

Tutorato di Basi di dati e Web

Matthew Rossi

matthew.rossi@unibg.it



JavaScript (JS)

Con HTML e CSS, JavaScript è una delle tecnologie fondamentali del Web

È presente in oltre il 97% dei siti web al fine di supportare comportamenti dinamici client-side

Tutti i principali browser hanno un motore di esecuzione Javascript built-in

- Chrome → V8
- Safari → JavaScriptCore
- Firefox → SpiderMonkey

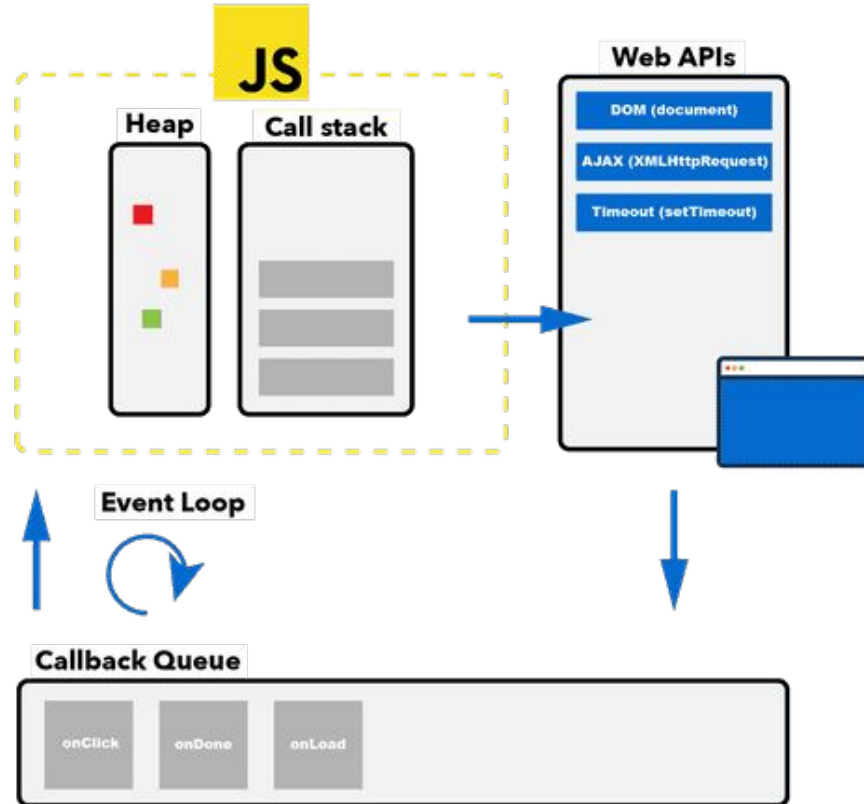
JavaScript (JS) - Caratteristiche

- Imperativo
- Interpretato
- Debolmente tipizzato
- Orientato agli oggetti
- Implementato a **singolo thread** in tutte le sue implementazioni moderne

Esecuzione sincrona/asincrona

- Esecuzione **sincrona**: è necessario attendere per il termine dell'esecuzione di un'istruzione / funzione, per poter eseguire la successiva
- Esecuzione **asincrona**: è possibile proseguire con l'esecuzione di nuove istruzioni / funzioni nonostante il risultato della precedente non sia ancora disponibile

Come funziona JavaScript?



Stack delle chiamate

Essendo JavaScript single-threaded, ha un unico stack delle chiamate

Quando una funzione viene chiamata, questa è aggiunta sulla cima dello stack, mentre al suo termine questa viene rimossa

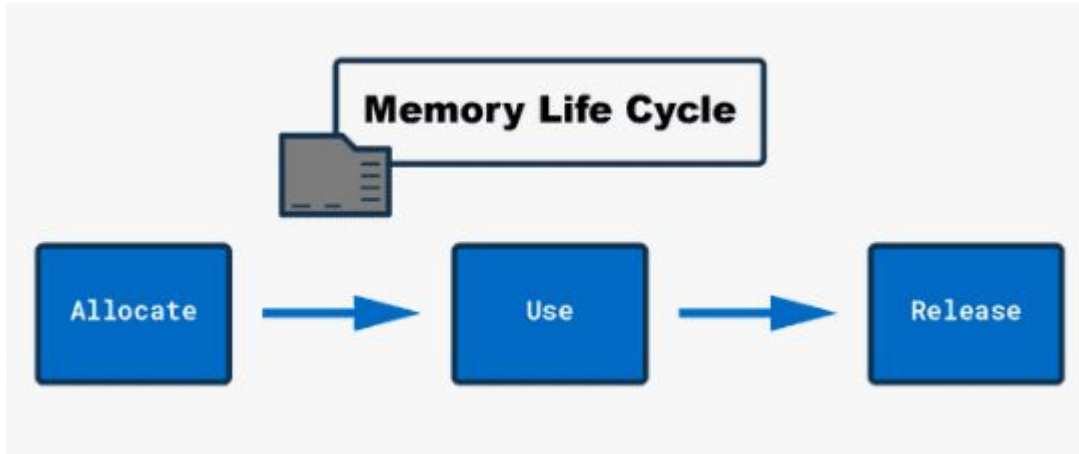
Ogni qual volta una funzione, ne chiama un'altra questa è aggiunta sulla cima dello stack, al di sopra della funzione chiamante

```
const addOne = (value) => value + 1;
const addTwo = (value) => addOne(value + 1);
const addThree = (value) => addTwo(value + 1);
const calculation = () => {
  return addThree(1) + addTwo(2);
};
calculation();
```

Heap

È dove sono salvati gli oggetti che vengono creati a seguito della definizione di funzioni e variabili

Assume una certa importanza quando si è interessati a minimizzare l'uso di memoria lato client



Web APIs

JavaScript può fare una sola cosa alla volta, ma il browser può fare più cose concorrentemente, e JavaScript può fare richieste al browser mediante le Web APIs

Le Web APIs non sono bloccanti

Permettono:

- Richieste AJAX
- Manipolare il DOM
- Molto altro (geolocalizzazione, accesso allo storage locale, ...)

Callback (1)

Le Web APIs abilitano la possibilità di fare attività concorrenti al di fuori dell'interprete JavaScript, ma come può il codice reagire ai risultati ottenuti mediante l'invocazione di una Web API (ad esempio una richiesta AJAX)?

CALLBACK

Mediante il loro uso è possibile specificare del codice da eseguire quando una API call è terminata

Callback (2)

Le callback sono delle funzioni passate come argomento ad un'altra funzione e sono in genere eseguite al termine dell'esecuzione

```
const a = () => console.log('a');  
const b = () => setTimeout(() => console.log('b'), 100);  
const c = () => console.log('c');  
  
a();  
b();  
c();
```

Coda delle callback

L'esecuzione di **setTimeout** aggiunge la callback alla coda delle callback

Il loop degli eventi prende la callback dalla coda e lo aggiunge allo stack non appena quest'ultimo è vuoto

Loop degli eventi


Il loop degli eventi prende la prima callback dalla coda e la aggiunge allo stack delle chiamate non appena quest'ultimo è vuoto

Se lo stack delle chiamate necessita di molto tempo per svuotarsi, il loop degli eventi è bloccato

È fondamentale non bloccare lo stack delle chiamate, in quanto questo rende il sito non responsivo alle interazioni dell'utente

WHAT THE HECK IS CALLBACK HELL?

```
2
3
4 a(function (resultsFromA) {
5     b(resultsFromA, function (resultsFromB) {
6         c(resultsFromB, function (resultsFromC) {
7             d(resultsFromC, function (resultsFromD) {
8                 e(resultsFromD, function (resultsFromE) {
9                     f(resultsFromE, function (resultsFromF) {
10                         console.log(resultsFromF);
11                     })
12                 })
13             })
14         })
15     })
16 });
17
```



Promise (1)

Sono la soluzione per evitare di incappare nel callback hell

Un oggetto JavaScript che corrisponde alla *promessa* di ottenere un risultato una volta terminata l'esecuzione di una funzione

```
let promise = new Promise(function(resolve, reject)
{
  // executor (the producing code, "singer")
});
```

Quando viene creata la funzione passata viene chiamata e produrrà eventualmente il risultato

resolve e **reject** sono callback fornite da JavaScript

Promise (2)

Quando l'esecutore ottiene il risultato chiama:

- `resolve(value)`: se la funzione è terminata con successo, con risultato `value`
- `reject(error)`: se si è verificato un errore

Promise (3)

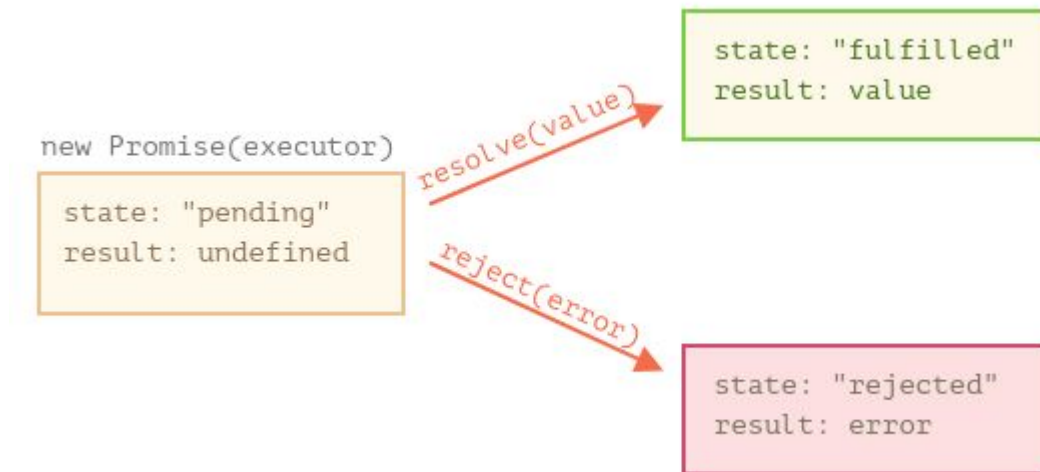
L'oggetto promise ha le seguenti proprietà interne:

- state

- "pending"
- "fulfilled"
- "rejected"

- result

- undefined
- value / error



Promise (4)

```
let promise = new Promise(function(resolve, reject) {  
  // the function is executed automatically when the promise is constructed  
  
  // after 1 second signal that the job is done with the result "done"  
  setTimeout(() => resolve("done"), 1000);  
});
```

new Promise(executor)

state: "pending"
result: undefined

resolve("done")

state: "fulfilled"
result: "done"

```
let promise = new Promise(function(resolve, reject) {  
  // after 1 second signal that the job is finished with an error  
  setTimeout(() => reject(new Error("Whoops!")), 1000);  
});
```

new Promise(executor)

state: "pending"
result: undefined

reject(error)

state: "rejected"
result: error

Promise consumers - then

È il consumer più importante, esegue quando il risultato della promessa è stabilito

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => resolve("done!"), 1000);  
});  
  
// resolve runs the first function in .then  
promise.then(  
  result => alert(result), // shows "done!" after 1 second  
  error => alert(error) // doesn't run  
);
```

Promise consumers - catch

Se siamo interessati solo ad errori è possibile usare:

- `.then(null, errorHandlingFunction)`
- `.catch(errorHandlingFunction)`

```
let promise = new Promise((resolve, reject) => {  
  setTimeout(() => reject(new Error("Whoops!")), 1000);  
});  
  
// .catch(f) is the same as promise.then(null, f)  
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

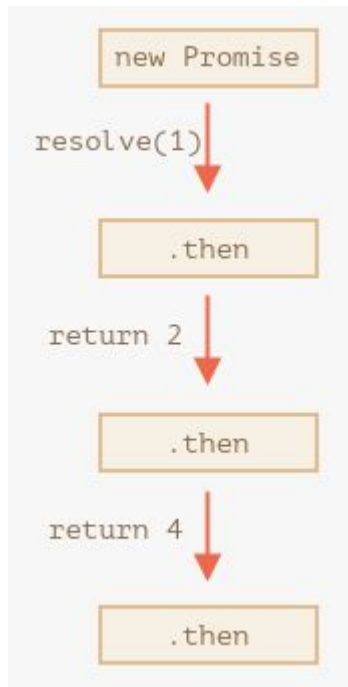
Promise consumers - then

È il consumer più importante, esegue quando il risultato della promessa è stabilito

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => resolve("done!"), 1000);  
});  
  
// resolve runs the first function in .then  
promise.then(  
  result => alert(result), // shows "done!" after 1 second  
  error => alert(error) // doesn't run  
);
```

Promise chaining

```
new Promise(function(resolve, reject) {  
    setTimeout(() => resolve(1), 1000); // (*)  
}).then(function(result) { // (**)  
    alert(result); // 1  
    return result * 2;  
}).then(function(result) { // (***)  
    alert(result); // 2  
    return result * 2;  
}).then(function(result) {  
    alert(result); // 4  
    return result * 2;  
});
```



Coda dei job

In aggiunta alla coda delle callback, ne esiste un'altra che accetta solo **promise**, la coda dei job

Questa coda ha priorità sulla coda di callback

```
console.log('a');  
setTimeout(() => console.log('b'), 0);  
new Promise((resolve, reject) => {  
  resolve();  
})  
.then(() => {  
  console.log('c');  
});  
console.log('d');
```

Richieste

Finora abbiamo visto la chiamata `setTimeout`, che introduce un delay prima di eseguire una funzione

Le richieste fatte dal client al server introducono un delay e possono compromettere la responsività del sito qualora gestite in modalità sincrona

Richieste sincrone - XMLHttpRequest

```
var request = new XMLHttpRequest();  
request.open('GET', '/bar/foo.txt', false); // `false` makes the request  
synchronous  
request.send(null);  
  
if (request.status === 200) {  
    console.log(request.responseText);  
}
```


Richieste asincrone - XMLHttpRequest

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/bar/foo.txt", true);
xhr.onload = function (e) {
    if (xhr.readyState === 4) {
        if (xhr.status === 200) {
            console.log(xhr.responseText);
        } else {
            console.error(xhr.statusText);
        }
    }
};
xhr.onerror = function (e) {
    console.error(xhr.statusText);
};
xhr.send(null);
```

Fetch (1)

L'utilizzo di XMLHttpRequest per la gestione di chiamate HTTP da JavaScript risulta abbastanza prolisso e scomodo

```
let promise = fetch(url, [options])
```

Fetch ha una sintassi più semplice basata sulle promise

- url: identifica l'URL a cui inviare la richiesta
- options: parametri opzionali come headers, metodi, etc

Example:

```
fetch("http://www.example.com" )  
  .then(response => {  
    console.log(response);  
  })  
  .catch(error => console.log("Si è verificato un errore!" ))
```

Fetch (2)

Nel caso di successo la promise verrà risolta e la risposta del server viene fornita mediante l'oggetto **Response**

Metodo	Descrizione
<code>text()</code>	Restituisce il contenuto sotto forma di testo
<code>json()</code>	Effettua il parsing del contenuto e lo restituisce sotto forma di oggetto
<code>blob()</code>	Restituisce il contenuto sotto forma di dati non strutturati (<i>blob</i>)
<code>arrayBuffer()</code>	Restituisce il contenuto strutturato in un <i>arrayBuffer</i>

Fetch (3)

```
fetch("https://www.example.com/api/articoli/123").then(response => {  
  if (response.ok) {  
    return response.json();  
  }  
})  
.then(articolo => console.log(articolo.titolo))  
.catch(error => console.log("Si è verificato un errore!"))
```