Softbinator
TECHNOLOGIES

Programarea aplicatiilor in NodeJS

Curs 3

Cerinte proiect

Recapitulare

- JS a aparut in 1995, creatorul este Brendan Eich.
- A fost creat in 10 zile, cu intenția de a fi folosit de către ceilalți programatori împreuna cu Java (care era foarte popular la vremea respectivă)
- A evoluat foarte mult și foarte diferit fata de Java, incat doar numele le mai leaga in vreun fel
- 1997 - ECMAScript, un standard care asigura interoperabilitatea aplicațiilor web pe orice browser
- 2009 - a aparut Node.js - JS se poate folosi fullstack
- Ecosistem mare de librarii si frameworks, atat pentru frontend cat și pentru backend

Recapitulare

- Javascript Engine - un program care executa cod JS (e.g. V8, SpiderMonkey si JavaScriptCore)
- ECMAScript a evoluat foarte mult în ultimii ani, in special in 2015 cand au adaugat foarte multe features care au îmbunătățit developer experience
- Javascript Runtime Environment:
  - Javascript Engine (parseaza, compileaza si executa codul)
  - APIs
  - Event loop

Recapitulare

- Tipuri de date: number, string, bigint, boolean, null, undefined, symbol, object (array este un object)
- Variabile: var (nu se mai foloseste), let si const
- If, else if, else, switch
- For, while, for x in object (itereaza peste keys), for y of array (itereaza peste valori)
- Functii:

```
function greet() {
  console.log("Hello World!");
}
```

```
const greet = function() {
  console.log("Hello World!");
};
```

```
const add = (a, b) => {
  return a + b;
};
```

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**.

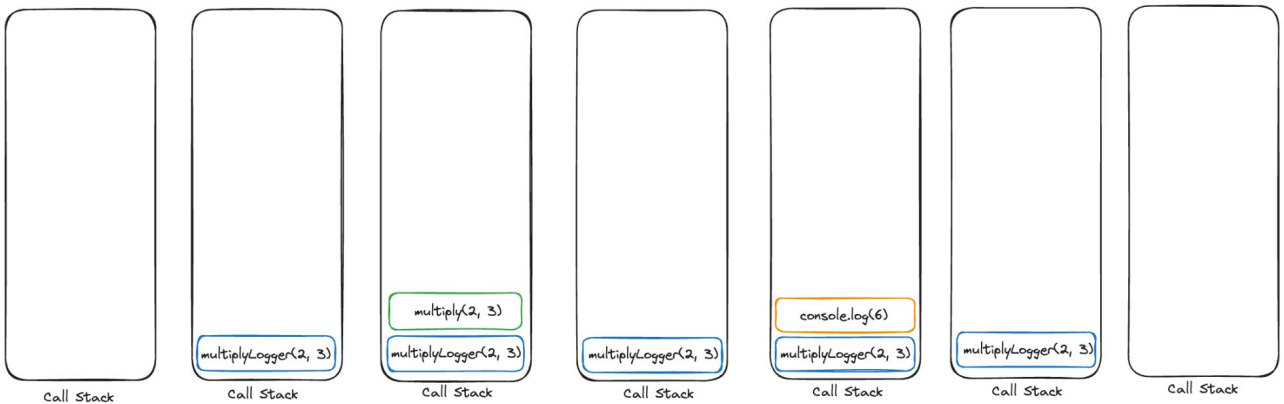
```
console.log("First");

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);
```



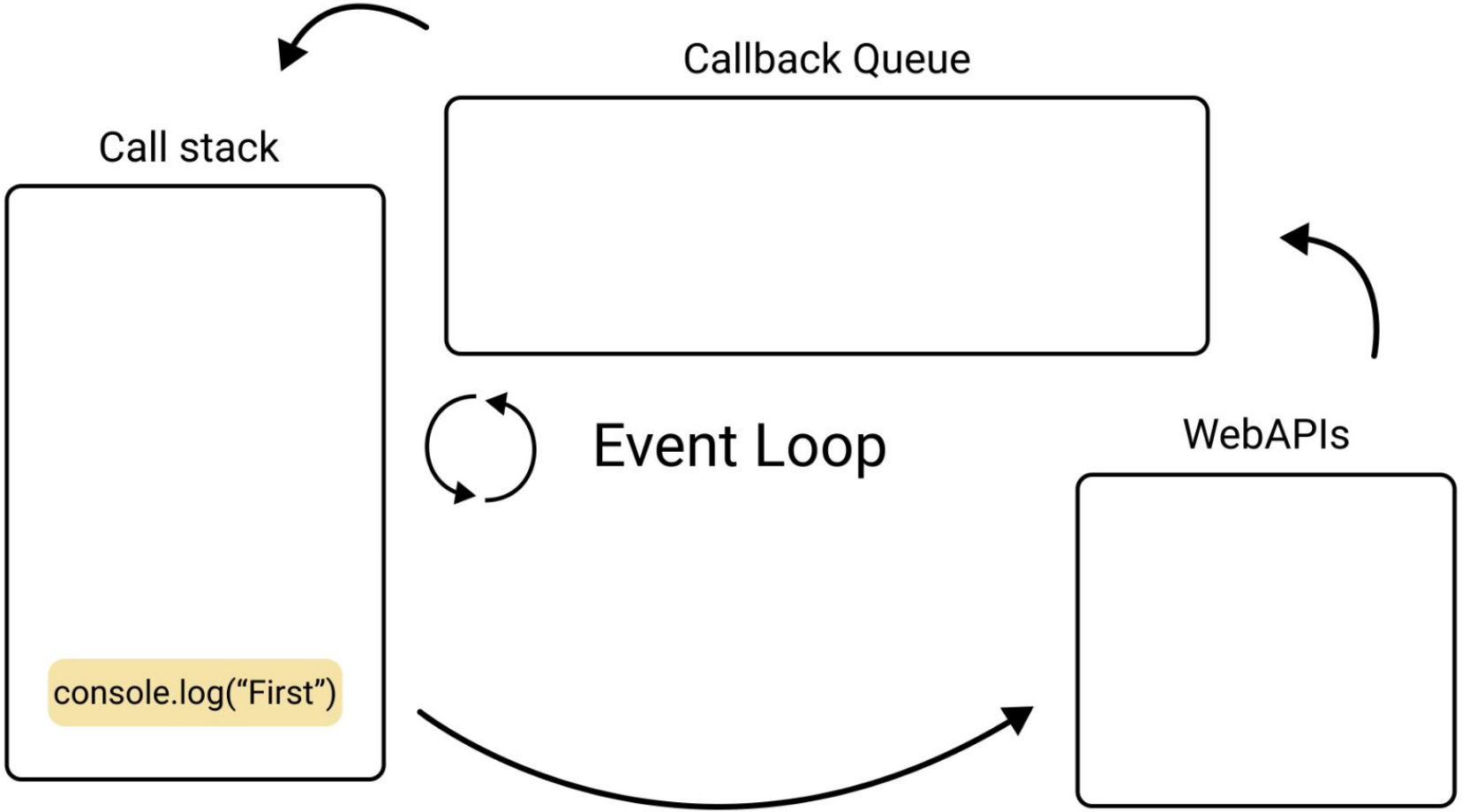
Advanced JS - Event Loop

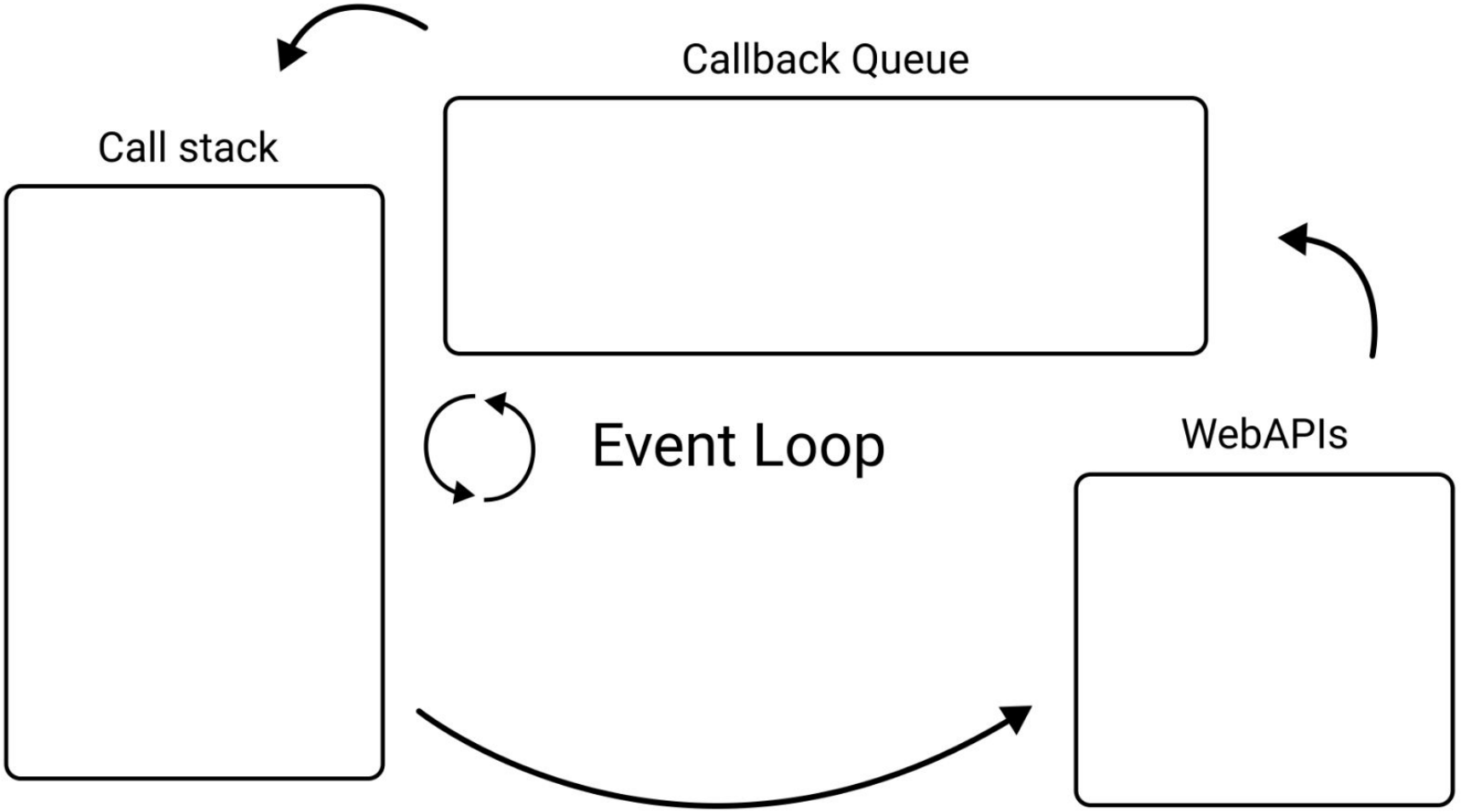
2. Web APIs/Node APIs
   - Daca API-ul este nevoit sa faca ceva asincron, atunci se foloseste de alt thread pentru a se executa
   - Dupa terminarea executiei, rezultatul se trimite intr-un callback queue
   - E.g. setTimeout, setInterval
   - E.g. filesystem (fs) - worker threads (4)
   - E.g. network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

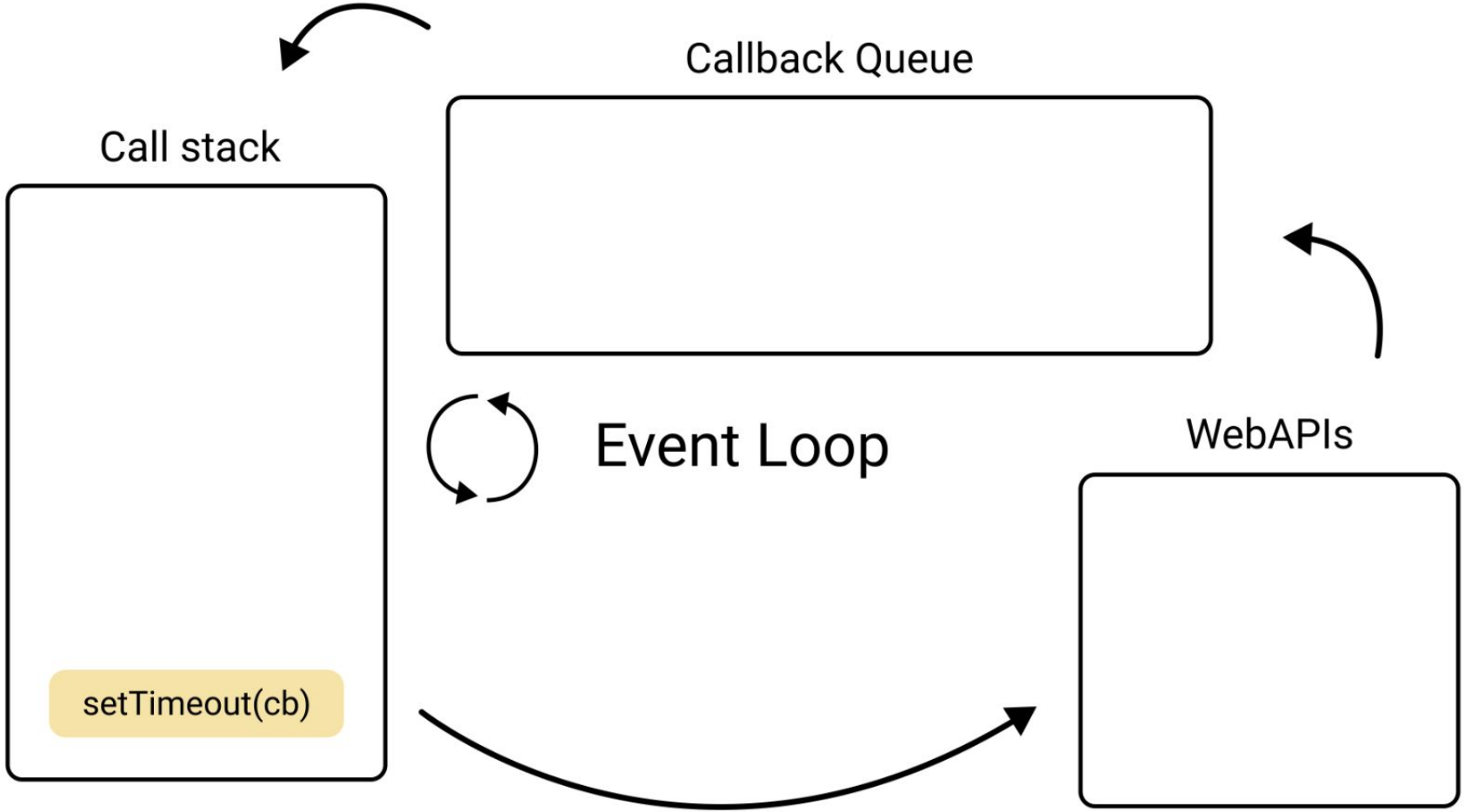
Advanced JS - Event Loop

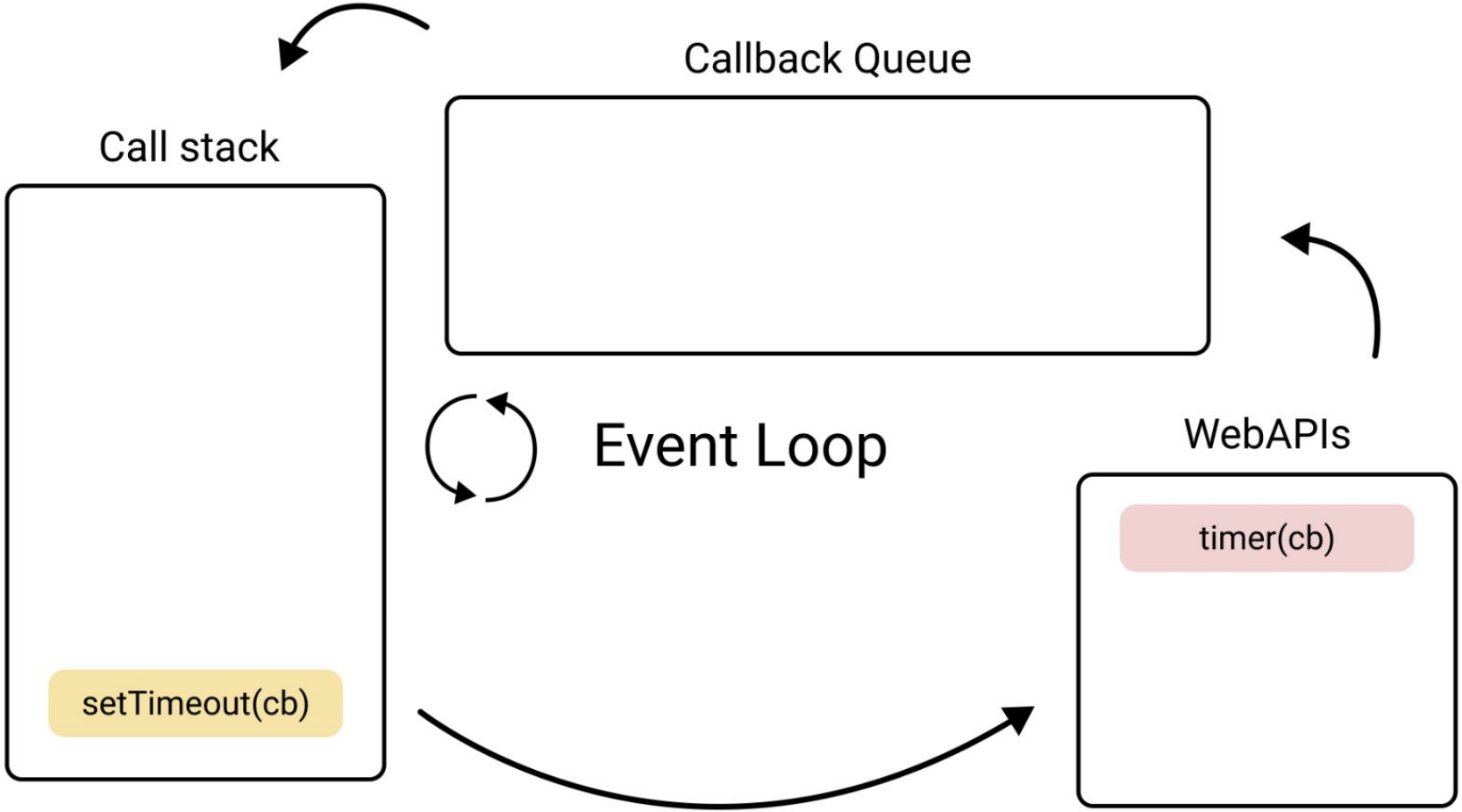
3. Callback queue
   - Dupa ce APIs isi termina “munca”, trimit rezultatul mai departe in callback queue
   - Cand **Call Stack** este gol, se extrag pe rand “rezultatele” din callback queue

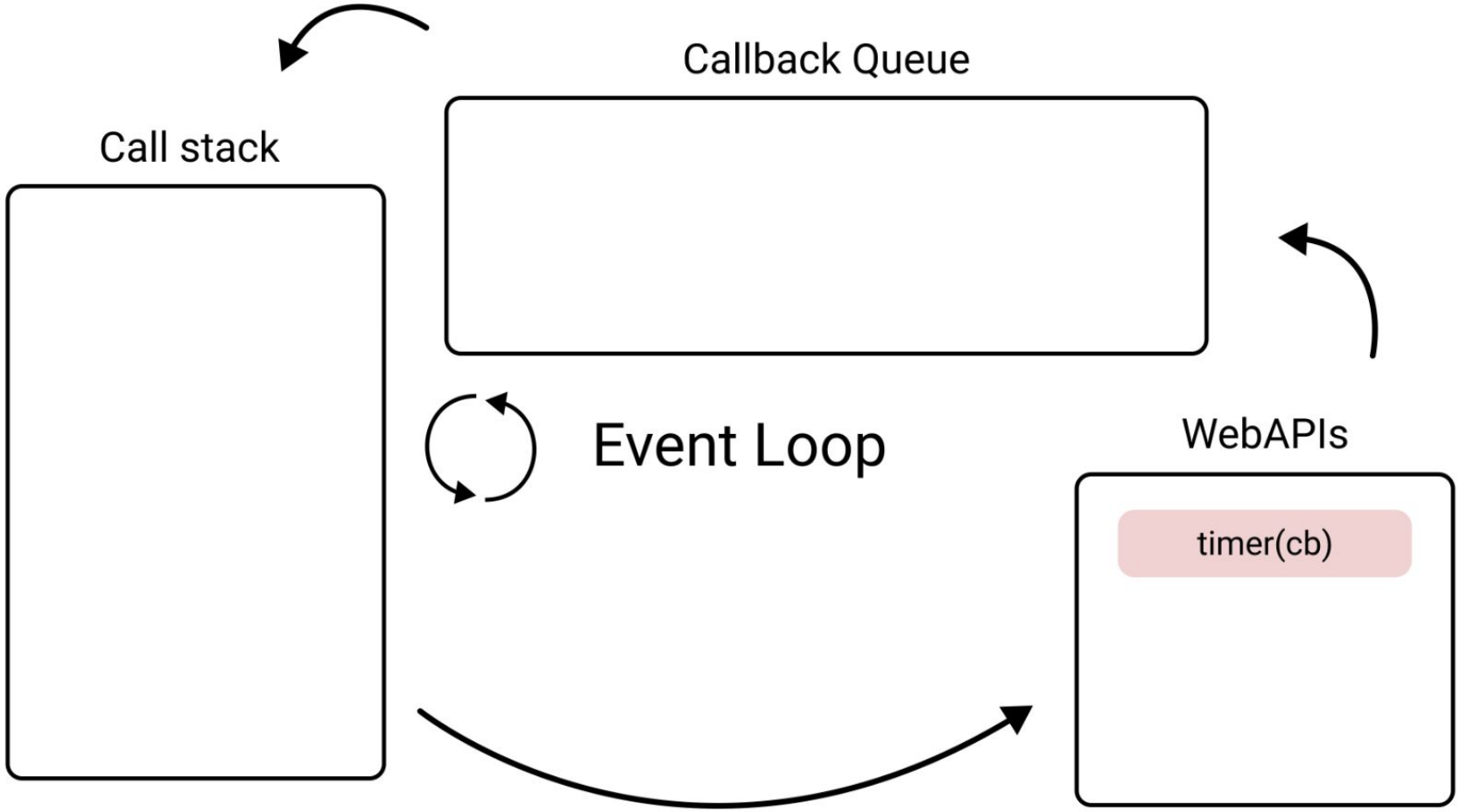
```
console.log("First");

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

console.log("Third");

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

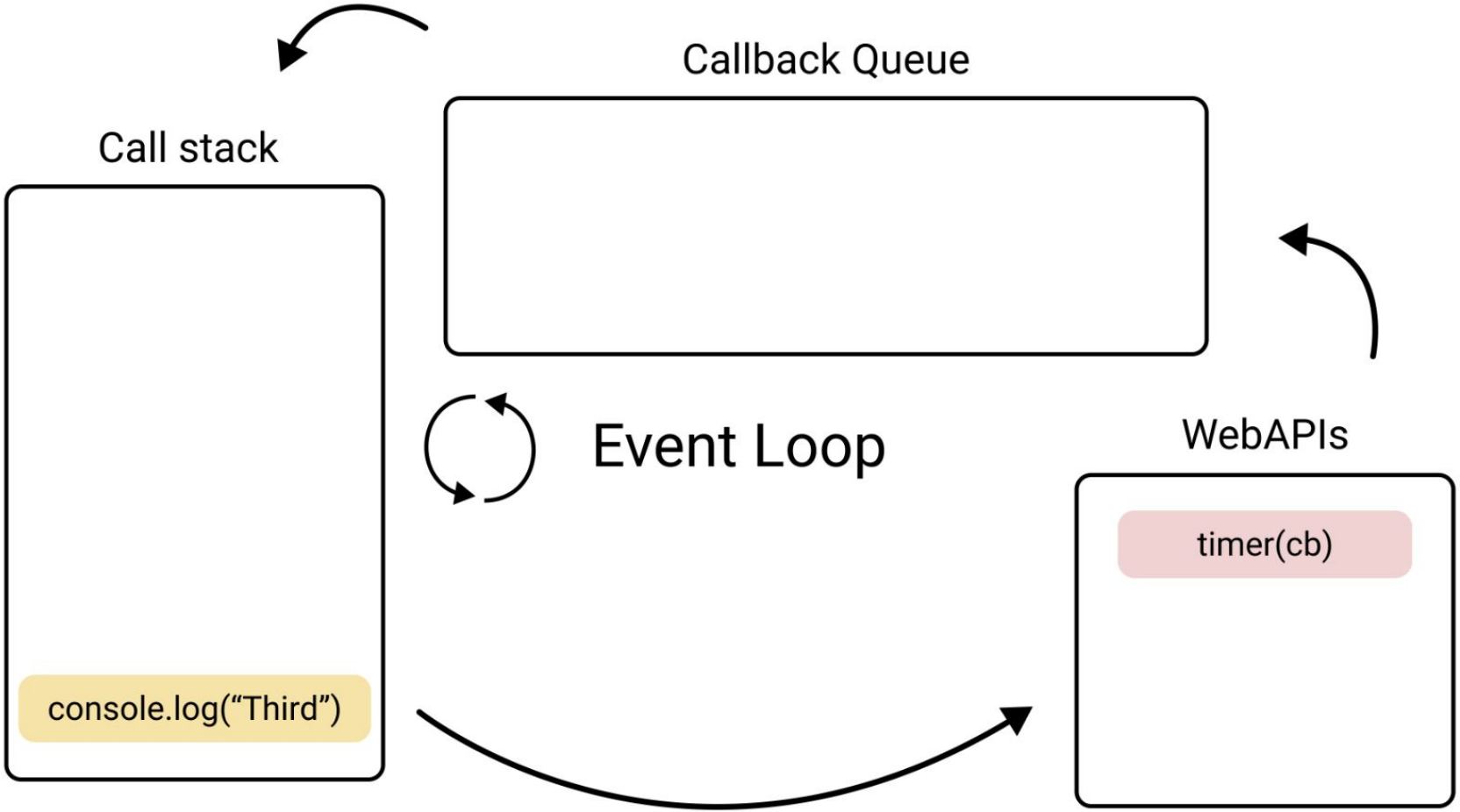

Callback Queue
Call stack
Event Loop
WebAPIs
console.log(“First”)

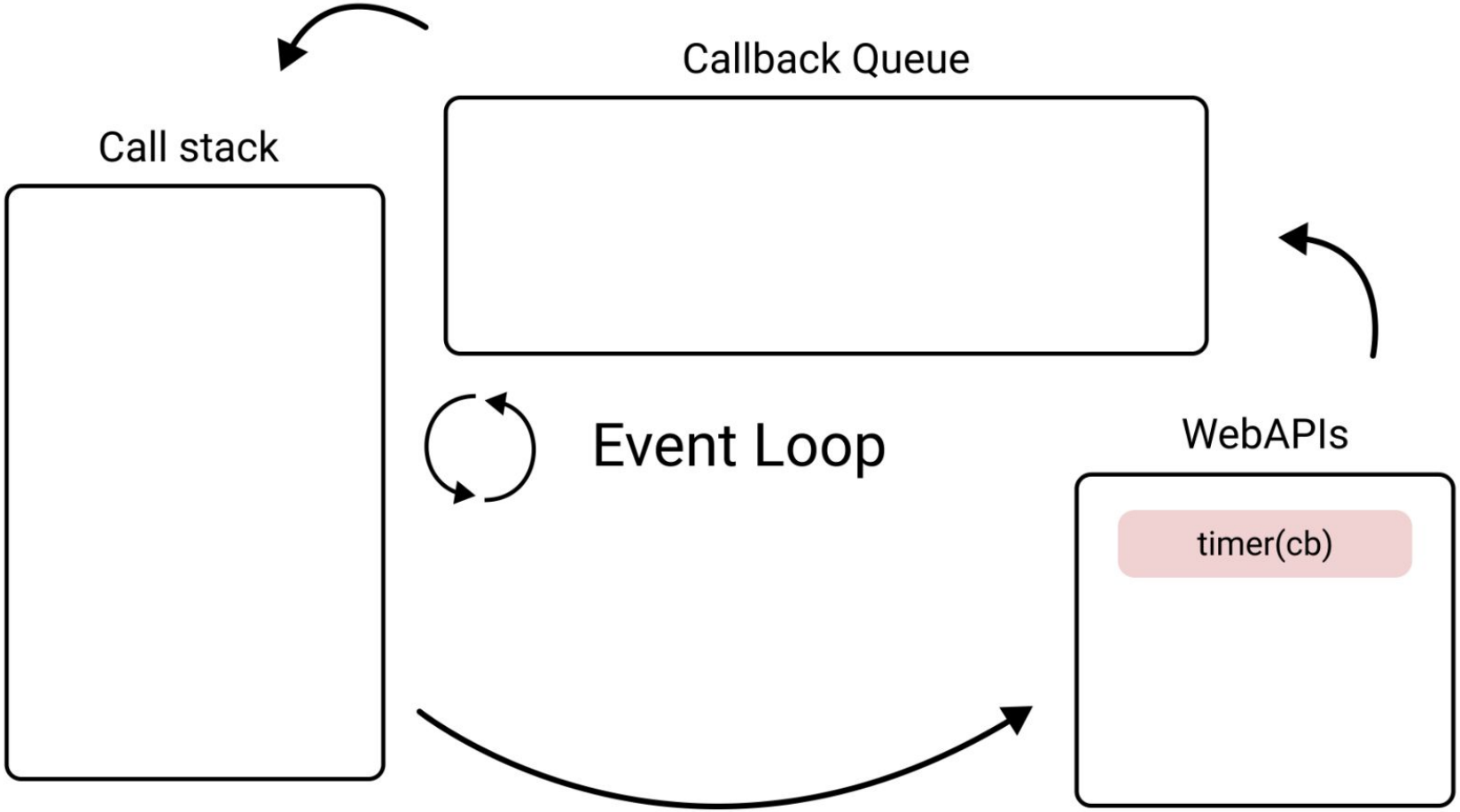

Callback Queue
Call stack
Event Loop
WebAPIs

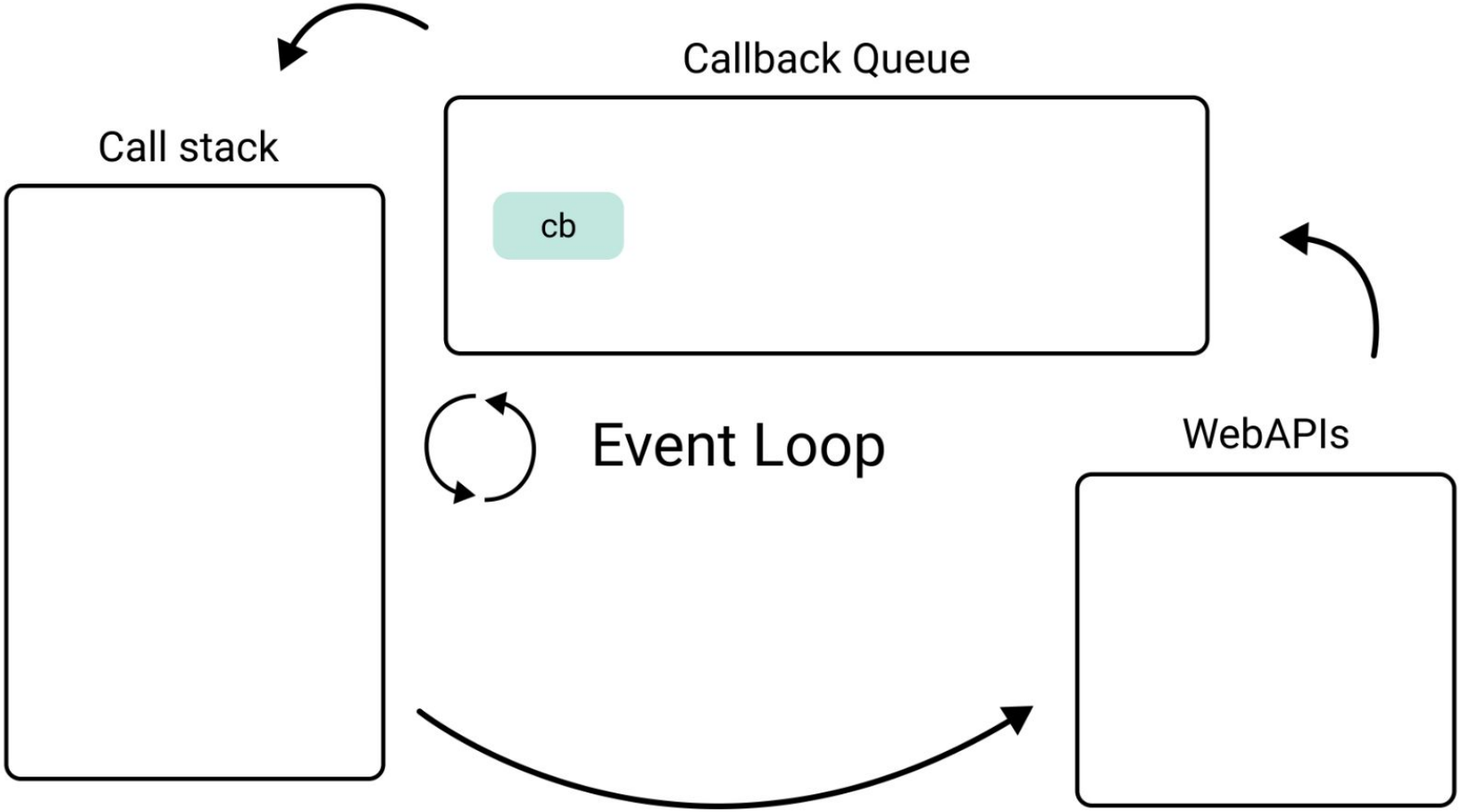

Callback Queue
Call stack
Event Loop
WebAPIs
setTimeout(cb)

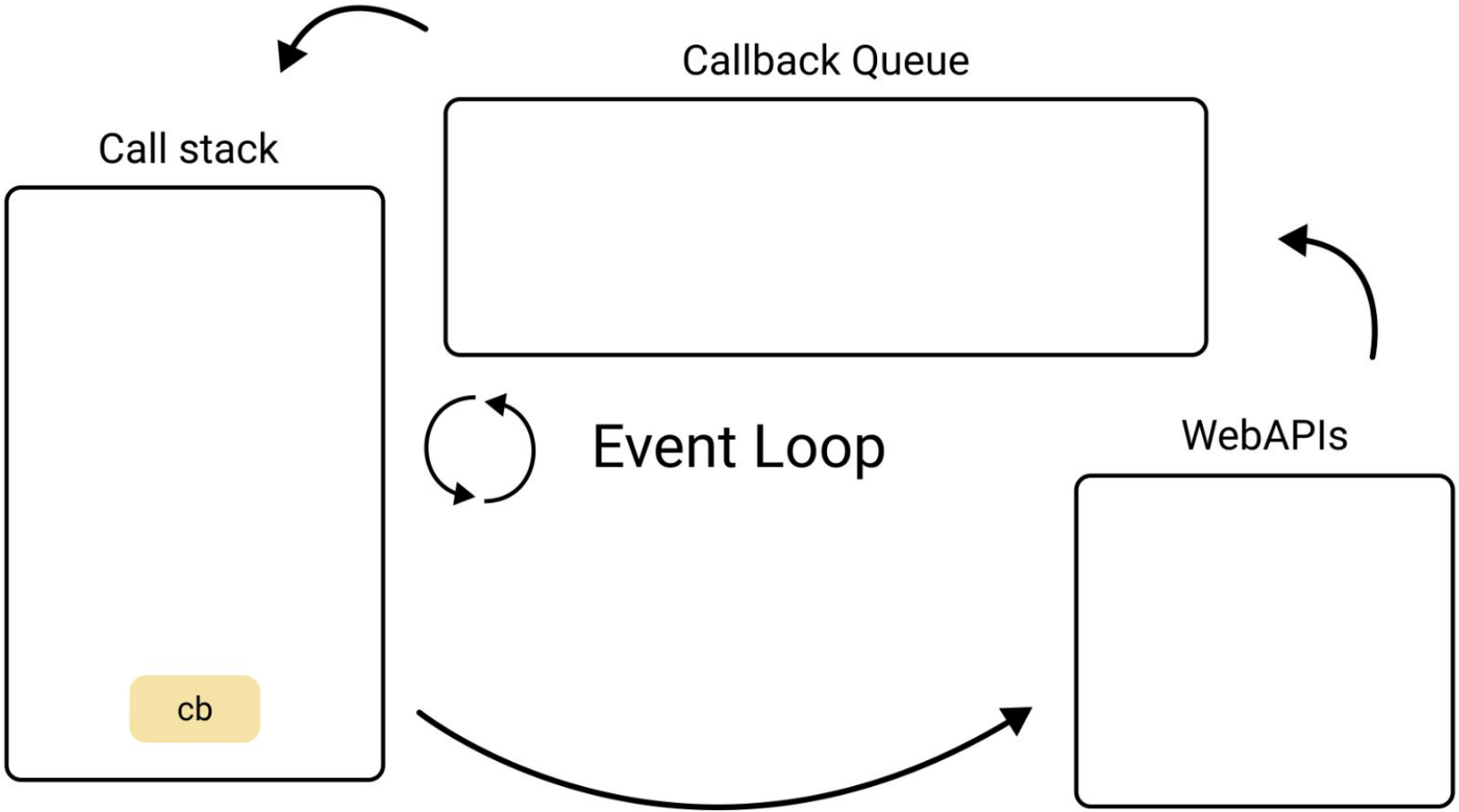

Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)
setTimeout(cb)

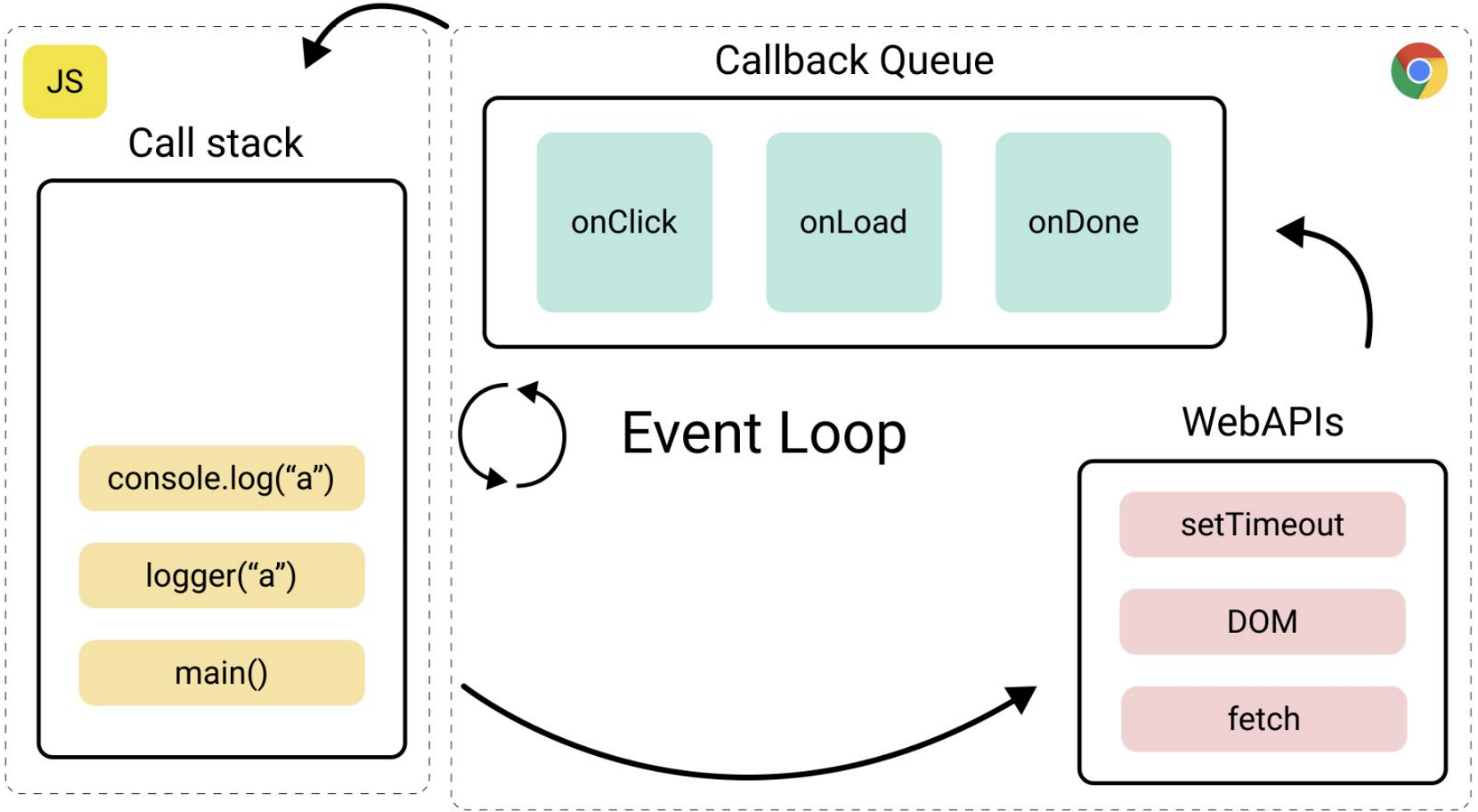

Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)


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

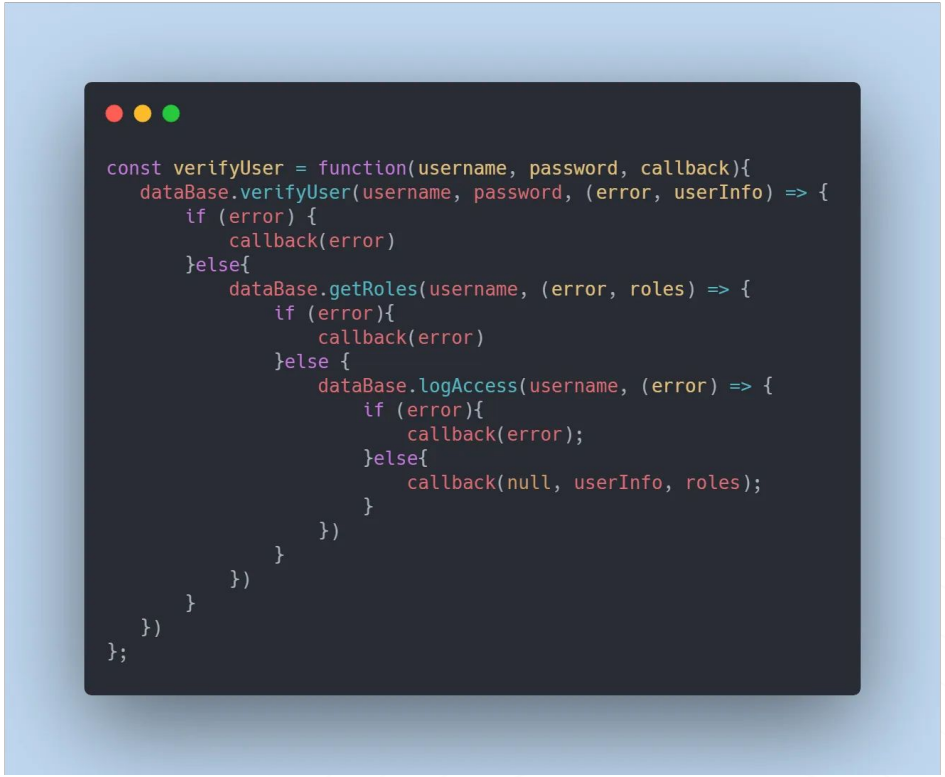

Callback Queue
Call stack
Event Loop
WebAPIs
timer(cb)


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();
```

Advanced JS - Promises