

JAVASCRIPT

Riccardo Cattaneo

Lezione 10



Fetch API

Che cos'è fetch? Dall'inglese «prelevare», è un API che ci permette di manipolare le chiamate HTTP. Vediamo un esempio pratico su come utilizzare l'API Fetch per prelevare oggetti JSON tramite il protocollo http.

Per farlo utilizziamo un servizio gratuito su internet che ci mette a disposizione un web service con dati fake : <https://jsonplaceholder.typicode.com/>

Routes

All HTTP methods are supported. You can use http or https for your requests.


GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Note: see [guide](#) for usage examples.

Per fare una prova creiamo un file .js che successivamente lo importeremo all'interno di una pagina html, e come prima cosa memorizziamo l'url dell'API all'interno di una variabile :

```
JS app.js > ...  
1  
2 let url = 'https://jsonplaceholder.typicode.com/posts/';  
3
```

A questo punto possiamo chiamare la **funzione globale fetch (che torna una promises)** e prende in ingresso come parametro l'url di destinazione dalla quale possiamo eseguire direttamente il then / catch :

```
JS app.js >  then() callback
1
2   let url = 'https://jsonplaceholder.typicode.com/posts/';
3
4   fetch(url).then( result => {
5     |   console.log(result);
6   }).catch( error  => {
7     |   console.log(error);
8   })
```

urces Network Performance Memory Application Security Lighthouse

```
▼ Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts/',  
  body: (...)  
  bodyUsed: false  
  ▶ headers: Headers {}  
  ok: true  
  redirected: false  
  status: 200  
  statusText: ""  
  type: "cors"  
  url: "https://jsonplaceholder.typicode.com/posts/"  
  ▶ [[Prototype]]: Response
```

Come possiamo vedere, abbiamo specificato l'URL su cui effettuare la richiesta HTTP come parametro della funzione `fetch()` ed abbiamo gestito la risposta come una promise. In caso di successo la promise verrà risolta ed entreremo nel ramo `then()`, in cui ci verrà fornita la risposta del server sotto forma di oggetto di tipo `Response`.

Occorre sottolineare che la promise restituita dalla funzione `fetch()` viene risolta ogni qualvolta c'è una risposta da parte del server, non solo quando otteniamo un codice di stato 200.

In altre parole, se entriamo nel ramo `then()` del codice precedente non dobbiamo dare per scontato di aver ottenuto il contenuto richiesto al server. È buona norma verificare il codice di stato della risposta e gestirlo opportunamente.

JS app.js > ...

```
1
2  let url = 'https://jsonplaceholder.typicode.com/posts/';
3
4  fetch(url).then( result => {
5
6      console.log(result);
7
8      if(result.ok == true){
9          console.log('ok tutto corretto');
10     }else if(result.status >= 400 && result.status <= 499){
11         console.log('richiesta errata');
12     }
13
14 }).catch( error => {
15     console.log(error);
16 })
17
```


```
▼ Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts/', redirected:  
  body: (...)  
  bodyUsed: false  
  ▶ headers: Headers {}  
  ok: true  
  redirected: false  
  status: 200  
  statusText: ""  
  type: "cors"  
  url: "https://jsonplaceholder.typicode.com/posts/"  
  ▶ [[Prototype]]: Response
```

ok tutto corretto

>

A questo punto non ci resta che estrapolare dal risultato il nostro oggetto json attraverso l'apposita funzione che ci mette a disposizione il linguaggio (`response.json()`) in questo modo :

```
if(result.ok == true){  
    console.log('ok tutto corretto');  
    console.log(result.json());  
}else if(result.status >= 400 && result.status <= 499){  
    console.log('richiesta errata');  
}
```



► Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts/', redirected: false, status: 200,

ok tutto corretto

▼ Promise {<pending>} ⓘ

► [[Prototype]]: Promise

[[PromiseState]]: "fulfilled"

▼ [[PromiseResult]]: Array(100)

- 0: {userId: 1, id: 1, title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- 1: {userId: 1, id: 2, title: 'qui est esse', body: 'est rerum tempore vitae\nsequi sint nihil reprehend.
- 2: {userId: 1, id: 3, title: 'ea molestias quasi exercitationem repellat qui ipsa sit aut', body: 'et il
- 3: {userId: 1, id: 4, title: 'eum et est occaecati', body: 'ullam et saepe reiciendis voluptatem adipisc
- 4: {userId: 1, id: 5, title: 'nesciunt quas odio', body: 'repudiandae veniam quaerat sunt sed\nnalias aut
- 5: {userId: 1, id: 6, title: 'dolorem eum magni eos aperiam quia', body: 'ut aspernatur corporis harum r
- 6: {userId: 1, id: 7, title: 'magnam facilis autem', body: 'dolore placeat quibusdam ea quo vitae\nmagni
- 7: {userId: 1, id: 8, title: 'dolorem dolore est ipsam', body: 'dignissimos aperiam dolorem qui eum\nfac
- 8: {userId: 1, id: 9, title: 'nesciunt iure omnis dolorem tempora et accusantium', body: 'consectetur ar
- 9: {userId: 1, id: 10, title: 'optio molestias id quia eum', body: 'quo et expedita modi cum officia vel
- 10: {userId: 2, id: 11, title: 'et ea vero quia laudantium autem', body: 'delectus reiciendis molestiae
- 11: {userId: 2, id: 12, title: 'in quibusdam tempore odit est dolorem', body: 'itaque id aut magnam\npra

Promise.all()

Promise.all() ci consente di eseguire diverse promise in parallelo e avere tutti i dati alla conclusione dell'ultima promise, non importa quale sarà risolta per prima, noi le processeremo tutte insieme una volta terminate tutte.

Questo si fa passando come **parametro un array di promise**, se una promise dell'array viene rigettata blocca tutte le altre promise in esecuzione.

Andiamo a testarlo provando con il `setTimeout` perchè tutto sia più chiaro, passiamo i risultati del nostro `fetch` ad un'altra funzione che andrà a creare due promise abbastanza lunghe e complicate, che richiederanno una 5 secondi, l'altra 10.

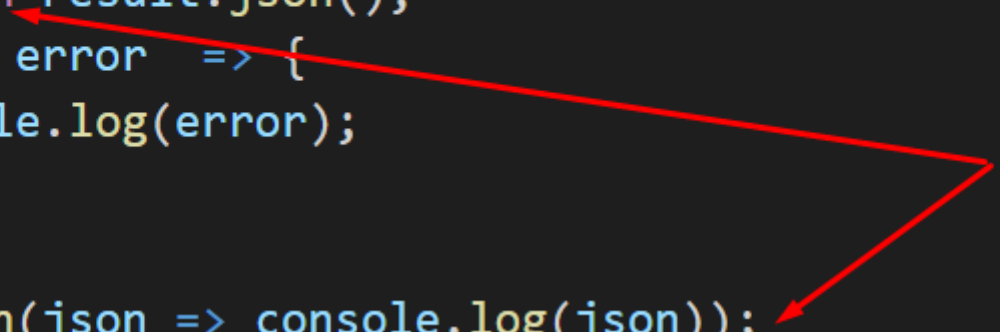
Noi dovremmo svolgere delle operazioni quando entrambe queste promise saranno terminate, ma nello stesso tempo non vogliamo far partire la seconda promise dopo che ha finito la prima, ma contemporaneamente, in modo che tutto sia più veloce. Tanto nessuna delle due promise dipende dall'altra. Iniziamo quindi a richiamare una funzione al termine del nostro `fetch` risolto:

Per esempio possiamo creare due promise, la prima che torna tutti i post e la seconda che torna tutti i commenti del primo post :

```
let url = 'https://jsonplaceholder.typicode.com/posts/';

let tutti = fetch(url).then( result => {
  return result.json();
}).catch( error => {
  console.log(error);
});

tutti.then(json => console.log(json));
```

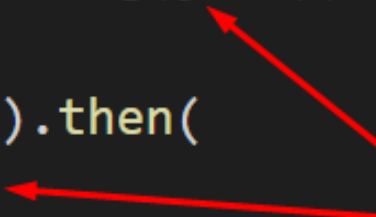
A diagram consisting of two red arrows. The first arrow originates from the `return result.json();` line within the `then` block of the `tutti` variable and points to the `then` method of the `tutti.then(json => console.log(json));` line. The second arrow originates from the `tutti.then(json => console.log(json));` line and points to the `console.log(json);` statement within the same `then` block, illustrating the sequential execution of promises.

JS app.js > ...

```
1
2  let url = 'https://jsonplaceholder.typicode.com/posts/';
3
4  let tutti = fetch(url).then( result => {
5    |   return result.json();
6  }).catch( error => {
7    |   console.log(error);
8  });
9
10 tutti.then(json => console.log(json));
11
12 let commenti = fetch(url+'1/comments').then( result => {
13   |   return result.json();
14 }).catch( error => {
15   |   console.log(error);
16 });
17
18 commenti.then(json => console.log(json));
```



A questo punto, invece di eseguirle, le passiamo in ingresso ad una `promise.all` che ricordiamolo prende in ingresso un array di promise

```
18  // commenti.then(json => console.log(json));
19
20  Promise.all([ tutti, commenti ]).then(
21    resp => {
22      console.log(resp);
23    }
24  )
```

A red arrow originates from the `Promise.all` call on line 20 and points to the `resp` parameter in the `then` callback on line 21. A second red arrow originates from the `console.log(resp)` call on line 22 and points back to the `resp` parameter on line 21, illustrating the data flow.

Per visualizzare ora i dati all'interno della nostra pagina html è sufficiente aggiungere un tag html con un id specifico :

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8      <script src="./app.js"></script>
9  </head>
10 <body>
11     <div id="posts-all"></div>
12     <div id="comments-all"></div>
13 </body>
14 </html>
```



```
Promise.all([ tutti, commenti ]).then(
  resp => {
    let posts = resp[0];
    let div = document.getElementById('posts-all');
    posts.forEach(element => {
      let miodiv = document.createElement('div');
      miodiv.innerHTML = element.body;
      div.appendChild(miodiv);
    });

    let comm = resp[1];
    let div2 = document.getElementById('comments-all');
    comm.forEach(element => {
      let miodiv = document.createElement('div');
      miodiv.innerHTML = element.body;
      div2.appendChild(miodiv);
    });
  }
)
```