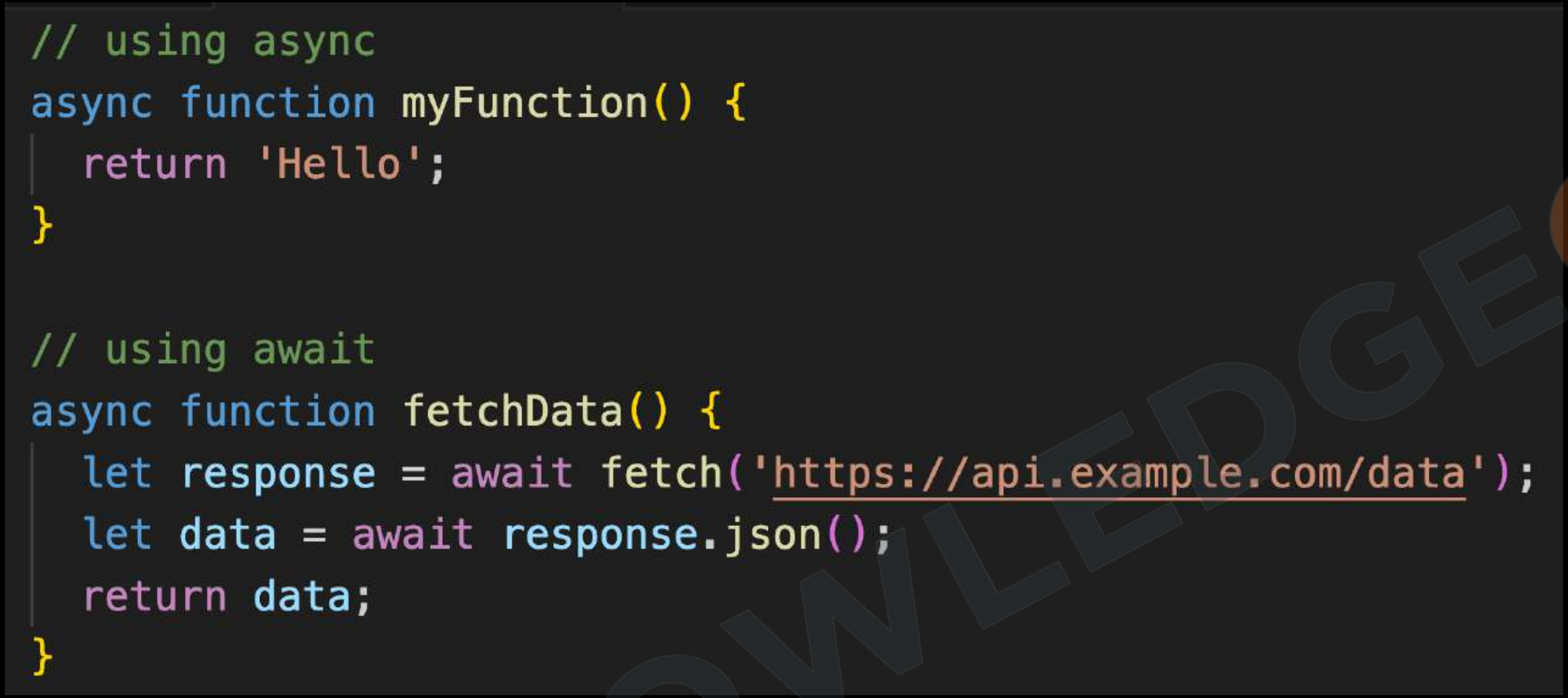
Async / Await

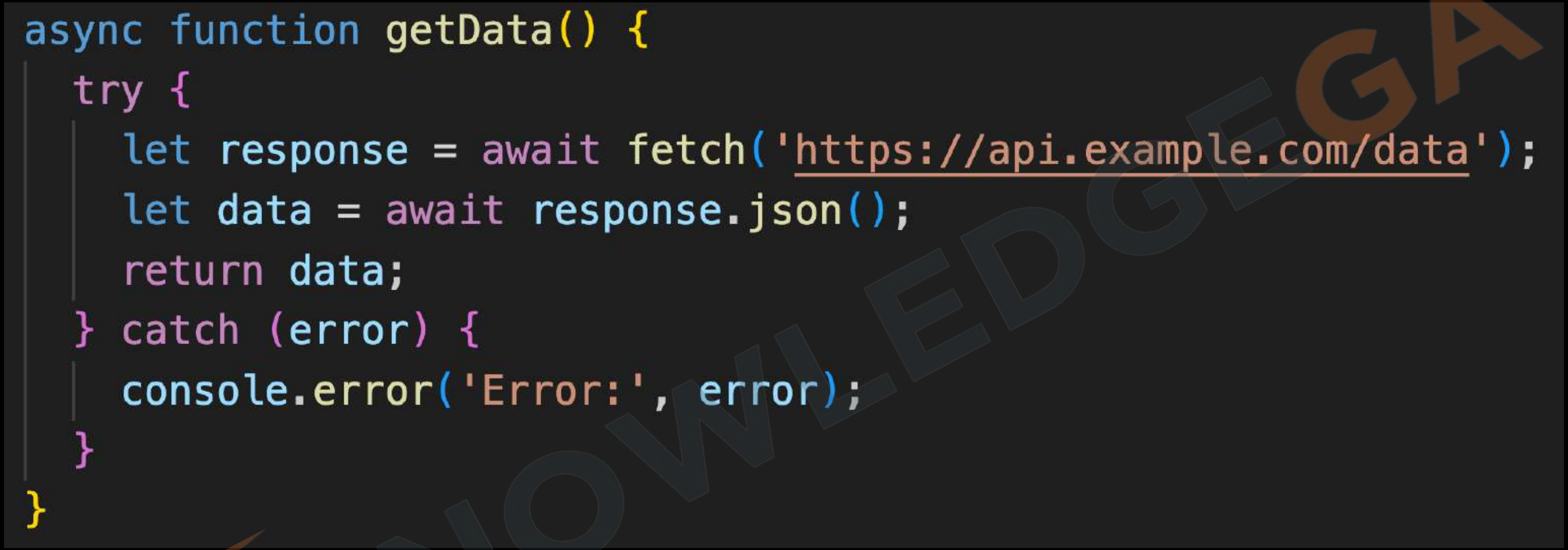


```
// using async
async function myFunction() {
  return 'Hello';
}

// using await
async function fetchData() {
  let response = await fetch('https://api.example.com/data');
  let data = await response.json();
  return data;
}
```

1. Syntax Sugar for Promises: async/await is built on top of promises, providing a cleaner and more readable way to work with asynchronous code.
2. Defining Async Functions: An async function is declared using the async keyword before the function definition. This function always returns a promise.
3. The await keyword is used to pause the execution of an async function until a promise is resolved. It can only be used inside an async function.

Async / Await

(Handling Exceptions)

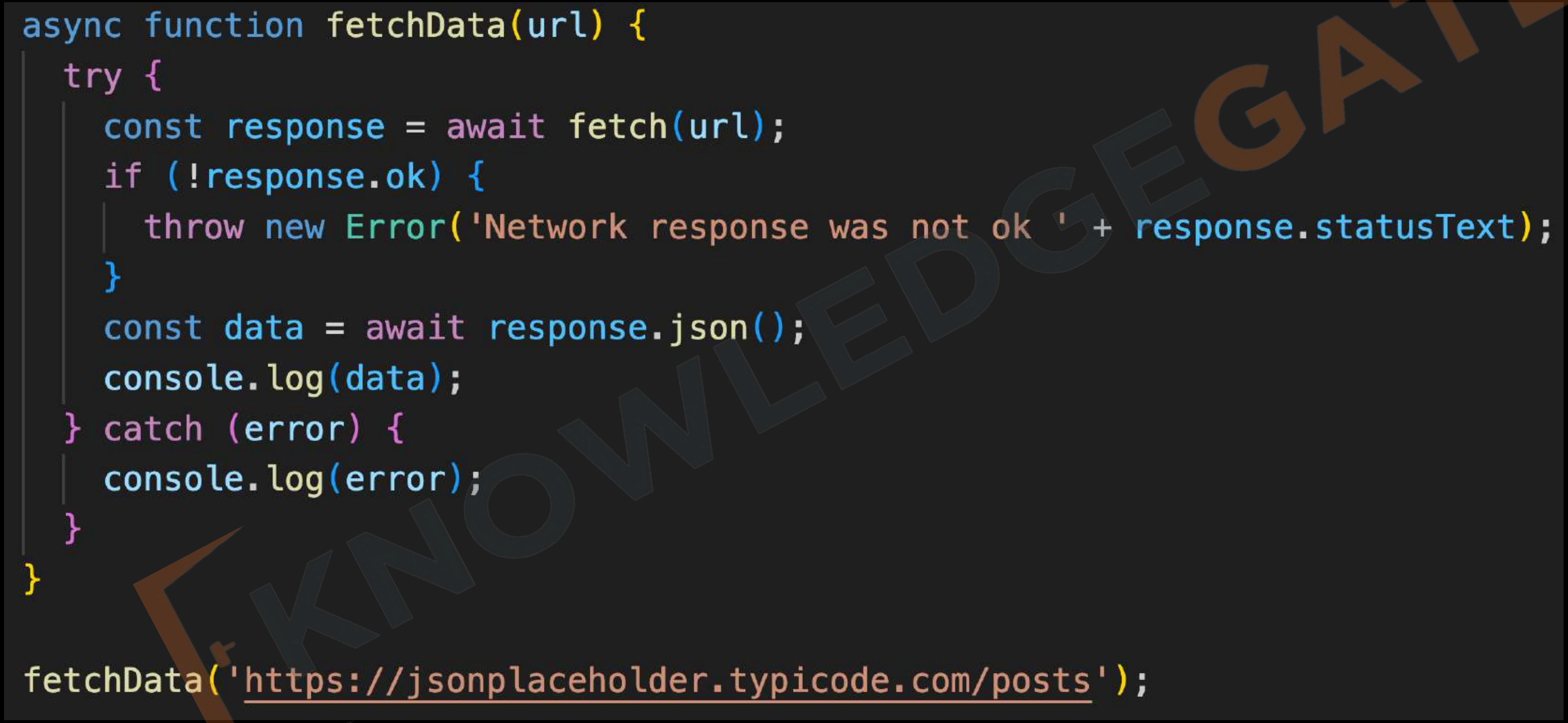


```javascript
async function getData() {
  try {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    return data;
  } catch (error) {
    console.error('Error:', error);
  }
}
```

Errors in async functions can be handled using try...catch blocks, making error management straightforward and consistent with synchronous code.

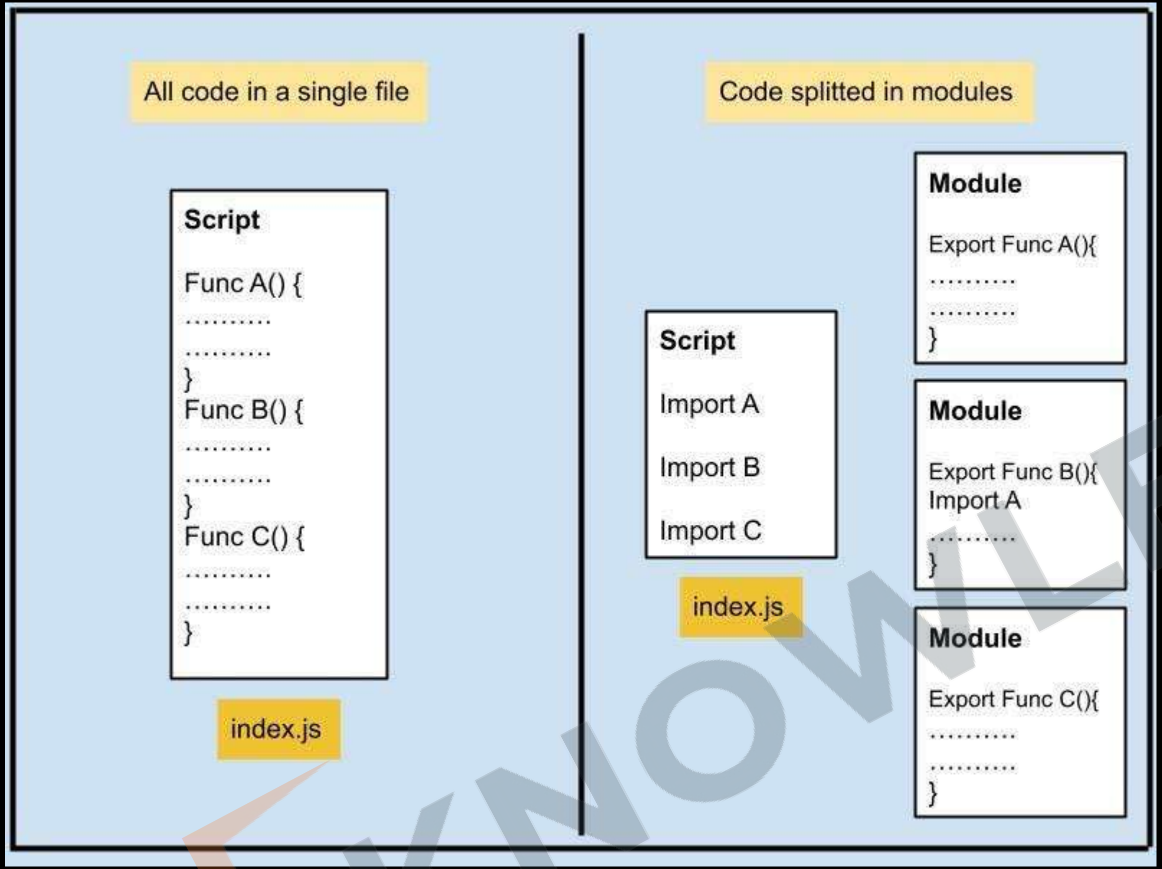
Async / Await

(Fetch API using async/await)



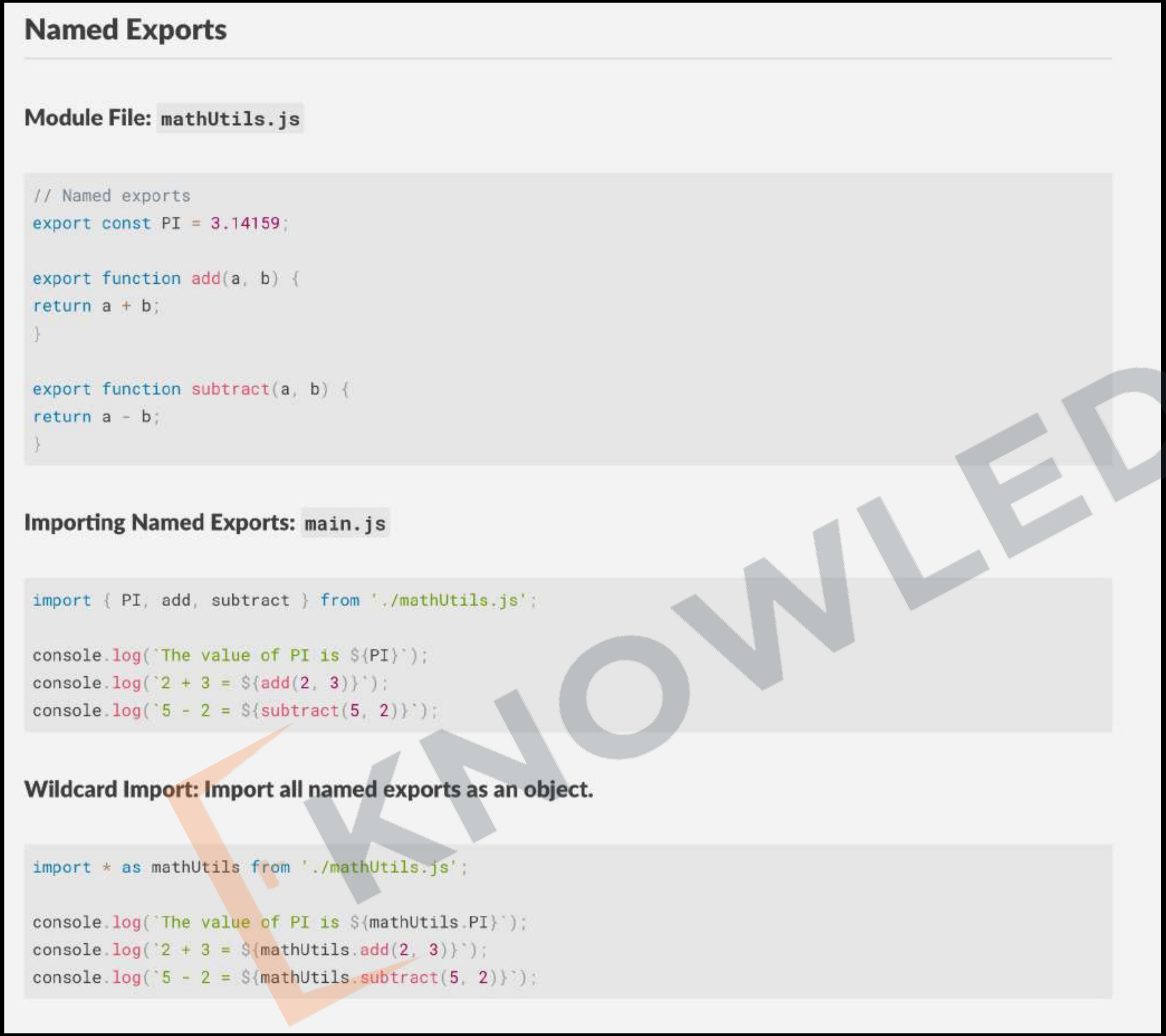
```
async function fetchData(url) {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error('Network response was not ok ' + response.statusText);
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log(error);
  }
}

fetchData('https://jsonplaceholder.typicode.com/posts');
```

Modules



- Modules are used to organize and manage code by dividing it into separate files or modules.
- This modular approach enhances code maintainability, reusability, and scalability.
- Modules can be imported and exported using the import and export statements.
- Modules are JavaScript files that encapsulate code and expose specific parts using the export keyword.
- The import keyword is used to bring in the exported features from one module to another.

Modules (Named Exports)



Named Exports

Module File: `mathUtils.js`

```
// Named exports
export const PI = 3.14159;

export function add(a, b) {
return a + b;
}

export function subtract(a, b) {
return a - b;
}
```

Importing Named Exports: `main.js`

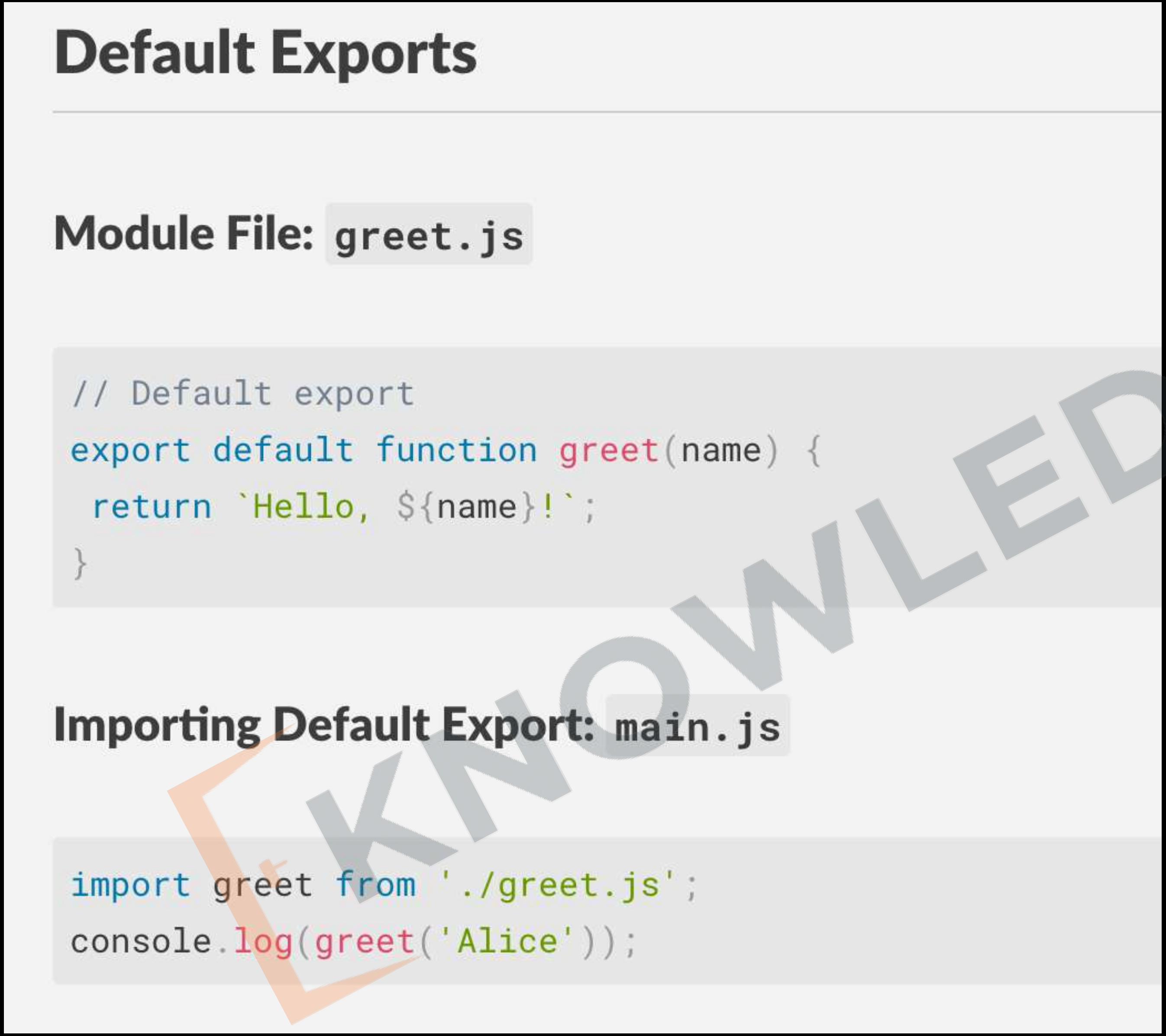
```
import { PI, add, subtract } from './mathUtils.js';

console.log(`The value of PI is ${PI}`);
console.log(`2 + 3 = ${add(2, 3)}`);
console.log(`5 - 2 = ${subtract(5, 2)}`);
```

Wildcard Import: Import all named exports as an object.

```
import * as mathUtils from './mathUtils.js';

console.log(`The value of PI is ${mathUtils.PI}`);
console.log(`2 + 3 = ${mathUtils.add(2, 3)}`);
console.log(`5 - 2 = ${mathUtils.subtract(5, 2)}`);
```

Allow you to export multiple values from a module. Each export must be imported using its exact name.

Modules (Default Exports)



Default Exports

Module File: `greet.js`

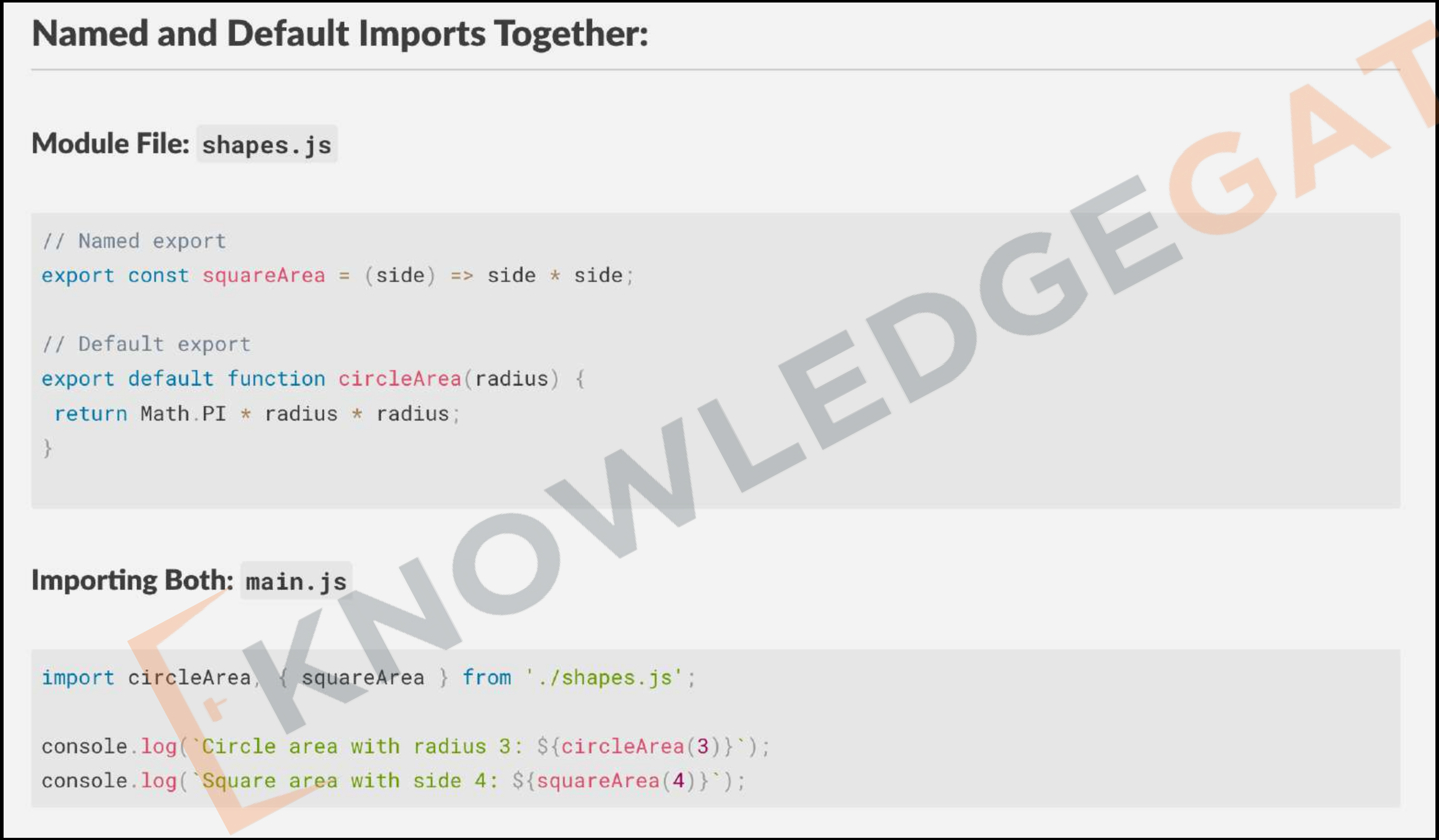
```
// Default export
export default function greet(name) {
 return `Hello, ${name}!`;
}
```

Importing Default Export: `main.js`

```
import greet from './greet.js';
console.log(greet('Alice'));
```

Allow you to export a single default value from a module. The importing module can choose any name for the default export.

Modules (Together)



Named and Default Imports Together:

Module File: `shapes.js`

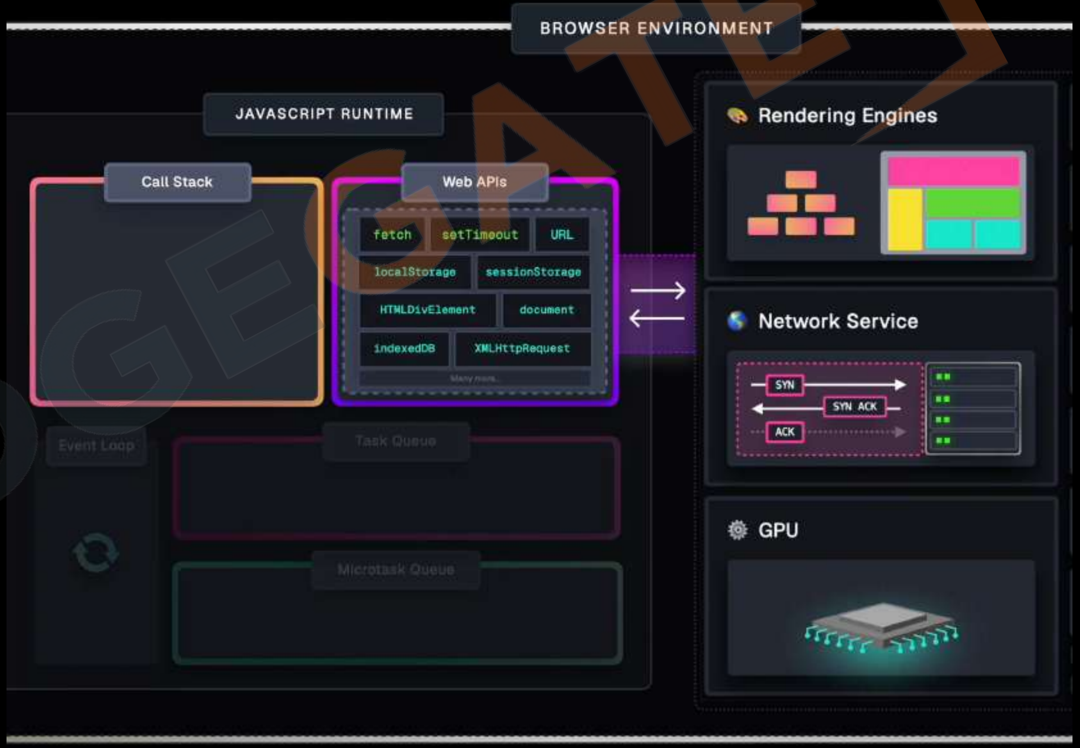
```javascript
// Named export
export const squareArea = (side) => side * side;

// Default export
export default function circleArea(radius) {
 return Math.PI * radius * radius;
}
```

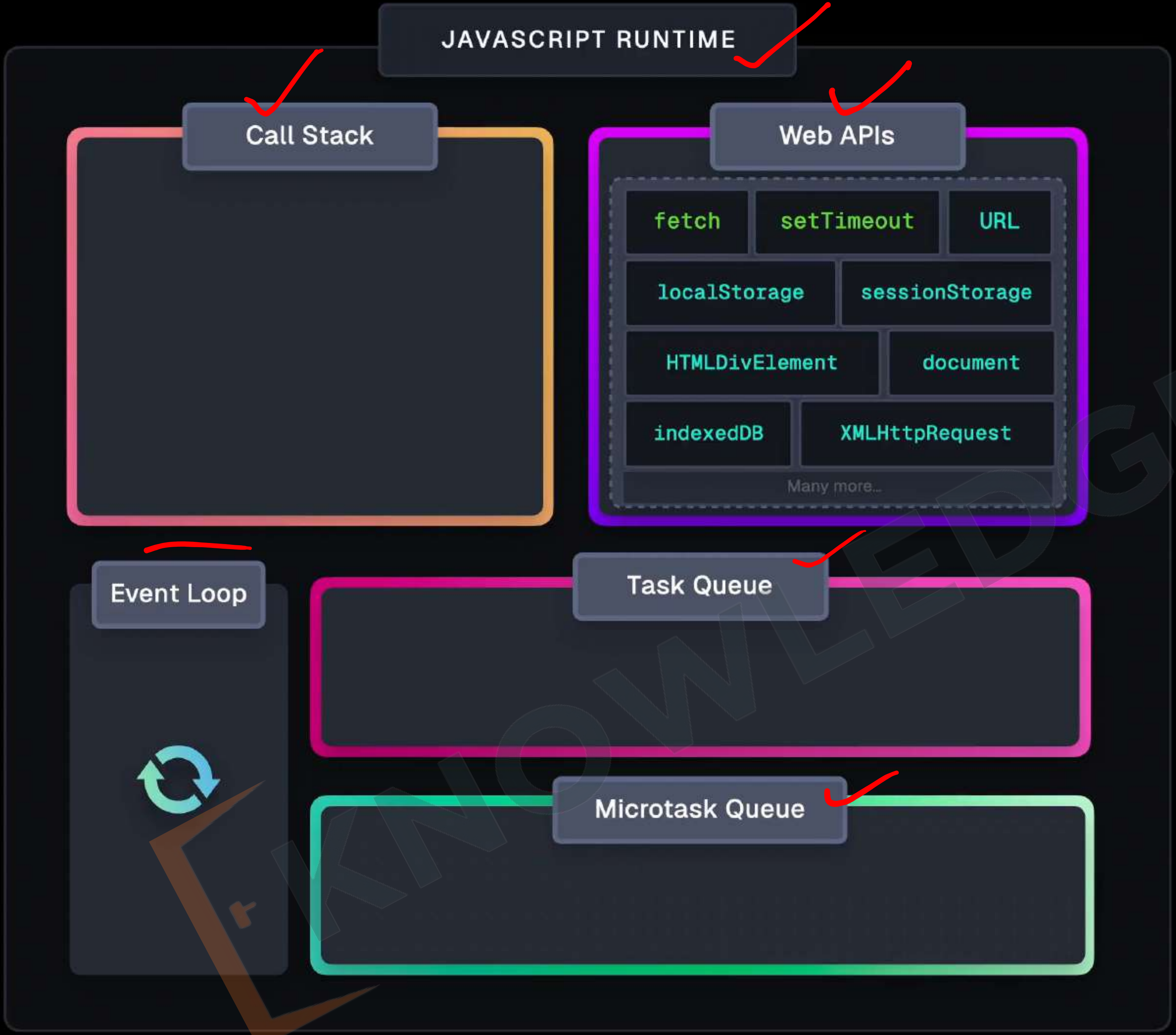
Importing Both: `main.js`

```javascript
import circleArea, { squareArea } from './shapes.js';

console.log(`Circle area with radius 3: ${circleArea(3)}`);
console.log(`Square area with side 4: ${squareArea(4)}`);
```

How JavaScript Works?

- Event Loop
- Call Stack
- Web APIs & Browser Env
- Async Task API
- Popular Web APIs
- Geo Location (Callback based API)
- Task Queue
- setTimeout (Callback based API)
- Microtask Queue
- Fetch (Promise based API)

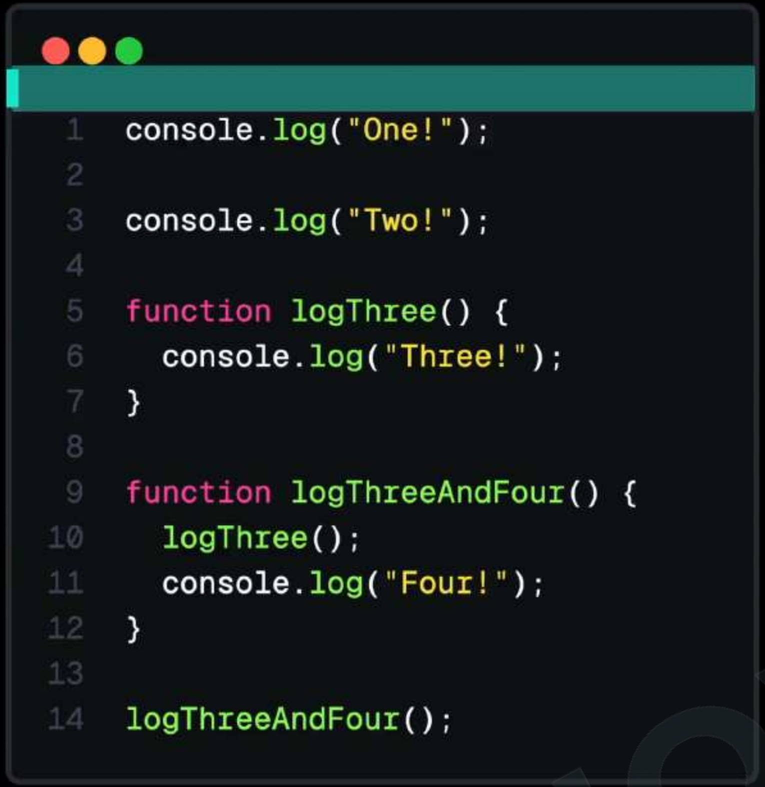




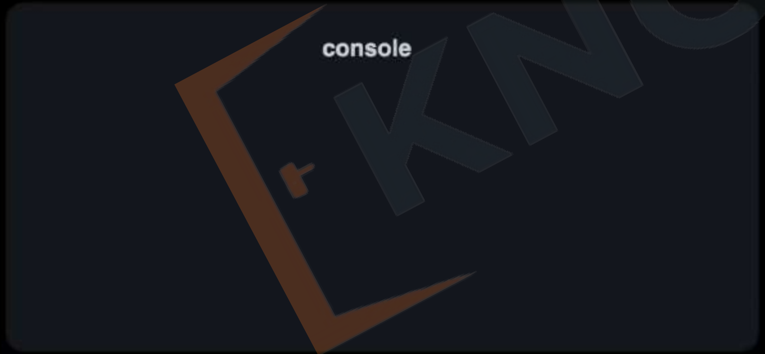
Event Loop

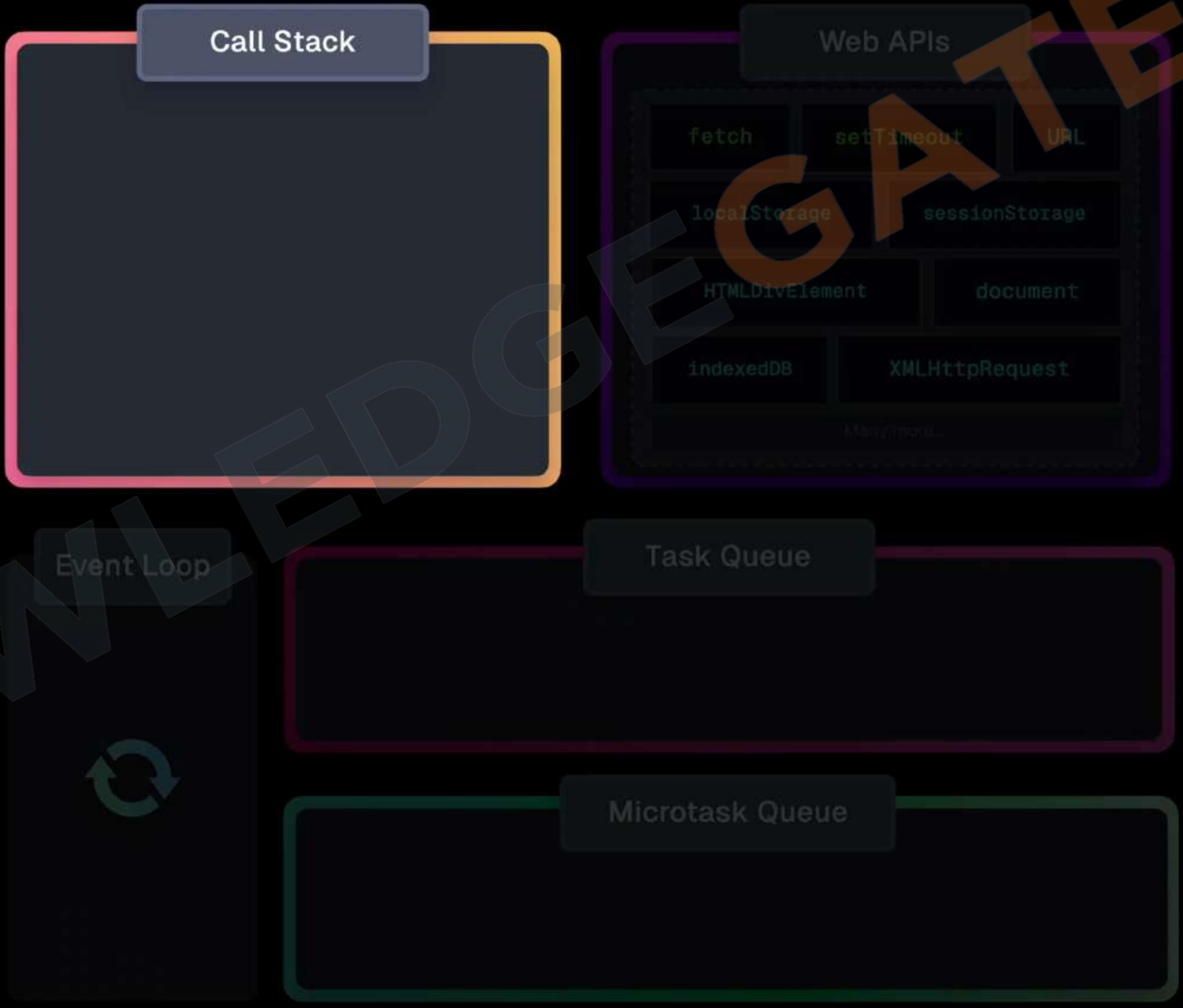
1. it's just a tiny component within the JavaScript runtime!
2. The event loop is a mechanism in JavaScript that handles asynchronous operations, ensuring that non-blocking tasks are executed efficiently.
3. JavaScript runs on a single thread, meaning it can only perform one operation at a time. The event loop helps manage multiple tasks without blocking the main thread.

Call Stack

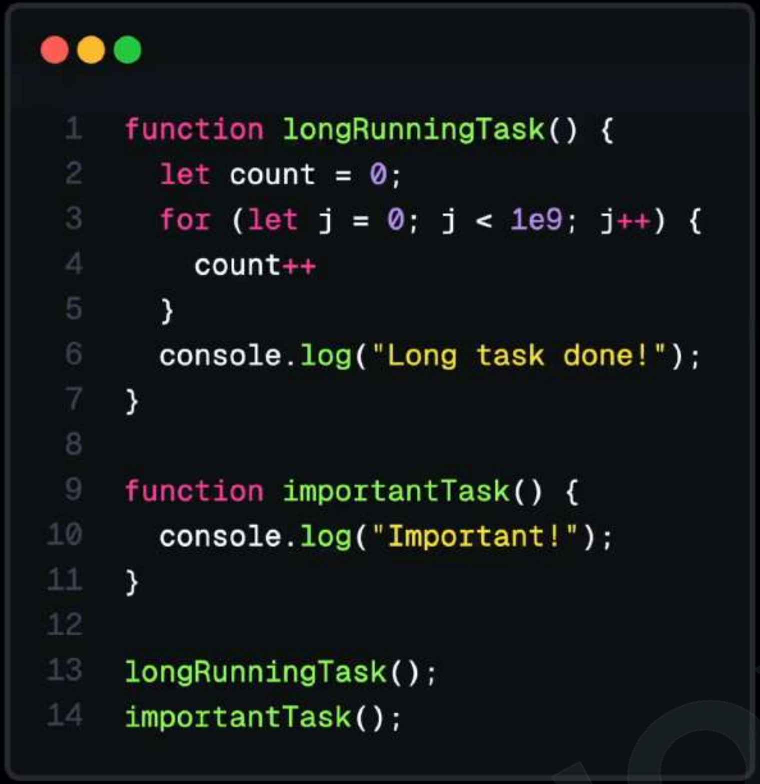


```
console.log("One!");

console.log("Two!");

function logThree() {
  console.log("Three!");
}

function logThreeAndFour() {
  logThree();
  console.log("Four!");
}

logThreeAndFour();
```

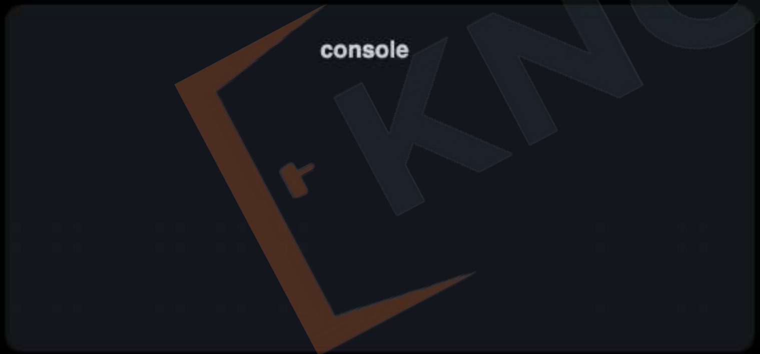


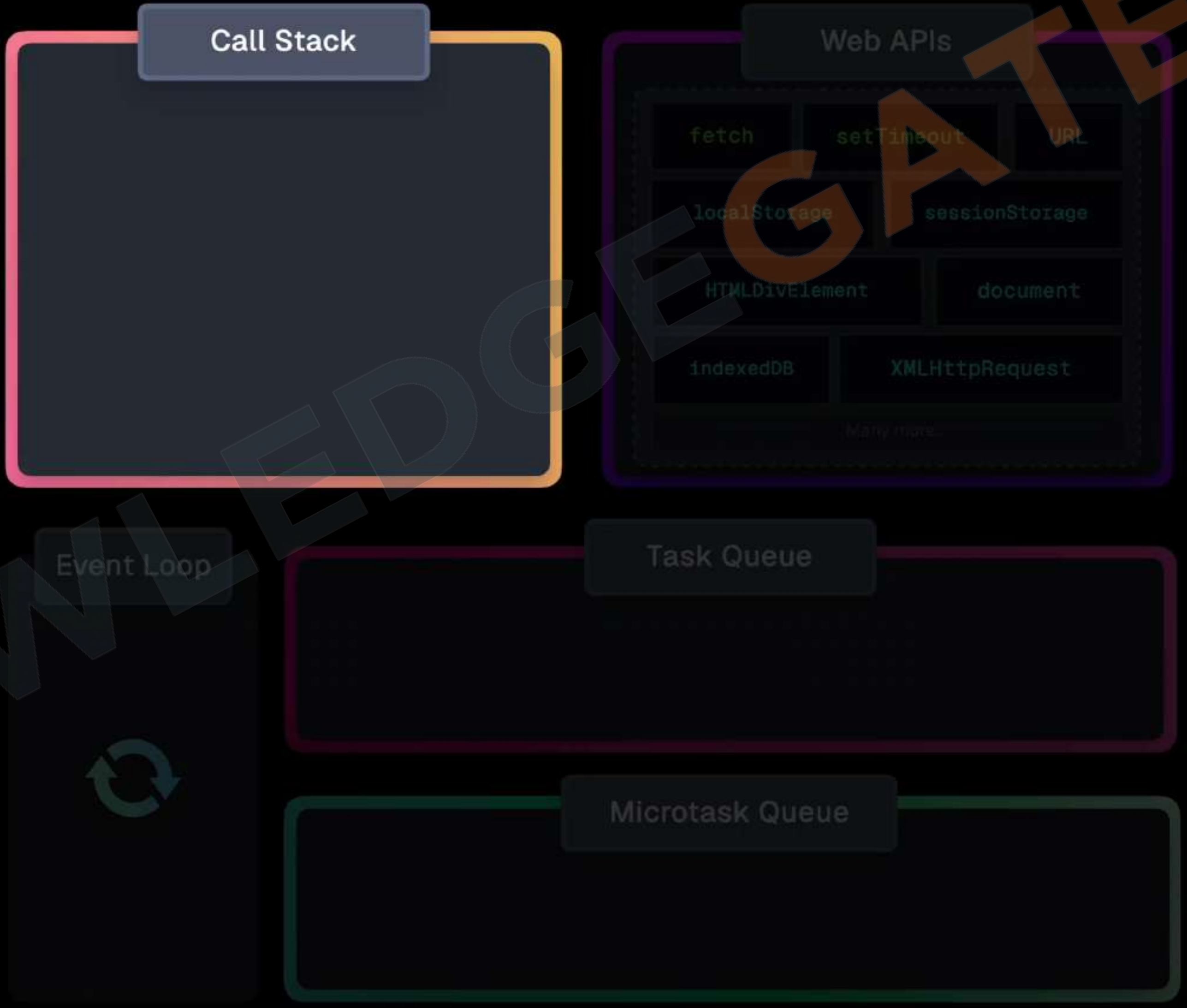


Call Stack (Problem with only one task at a time)



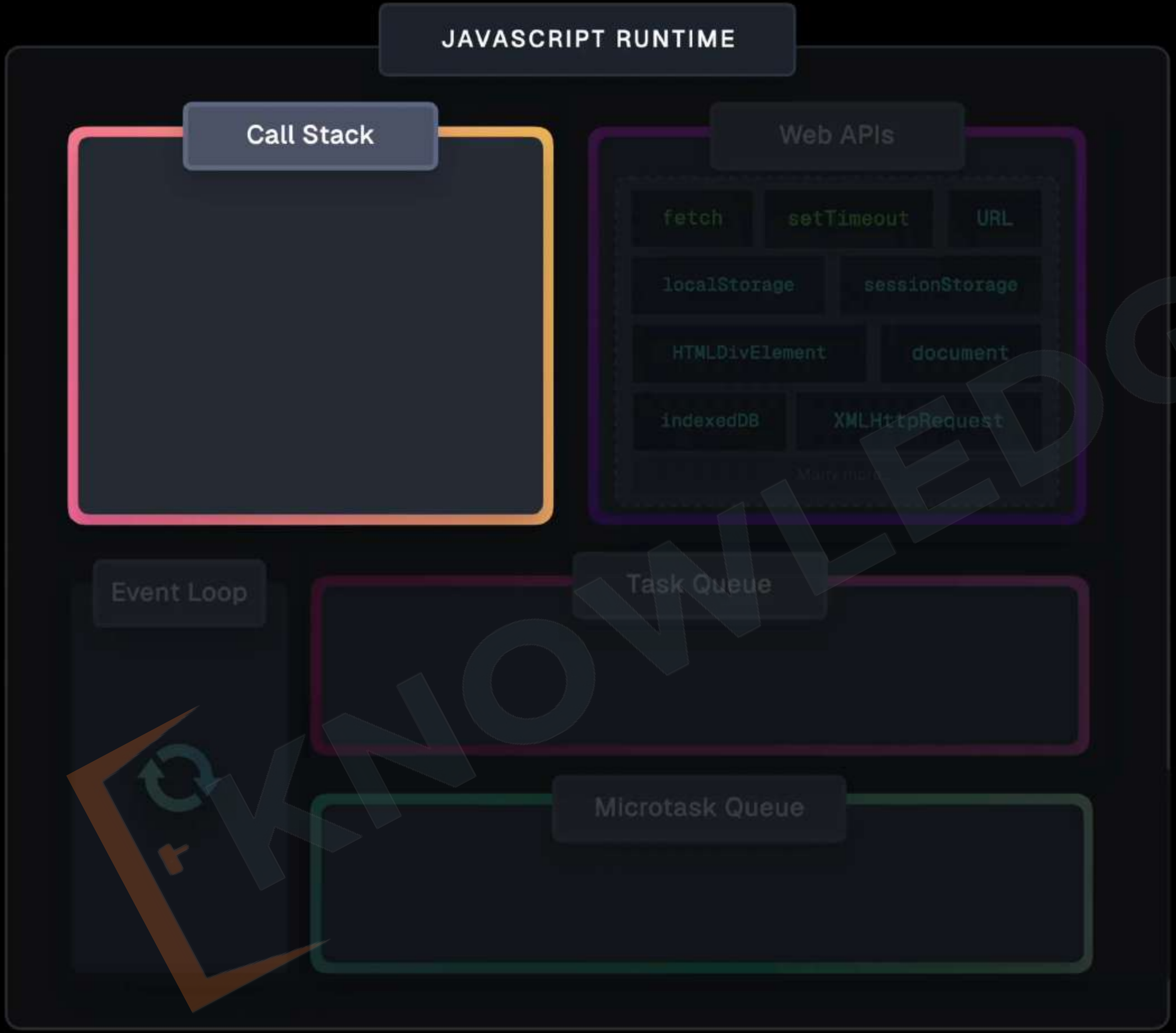
```
function longRunningTask() {
  let count = 0;
  for (let j = 0; j < 1e9; j++) {
    count++
  }
  console.log("Long task done!");
}

function importantTask() {
  console.log("Important!");
}

longRunningTask();
importantTask();
```



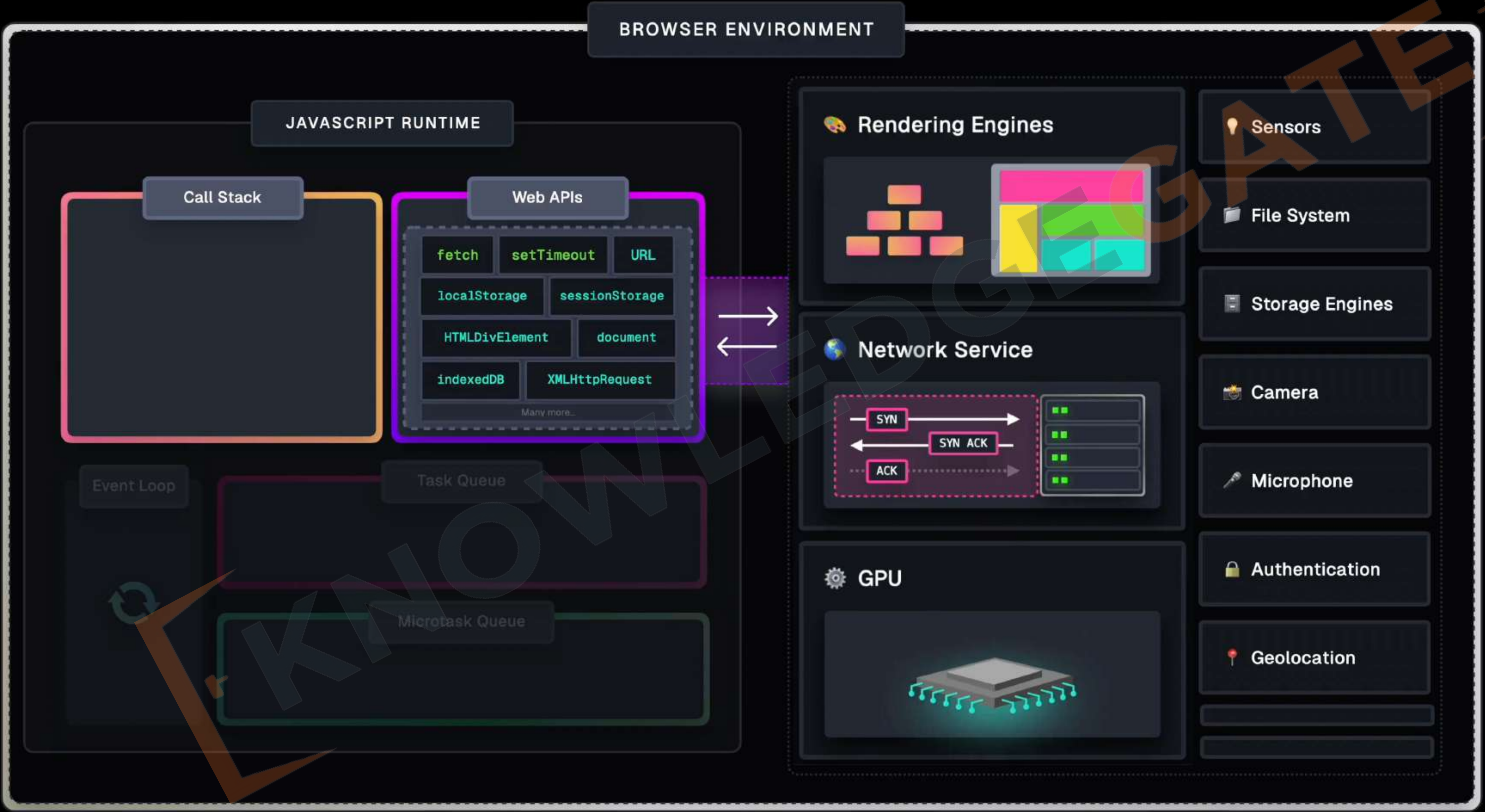


Web APIs & Browser Environment

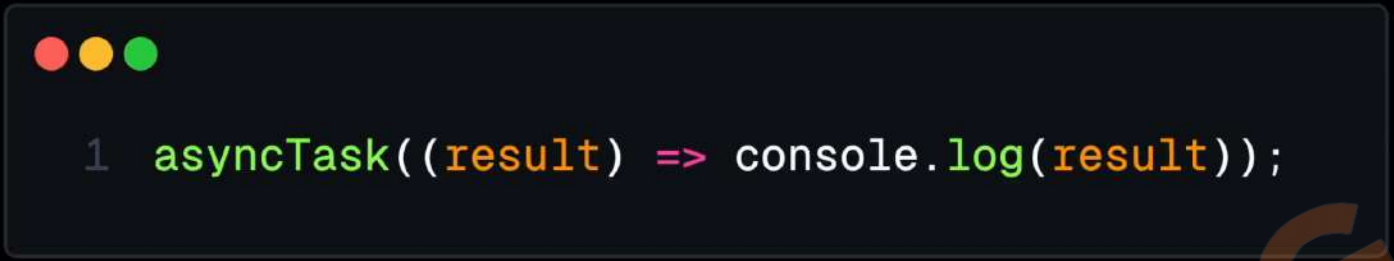
Solution of only one task at a time.



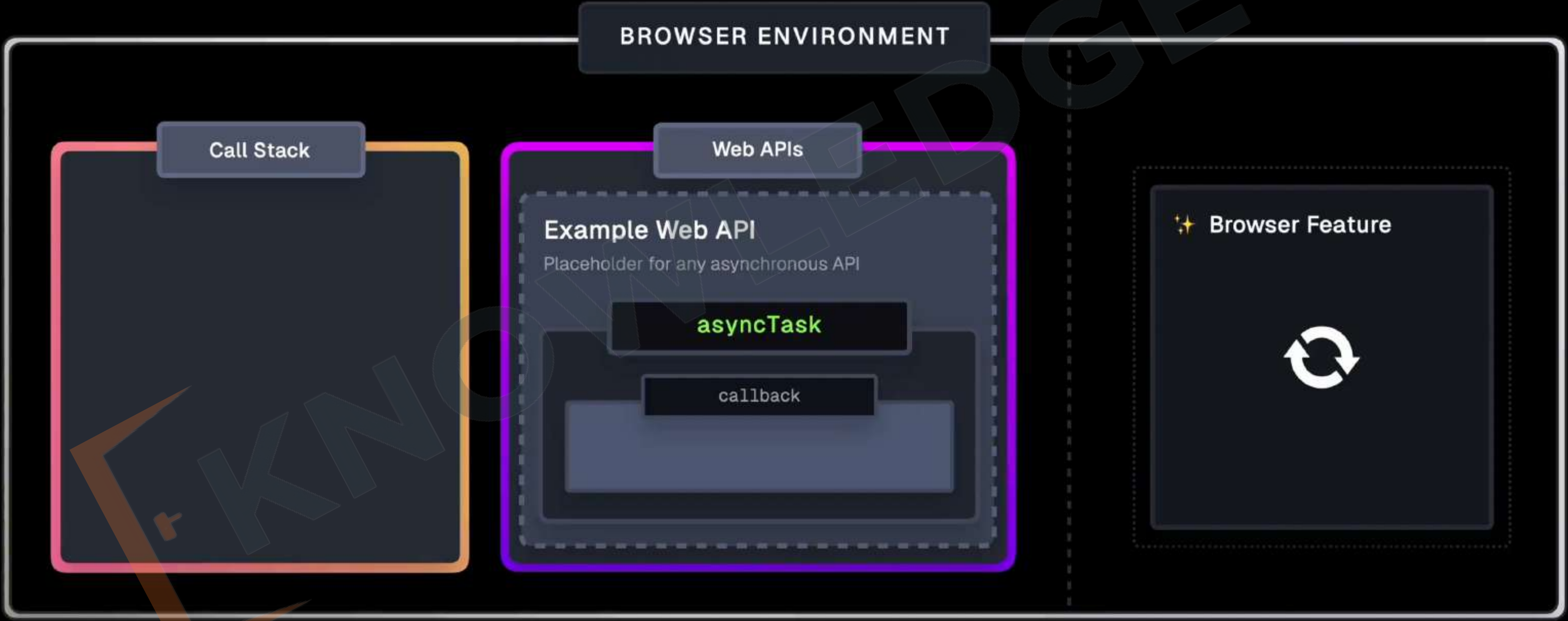
Web APIs & Browser Environment



Async Task API

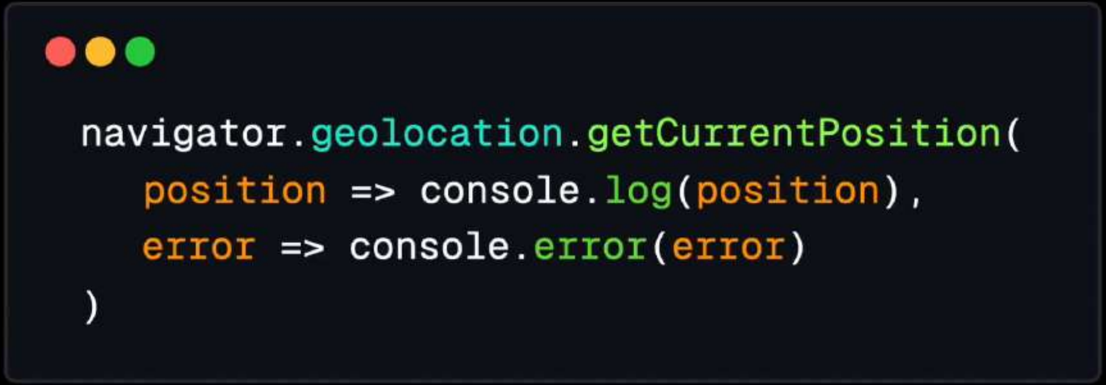


```
asyncTask((result) => console.log(result));
```

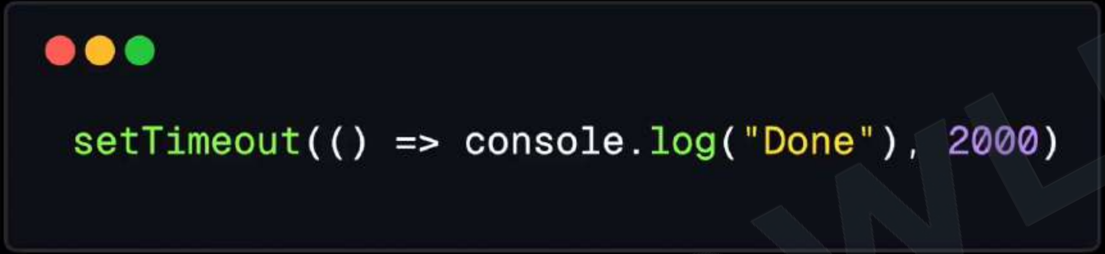


Popular Web APIs

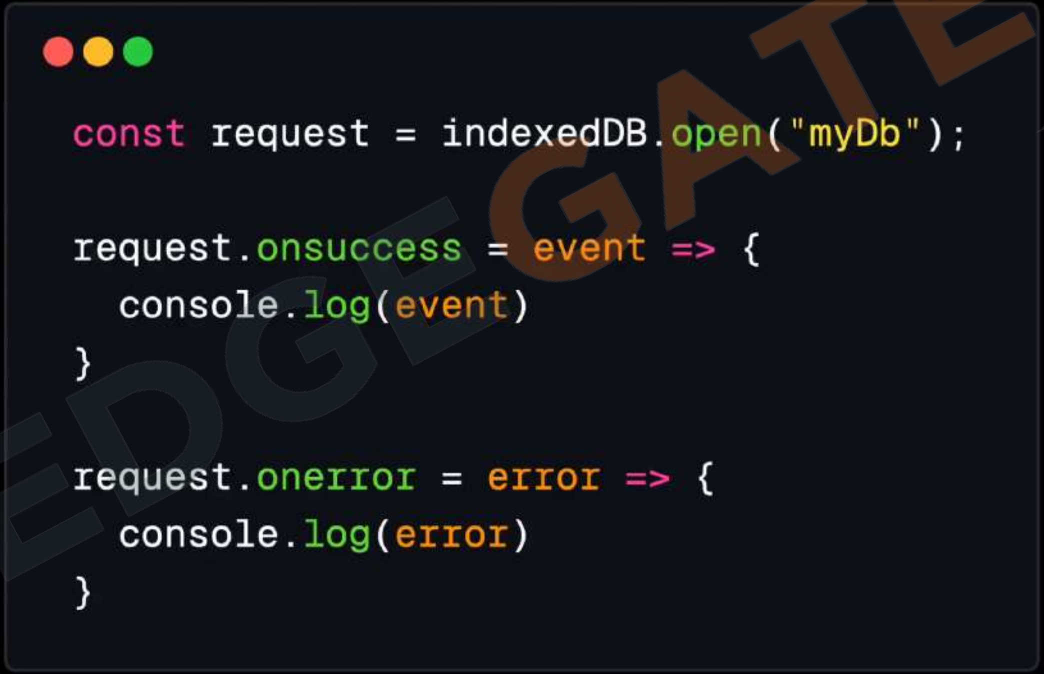
CALLBACKS, FOR EXAMPLE:



```
navigator.geolocation.getCurrentPosition(
   position => console.log(position),
   error => console.error(error)
)
```

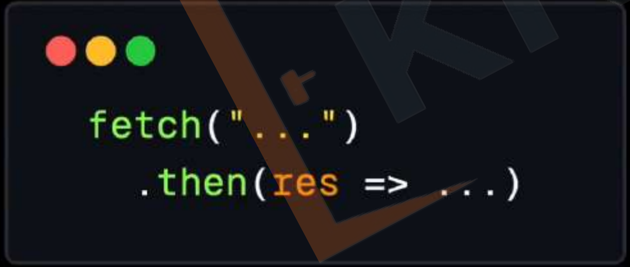


```
setTimeout(() => console.log("Done"), 2000)
```

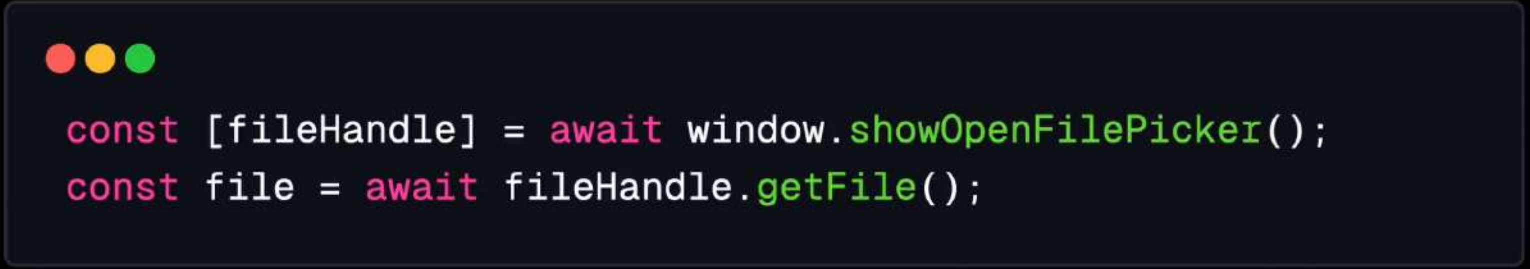


```
const request = indexedDB.open("myDb");

request.onsuccess = event => {
  console.log(event)
}

request.onerror = error => {
  console.log(error)
}
```

PROMISES, FOR EXAMPLE:

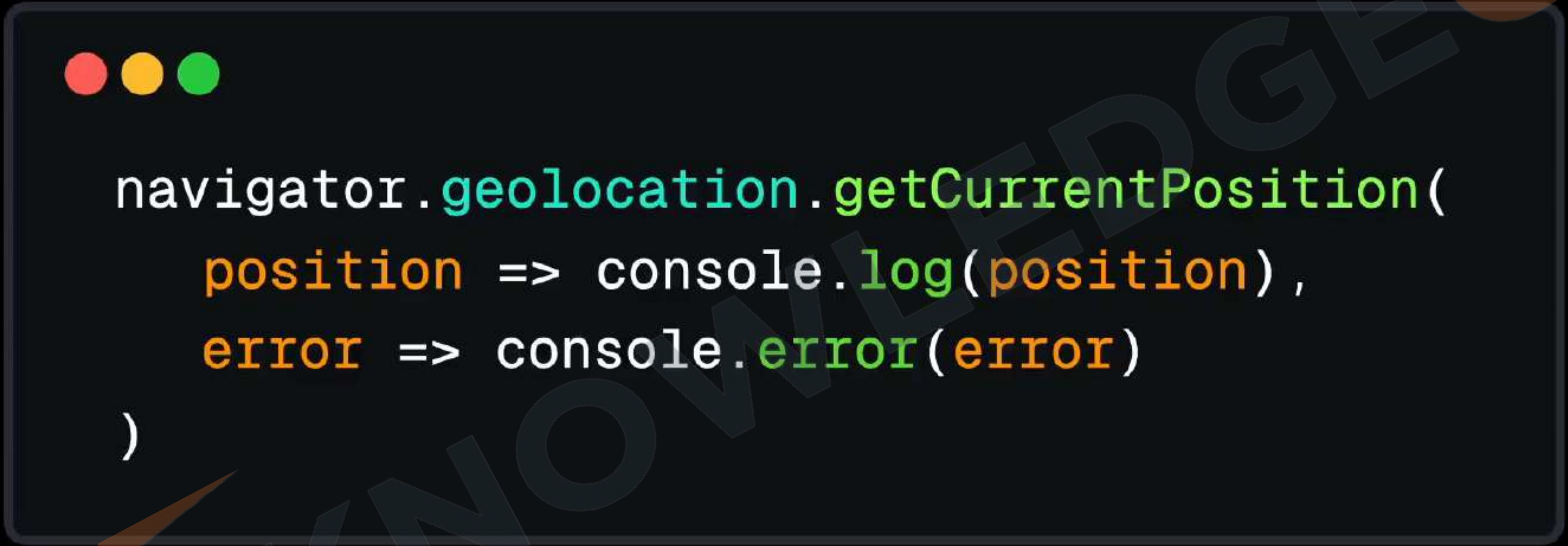


```
fetch("...")
  .then(res => ...)
```



```
const [fileHandle] = await window.showOpenFilePicker();
const file = await fileHandle.getFile();
```

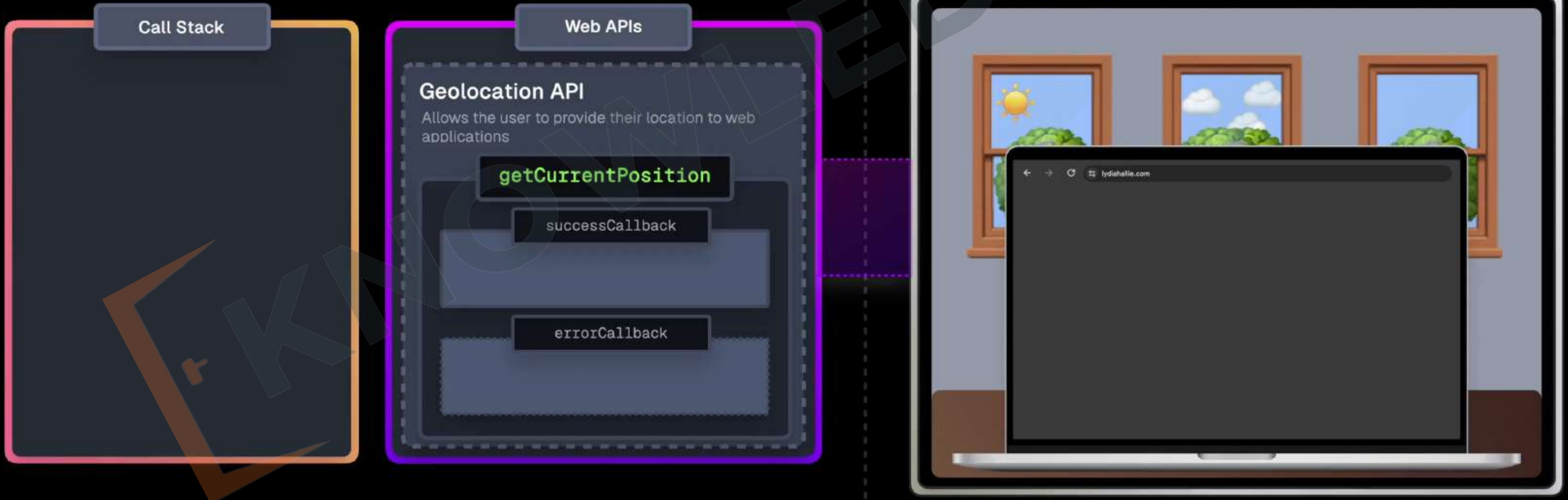
Geo Location (Callback based API)



```
navigator.geolocation.getCurrentPosition(
   position => console.log(position),
   error => console.error(error)
)
```

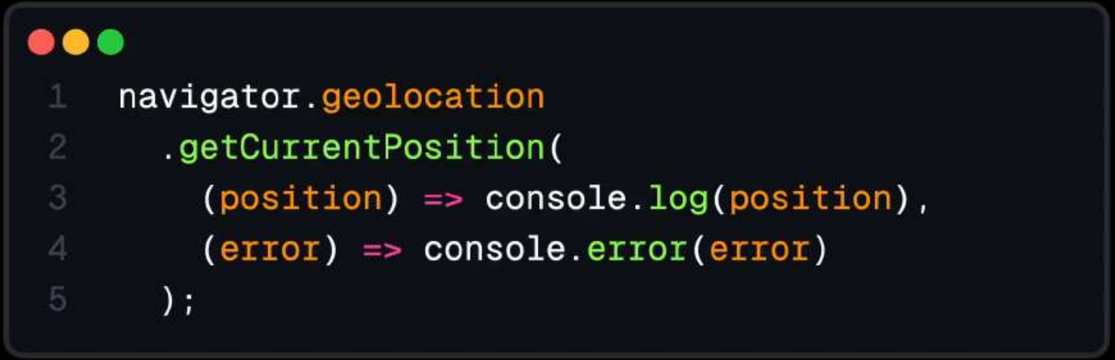
Geo Location
(Call Initiation)

```
navigator.geolocation
  .getCurrentPosition(
    (position) => console.log(position),
    (error) => console.error(error)
  );
```

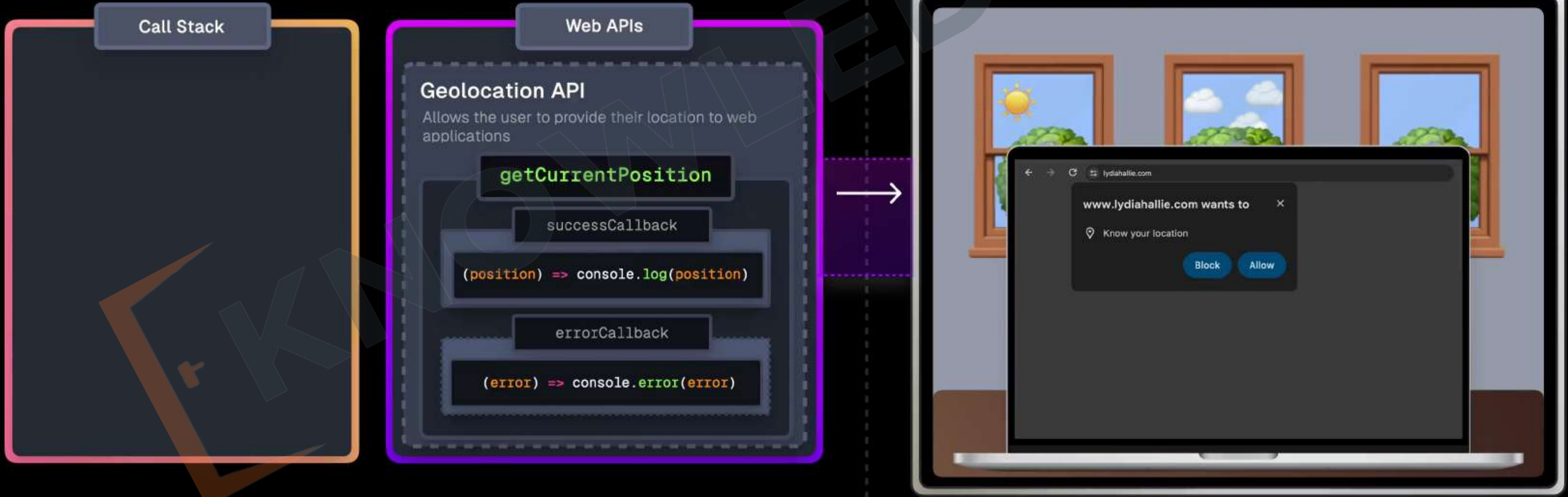


Geo Location

(Call stack executing other tasks)

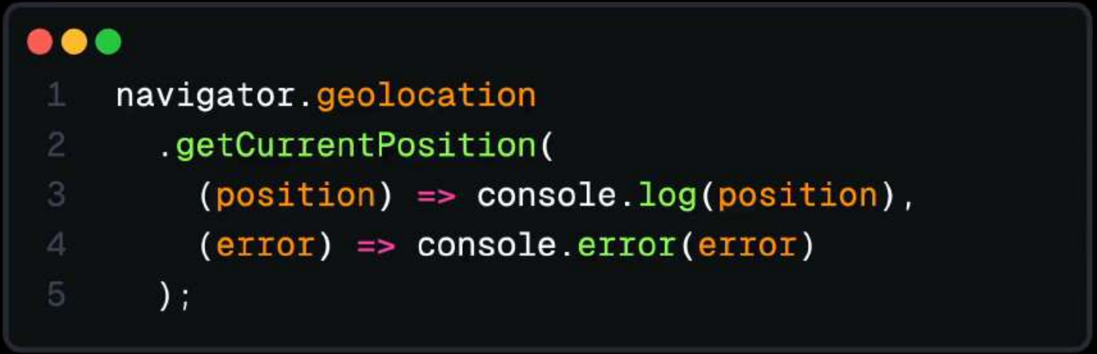


```
navigator.geolocation
  .getCurrentPosition(
    (position) => console.log(position),
    (error) => console.error(error)
  );
```

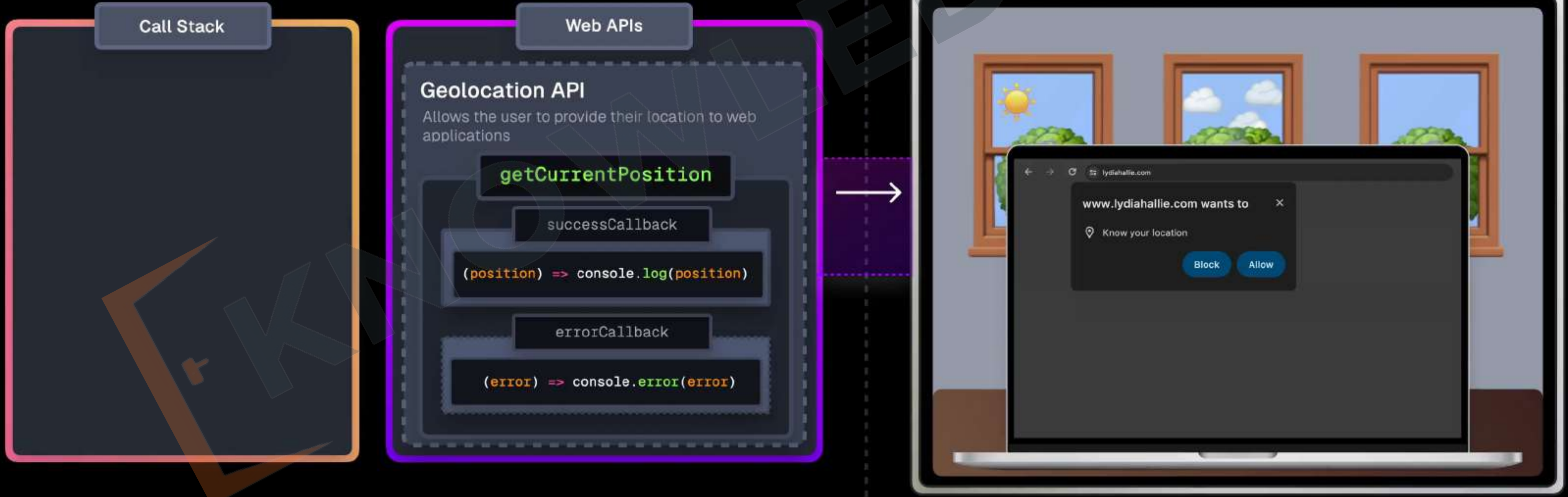


Call Stack
Web APIs
Geolocation API
Allows the user to provide their location to web applications
getCurrentPosition
successCallback
(position) => console.log(position)
errorCallback
(error) => console.error(error)
lydiahallie.com
www.lydiahallie.com wants to
Know your location
Block
Allow

Geo Location
(Execution of Callback)



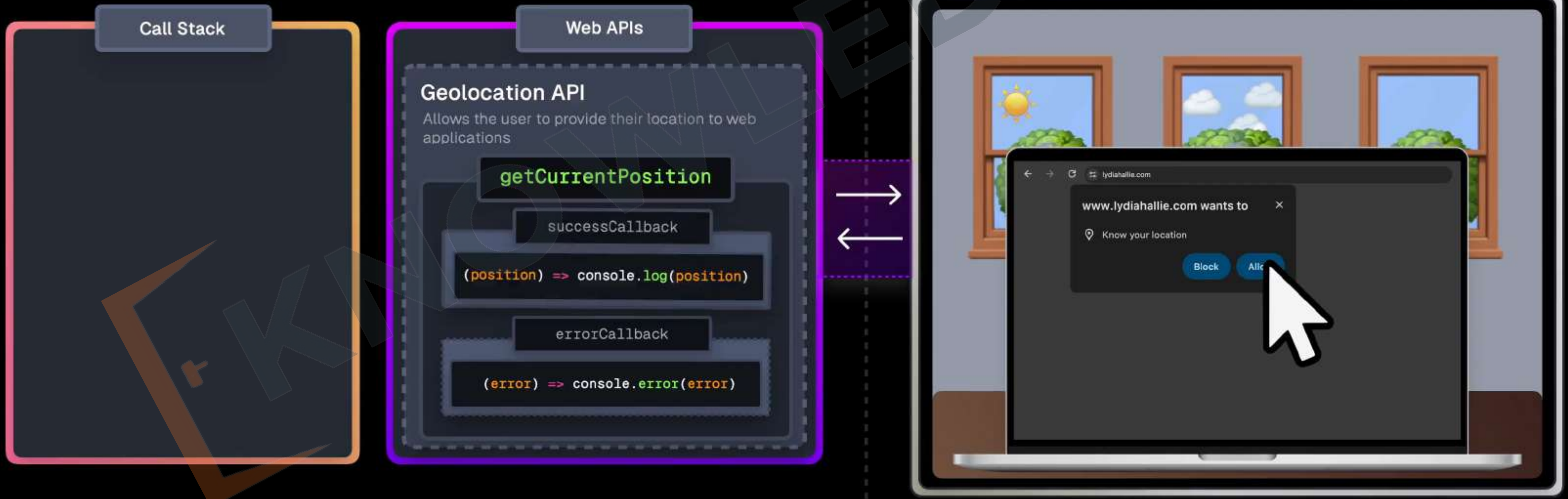
```
navigator.geolocation
  .getCurrentPosition(
    (position) => console.log(position),
    (error) => console.error(error)
  );
```



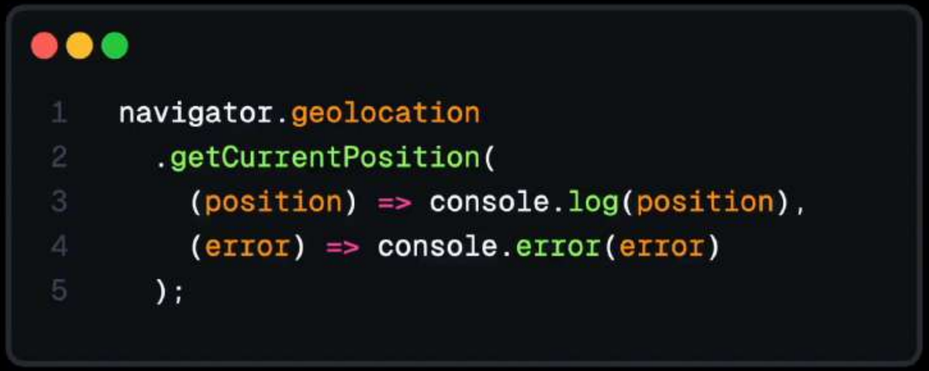
Call Stack
Web APIs
Geolocation API
Allows the user to provide their location to web applications
getCurrentPosition
successCallback
(position) => console.log(position)
errorCallback
(error) => console.error(error)
lydiahallie.com
www.lydiahallie.com wants to
Know your location
Block
Allow

Task Queue

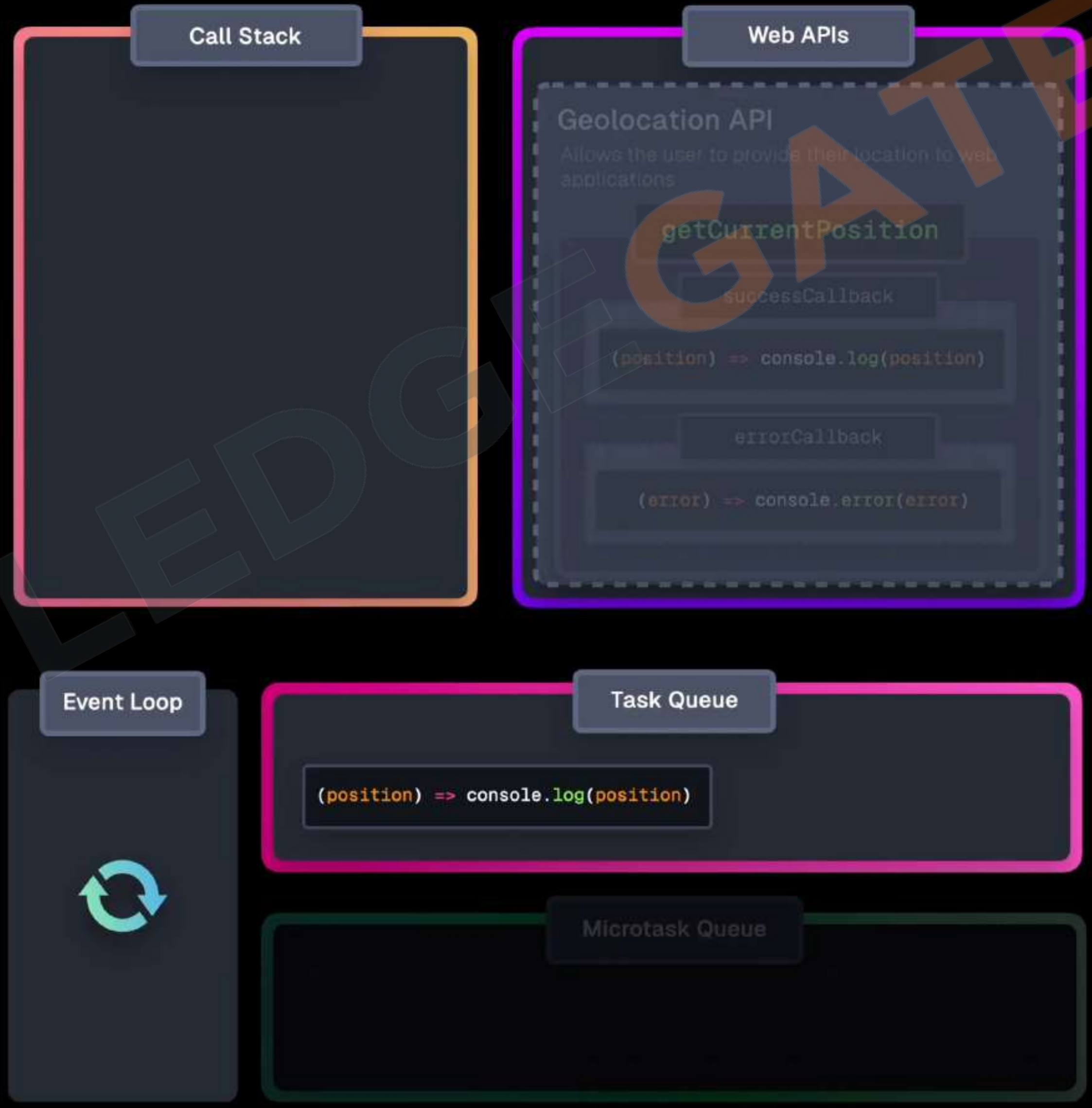
```
navigator.geolocation
  .getCurrentPosition(
    (position) => console.log(position),
    (error) => console.error(error)
  );
```


Call Stack
Web APIs
Geolocation API
Allows the user to provide their location to web applications
getCurrentPosition
successCallback
(position) => console.log(position)
errorCallback
(error) => console.error(error)
lydiahallie.com
www.lydiahallie.com wants to
Know your location
Block
Allo

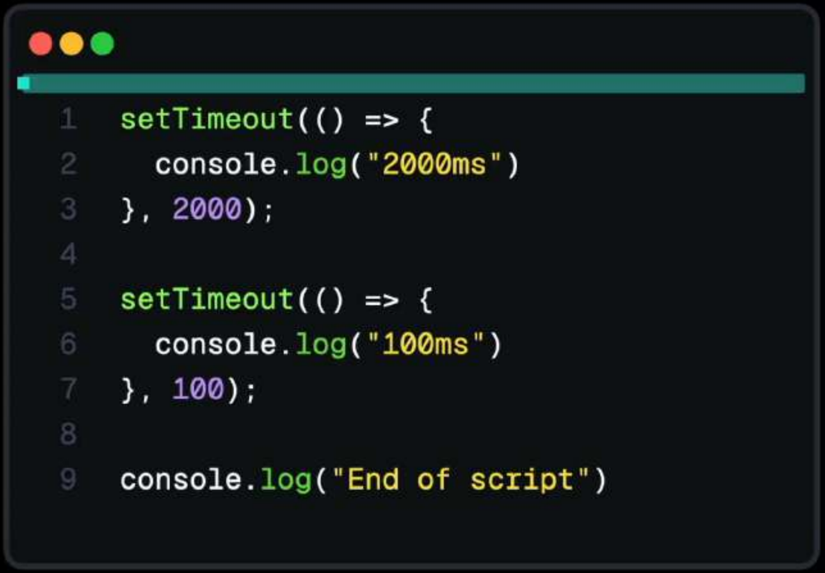
Task Queue & Event Loop



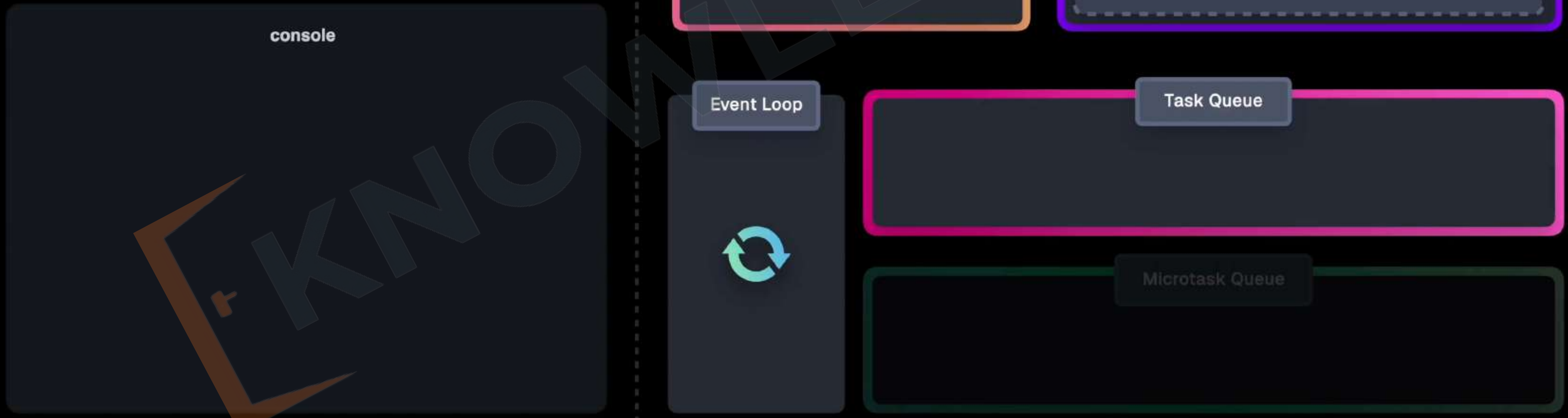
```
navigator.geolocation
  .getCurrentPosition(
    (position) => console.log(position),
    (error) => console.error(error)
  );
```


console
Call Stack
Web APIs
Geolocation API
Allows the user to provide their location to web applications
getCurrentPosition
successCallback
(position) => console.log(position)
errorCallback
(error) => console.error(error)
Event Loop
Task Queue
(position) => console.log(position)
Microtask Queue

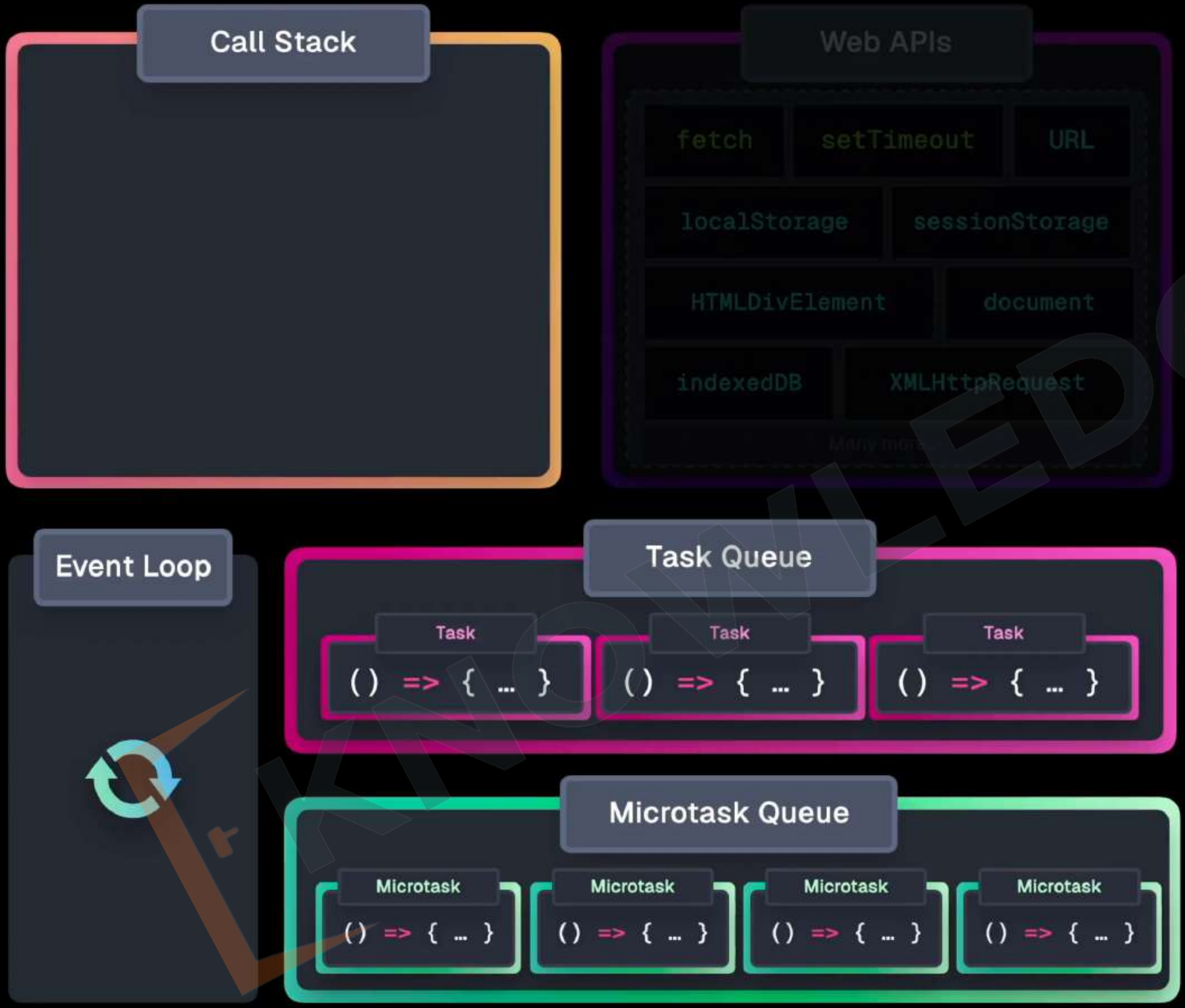
setTimeout (Callback based API)



```
setTimeout(() => {
  console.log("2000ms")
}, 2000);

setTimeout(() => {
  console.log("100ms")
}, 100);

console.log("End of script")
```



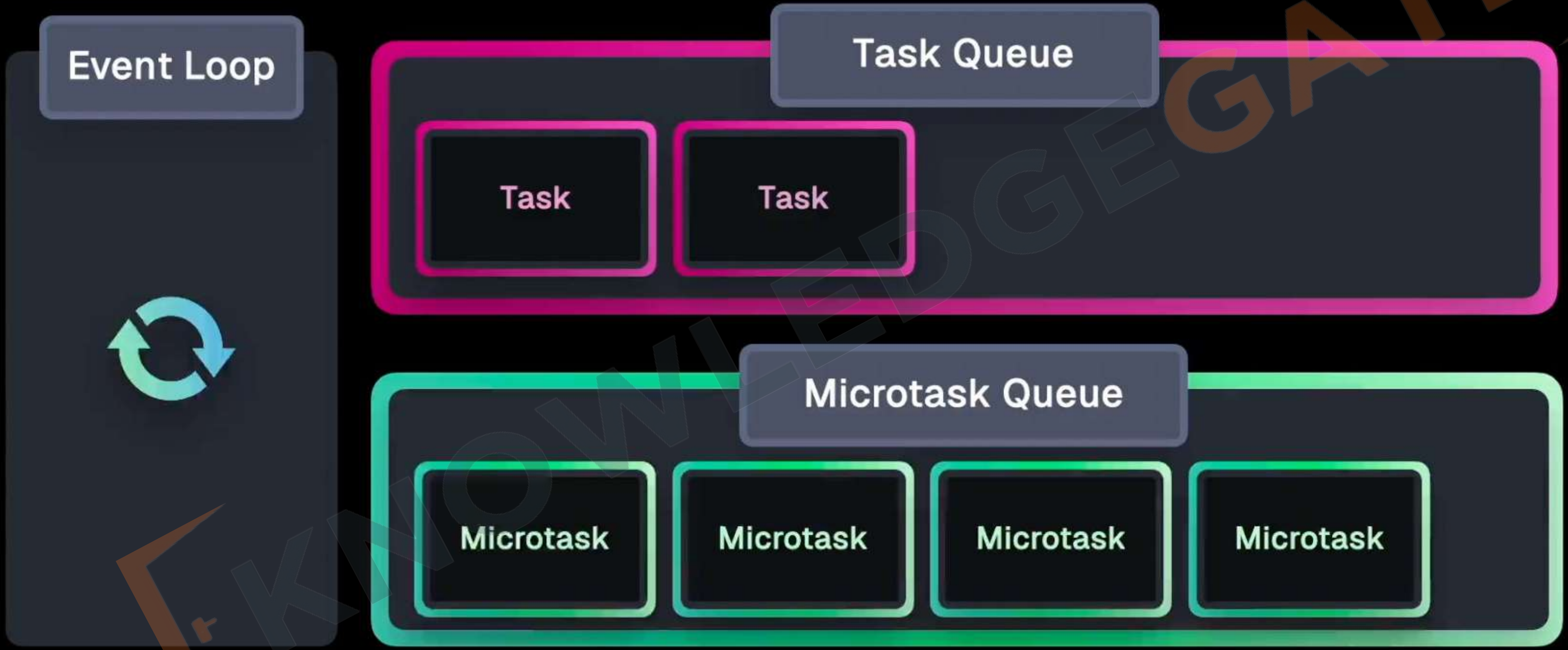
Microtask Queue



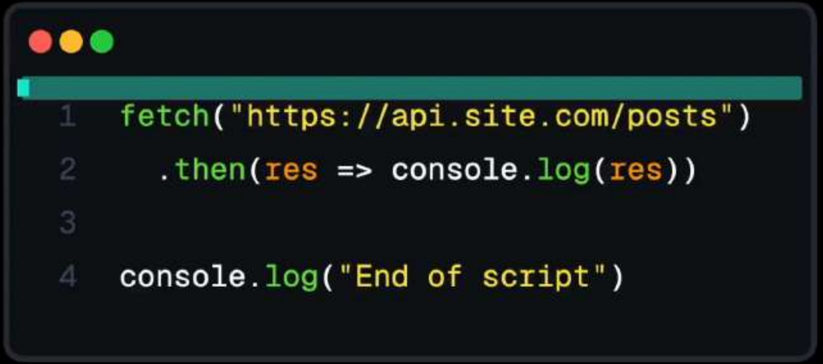
The Microtask Queue is another queue in the runtime with a higher priority than the Task Queue. This queue is specifically dedicated to:

- Promise handler callbacks (then(callback), catch(callback), and finally(callback))
- Execution of async function bodies following await
- MutationObserver callbacks
- queueMicrotask callbacks

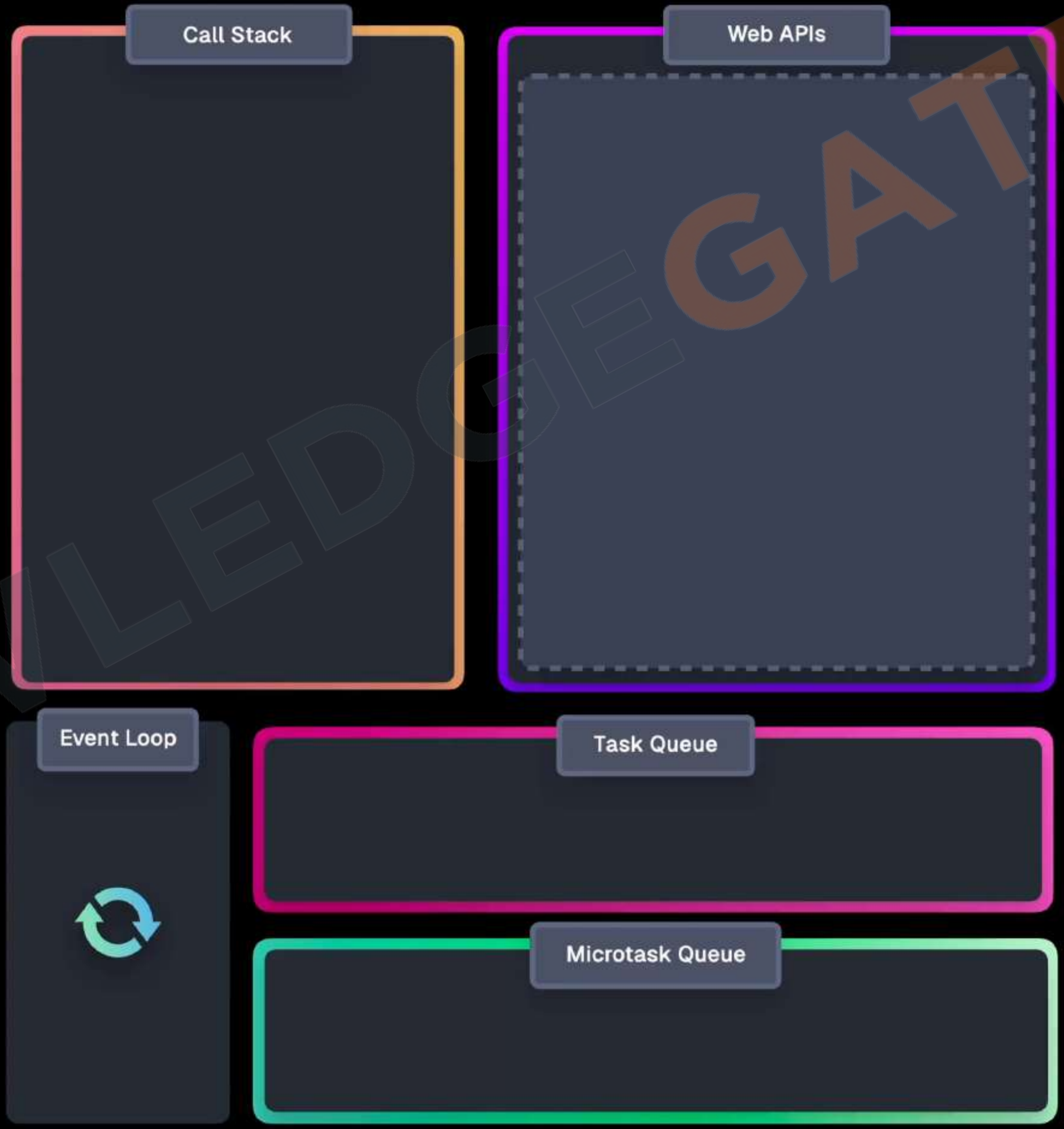
Microtask Queue (Infinite Loop)



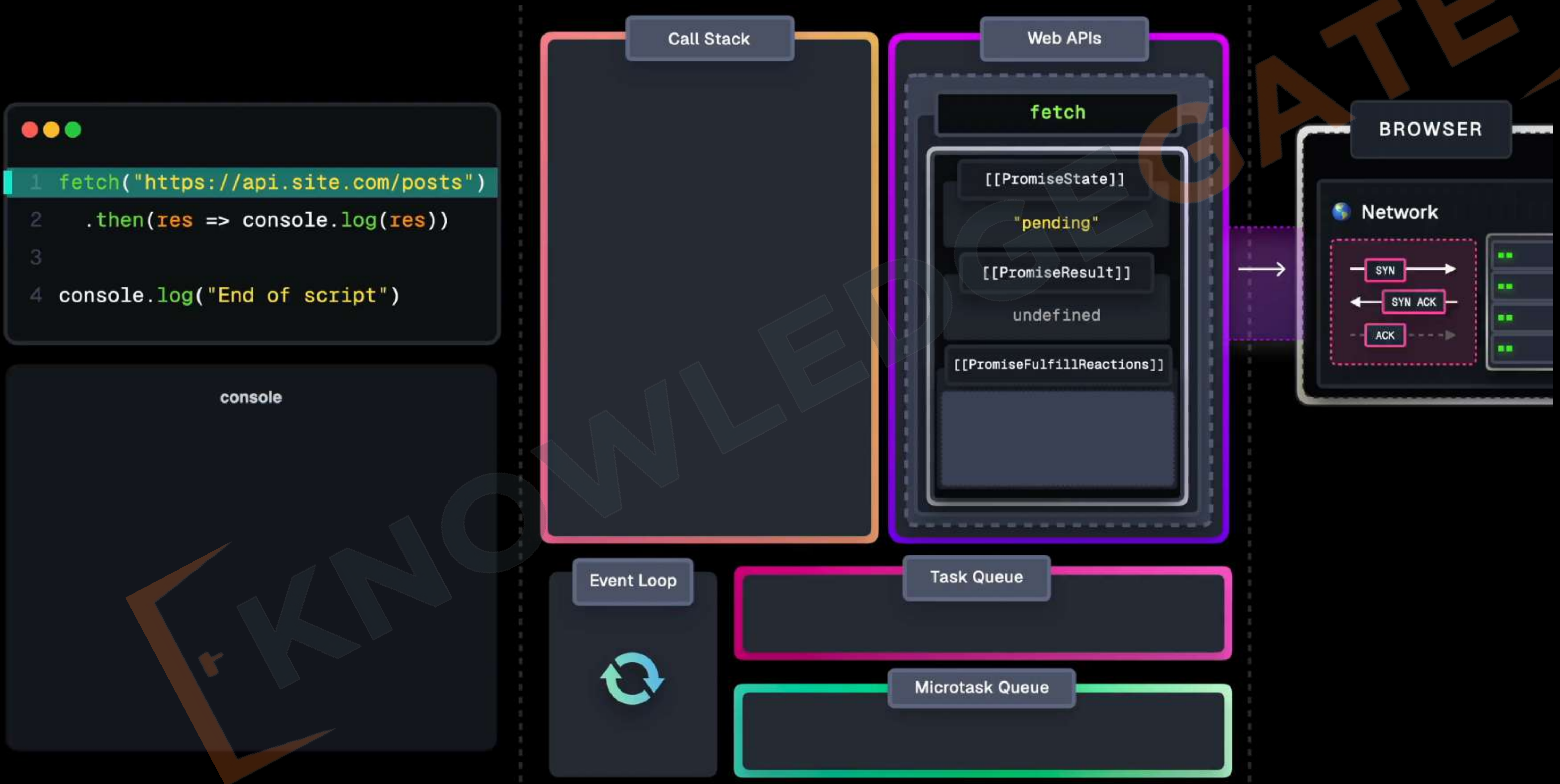
Fetch (Promise based API)



```
fetch("https://api.site.com/posts")
  .then(res => console.log(res))

console.log("End of script")
```

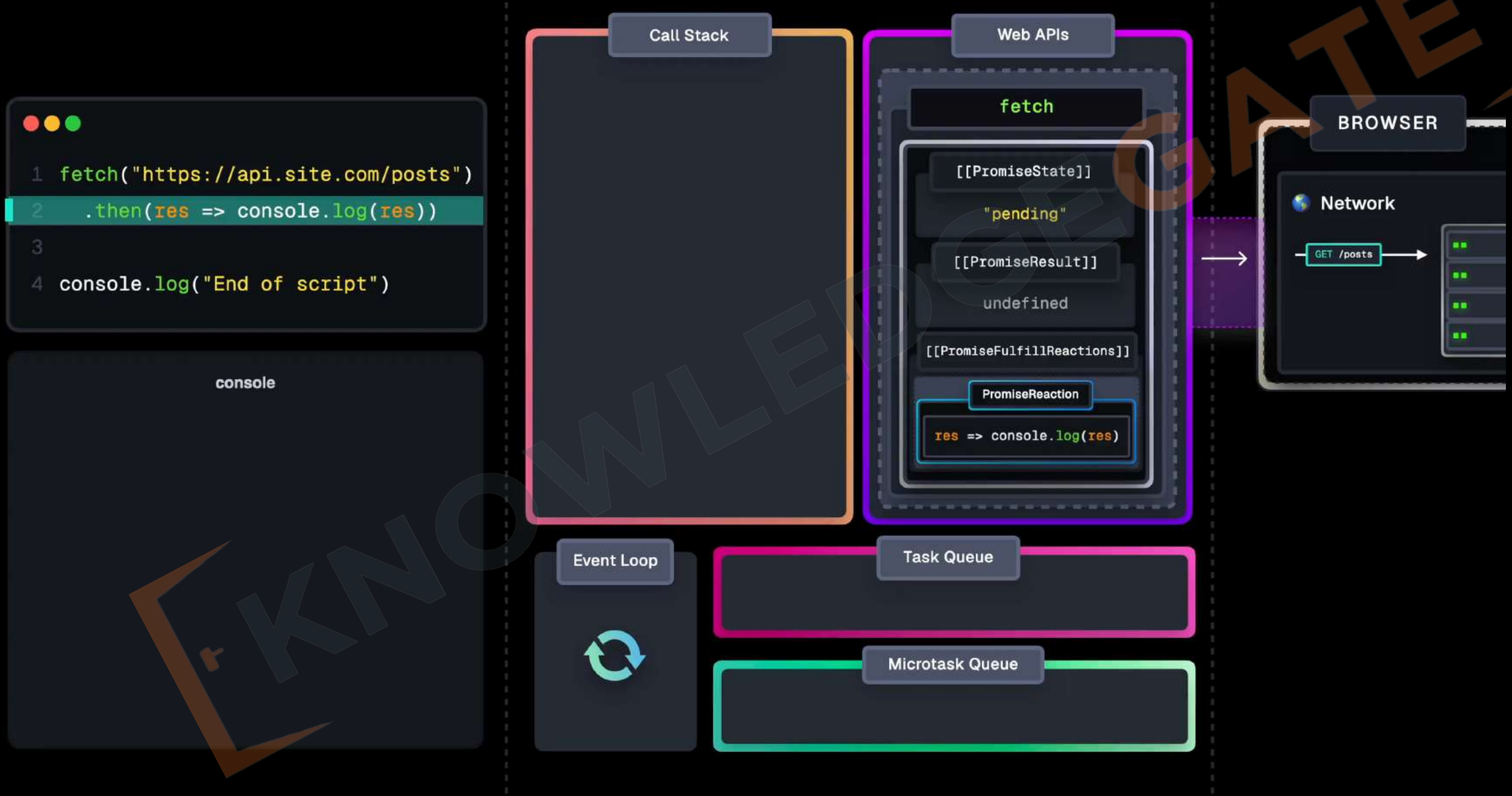


Fetch (Promise Reaction Record)



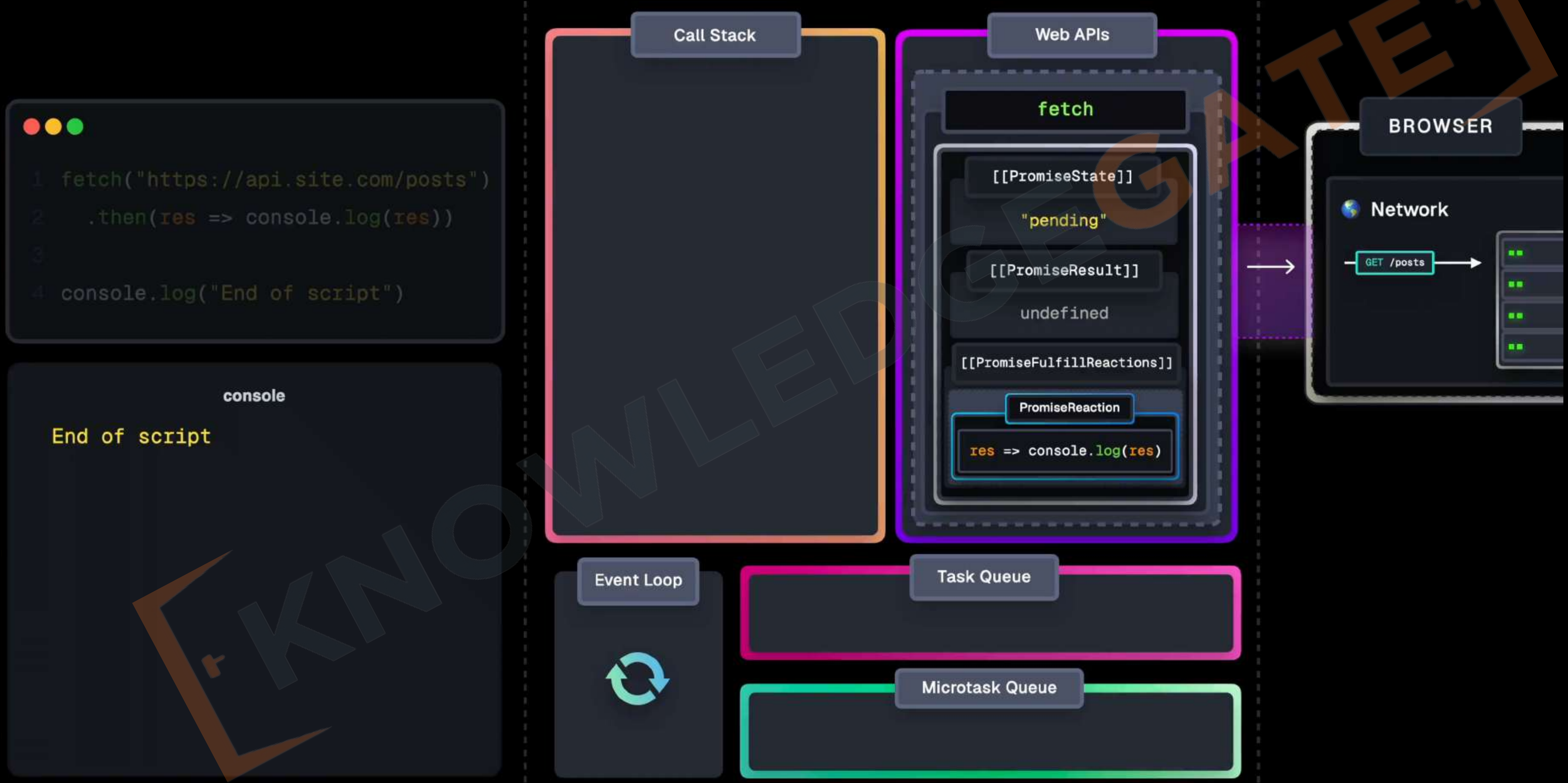
```
fetch("https://api.site.com/posts")
  .then(res => console.log(res))

console.log("End of script")
```

Fetch (Normal Execution Continues)



```
fetch("https://api.site.com/posts")
  .then(res => console.log(res))

console.log("End of script")
```

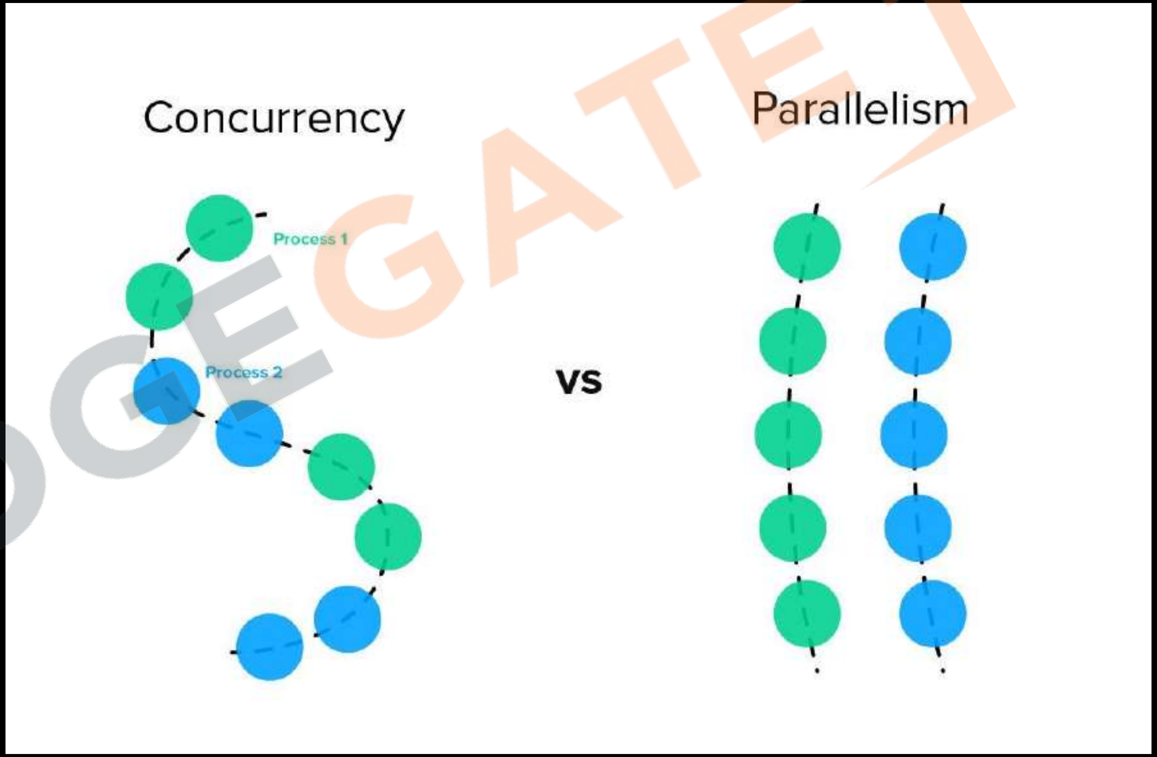
Fetch (Promise Reaction pushed to Microtask queue)



```
fetch("https://api.site.com/posts")
  .then(res => console.log(res))

console.log("End of script")
```

Concurrency

- Concurrency allows multiple operations to overlap in execution, but not necessarily run simultaneously.
- JavaScript can achieve parallelism through web workers, which allow scripts to run in background threads.
- Concurrency in JavaScript is handled through the event loop and asynchronous programming.
- Asynchronous constructs like callbacks, promises, and async/await allow non-blocking operations.
- Web APIs and timers enable asynchronous task execution without blocking the main thread.
- Understanding concurrency is essential for writing efficient, responsive JavaScript applications, especially for handling I/O operations and user interactions.


Concurrency
Process 1
Process 2
vs
Parallelism