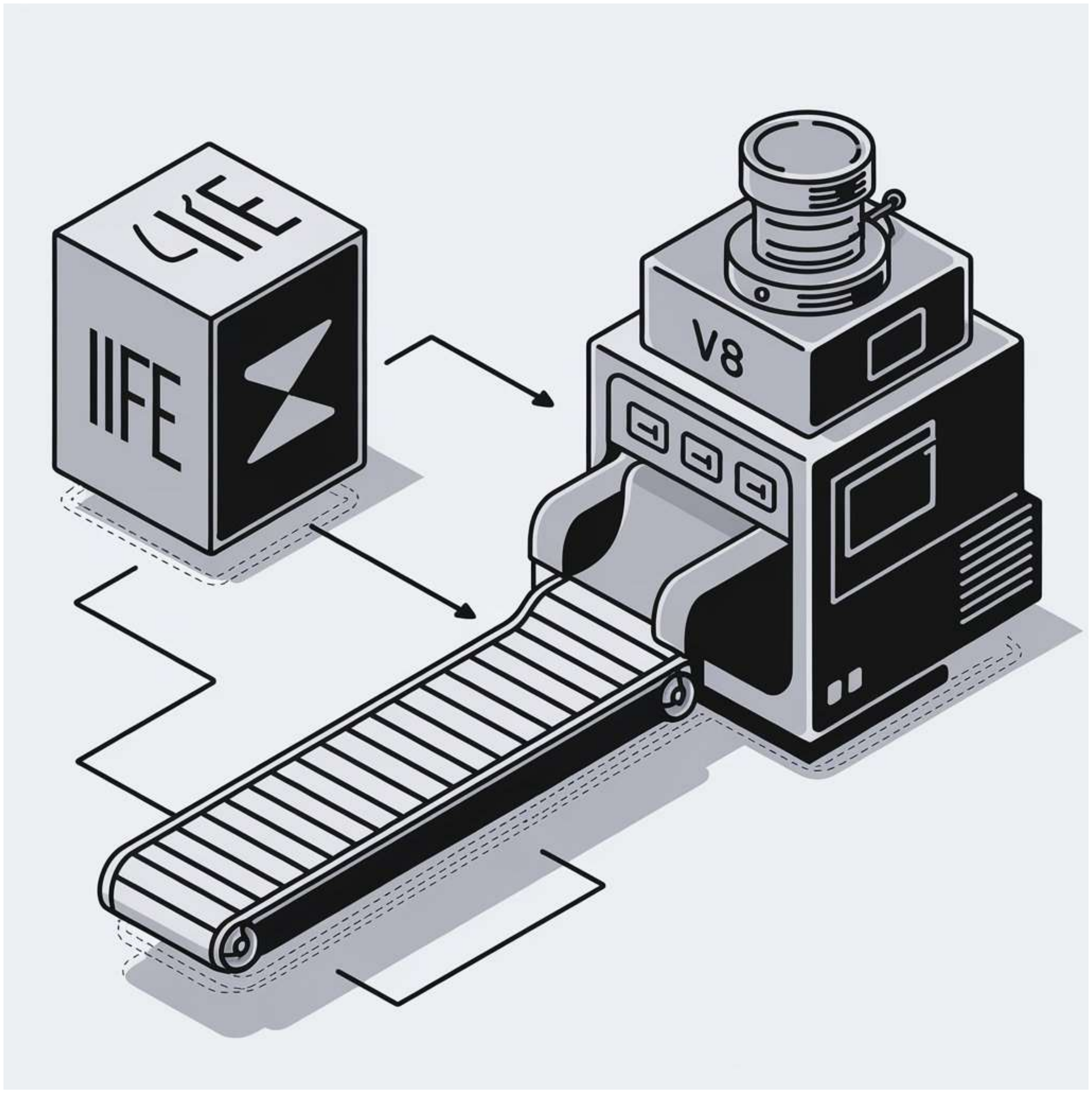
1. **Immediately Invoke Code:** The function runs as soon as it is defined.
2. **Keep Variables and Functions Private:** By encapsulating the code within the IIFE, it prevents variables and functions from interfering with other parts of the code. This ensures that the code within the IIFE remains independent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

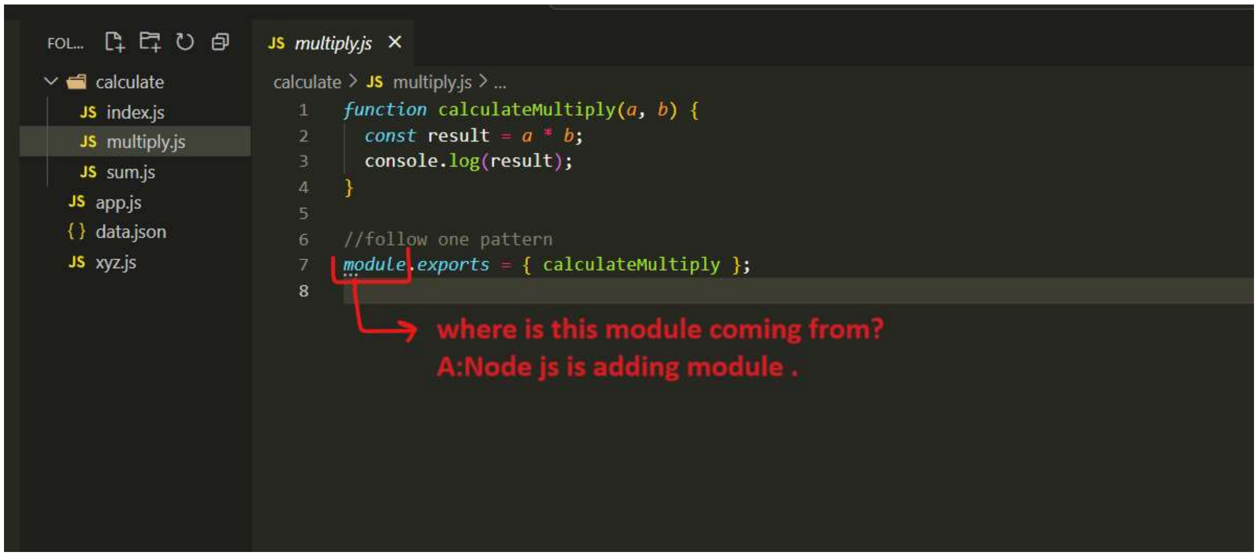
because of IIFE and the requirement (statement) wrapping code inside IFE.



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

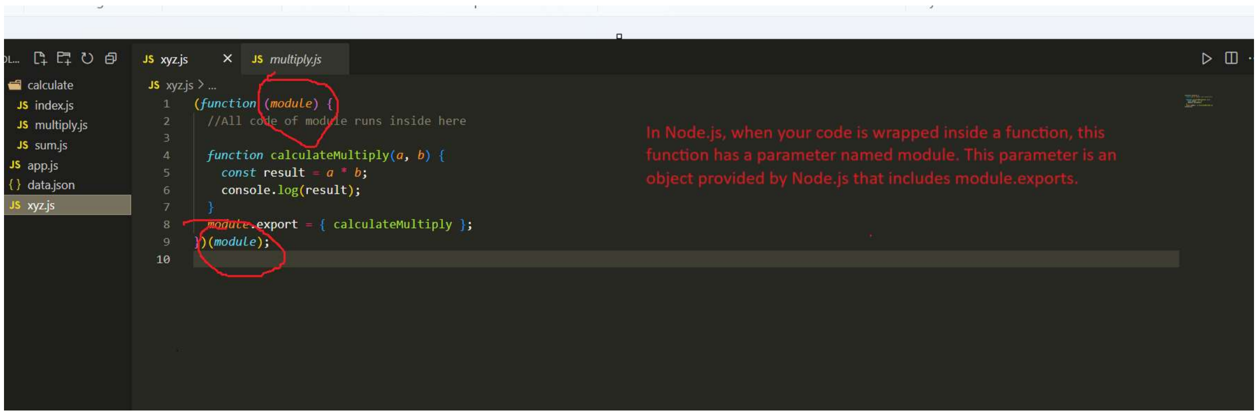
A:

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`.



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

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

where is this module coming from?
A:Node js is adding module .



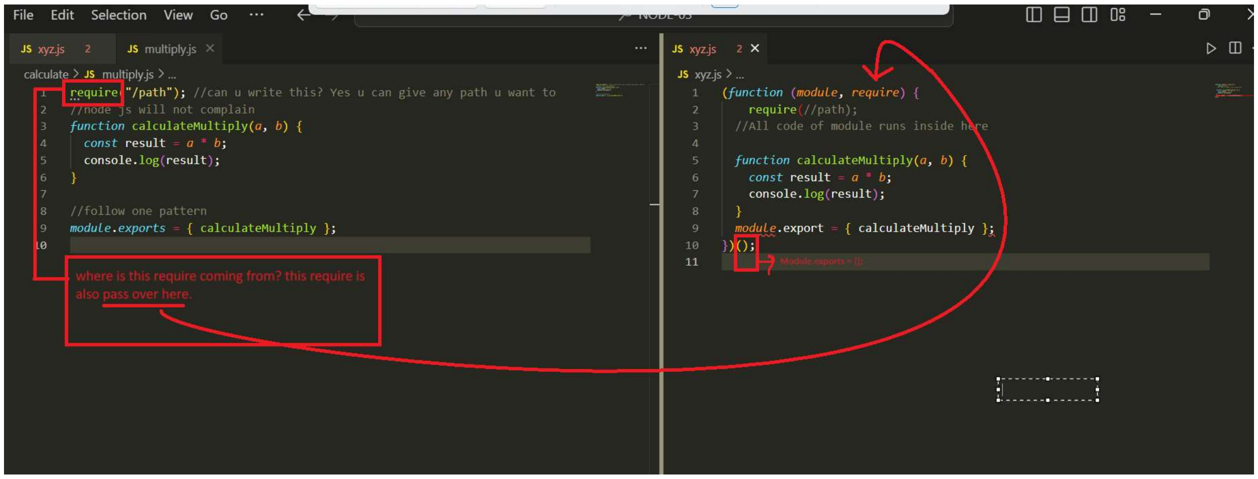
```js
(function (module) {
  //All code of module runs inside here

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

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.

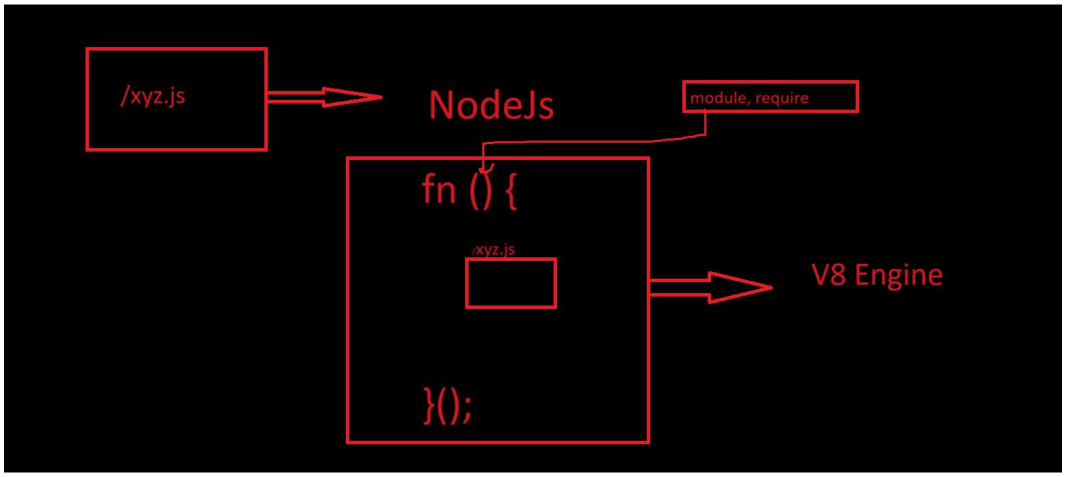


```js
require("/path"); //can u write this? Yes u can give any path u want to
//node js will not complain
function calculateMultiply(a, b) {
  const result = a * b;
  console.log(result);
}

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

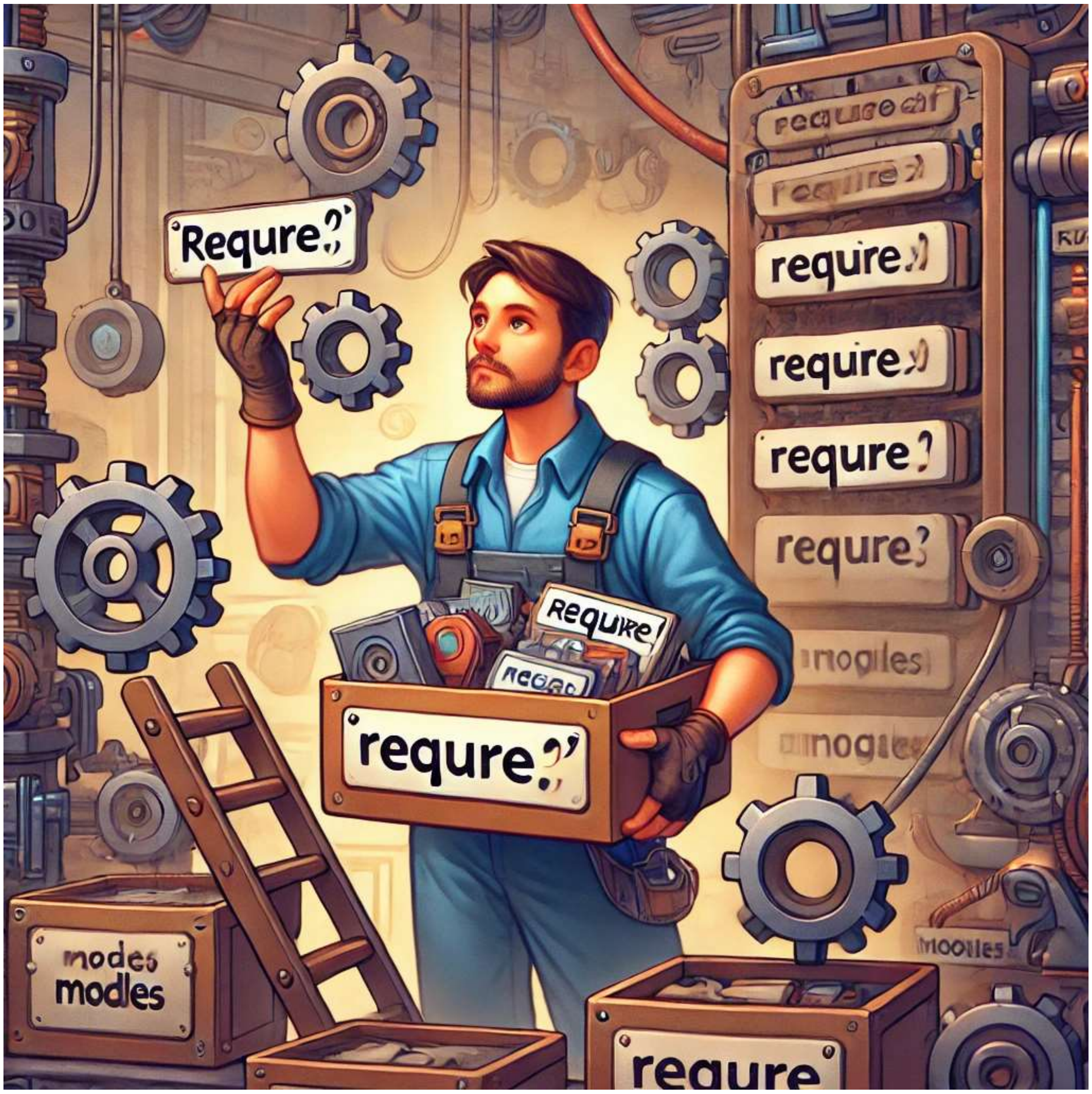
where is this require coming from? this require is also pass over here.

```js
(function (module, require) {
    require(//path);
  //All code of module runs inside here

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

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 M`odule 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

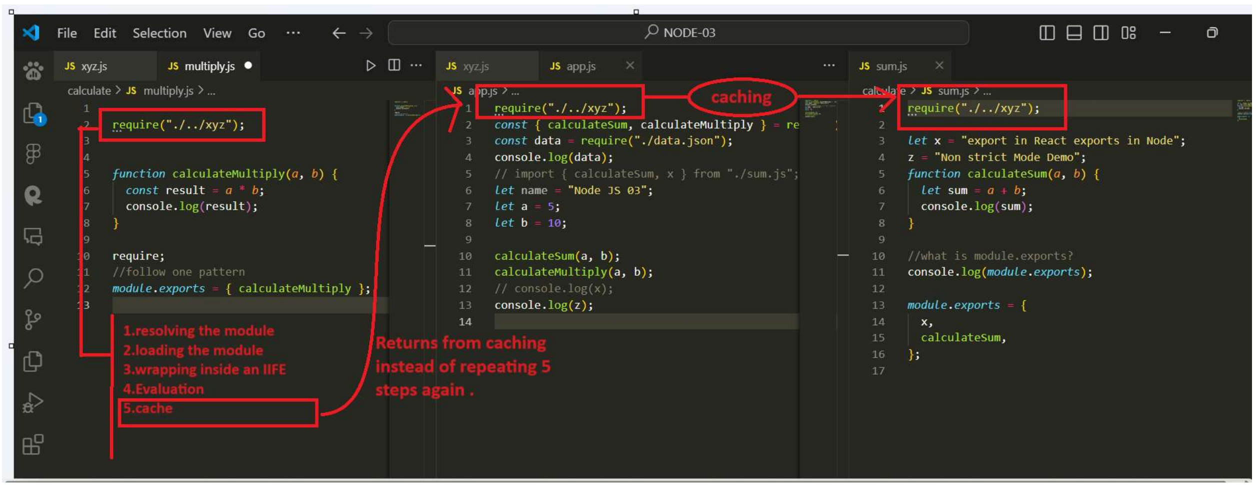
- **Scenario:** Suppose you have three files: `sum.js`, `app.js`, and `multiply.js`. All three files require a common module named `xyz`.
- **Initial Require:**
  - When `sum.js` first requires `xyz` with `require('./xyz')`, Node.js performs the full `require()` process for `xyz`:
    1. **Resolving**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



```javascript
require("./../xyz");


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

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

1.resolving the module
2.loading the module
3.wrapping inside an IIFE
4.Evaluation
5.cache

Returns from caching instead of repeating 5 steps again .

```javascript
require("./../xyz");
const { calculateSum, calculateMultiply } = re
const data = require("./data.json");
console.log(data);
// import { calculateSum, x } from "./sum.js";
let name = "Node JS 03";
let a = 5;
let b = 10;

calculateSum(a, b);
calculateMultiply(a, b);
// console.log(x);
console.log(z);
```

```javascript
require("./../xyz");

let x = "export in React exports in Node";
z = "Non strict Mode Demo";
function calculateSum(a, b) {
  let sum = a + b;
  console.log(sum);
}

//what is module.exports?
console.log(module.exports);

module.exports = {
  x,
  calculateSum,
};
```

Impact on Performance:

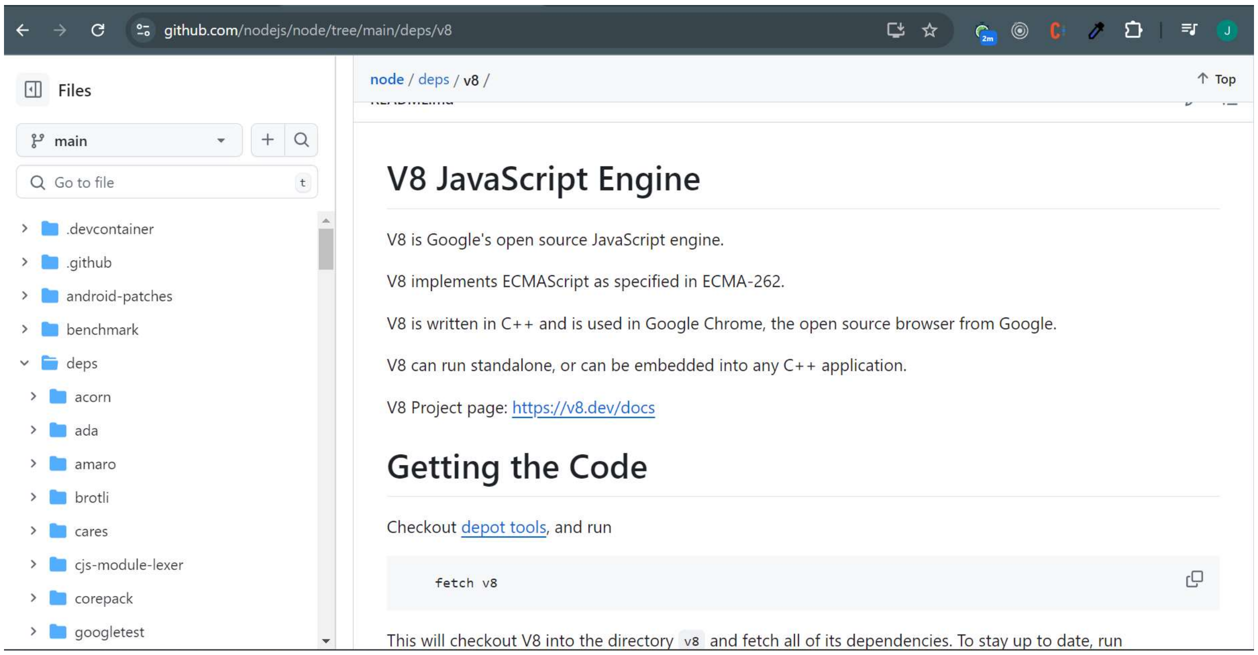
- If caching did not exist, each `require('./xyz')` call would repeat the full module loading and evaluation process. This would result in a performance overhead, especially if `xyz` is a large or complex module and is required by many files.
- With caching, Node.js efficiently reuses the module's loaded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

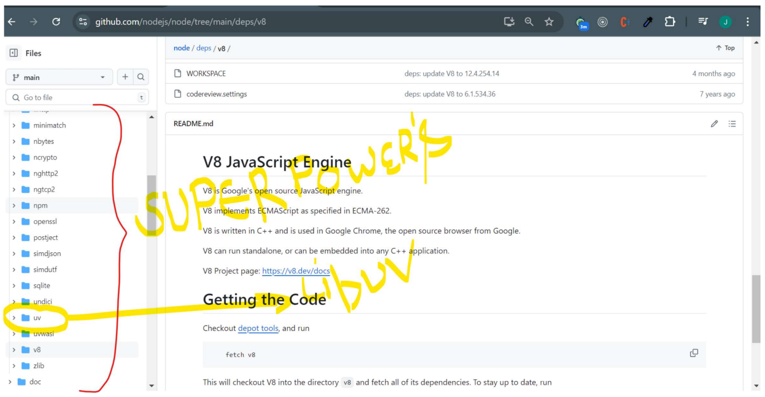
Getting the Code

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**



WORKSPACE	deps: update V8 to 12.4.254.14	4 months ago
codereview.settings	deps: update V8 to 6.1.534.36	7 years ago

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

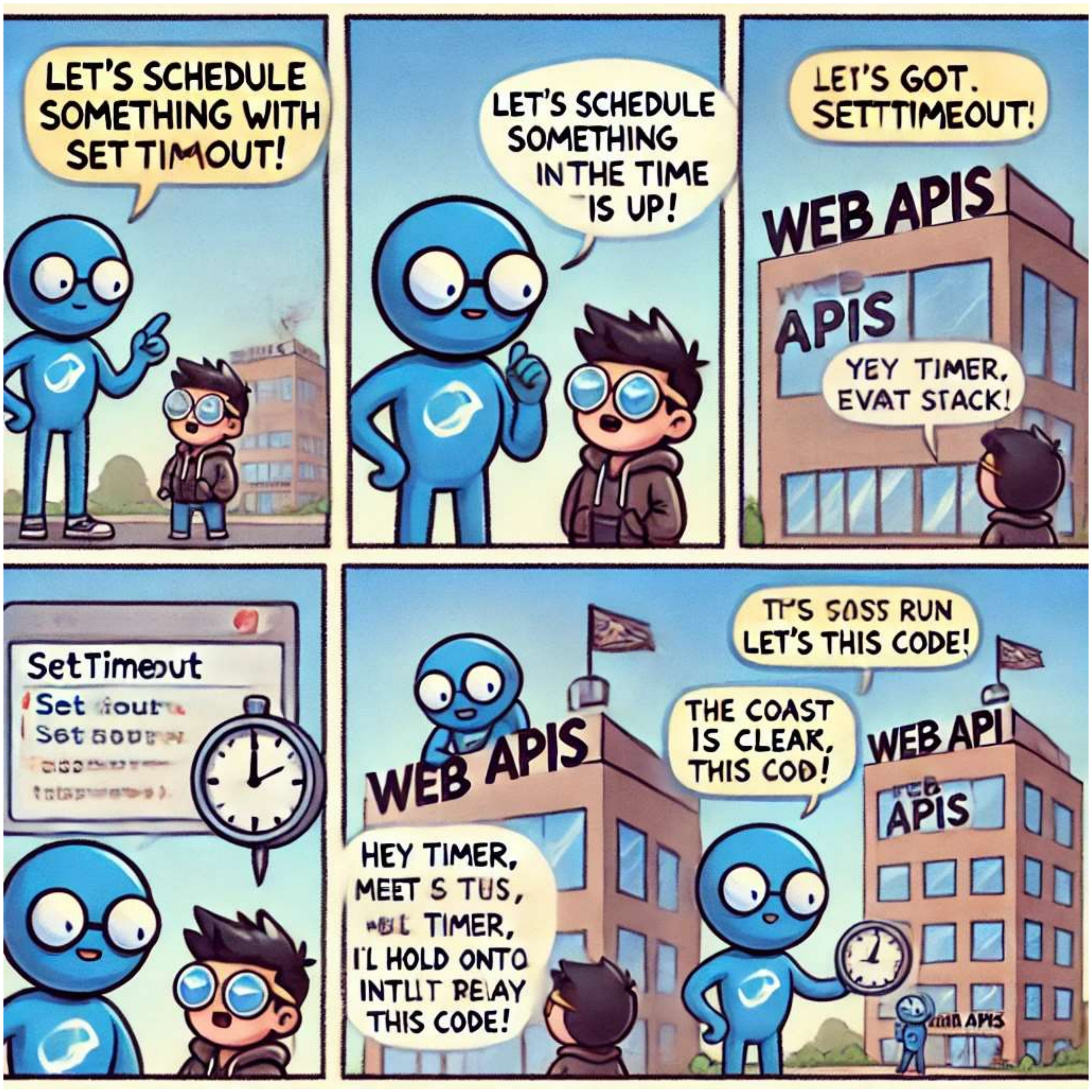
4. **Libuv** is the most amazing superpower

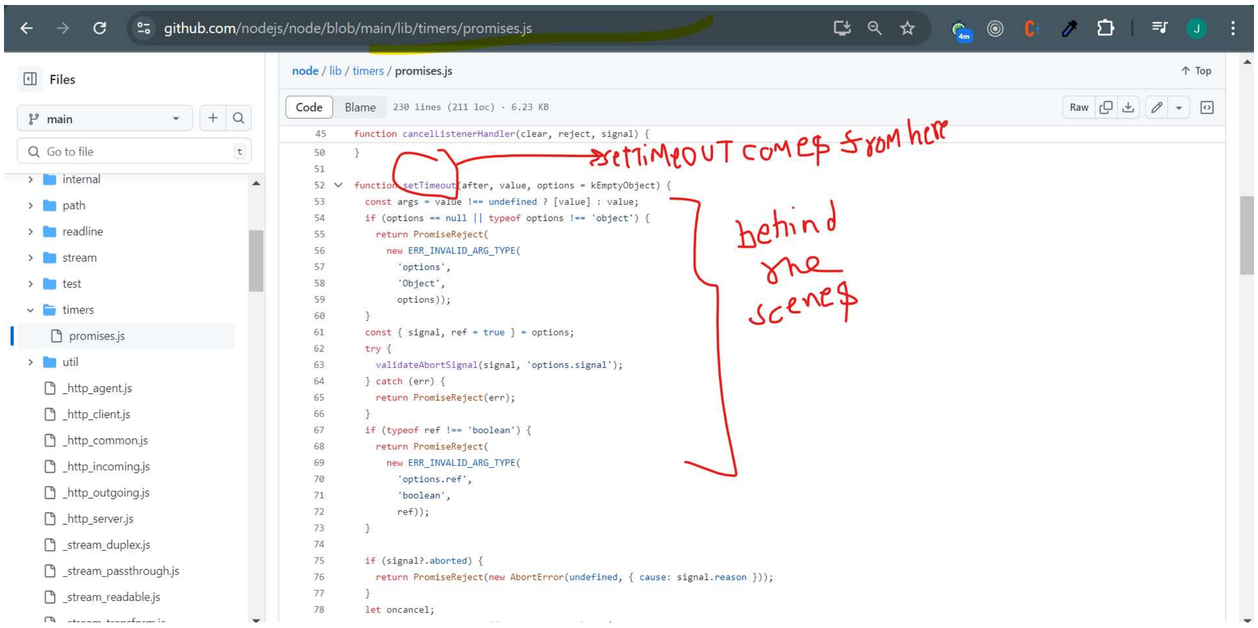
- **Node.js is popular just because of libuv**
- **libuv plays a critical role in enabling Node.js's high performance and scalability. It provides the underlying infrastructure for asynchronous I/O, event handling, and cross-platform compatibility.**

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 ?

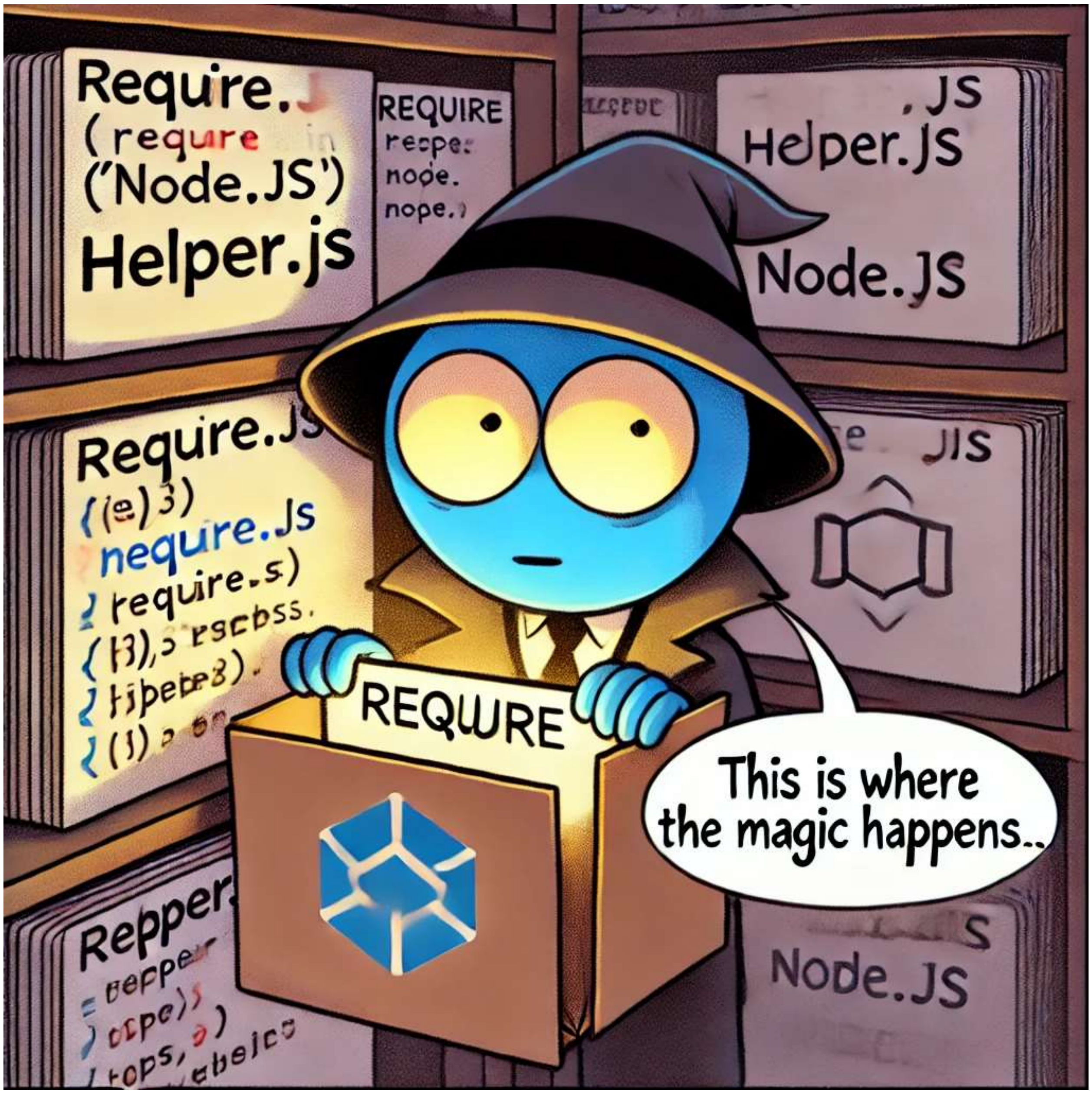
https://github.com/nodejs/node/blob/main/lib/timers/promises.js


LET'S SCHEDULE SOMETHING WITH SET TIMOUT!
LET'S SCHEDULE SOMETHING IN THE TIME IS UP!
LET'S GOT. SETTTIMEOUT!
WEB APIS
APIS
YEY TIMER, EVAT STACK!
SetTimeout
THE'S SOSS RUN LET'S THIS CODE!
THE COAST IS CLEAR, THIS COD!
WEB APIS
WEB API
APIS
HEY TIMER, MEET S TUS, TIMER, I'L HOLD ONTO INTLIT REIAY THIS CODE!



```
function cancelListenerHandler(clear, reject, signal) {
}

function setTimeout(after, value, options = kEmptyObject) {
  const args = value !== undefined ? [value] : value;
  if (options == null || typeof options !== 'object') {
    return PromiseReject(
      new ERR_INVALID_ARG_TYPE(
        'options',
        'Object',
        options));
  }
  const { signal, ref = true } = options;
  try {
    validateAbortSignal(signal, 'options.signal');
  } catch (err) {
    return PromiseReject(err);
  }
  if (typeof ref !== 'boolean') {
    return PromiseReject(
      new ERR_INVALID_ARG_TYPE(
        'options.ref',
        'boolean',
        ref));
  }

  if (signal?.aborted) {
    return PromiseReject(new AbortError(undefined, { cause: signal.reason }));
  }
  let oncancel;
```

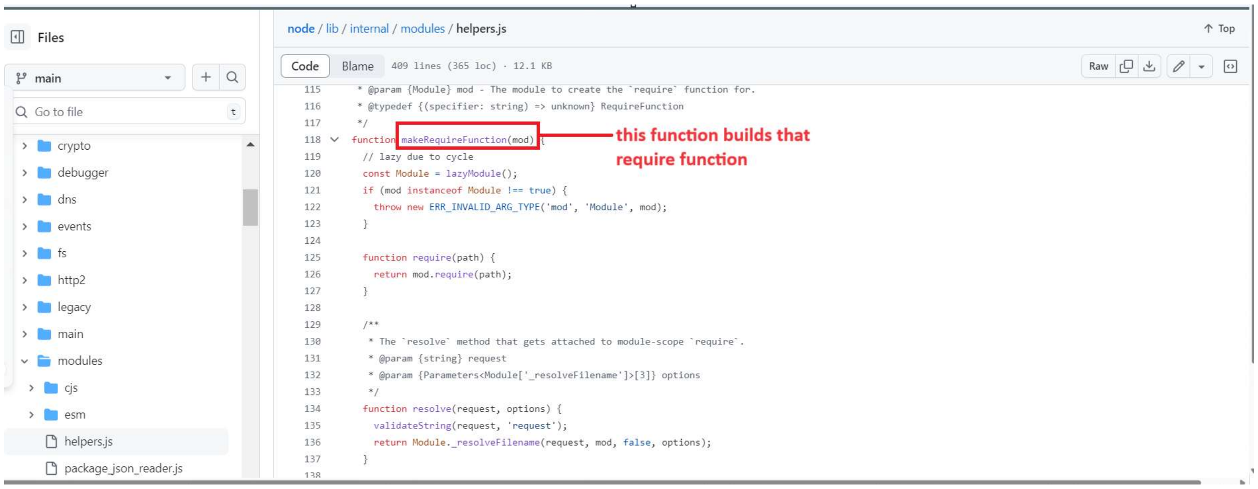
require in nodejs repo



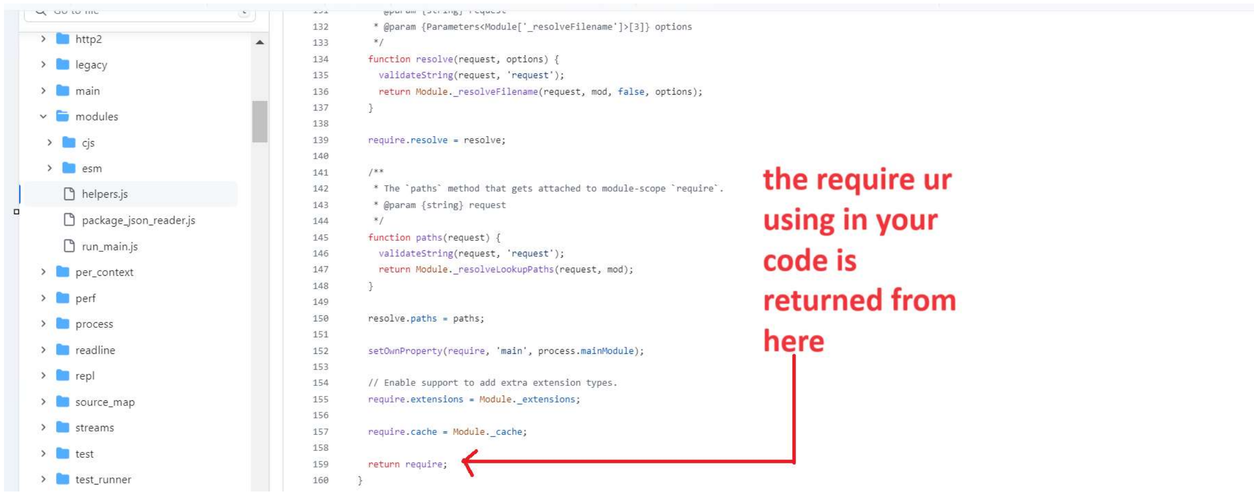
In helper.js file, you can find the actual implementation of require method Here is where the required function is formed

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

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



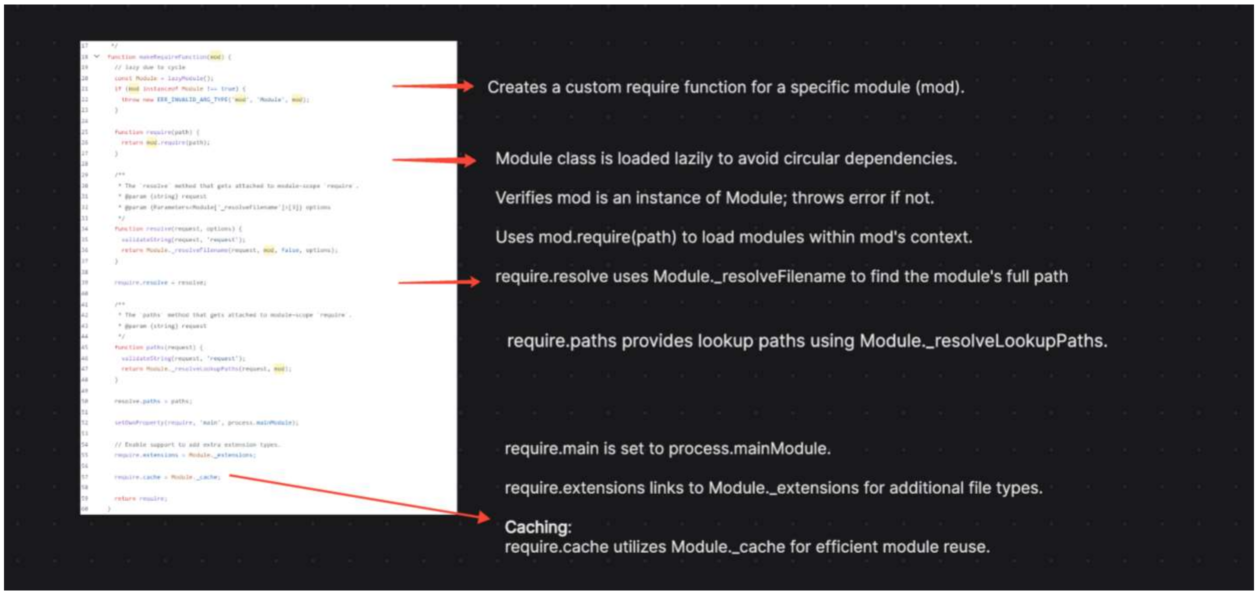
```js
 * @param {Module} mod - The module to create the `require` function for.
 * @typedef {(specifier: string) => unknown} RequireFunction
 */
function makeRequireFunction(mod) {
  // lazy due to cycle
  const Module = lazyModule();
  if (mod instanceof Module !== true) {
    throw new ERR_INVALID_ARG_TYPE('mod', 'Module', mod);
  }

  function require(path) {
    return mod.require(path);
  }

  /**
   * The `resolve` method that gets attached to module-scope `require`.
   * @param {string} request
   * @param {Parameters<Module['_resolveFilename']>[3]} options
   */
  function resolve(request, options) {
    validateString(request, 'request');
    return Module._resolveFilename(request, mod, false, options);
  }
```



```js
   * @param {Parameters<Module['_resolveFilename']>[3]} options
   */
  function resolve(request, options) {
    validateString(request, 'request');
    return Module._resolveFilename(request, mod, false, options);
  }

  require.resolve = resolve;

  /**
   * The `paths` method that gets attached to module-scope `require`.
   * @param {string} request
   */
  function paths(request) {
    validateString(request, 'request');
    return Module._resolveLookupPaths(request, mod);
  }

  resolve.paths = paths;

  setOwnProperty(require, 'main', process.mainModule);

  // Enable support to add extra extension types.
  require.extensions = Module._extensions;

  require.cache = Module._cache;

  return require;
}
```

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`.



```
 */
function makeRequireFunction(mod) {
  // lazy due to cycle
  const Module = lazyModule();
  if (mod instanceof Module !== true) {
    throw new ERR_INVALID_ARG_TYPE('mod', 'Module', mod);
  }

  function require(path) {
    return mod.require(path);
  }

  /**
   * The `resolve` method that gets attached to module-scope `require`.
   * @param {string} request
   * @param {Parameters<Module['_resolveFilename']>[3]} options
   */
  function resolve(request, options) {
    validateString(request, 'request');
    return Module._resolveFilename(request, mod, false, options);
  }

  require.resolve = resolve;

  /**
   * The `paths` method that gets attached to module-scope `require`.
   * @param {string} request
   */
  function paths(request) {
    validateString(request, 'request');
    return Module._resolveLookupPaths(request, mod);
  }

  resolve.paths = paths;

  setOwnProperty(require, 'main', process.mainModule);

  // Enable support to add extra extension types.
  require.extensions = Module._extensions;

  require.cache = Module._cache;

  return require;
}
```

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.

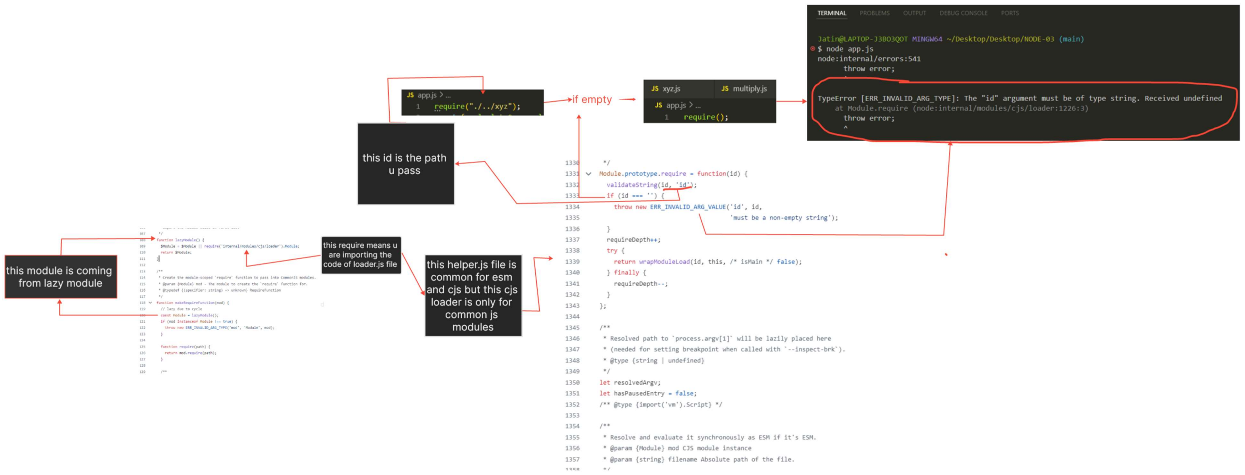
require.main is set to process.mainModule.

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



this module is coming from lazy module
this require means u are importing the code of loader.js file
this helper.js file is common for esm and cjs but this cjs loader is only for common js modules
this id is the path u pass
if empty
app.js
xyz.js
multiply.js
TERMINAL
PROBLEMS
OUTPUT
DEBUG CONSOLE
PORTS

```
require("./../xyz");
```

```
require();
```

```
 */
Module.prototype.require = function(id) {
  validateString(id, 'id');
  if (id === '') {
    throw new ERR_INVALID_ARG_VALUE('id', id,
                                    'must be a non-empty string');
  }
  requireDepth++;
  try {
    return wrapModuleLoad(id, this, /* isMain */ false);
  } finally {
    requireDepth--;
  }
};

/**
 * Resolved path to `process.argv[1]` will be lazily placed here
 * (needed for setting breakpoint when called with `--inspect-brk`).
 * @type {string | undefined}
 */
let resolvedArgv;
let hasPausedEntry = false;
/** @type {import('vm').Script} */

/**
 * Resolve and evaluate it synchronously as ESM if it's ESM.
 * @param {Module} mod CJS module instance
 * @param {string} filename Absolute path of the file.
 */
```

```
Jatin@LAPTOP-J3BO3QOT MINGW64 ~/Desktop/Desktop/NODE-03 (main)
$ node app.js
node:internal/errors:541
      throw error;
      ^

TypeError [ERR_INVALID_ARG_TYPE]: The "id" argument must be of type string. Received undefined
    at Module.require (node:internal/modules/cjs/loader:1226:3)
      throw error;
      ^
```

- 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

We're so glad you were here! Until next time, Keep coding and stay curious!
the next episode!
V8
Javesount Engine
Lbuv