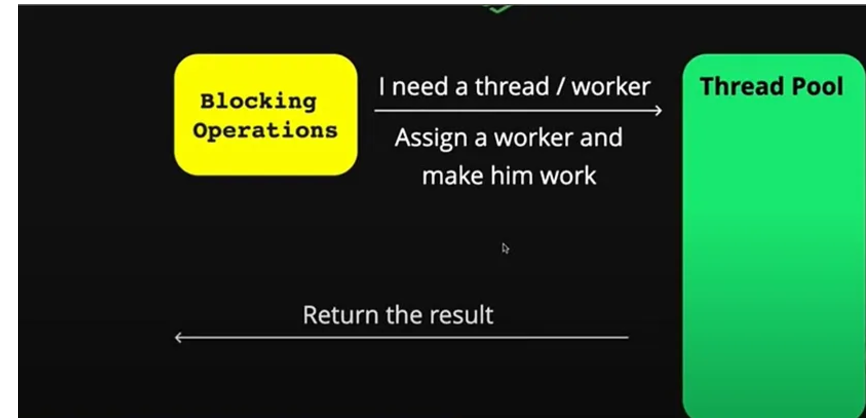
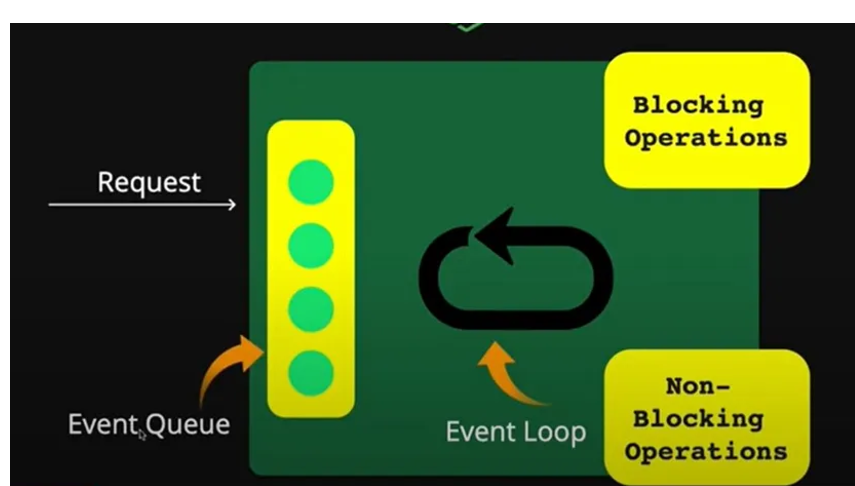
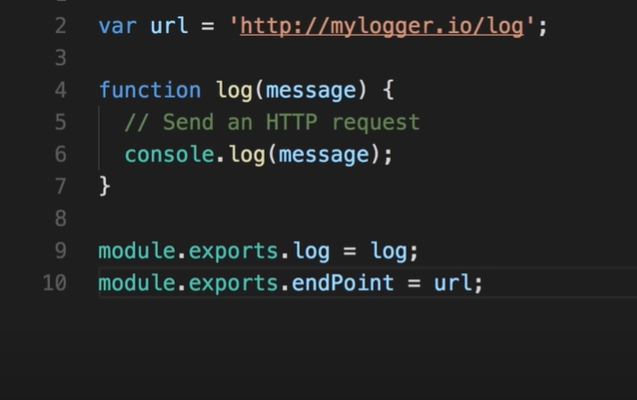
**Node.JS**

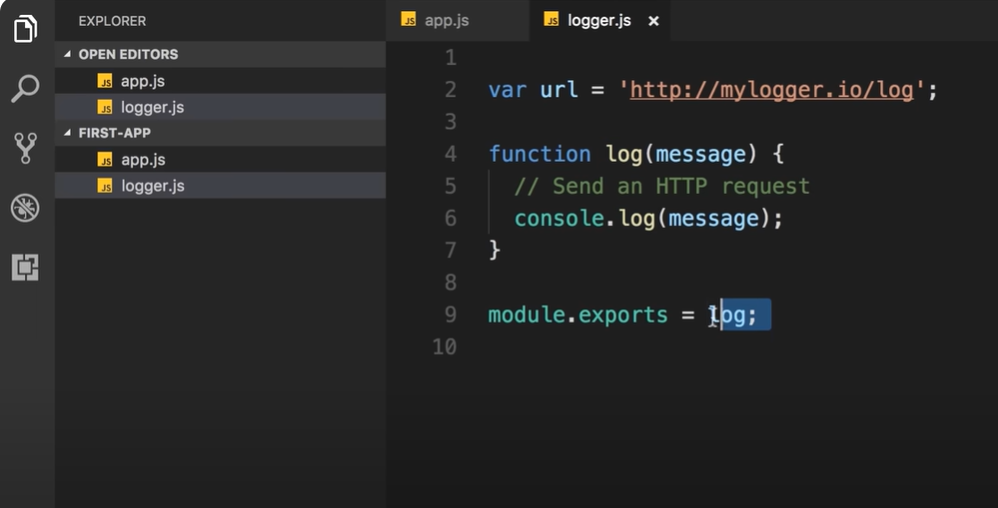
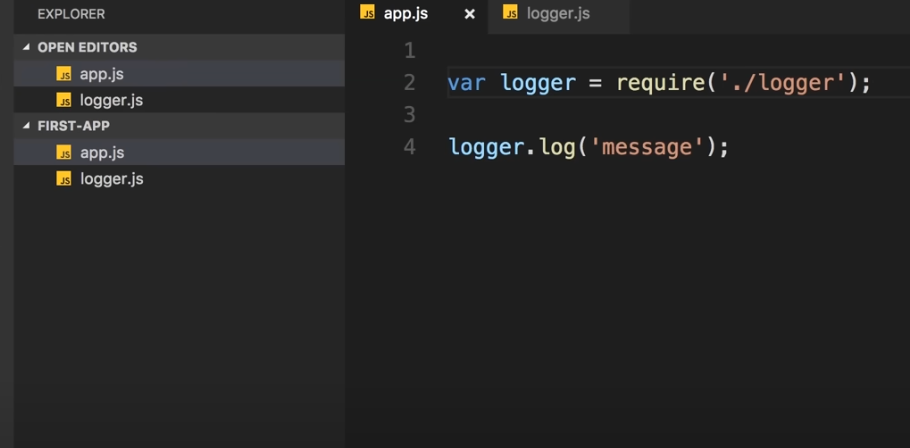
* When declaring a className, it should be like *EventEmitter.* 1st letter of all word must be in caps.
* when a variable is declared above a function it can be used inside the function, this s lexical scope. The function looks for variables in its own scope first, and then in the outer (parent) scopes.
* **What is Node.js?**
  + Runtime environment for running JS code outside browser.
  + Node.js app runs in a single process without creating new thread for every request.
* **Why use node.js?**
  + Node.js is used to run code outside the browser.
  + It is fast, lightweight, scalable, efficient, event-driven programming, non-blocking I/O capable.
* **What is libluv?**
  + Libluv is a C library used for performing I/O operations.
  + I/O operations have to do with sending requests to the computer and receiving responses. These operations include reading and writing files, making network requests, and so on.
* **How nodejs works?**

Create request 🡪 node.js retrieves request and adds to event queue (request will be added line by line) 🡪 1st added request will be performed 1st 🡪while performing the request is added to event loop 🡪 checks the req if blocking or non-blocking 🡪 if non-blocking req it is performed 1st by passing the operation to libluv librry and res send to user, if blocking it is transferred to thread pool

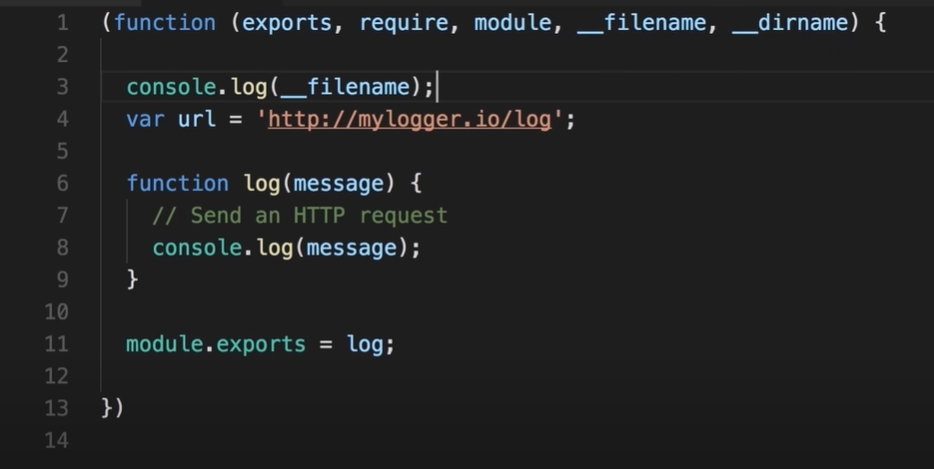


* **Global object**
  + In JS we have **window** object which can be accessed anywhere in the function.
  + But in node we have **global** object.
  + Any variable declared in JS will be added to window object. This can be accessed by window.var
  + But variable declared in node is not added to window object
  + In node, each file is a module. So variable/ function declared in that module cannot be used in another module. So those variables/ functions are not added in global object.
  + Module is not a global object. 1st image is **console.log(modules)**

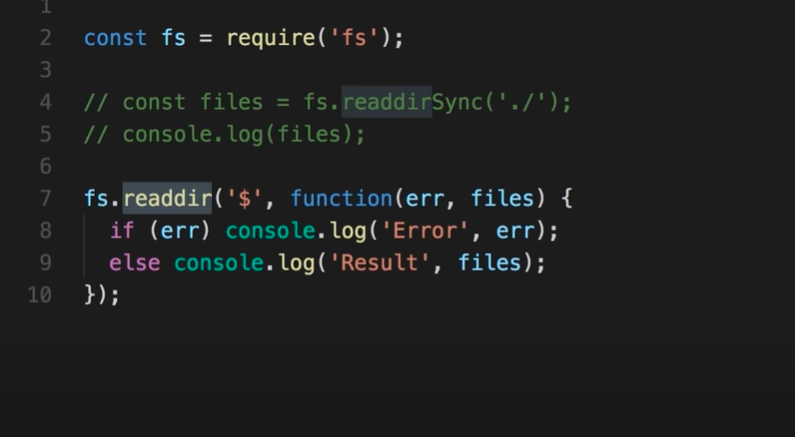




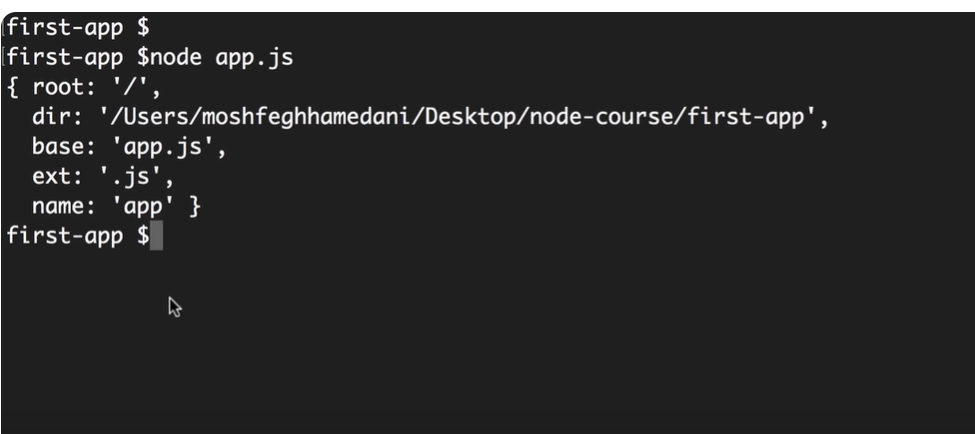
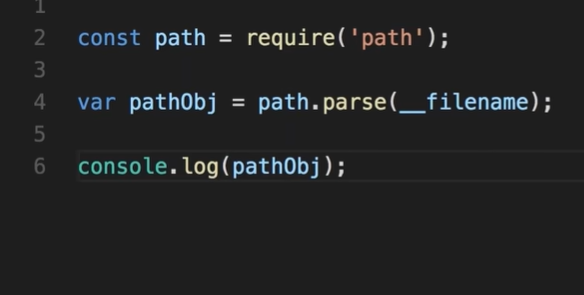
* 4th picture is another way to export function rather than exporting as an object.
* Node doesn’t executes code directly, it wraps inside a function 🡪 module wrapper function



* Some built-in modules have both **sync and async** method. But it is best to use async method so that all clients can be served without asking them to wait.
* Every async method has 2 parameters.
  + 1st parameter 🡪to be found
  + 2nd parameter 🡪callback function
    - Callback function has 2 parameters 🡪err,res



* Some of the built-in modules:
  + ‘path’
  + Imported by using **require().** By default node assumes parameter inside require is a js file. So extension is not needed.



* **History of Node.js**
  + Node.js was written by Ryan Dahl in 2009.
  + 13 years later introduction of 1st server-side JS environment is released.
* **Modules**
  + Modules in node.js are nothing but files.
  + We split our code into different files to maintain, organize and reuse code whenever possible.
  + Module allows to import code written by other developers and export our code.
  + Node.js has many inbuilt modules that comes with installation of node. Ex: HTTP, fs, os…
  + EX for importing built-in modules

const http = require("http");

const server = http.createServer(function (\_req, res) {

res.writeHead(200);

res.end("Hello, World!");

});

server.listen(8080);

* + Types of modules
    - CommonJS modules
    - ES modules
* **CommonJS modules**
  + Node.js will treat files with extension of .js, .cjs as commonjs modules.
  + It uses synchronous way of loading. This means that when a module is imported, the code execution is blocked until the module is loaded.
  + It uses **require** function to import modules and **module.exports** to export modules

**EX:** // logger.js

function log(message) {

const timestamp = new Date().toISOString();

console.log(`${timestamp}: ${message}`);

}

module. Exports = log;

// app.js

const log = require('./logger');

log("Starting the application...");

// Additional application logic.

log("Application is running");

// More application logic.

log("Application finished execution");

* **ES Modules**
  + It is asynchronous way of loading.
  + It is introduced in ES6
  + It uses import statement to import modules. Not supported fully in old browsers

**EX:** // logger.mjs

export function log(message) {

const timestamp = new Date().toISOString();

console.log(`${timestamp}: ${message}`);

}

// app.mjs

import { log } from './logger.mjs';

log("Starting the application...");

// Additional application logic.

log("Application is running");

// More application logic.

log("Application finished execution");

* **What is npm?**
  + NPM – node package manager
  + It is the standard package manager for Node.js
  + Package is a collection of modules bundled together.
  + package.json file provides metadata about the package. Express, react are one of the packages in node.js.

**COMMAND:** npm install <package\_name>

That package must be in./node\_modules/.bin/your-package-name

That package must be exist in package.json file

To run npm

npm run <package\_name>

* **What is npx?**
  + NPX – Node Pakage eXecute.
  + NPX is an NPM package used to execute any packae on NPM directly without installing locally and globally.
  + When running NPM executables with NPX, if a package is installed, NPX will search for the package binaries and then run the package.
  + If the package was not previously installed, NPX will not install the package in your system; instead, it will create a temporary cache that will hold the package binaries.
  + Once the execution is over, NPX will remove the installed cache binaries from the system.
  + This way, global stays clean. This saves disk space and allows you to run a package only when it’s needed. It also gives you the advantage of testing packages without having to install them.

COMMAND: npx <package\_name>

File (modules – can be imported/ exported)

Package (Collection of modules/ files). Includes package.json

Package can be installed in node\_modules of local folder and run using **npm install express**

Package can be temporarily installed, run and deleted using **npx create react-app.**

**EX:**

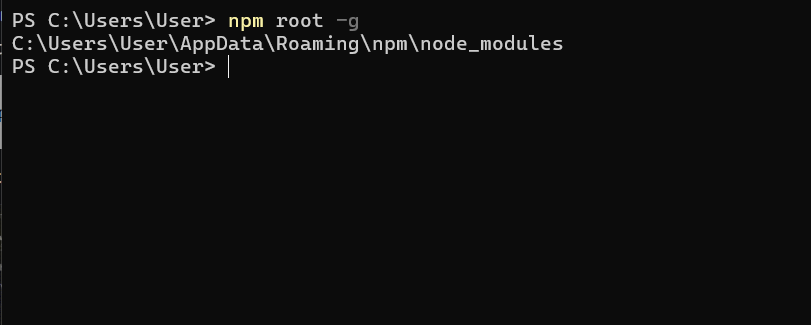
* If you use **npm install express**, express is downloaded only in that project folder and can be reused without reinstalling.
* If you use **npx express** (or similar), the package is only temporarily executed. Changes won’t persist after execution, and you’ll need to run it again each time.

**INSTALLATION PACKAGES:**

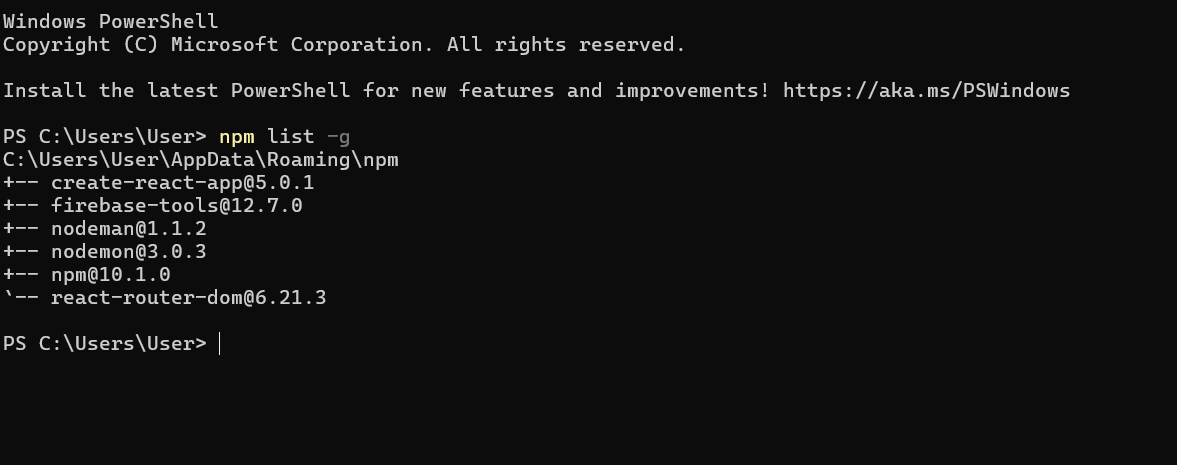
* Dependencies – a piece of code/ file/ module used to run a package. Stored in node\_modules
* Dependencies can be installed either locally/ globally.
* Package.json lists all the dependencies required for node project
* By installing packages globally, it is not possible to have versioning of packages for different projects. This is possible if packages are installed locally.
* Types of installations are
  + Global installation
  + Local installation
* Global installation
  + Refers to installing packages such that they are available system-wide, rather being restricted to a single project.
  + When a package is installed globally, it can be used from any location on your computer without needing to be installed again in each project directory.
  + WORKING - When you globally install a package using npm (Node Package Manager), the package is placed in a specific directory on system that is part of system’s PATH environment variable. This means you can access the package and its executables from any command line or terminal session.
  + Path of global variable -C:\Users\YourUsername\AppData\Roaming\npm\node\_modules

**SYNTAX** - run npm install -g <package-name>

* + **npm root – g** 🡪 used to print location on system where all global modules are installed



* + **npm list – g 🡪** to check which packages are installed globally



* + **npm uninstall –g <package name> 🡪** removes package and its executables from your system.
* **Local installation:**
  + This installs packages in current working directory.
  + Node modules go in ./node\_modules, executables go in ./node\_modules/.bin
  + In your project folder, run **npm sintall <package name>**
* **npm link:**
  + mostly used for testing purpose.
  + You create a package and if you want to test it
  + go to the package folder and run **npm link.** This creates a link of package globally, which can be accessed from anywhere.
  + To use that package in your project, go to project folder and run

**npm link <packagename>.**

* + This links the globally available package to your local project for testing purpose.
* **Updating packages**
  + Local updates:
    - To update local packages, npm install <package-name>@<version>
    - If updated locally, versions in dependencies will be automatically reflected in package.json
    - If you want to allow both minor and the major updates, you can use \* or manually can change the version range.
    - To verify, **npm list <package-name>**
    - To update all dependencies at once, use **npm update**
  + Global updates:
    - To update global packages, npm update -g <package\_name>
    - Or npm install -g <package-name>
* **EXAMPLE OF TERMINILOGIES:** I have a project. where i create many files like app.js etc. it is known as module. it can be exported and imported using commonJS or EJS methods. i need additional built-in package (collection of modules) like express in my project. so i use npm install express to install it locally in project. this installs all needed files of express in node\_modules. those metadata, scripts are stored in package-lock.json. if we need we can install package globally using npm install -g express. also there is npx. by running this that package will temporily installed in cache executed by project and deleted in cache after sometime. is this flow correct?
* **Running scripts:**
  + To execute a JS file run **node <file.js>**
* **NPM workspaces:**
  + NPM workspaces are used to organize and manage multi-package projects in a single repository. They allow to handle dependencies and scripts across multiple packages, making development process smooth and efficient.
  + If I have both server (a package) and client (a package) in single repo, managing those npm packages are difficult. So npm workspace are created.
  + Common dependencies are hoisted to the root node\_modules to optimize disk space.
* **Creating packages:**
  + Create a project folder. Navigate to it and run **npm init –y**, to install package.json file
  + Create index.js in the folder and write the code for package and export it.
  + Navigate to the folder and run npm link to make it available globally.
  + To test the package, create a test folder and create a script.js file to import the package and test it.
  + After importing the package, it will create a node\_modules folder containing the package files.
  + To publish the package, 1st run npm login. Enter your username and password. Then run npm publish to publish the package.
  + This can be checked in npm official website.
  + To create a scoped package like @ananya/hello-npm as package name, run npm init --scope=@your-username
  + Then run npm publish --access public
* **Semantic versioning:**
  + Versioning scheme is designed to convey meaning about changes in a release.
  + 3 parts version number 🡪MAJOR.MINOR.PATCH
  + In software development, changes are inevitable. Whether it’s fixing bugs, adding new features, or making breaking changes, developers need a reliable way to communicate these updates.
  + MAJOR: Incremented for incompatible API changes. 1.0.0 to 2.0.0
    - No backward compatible
  + MINOR: Incremented for adding functionality in a backwards-compatible manner. 1.1.0 – 1.2.0
    - Backwords-compatible manner 🡪meaning this can work with older versions/ previous iterations
  + PATCH: Incremented for backwards-compatible bug fixes. 1.1.0 to 1.1.1
* **Async programming:**
  + **setTimeOut –** delay an operation. Provided by Node.js timers module
  + **setTimeout(callbackfn, milliseconds, ..args);**
    - args are additional arguments passed to callback function

setTimeout(() => {

console.log('Executed after 2 seconds');

}, 2000);

* + use **clearTimeOut()** to prevent timeout to execute.
  + clearTimeout cancels the scheduled execution of the callback.

**EX: const timeId = setTimeOut(() =>{**

**console.log(“Logged after 2seccs”)**

**}, 2000)**

**clearTimeOut(timeId);**

* This is used when I want to stop the loading before 2secs. So that the callback function will not at all be executed.
* **setInterval**
  + It repeatedly executes the callback function at regular intervals until it is explicitly cleared using clearInterval.
  + **EX: const intervalId = setInterval(() => {**

**console.log('This message repeats every 2 seconds');**

**}, 2000);**

* + The callback console.log('This message repeats every 2 seconds') will be executed every 2000 milliseconds (2 seconds). This goes on for infinity
  + To stop the interval after a certain amount of time, you can use clearInterval.
  + **EX: const intervalId = setInterval(() => {**

**console.log('This will run every second');**

**}, 1000);**

**// Stop the interval after 5 seconds**

**setTimeout(() => {**

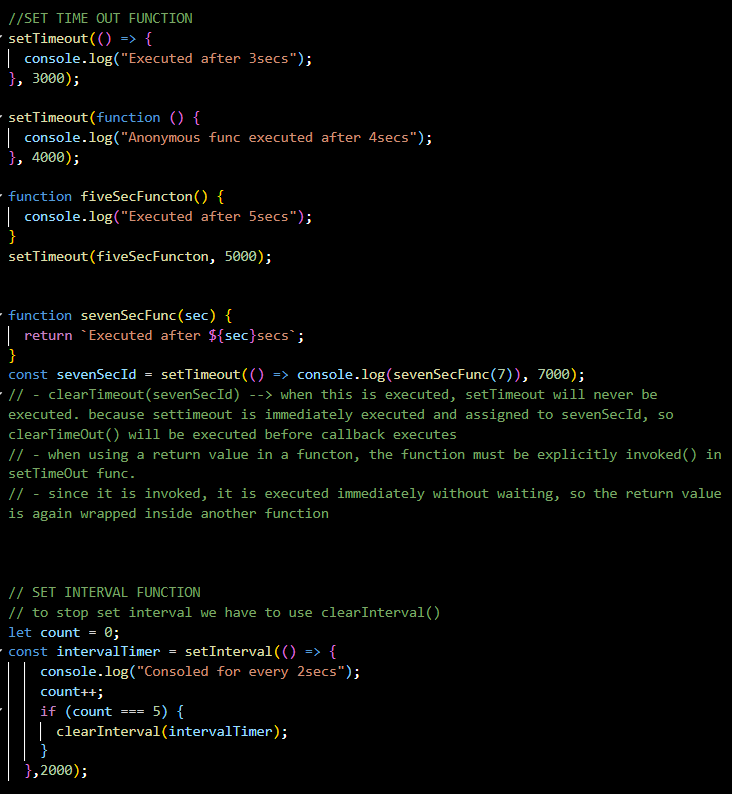
**clearInterval(intervalId);**

**console.log('Interval cleared');**

**}, 5000);**

* + setInterval will log a message every second.
  + After 5 seconds, clearInterval(intervalId) will stop the repeating execution of the callback, and "Interval cleared" will be logged.
  + So the output will be, This will run every second (printed 5 time)

Interval cleared

****

**Callbacks:**

* Node.js callbacks are a special type of function passed as an argument to another function.
* They're called when the function that contains the callback as an argument completes its execution, and allows the code in the callback to run in the meantime.

SYNTAX: function function\_name(argument, callback)

* Only when execution of function\_name is completed, callback function is called.
* The callback is used to define what happens when the function containing the callback as an argument completes its execution.
* For example, we can define a callback that prints the result after the parent function completes its execution.
* EX: setTimeOut() is an example for callbacks.
  + setTimeOut(()=>{console.log(“waited for 2sec”)},2000)
  + function addNumbers(a,b,callback){

const sum = a+b;

callback(sum);

}

function displaySum(res){

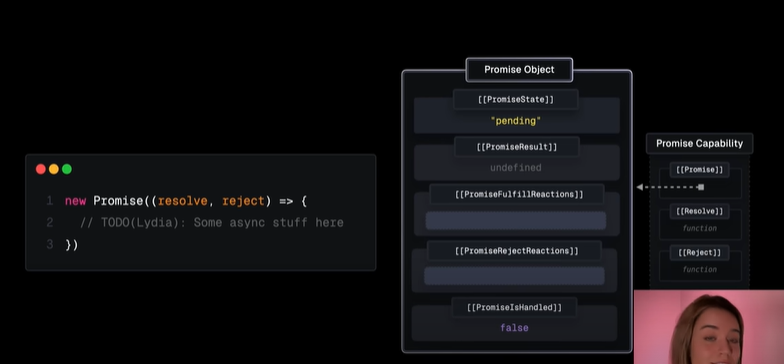
console.log(res);

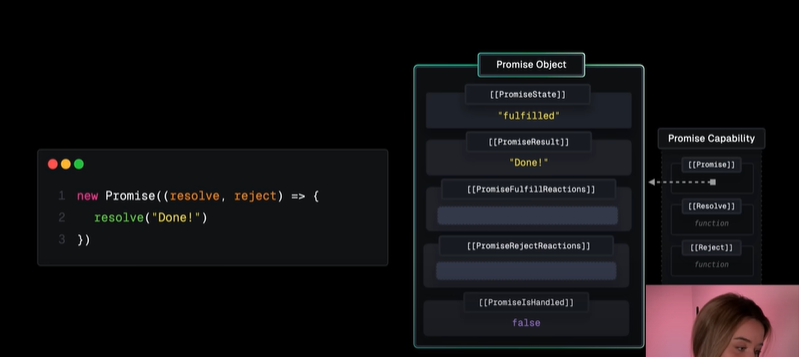
}

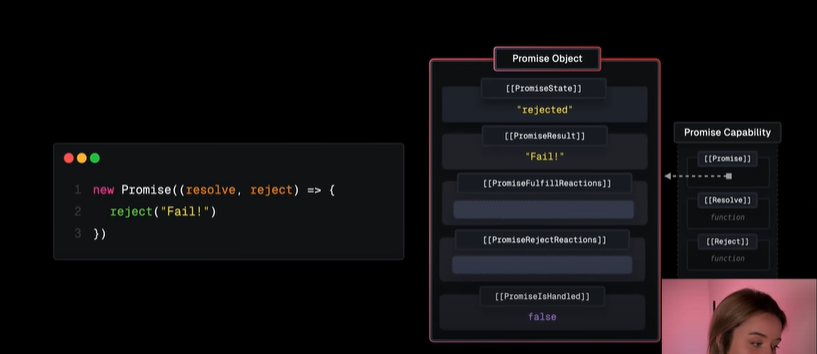
addNumbers(5,10,displaySum)

**Promise:**

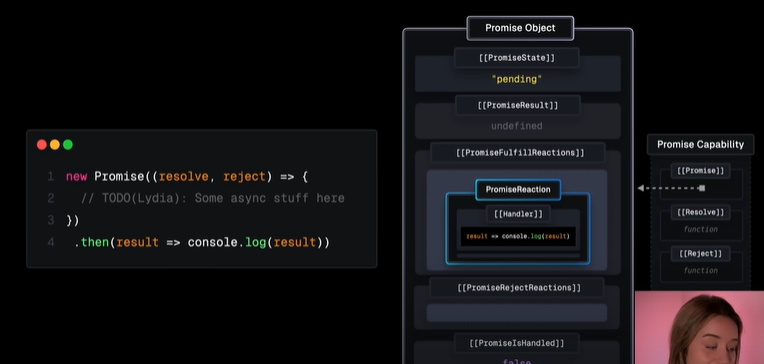
* Promises notify whether the req is fulfilled/ rejected.
* The .then() can be chained to handle the fulfillment and rejection whereas .catch() can be used for handling the errors(if any).
* pending - The initial state of a promise.
* fulfilled - The state of a promise representing a successful operation.
* rejected - The state of a promise representing a failed operation.
* When new promise constructor is used, **promise object** is created.

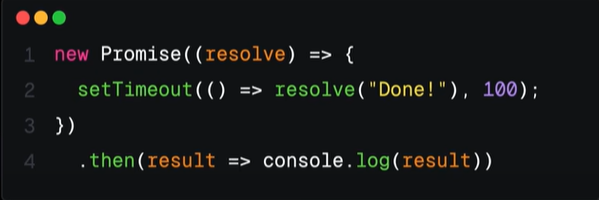


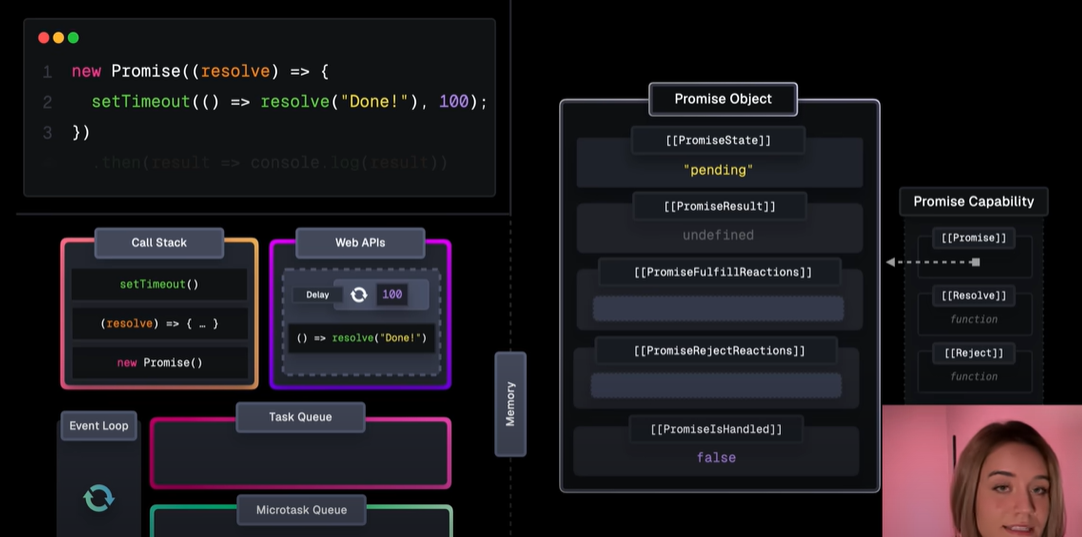


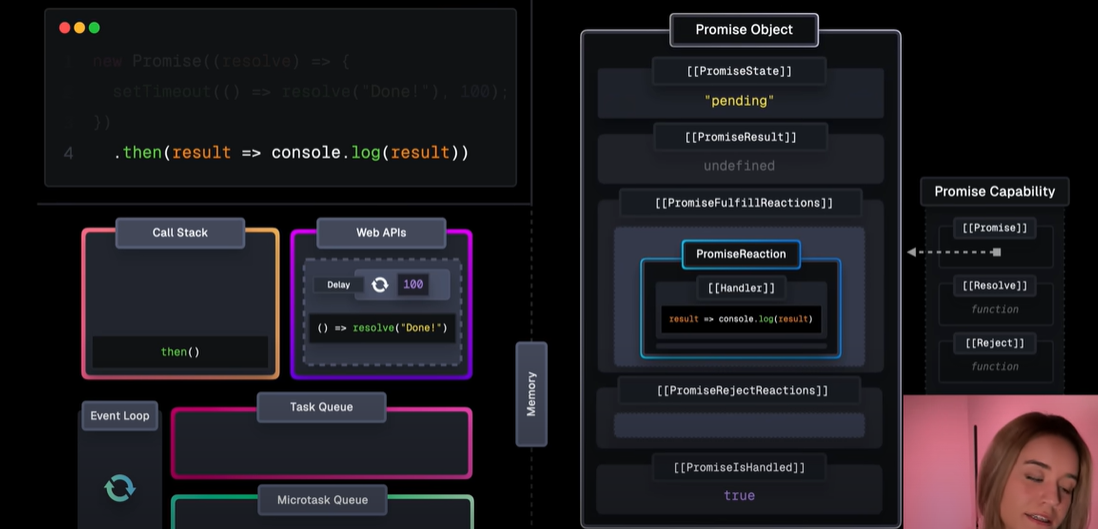


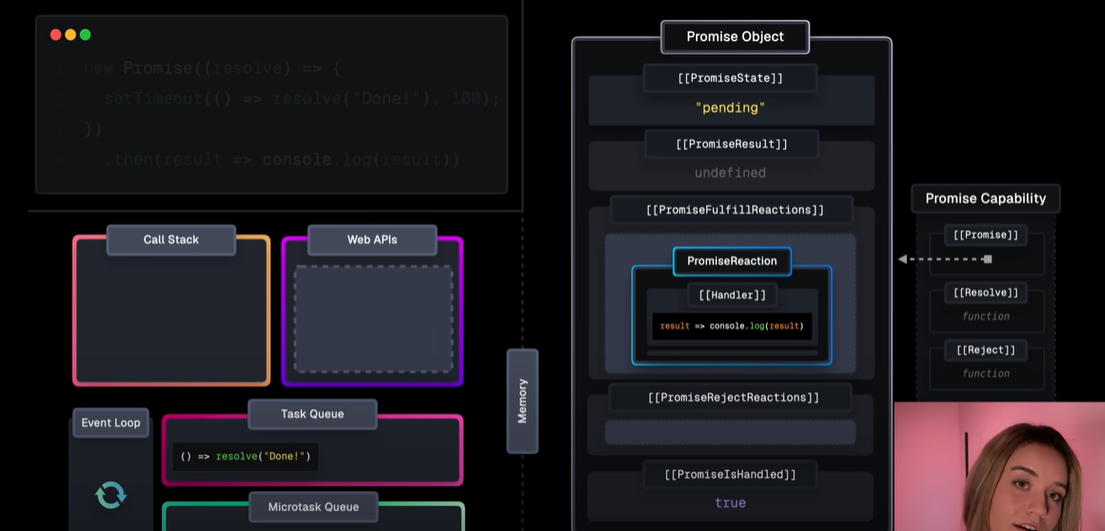
* When resolve is called, then promiseState is fulfilled, promiseResult is value passed inside resolve.
* When used .then(), .catch() promiseFulfilReaction is started.

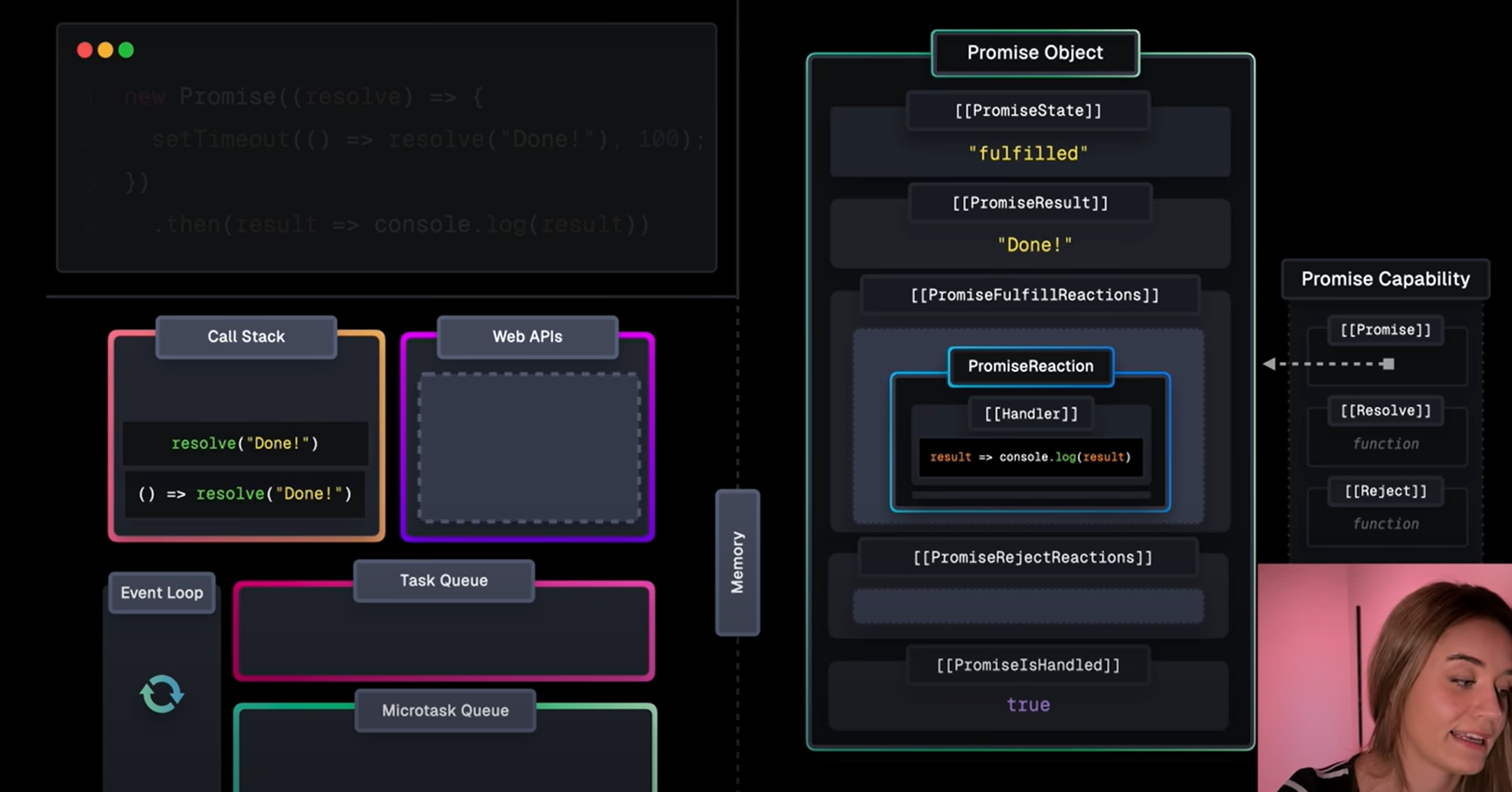


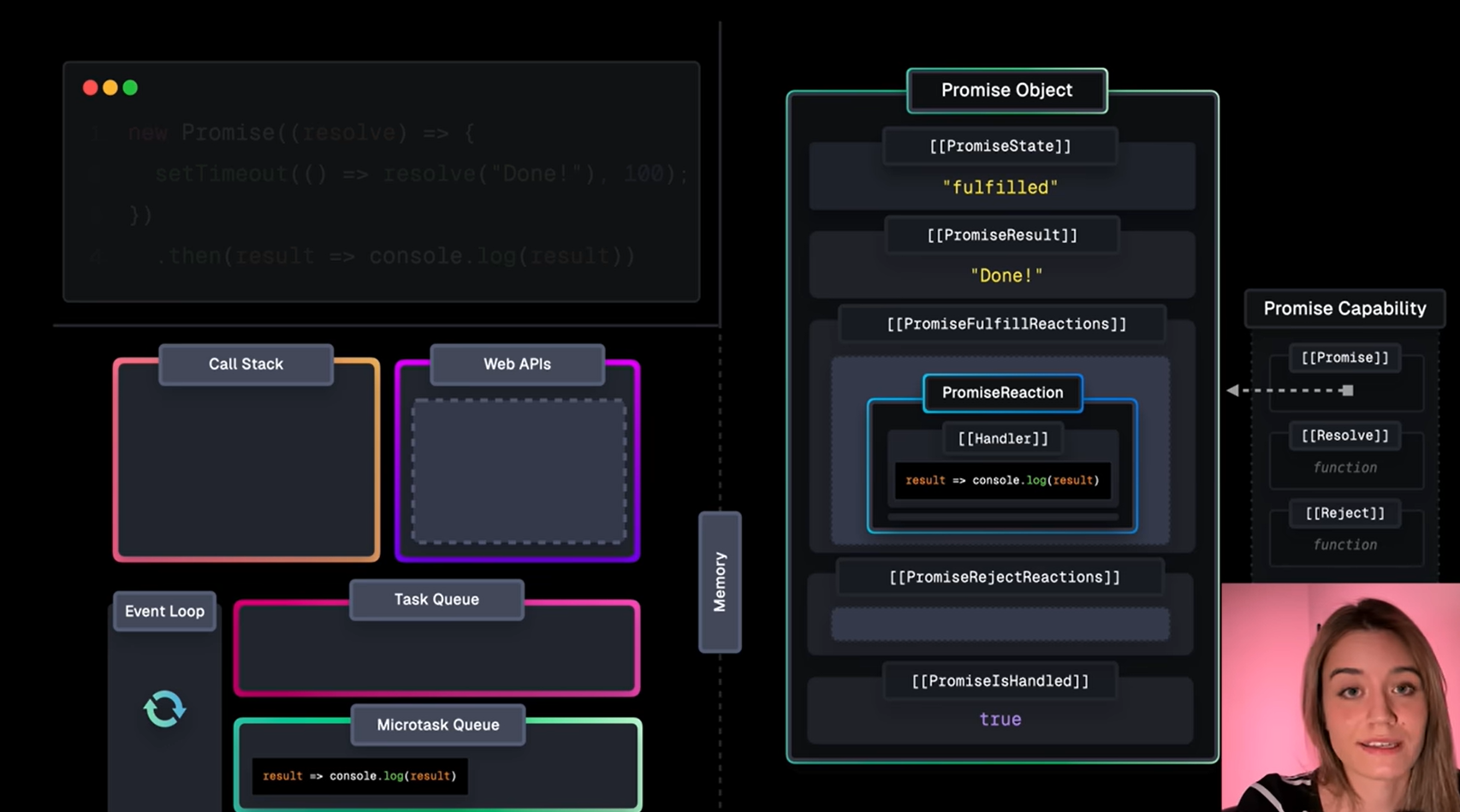


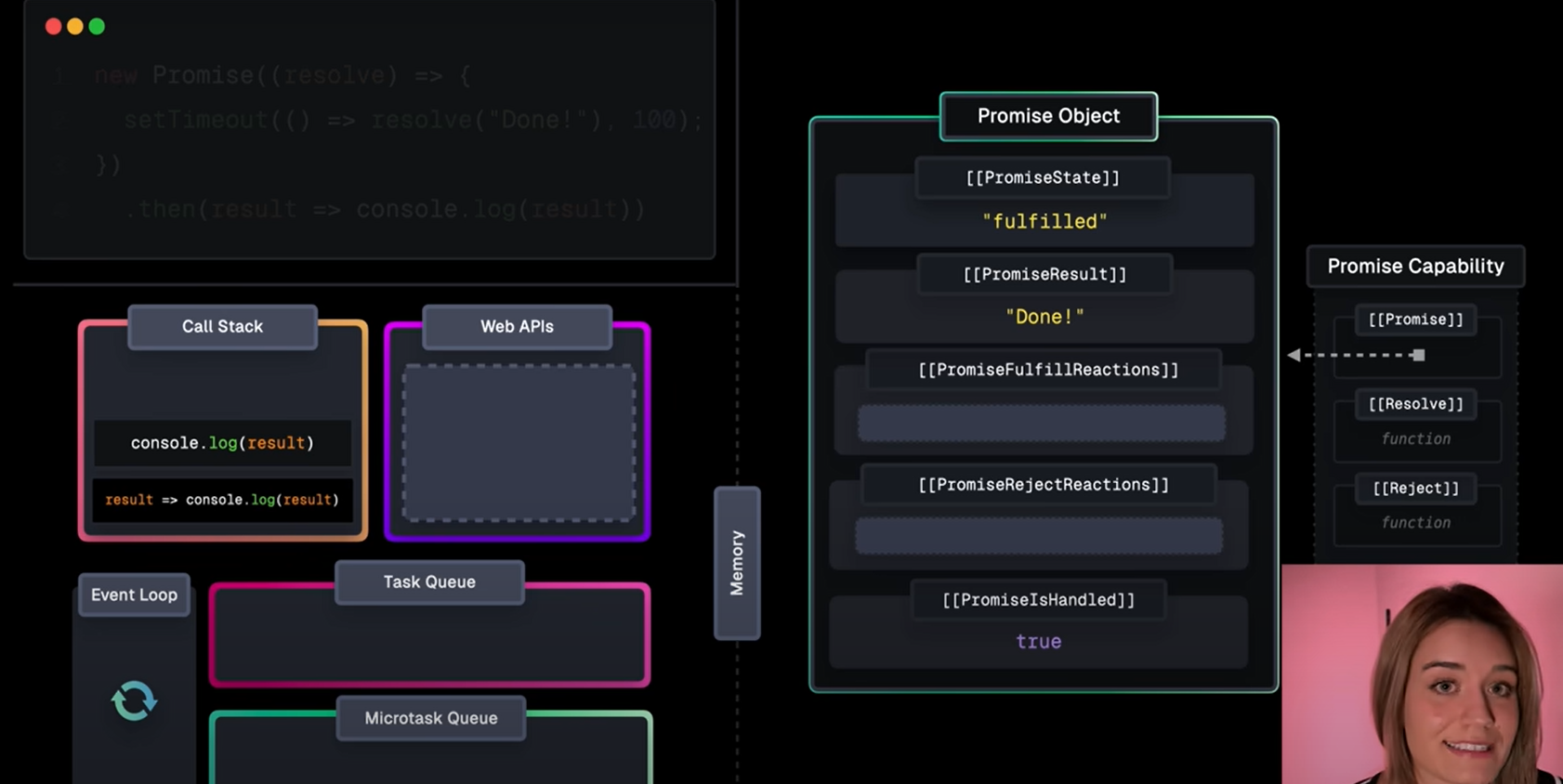












* When a promise is created, the executor function (the function inside new Promise((resolve, reject) => {...})) runs immediately.
* If resolve() is called, the promise is fulfilled, and its result is locked.
* Any subsequent calls to resolve() or reject() are ignored after the promise is settled.
* .then(), .catch(), .finally() creates a new promise object. If the callback func inside the then() is ready to execute then it is placed in microtask queue, then it is placed in call stack.

**Async/ await**

* Chaning of promises with .then() paved a way for async/ await.
* Async/Await is a special syntax to work with promises in a more comfortable fashion.
* It’s easy to understand and use.
* Adding the keyword async before a function ensures that the function returns a promise and the keyword await makes JavaScript wait until that promise settles and returns the result.
* By using try{}, catch{} error handling can be done.
* EX: async function fetchAndLogData() {

try {

const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');

const data = await response.json();

console.log(data);

return data;

}

catch (error) {

console.error('Error fetching data:', error);

}

}

fetchAndLogData();

**process.nextTick():**

* When we pass a function to process.nextTick(), we instruct the engine to invoke this function immediately after the current operation completes, before moving to the next phase in the event loop
* The event loop is busy processing the current function code. When this operation ends, the JS engine runs all the functions passed to nextTick calls during that operation.
* It's the way we can tell the JS engine to process a function asynchronously (after the current function), but as soon as possible, not queue it.
* Calling setTimeout(() => {}, 0) will execute the function at the end of next tick, much later than when using nextTick() which prioritizes the call and executes it just before the beginning of the next tick.

**Event Emitter:**

* It is a module used to handle and work with events.
* It allows to create and manage events in your application.
* **EventEmitter** is a class in the events module.
* An instance is created to use EventEmitter class.
* Events can be triggered and listeners can respond to those events.
* Event must be registered 1st and then triggered. Registration can be done using **myEmitter.on().** To emit an event **myEmitter.emit()** is used.
* myEmitter.on() is same as myEmitter.addListener()

