Softbinator
TECHNOLOGIES

Programarea aplicatiilor in NodeJS

Curs 4

Recapitulare

Advanced JS - Async Programming

Ce este programarea asincrona?

O tehnica prin care putem executa operații care durează mai mult timp (web requests, file operations, etc.) fără să blocăm threadul principal pe care rulează aplicația.

Cum poate JS să fie asincron dacă este single threaded?

Se folosește de un design pattern interesant, numit **Event Loop**.

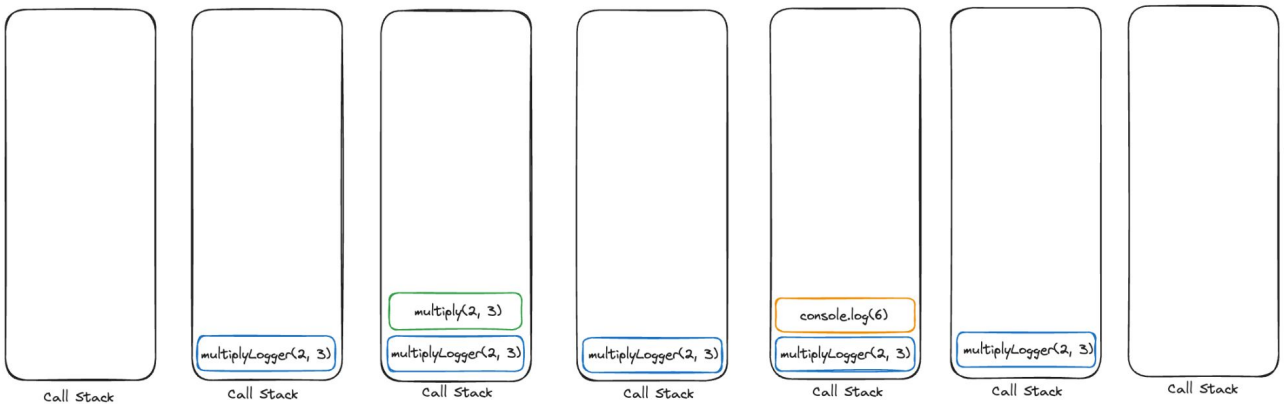
```
console.log("First");

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

console.log("Third");

// Expected output
// "First"
// "Third"
// "Second"
```

Advanced JS - Event Loop

Componente principale:

1. Call Stack

```
const multiply = (x, y) => {
  return x * y;
};

const multiplyLogger = (x, y) => {
  const value = multiply(x, y);
  console.log(value);
};

multiplyLogger(2, 3);
```



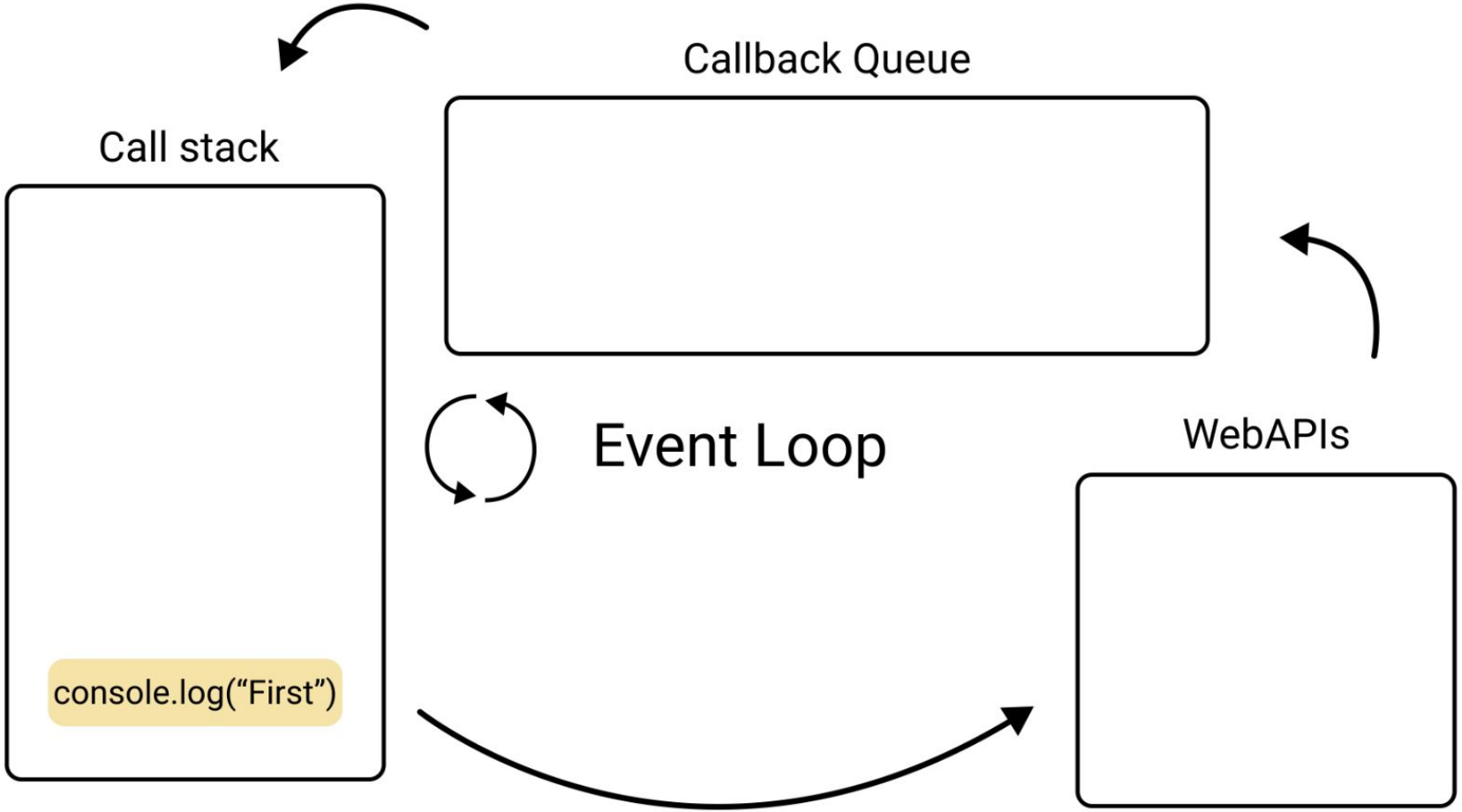
Advanced JS - Event Loop

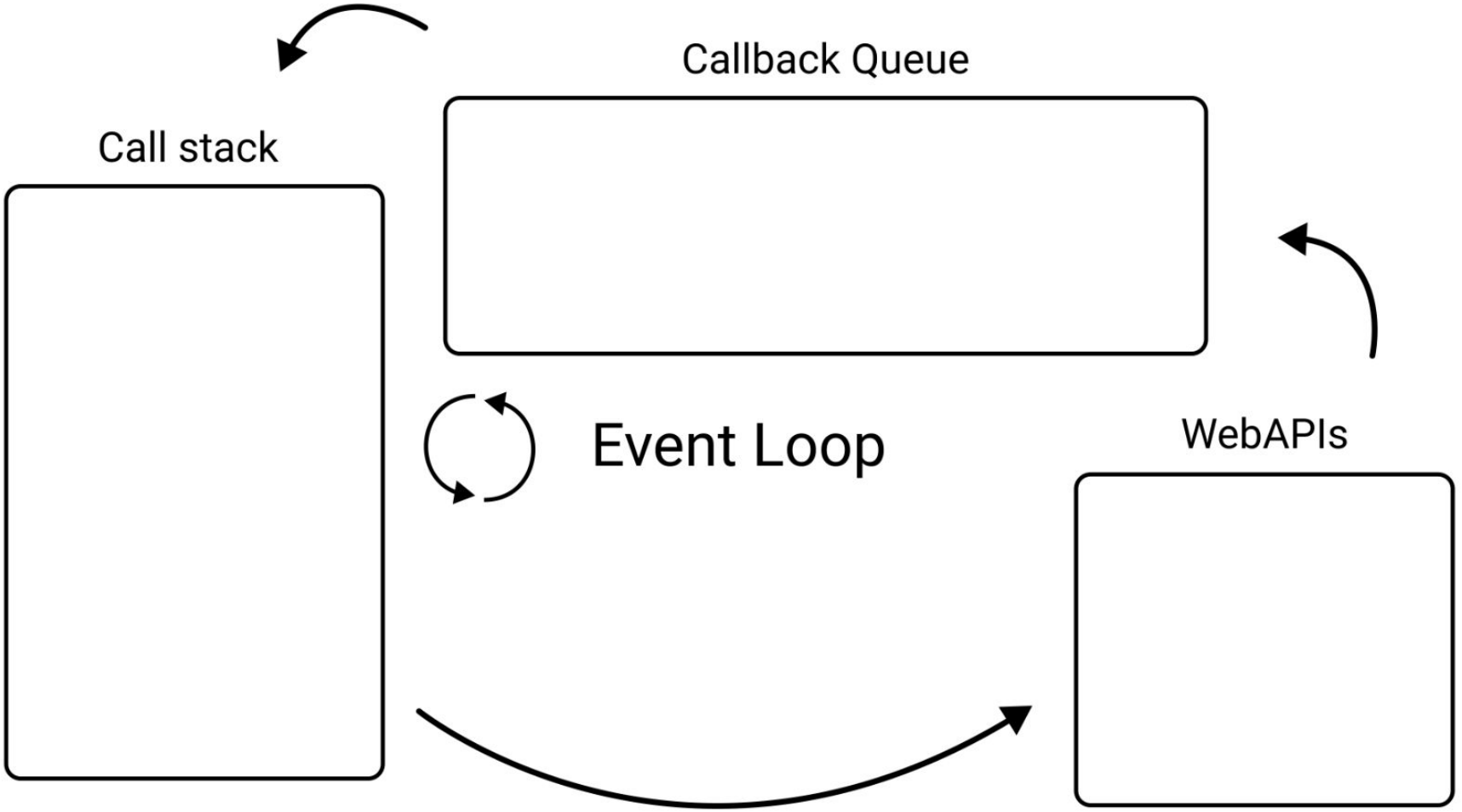
2. Web APIs/Node APIs
   - Daca API-ul este nevoit sa faca ceva asincron, atunci se foloseste de alt thread pentru a se executa
   - Dupa terminarea executiei, rezultatul se trimite intr-un callback queue
   - E.g. setTimeout, setInterval
   - E.g. filesystem (fs) - worker threads (4)
   - E.g. network (http requests)

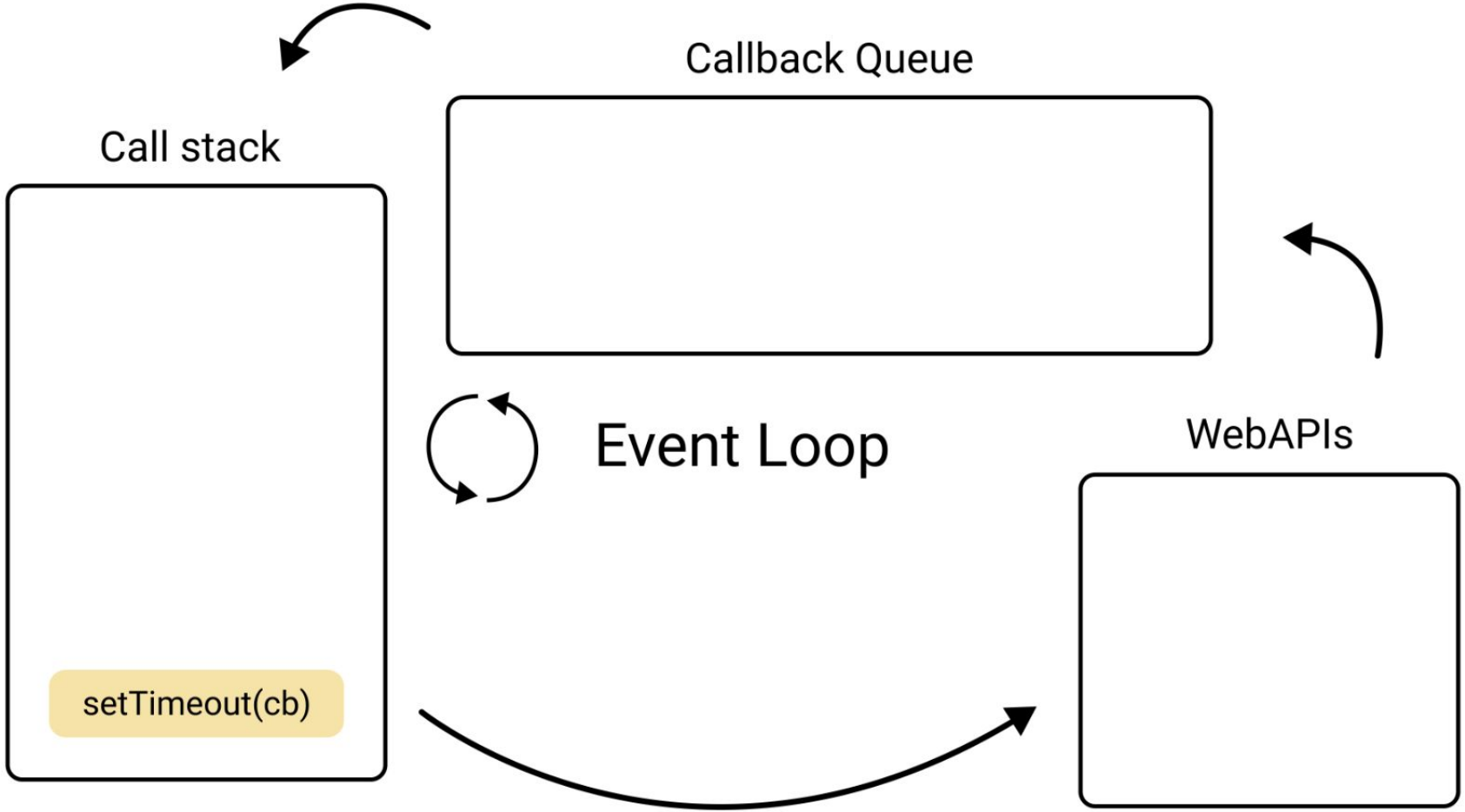
Advanced JS - Event Loop

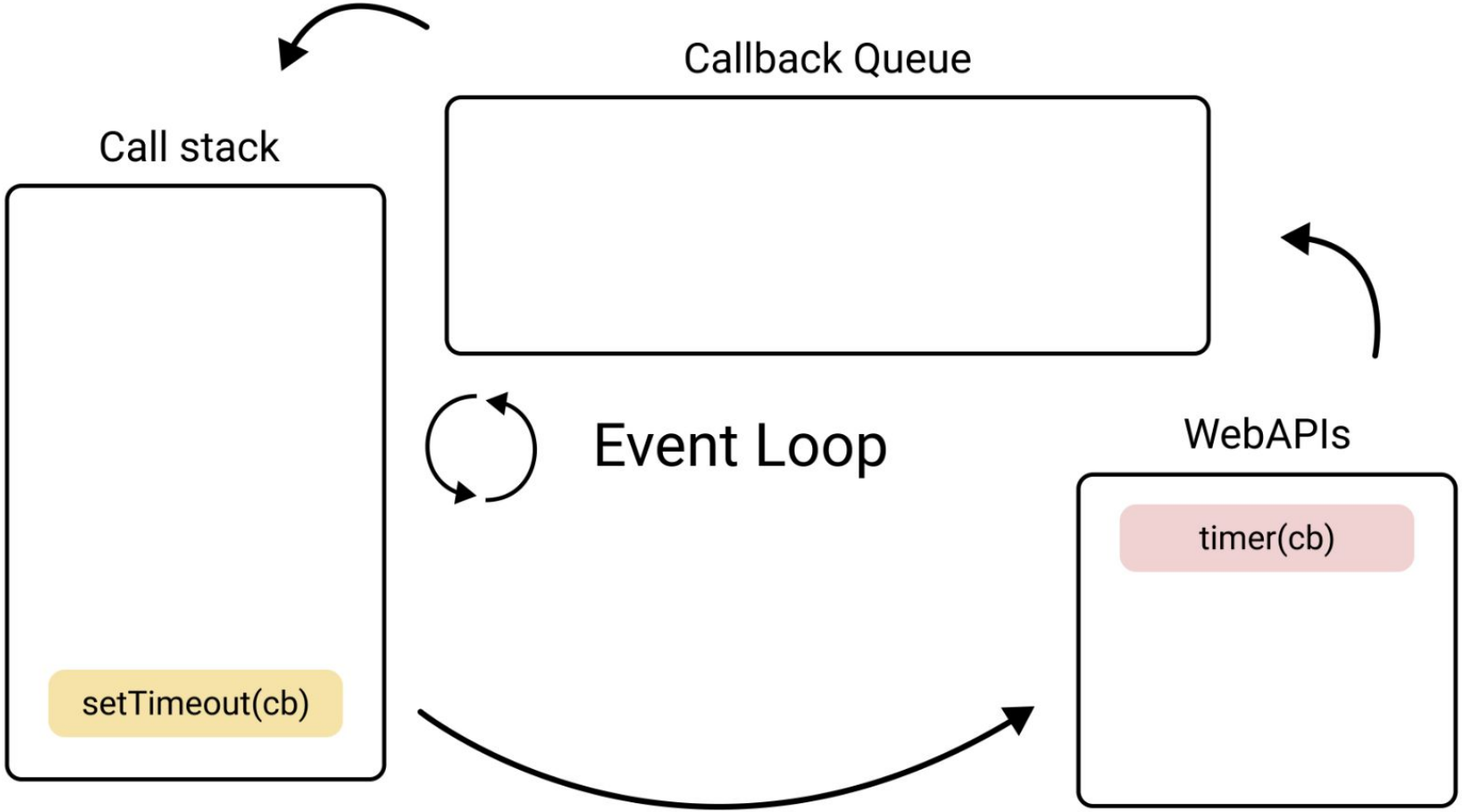
3. Callback queue
   - Dupa ce APIs isi termina “munca”, trimit rezultatul mai departe in callback queue
   - Cand **Call Stack** este gol, se extrag pe rand “rezultatele” din callback queue

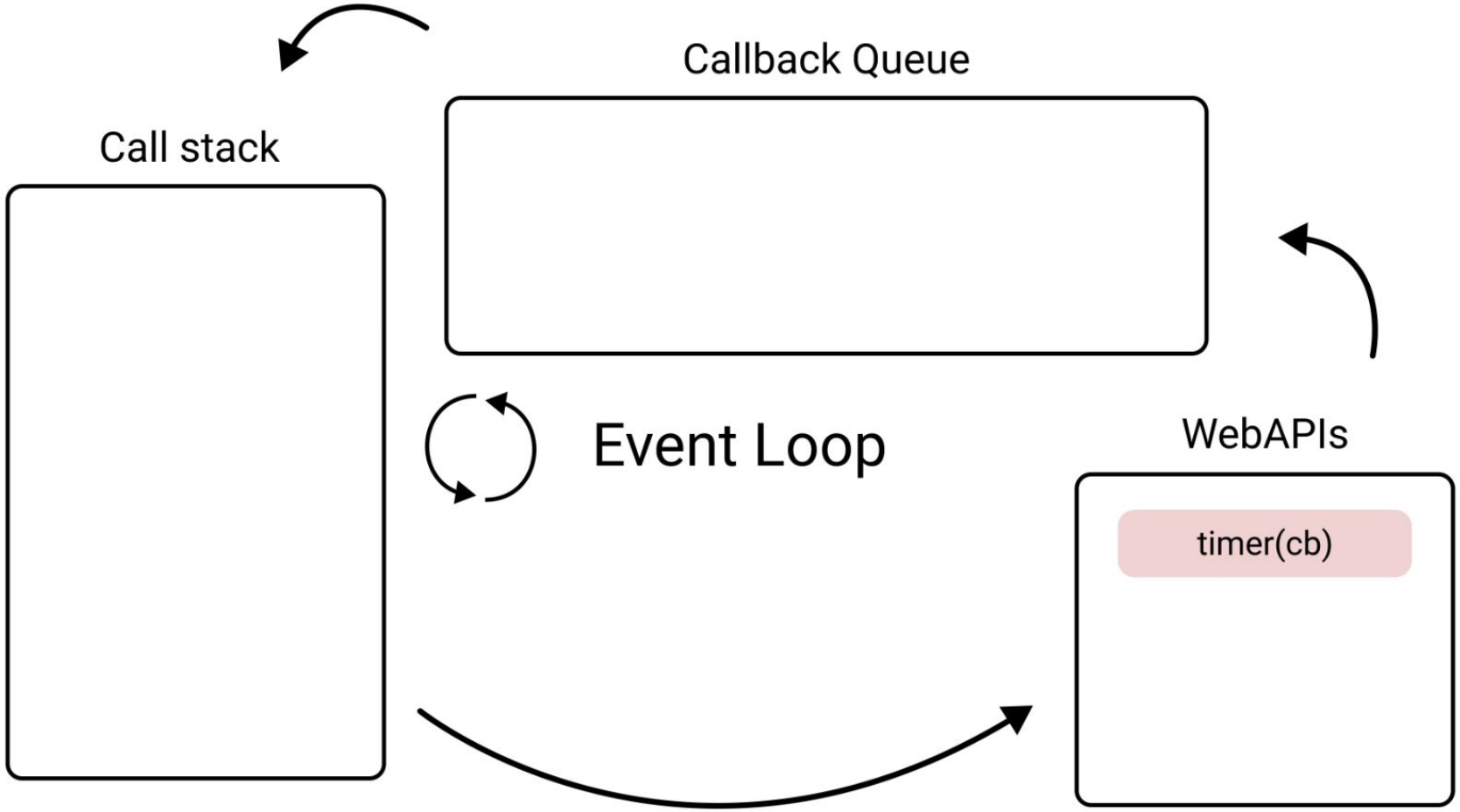
```
console.log("First");

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

console.log("Third");

// Expected output
// "First"
// "Third"
// "Second"
```

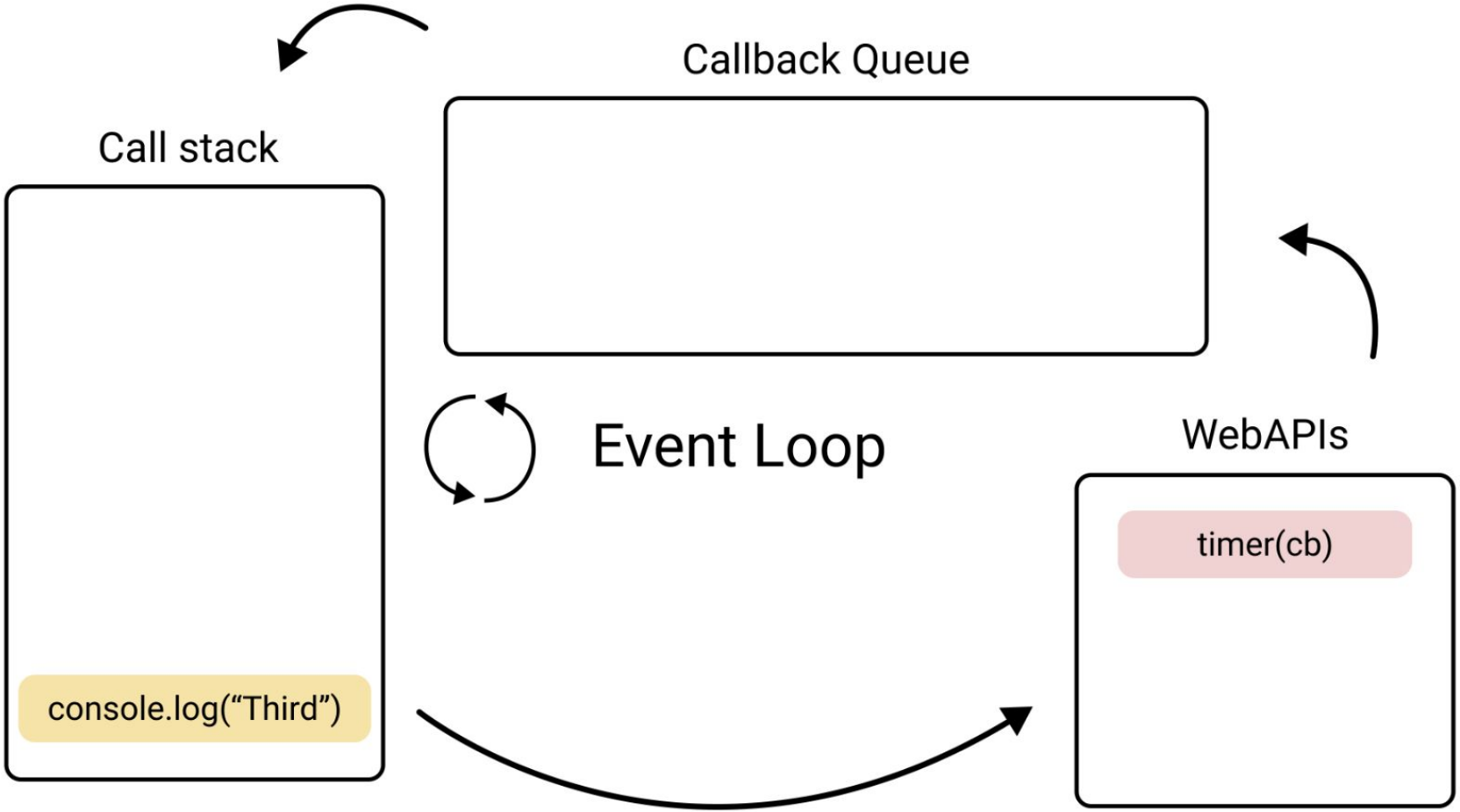

Callback Queue
Call stack
Event Loop
WebAPIs
console.log(“First”)

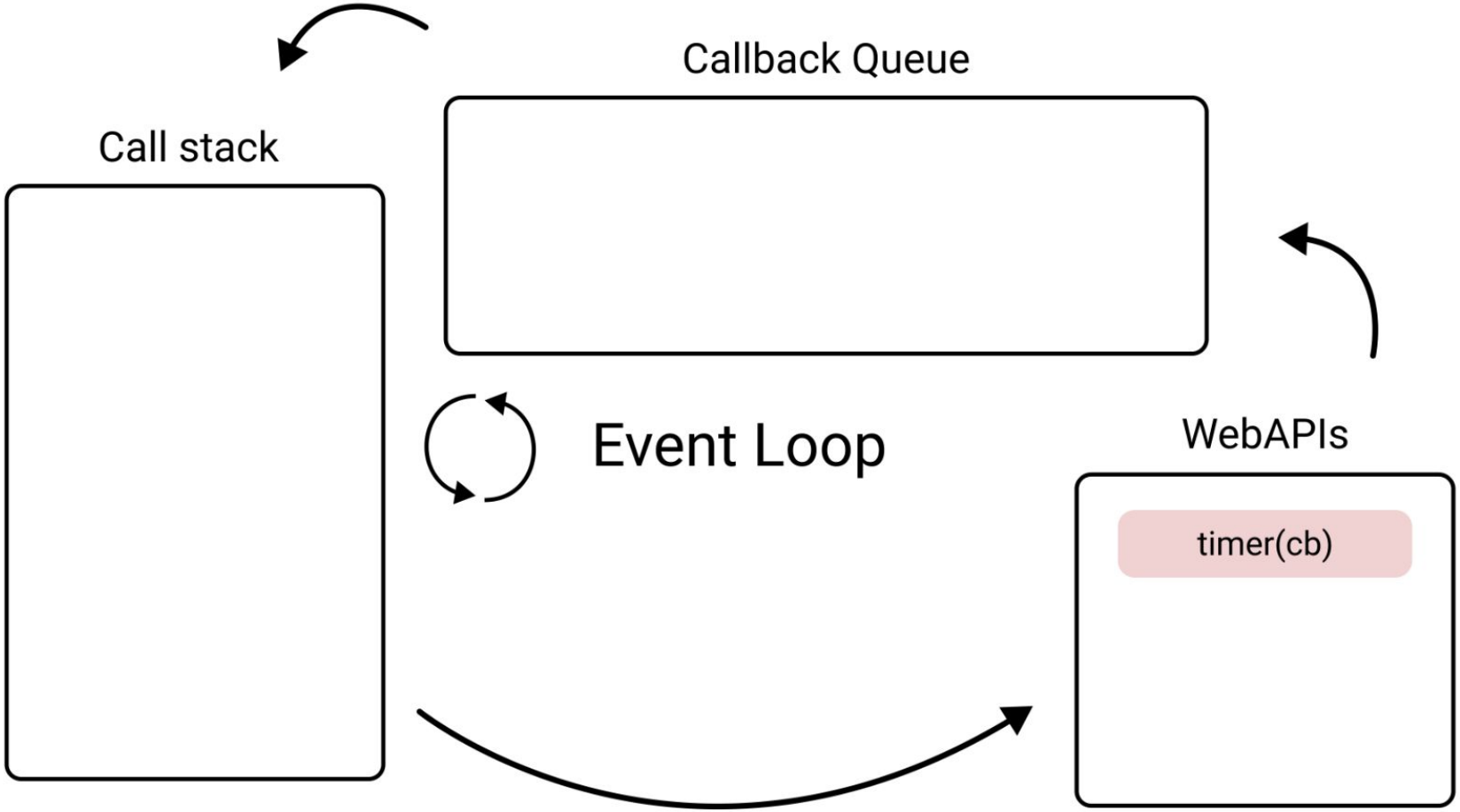

Callback Queue
Call stack
Event Loop
WebAPIs

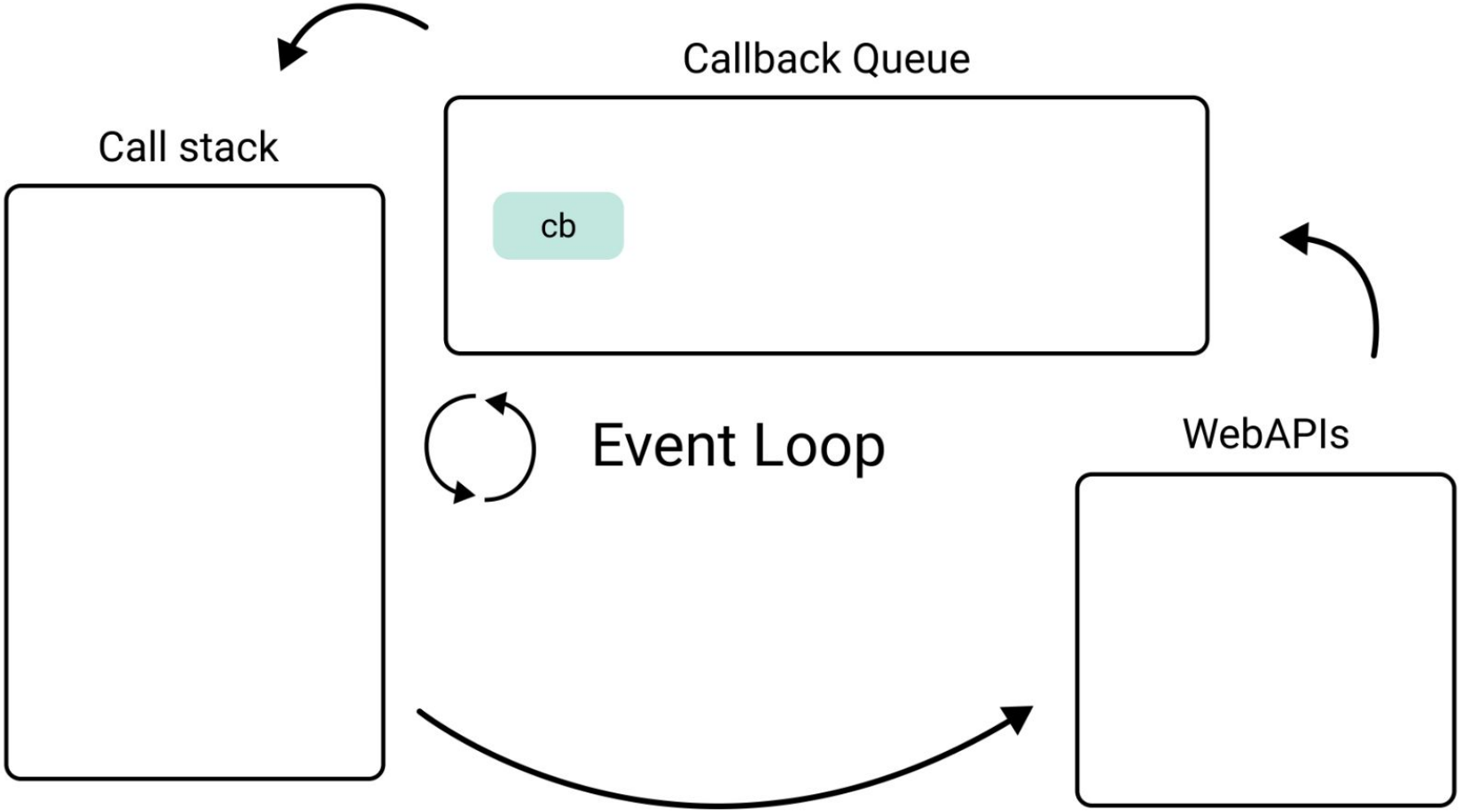

Callback Queue
Call stack
Event Loop
WebAPIs
setTimeout(cb)

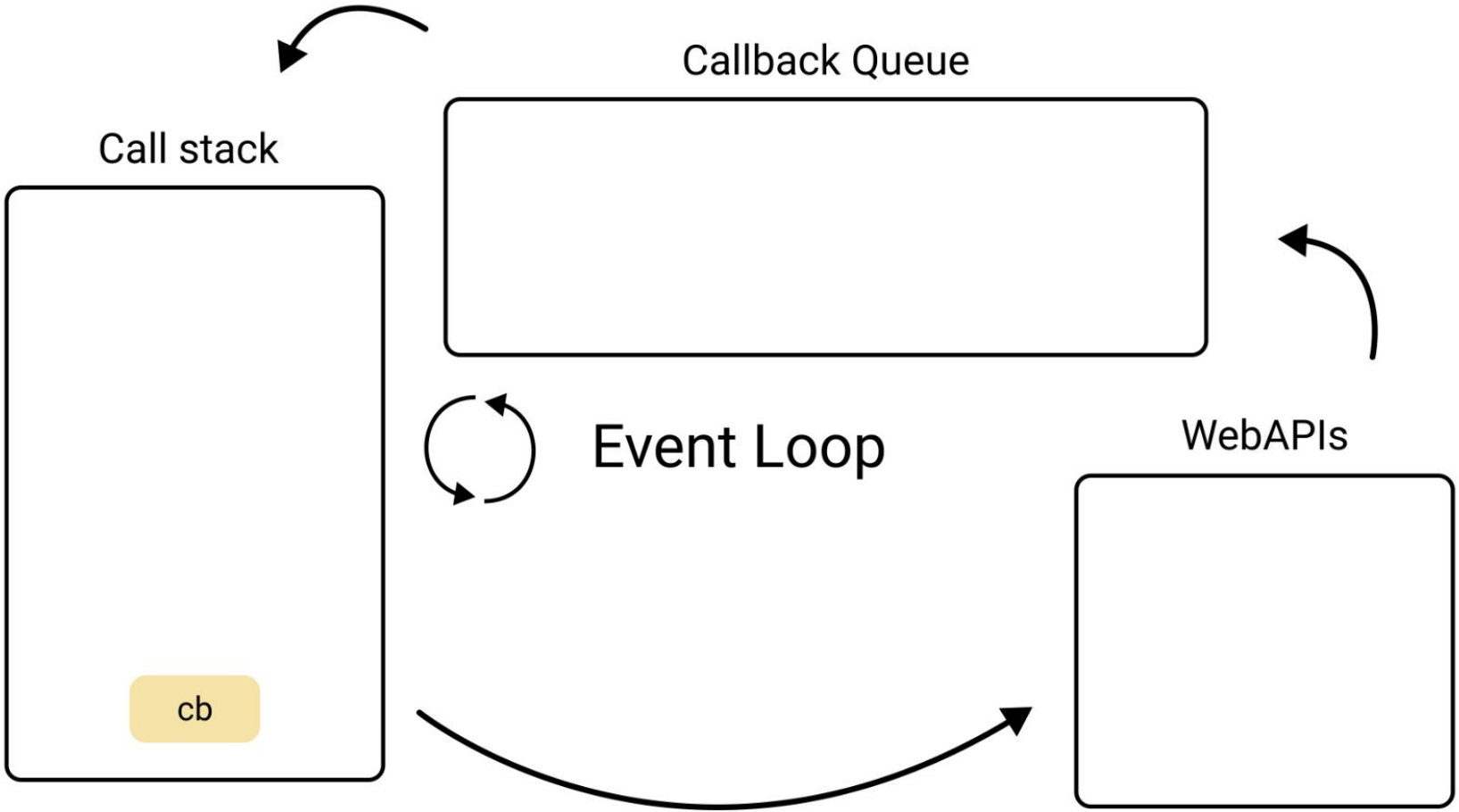

Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)
setTimeout(cb)

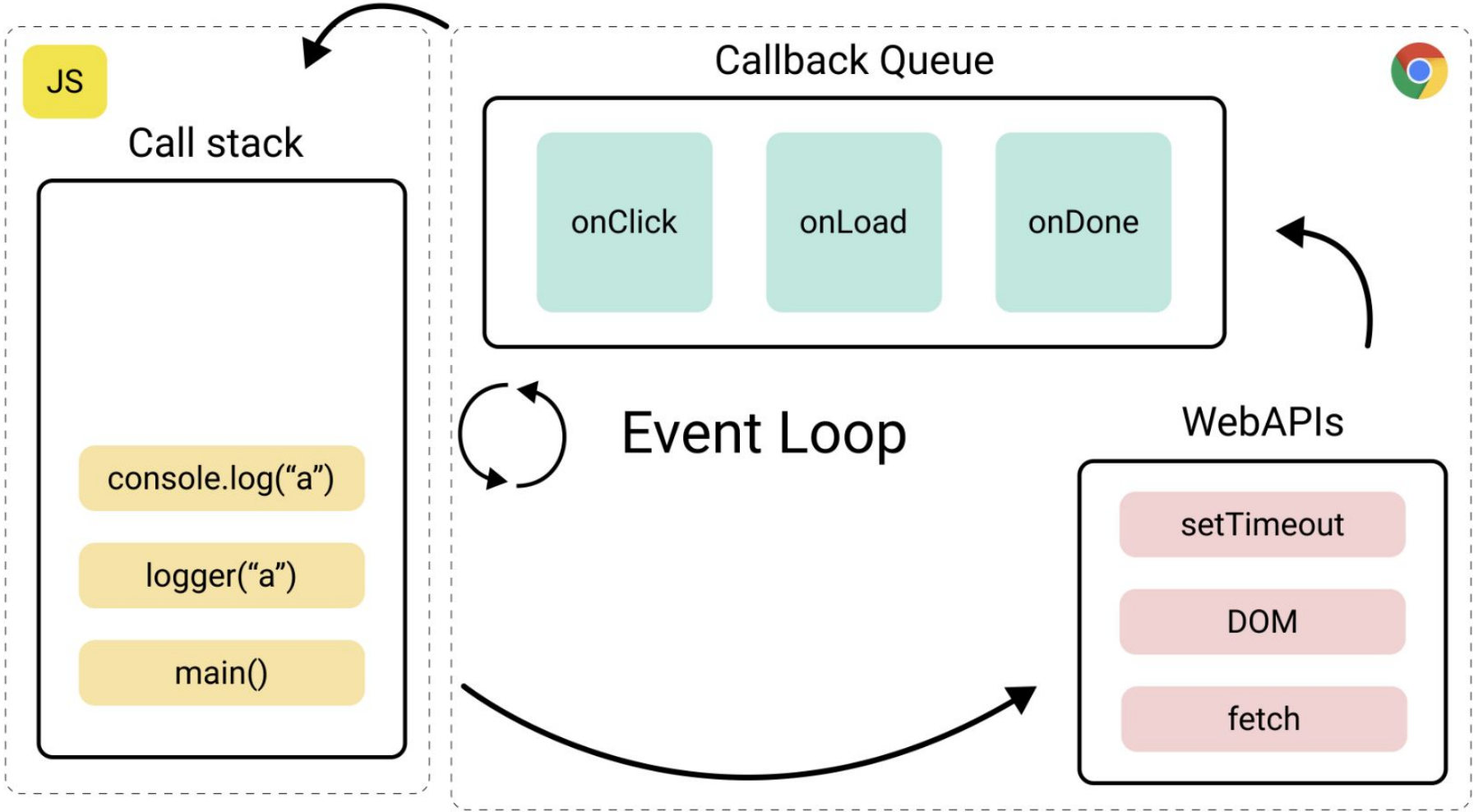

Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)


Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)
console.log(“Third”)

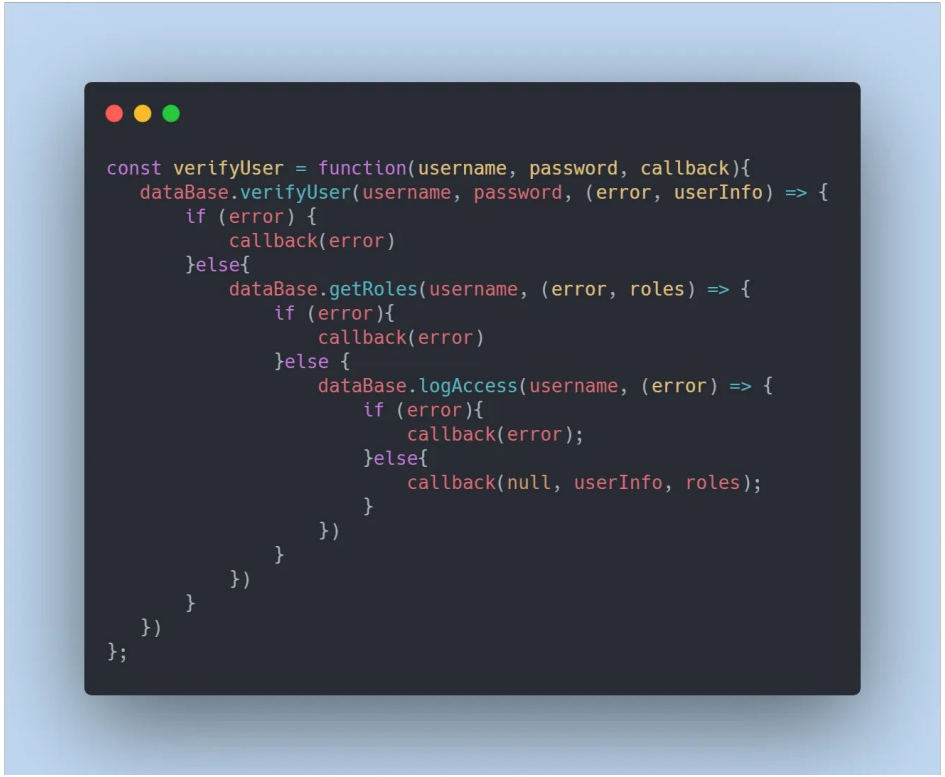

Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)


Callback Queue
Call stack
cb
Event Loop
WebAPIs


Callback Queue
Call stack
Event Loop
WebAPIs
cb


JS
Call stack
console.log("a")
logger("a")
main()
Callback Queue
onClick
onLoad
onDone
Event Loop
WebAPIs
setTimeout
DOM
fetch

Advanced JS - Callback Hell



```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       }else{
           dataBase.getRoles(username, (error, roles) => {
               if (error){
                   callback(error)
               }else {
                   dataBase.logAccess(username, (error) => {
                       if (error){
                           callback(error);
                       }else{
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

Advanced JS - Callback Hell Solution

Pentru a “combate” callback hell, in ES6 au fost adauge **“promises”.**
Un promise reprezinta rezultatul eventual al unei operatii asincrone. Rezultatul unui promise este un obiect, printre care contine si starea promisiunii: pending, fulfilled sau rejected.

Pending - starea inițială a promisiunii, indica faptul ca operația încă rulează
Fulfilled - stare finală a promisiunii dacă operația s-a executat cu success, conține și rezultatul final
Rejected - stare finală a promisiunii dacă operația a dat greș, conține date despre eroare

Promises simplifica foarte mult programarea asincrona in JS, și este mult mai ușor sa faci error management (prin try catch)

Advanced JS - Callback Hell Solution

Pentru a “combate” callback hell, in ES6 au fost adauge **“promises”.**
Un promise reprezinta rezultatul eventual al unei operatii asincrone. Rezultatul unui promise este un obiect, printre care contine si starea promisiunii: pending, fulfilled sau rejected.

Pending - starea inițială a promisiunii, indica faptul ca operația încă rulează
Fulfilled - stare finală a promisiunii dacă operația s-a executat cu success, conține și rezultatul final
Rejected - stare finală a promisiunii dacă operația a dat greș, conține date despre eroare

Promises simplifica foarte mult programarea asincrona in JS, și este mult mai ușor sa faci error management (prin try catch)

Advanced JS - Promises

Pentru a crea un promise, ne folosim de keyword-ul “new”, urmat de “Promise”.
Functia care se executa in interiorul unui promise are 2 argumente: resolve si reject

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation code here
  if (/* operation successful */) {
    resolve(value); // Fulfills the promise with value
  } else {
    reject(error); // Rejects the promise with error
  }
});
```

Advanced JS - Promises

Cum “consumam” rezultatul unui promise? Un promise ne pune la dispozitie trei variante: **then, catch** si **finally.**

- Then se executa cand promisiunea devine fulfilled
- Catch se executa cand promisiunea devine rejected
- Finally se executa cand promisiunea devine fulfilled sau rejected

```
myPromise
  .then(result => {
    console.log('Success:', result);
  })
  .catch(error => {
    console.error('Something went wrong:', error);
  })
  .finally(() => {
    console.log('This runs regardless of outcome.');
  });
```

Advanced JS - Promises

Promises pot fi inlantuite cu mai multe then-uri la rand

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .catch(error => console.error('Failed at some point:', error));
```

Advanced JS - Async/await

Pentru a face lucrul cu promises mai simplu, in ES2017 a fost adaugat “async” si “await”, 2 keyword-uri noi.
În sine ele funcționează destul de simplu, se adaugă “await” înaintea unei promisiuni pentru a aștepta rezultatul. Daca promisiunea da fail, atunci este “aruncata” o eroare. In cazul acesta, trebuie sa folosim sintaxa try/catch pentru a face error handling.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function add1(x) {
  const a = await resolveAfter2Seconds(20);
  const b = await resolveAfter2Seconds(30);
  return x + a + b;
}

add1(10).then(v => {
  console.log(v);  // prints 60 after 4 seconds.
});
```

```
async function getWithErrorHandling() {
  try {
    const result = await Promise.reject(new Error("Failed to fetch"));
    console.log(result); // This line will not be executed
  } catch (error) {
    console.error(error.message); // "Failed to fetch"
  }
}

getWithErrorHandling();
```

Node.js

- Node.JS este un Javascript Runtime Environment care foloseste V8 Chrome Engine
- A fost dezvoltat de catre Ryan Dahl, si lansat in 2009
- Cativa ani mai tarziu, Ryan Dahl a tinut o prezentare la JSConf EU intitulata “10 lucruri pe care le regret la Node.js”, si anunta un nou JRE la care lucreaza, numit Deno.
- Primeste updates destul de des - semantic versioning
- Are o comunitate foarte mare in spate, dispune de foarte multe librarii si framework-uri care sporesc foarte mult capabilitatile Node.js si nu numai

De ce Node.js?

- Javascript este un limbaj “easy to learn, hard to master”
- A fost gandit pentru a construi aplicatii scalabile, lucru datorat arhitecturii bazate pe eventuri.
- Are performante destul de bune, deoarece la baza foloseste V8. O parte din API-uri sunt scrie in C si C++, lucru care il face foarte rapid in anumite taskuri.
- Este foarte versatil, poate sa ruleze pe foarte multe device-uri

Node js frameworks

Fullstack:

- MEAN/MERN stack - un stack de tehnologii foarte folosit acum cativa ani, vine de la urmatoarele tehnologii (MongoDB, Express.js, Angular/React si Node.js), folosit pentru aplicatii full stack

Backend:

- Express.js - un framework foarte simplu, minimalist si flexibil. Este foarte popular, si sta in spatele multor altor framework-uri. Nu a mai primit update-uri majore in ultimii ani deoarece isi face treaba cum trebuie
- Koa - creat de aceeasi echipa din spatele Express.js, Koa este mai expresiv si mai robust, e recomandat pentru aplicatii unde performanta si scalabilitatea sunt prioritare

Node js frameworks

- NestJS - similar cu Angular, structura si arhitectura aplicatiei nu este la fel de flexibila, dar nu face niciun compromis la partea de performante

Frontend:

- Next.js & Nuxt.js - aceste frontend frameworks pot fi integrate cu Node.js pentru a crea server-side rendered apps.

Cum alegi un framework?

In functie de marimea si complexitatea proiectului, pentru proiecte mai simple cel mai recomandat ar fi Koa sau Express. Pentru “development speed”, atunci MERN/MEAN stack ar fi recomandat. Pentru o comunitate mai mare (care ofera mai multe resurse, plugins, extensions, etc.) atunci Express.js sau Next.js

Express.js

- Semestrul acesta vom folosi Express.js pentru toate motivele mentionate anterior
- Este foarte simplu, si poate fi considerat ca o mica abstractizare peste capabilitatile HTTP ale Node.js

Express.js

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

    // 1. Tell the browser everything is OK (Status code 200), and the c
    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });

    // 2. Write the announced text to the body of the page
    response.write('Hello, World!\n');

    // 3. Tell the server that all of the response headers and body have
    response.end();

}).listen(1337); // 4. Tells the server what port to be on
```

```
const express = require('express' 4.18.2 )
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Express.js

- Express are doua functionalitati foarte importante:
  - Routing (https://expressjs.com/en/guide/routing.html)
  - Middlewares (https://expressjs.com/en/guide/using-middleware.html)

NPM

- NPM (node package manager) este un manager de “librarii” pentru ecosistemul node, si nu numai. NPM are 2 componente principale:
  - Un CLI
  - O baza de date online numita npm registry
- Faciliteaza sharing code pentru orice developer de pe tot globul
- E cel mai mare software registry din lume
- Key features:
  - Dependency management
  - Scripts
  - Version control and distribution

NPM

- Cum instalezi o librarie/package?
  - Mai intai trebuie sa rulam comanda ‘npm init -y’ care va crea cateva fisiere importante in folderul in care se ruleaza comanda
  - Npm install <package-name>
- Package.json
  - Fisierul principal care are informatii despre fiecare dependinta, scripts, si multe alte lucruri despre proiect
- https://socket.dev/blog/2023-npm-retrospective