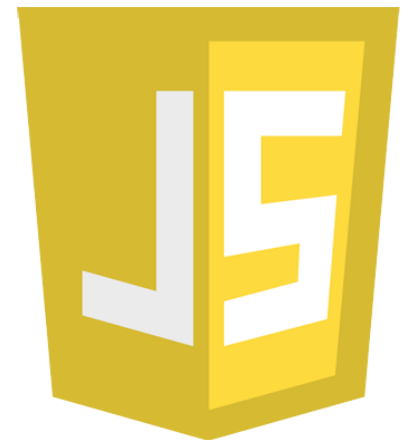


JAVASCRIPT

Riccardo Cattaneo

Lezione 11



async / await

async e await : Si tratta di due parole chiave che **abilitano la gestione di funzioni asincrone eseguite tramite un approccio sincrono**. Per comprendere l'utilità di queste nuove parole chiave, occorre innanzitutto capire quali sono gli approcci generalmente più utilizzati per l'esecuzione di operazioni asincrone in JavaScript.

```
function a() {  
    console.log("funzione A");  
}  
  
function b() {  
    console.log("funzione B");  
}  
  
function c() {  
    console.log("funzione C");  
}  
  
function d() {  
    console.log("funzione D");  
}  
  
function calcolo() {  
    a();  
    b();  
    c();  
    d();  
}  
  
calcolo();
```

Come possiamo notare il codice viene eseguito in sequenza con un approccio sincrono, viene prima eseguita la funzione prova1, prova2, prova3 e prova4. Ma cosa succede se inseriamo all'interno di prova 3 una funzione con approccio asincrono come ad esempio setTimeout ?

```
function a(){
  console.log("funzione A");
}

function b(){
  console.log("funzione B");

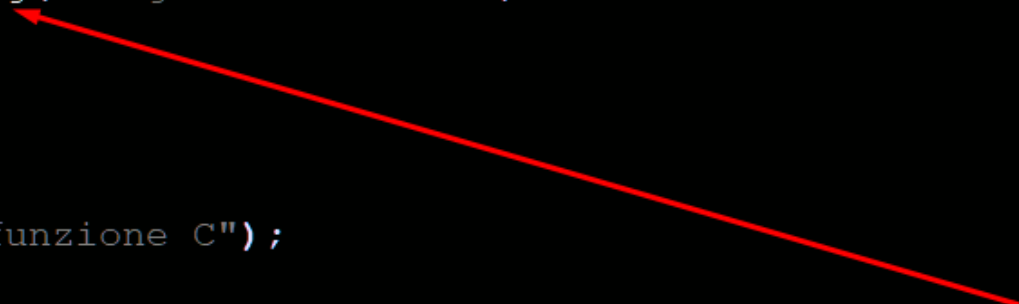
  setTimeout( ()=>{
    console.log("eseguo setTimeout");
  } , 3000)
}

function c(){
  console.log("funzione C");
}

function d(){
  console.log("funzione D");
}

function calcolo(){
  a();
  b();
  c();
  d();
}

calcolo();
```



Come possiamo notare vengono eseguite prima le funzioni a,b,c,d e poi alla fine setTimeout. Ma come abbiamo detto in precedenza, spesso ci capita di dover «aspettare» l'esecuzione della funzione b in quanto nell'esecuzione della chiamata asincrona vengono prelevati dati importanti per l'esecuzione delle funzioni c e d.

In questo caso ci vengono in aiuto `async / await`, che, regola importante, tornano una **promise**. Quindi per usarle dobbiamo fare in modo che la funzione deve tornare una promise **obbligatoriamente**.

Nel nostro esempio infatti, se applico semplicemente `async / await` sulla funzione `b()` ... non avrà alcun effetto.... proviamo (**async viene messo davanti alla funzione contenitore, await davanti alla funzione interna asincrona**) :

```
function a(){
  console.log("funzione A");
}

function b(){
  console.log("funzione B");

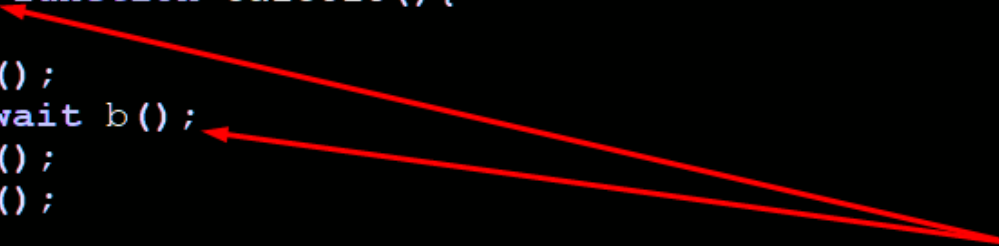
  setTimeout( ()=>{
    console.log("eseguo setTimeout");
  } , 3000)
}

function c(){
  console.log("funzione C");
}

function d(){
  console.log("funzione D");
}

async function calcolo(){
  a();
  await b();
  c();
  d();
}

calcolo();
```



The diagram consists of two red arrows. The first arrow originates from the 'await b()' line within the 'calcolo()' function and points to the 'setTimeout' call inside the 'b()' function. The second arrow originates from the same 'await b()' line and points to the 'setTimeout' call inside the 'b()' function. This illustrates that the execution of the 'calcolo()' function is paused at the 'await b()' statement until the promise returned by 'b()' (which is the 'setTimeout' promise) is resolved.

La prossima modifica da fare ora è far tornare la nostra funzione `b()` una promise in questo modo:

```
function b(){  
  console.log("funzione B");  
  let x = new Promise( (res,rej) => {  
    setTimeout( ()=>{  
      console.log("eseguo setTimeout");  
      res("ok");  
    } , 3000)  
  })  
  return x;  
}
```

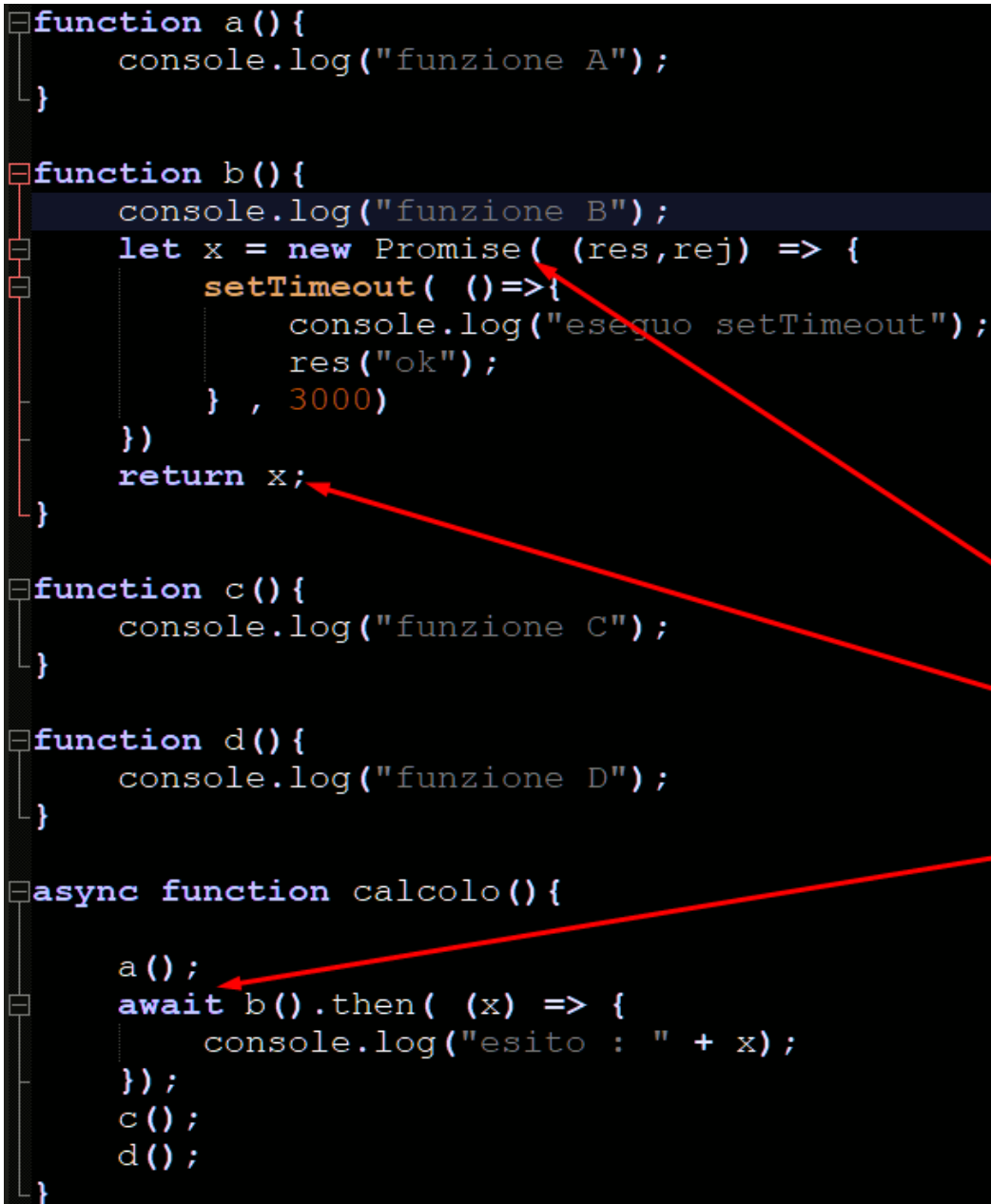
```
function a(){
  console.log("funzione A");
}

function b(){
  console.log("funzione B");
  let x = new Promise( (res,rej) => {
    setTimeout( ()=>{
      console.log("eseguo setTimeout");
      res("ok");
    } , 3000)
  })
  return x;
}

function c(){
  console.log("funzione C");
}

function d(){
  console.log("funzione D");
}

async function calcolo(){
  a();
  await b().then( (x) => {
    console.log("esito : " + x);
  });
  c();
  d();
}
```



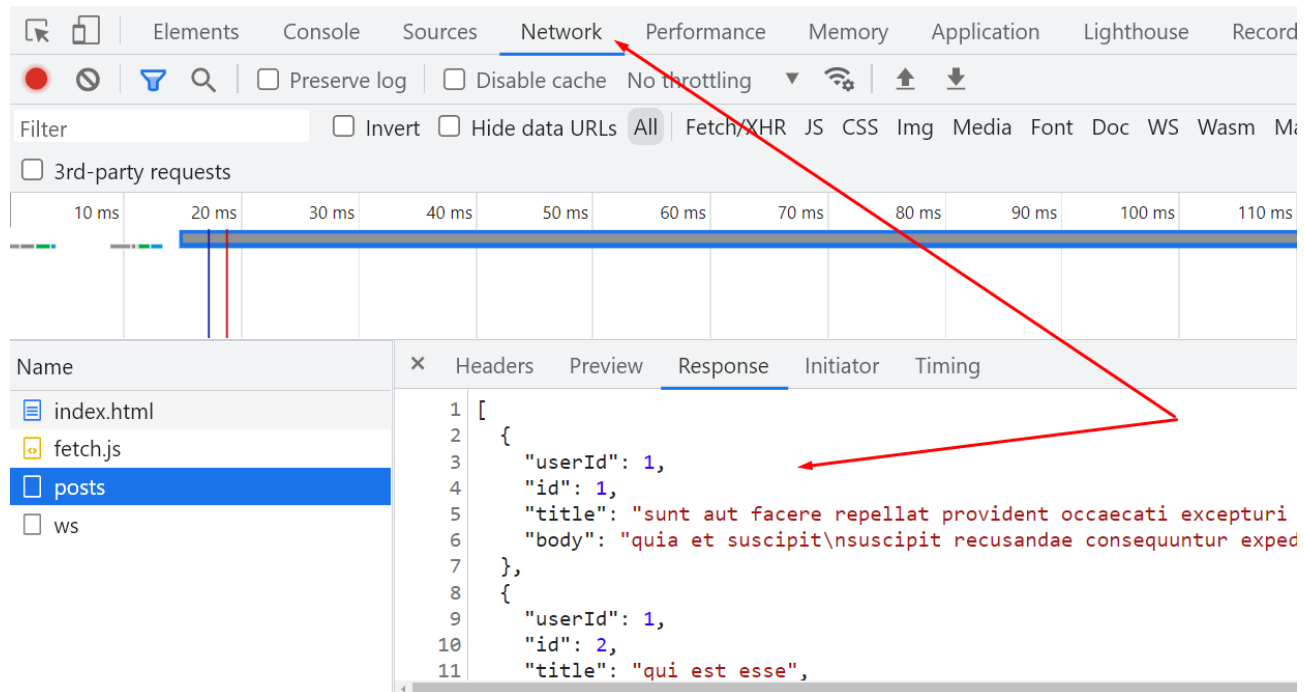
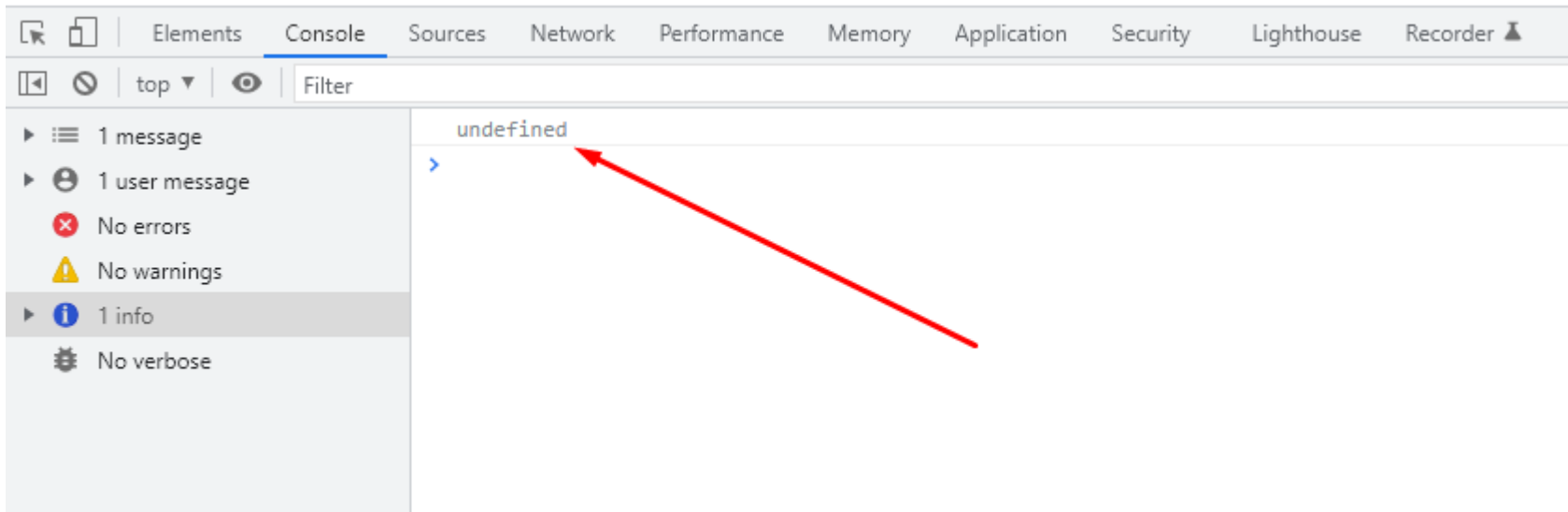
The diagram illustrates the execution flow of the provided JavaScript code. Red arrows indicate the sequence of calls:

- A red arrow points from the `await b().then(...)` line in the `calcolo()` function to the `return x;` line in the `b()` function.
- Another red arrow points from the `return x;` line in the `b()` function back to the `await b().then(...)` line in the `calcolo()` function.

Per capire il funzionamento e a cosa serve `async / await` facciamo un altro esempio e scriviamo una semplice `fetch` per prelevare tutti gli utenti dal nostro sito «jsonplaceholder» e attraverso la console andiamo a richiamare la funzione e stampare il risultato : come possiamo notare la funzione ci torna **undefined**... come mai? E se andiamo nella console del browser ed apriamo la preview della Network, possiamo notare che gli utenti ci sono... come mai ?

```
function prelevamentoDati(){  
    let indirizzo = "https://jsonplaceholder.typicode.com/posts";  
  
    fetch(indirizzo).then(  
        (ris) => {  
            let risultato = ris.json();  
            return risultato;  
        }  
    ).catch(  
        (err) => { console.log(err); }  
    );  
}
```

```
let risultato = prelevamentoDati();  
console.log(risultato); // undefined
```



JavaScript

Questo accade perché «fetch» ha un approccio asincrono e questo vuol dire che appena il compilatore incontra una chiamata fetch, il codice non attende che termina ma va avanti con l'esecuzione del codice (asincrono appunto).

Quando ritorniamo la variabile «utenti» la stessa è ancora undefined in quanto viene valorizzata all'interno di una chiamata fetch che come sappiamo torna una promise.

Grazie ad `async` / `await` possiamo dire alla funzione che è asincrona attraverso la parola chiave `async`, mentre con `await` specifichiamo quale chiamata asincrona deve gestire : in questo caso aspetta che finisce di caricare gli utenti e poi va avanti nella funzione `elencoutenti()`, questo vuol dire che arrivati all'assegnazione della variabile `let ris`, la promise sarà già risolta.

```
async function prelevamentoDati(){
```

```
  let indirizzo = "https://jsonplaceholder.typicode.com/posts";
```

```
  let esito = await fetch(indirizzo).then(  
    (ris) => {
```

```
    let risultato = ris.json();
```

```
    return risultato;
```

```
  }
```

```
  ).catch(  
    (err) => { console.log(err);}
```

```
  );
```

```
  return esito;
```

```
}
```

```
let risultato = prelevamentoDati();
```

```
risultato.then( (x) => {
```

```
  console.log(x);
```

```
} );
```


Ricordiamo che una funzione `async` torna **sempre** una `promise`. All'interno possiamo mettere da 0 a `n` `await`, verranno eseguiti uno alla volta e finché il primo non è terminato, non passa al secondo e così via...

Esempio

JS app2.js > ...

```
1
2  async function getBlogAndPhoto(userId) {
3
4      try {
5          let utente = await fetch("/utente/" + userId);
6          let blog = await fetch("/blog/" + utente.blogId);
7          let foto = await fetch("/photo/" + utente.albumId);
8          return {
9              utente,
10             blog,
11             foto
12         };
13     } catch (e) {
14         console.log("Si è verificato un errore!");
15     }
16 }
17
```

Essa carica i dati dell'utente, poi i dati del blog associato all'utente e quindi le foto associate all'utente. Infine la funzione restituisce un oggetto con tutte le informazioni recuperate.

Ciascuna operazione asincrona scatenata dall'invocazione a `fetch()` **viene eseguita dopo il completamento della precedente invocazione**. In altre parole, le operazioni asincrone non avvengono in parallelo, avendo quindi un potenziale impatto sulle prestazioni dell'applicazione.

Se volessimo trarre beneficio dall'esecuzione parallela delle chiamate HTTP, dovremmo utilizzare il metodo **`Promise.all()`**, come mostrato dal seguente codice:

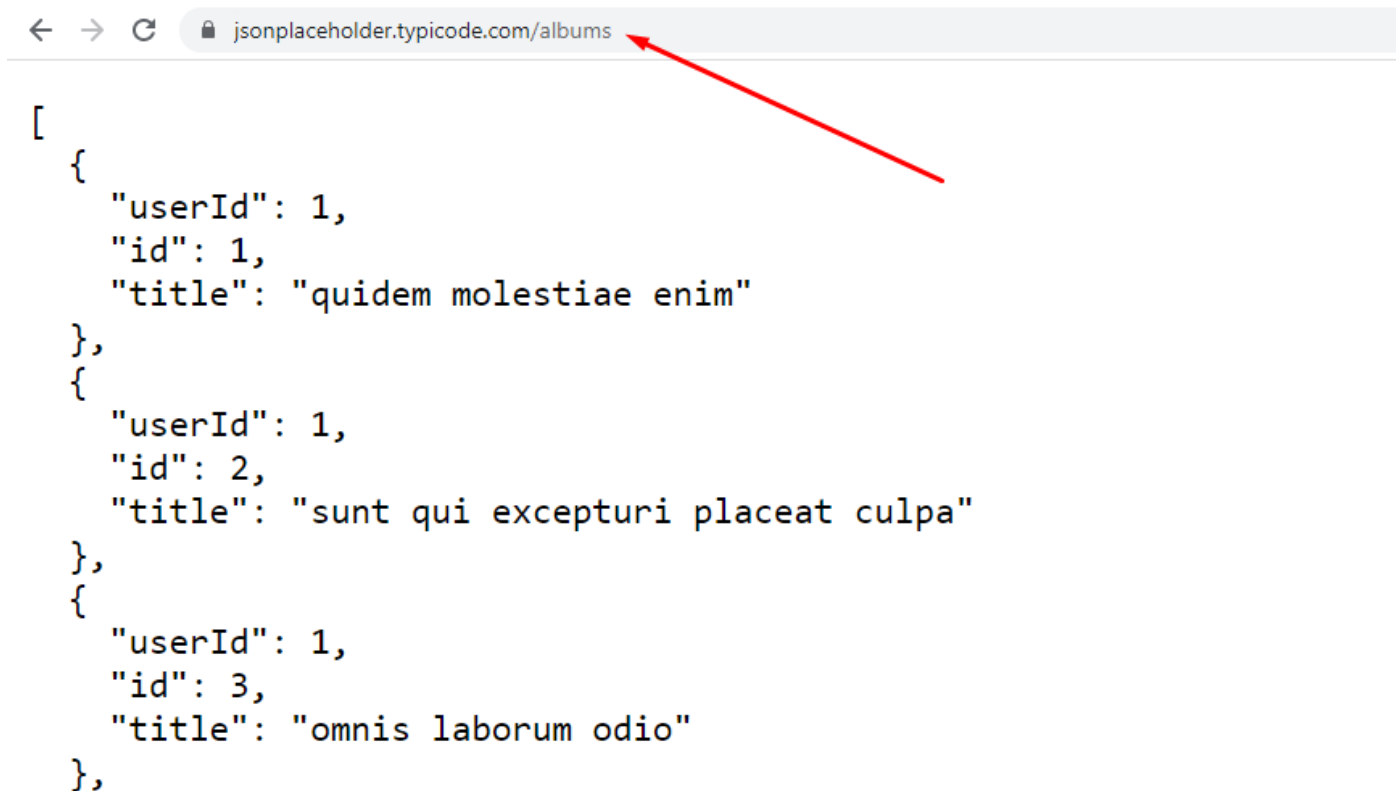
```
1
2 async function getBlogAndPhoto(userId) {
3     try {
4         let utente = await fetch("/utente/" + userId);
5         let result = await Promise.all([
6             fetch("/blog/" + utente.blogId),
7             fetch("/photo/" + utente.albumId)
8         ]);
9         return {
10             utente,
11             blog: result[0],
12             foto: result[1]
13         };
14     } catch (e) {
15         console.log("Si è verificato un errore!")
16     }
17 }
```

In questo caso attendiamo il completamento del caricamento dei dati dell'utente, requisito essenziale per recuperare le altre informazioni, e quindi rimaniamo in attesa del caricamento in parallelo dei dati del blog e delle foto.

In conclusione, le parole chiave `async` e `await` ci aiutano a semplificare il codice per la gestione delle operazioni asincrone, ma non si sostituiscono all'utilizzo delle `Promise`. Queste infatti continuano ad essere alla base dell'esecuzione di codice asincrono e in alcune situazioni risultano ancora insostituibili.

Esempio

Facciamo un altro esempio prendendo spunto dal nostro sito «jsonplaceholder» e prendiamo tutti gli album messi a disposizione :



Come possiamo vedere gli album contengono l'id dell'utente, ma noi supponiamo di voler recuperare anche il nome ed il cognome. Per prima cosa recuperiamo tutti gli album come già sappiamo fare :

```
1
2  const url = 'https://jsonplaceholder.typicode.com/';
3
4  async function elencoalbum() {
5
6      const albums = await fetch(url + 'albums').then( result => {
7          return result.json();
8      }).catch( error => {
9          console.log(error);
10      });
11
12      return albums;
13  }
14
15  let ris = elencoalbum();
16  ris.then( resp => console.log(resp)); // promise
17
```


A questo punto abbiamo gli albums ma noi abbiamo bisogno anche degli utenti, quindi, avendo a disposizione tutti gli album posso ciclare gli album e recuperare l'utente. Intanto andiamo a creare le promises degli utenti :

```
2  const url = 'https://jsonplaceholder.typicode.com/';
3
4  async function elencoalbum() {
5
6      const albums = await fetch(url + 'albums').then( result => {
7          return result.json();
8      }).catch( error => {
9          console.log(error);
10      });
11
12      const usersProm = albums.map(elemento => {
13          return fetch(url + 'users/' + elemento.userId).then(
14              resp => resp.json()
15          )
16      });
17
18      console.log(usersProm);
19
20      return {albums : albums, users : users};
21  }
22
23  let ris = elencoalbum();
24  ris.then( resp => console.log(resp)); // promise
```

map vs foreach

JavaScript ha 2 metodi che aiutano a iterare gli array. Sono `map()` e `forEach()`.

Il metodo **map** riceve una funzione come parametro. Poi lo applica ad ogni elemento e restituisce un array interamente nuovo popolato con i risultati delle chiamate della funzione data. Questo significa che **restituisce un nuovo array che contiene un'immagine di ogni elemento dell'array**. Ritournerà sempre lo stesso numero di elementi.

```
JS app2.js > ...  
1  
2   const mioArray = [10, 5, 4, 3, 2]  
3  
4   mioSplendidoArray.map(x => x * x)  
5  
6   // Output: [100, 25, 16, 9, 4]  
7
```

Come `map`, il metodo `forEach` riceve una funzione come argomento e la esegue una volta per ogni elemento dell'array. Però, invece di restituire un nuovo array come `map`, restituisce `undefined`.

```
JS app2.js > ...
```

```
1
```

```
2   const mioArray = [10, 5, 4, 3, 2];
```

```
3
```

```
4   mioArray.forEach(x => x * x)
```

```
5   // valore restituito: undefined
```

```
6
```

```
7   mioArray.map(x => x * x)
```

```
8   // valore restituito: [100, 25, 16, 9, 4]
```

Con questo codice abbiamo generato 100 promises che tramite una Promise.all possiamo gestirle tutte insieme contemporaneamente :

```
1
2  const url = 'https://jsonplaceholder.typicode.com/';
3
4  async function elencoalbum() {
5
6      const albums = await fetch(url + 'albums').then( result => {
7          return result.json();
8      }).catch( error => {
9          console.log(error);
10     });
11
12     const usersProm = albums.map(elemento => {
13         return fetch(url + 'users/' + elemento.userId).then(
14             resp => resp.json()
15         )
16     });
17
18     const users = await Promise.all(usersProm);
19
20     return {albums : albums, users : users};
21 }
22
23 let ris = elencoalbum();
24 ris.then( resp => console.log(resp)); // promise
25
```