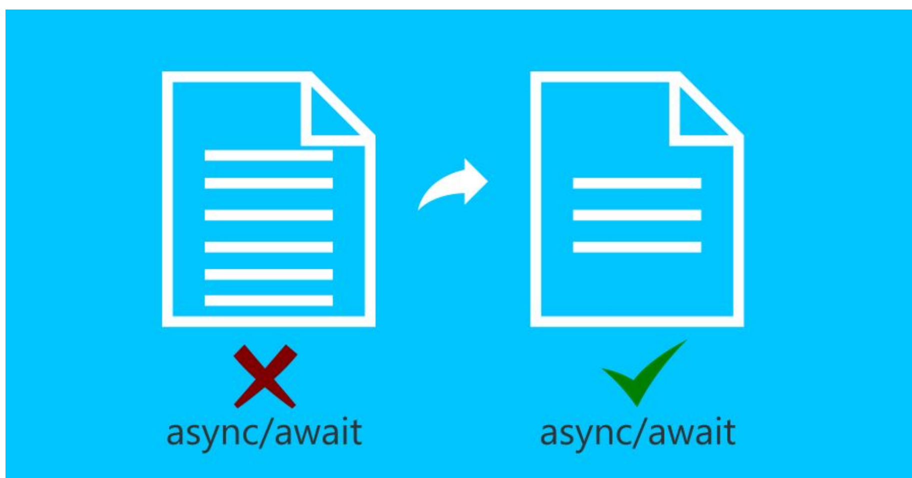
Callback, promise e async await: comprenderli ed usarli per semplificare il codice + come applicare tale tecnica in Angular

Come funzionano le chiamate asincrone? Cosa sono le callback? Cosa sono le Promise in Javascript? Async await è una tecnica utile per migliorare la lettura del codice? In questo articolo risponderemo a tutte queste domande e vedremo come è elegante scrivere codice asincrono come se fosse sincrono.

Applicheremo poi tale tecnica ad *Angular* per trasformare il servizio *HttpClient* in un servizio più semplice da usare dal punto di vista dello sviluppatore, migliorando notevolmente l'esperienza di sviluppo.



Esigenza

Per uno sviluppatore back-end abituato a programmare in *Java* o *C#* piombare sul front-end può essere un duro colpo. Cambiano molte cose, ma un aspetto che è decisamente diverso dal back-end riguarda la comunicazione con il server, che avviene in asincrono.

Nella programmazione sincrona siamo abituati al fatto che una riga di codice che “sta sotto“ nello stesso scope viene eseguita dopo.

```typescript
console.log("1")
console.log("2")
console.log("3")
console.log("4")
```

L’output di questo codice sarà quindi *1,2,3,4*.

Quando abbiamo a che fare con API asincrone, una riga di codice che “sta sotto” potrebbe essere eseguita prima di altre righe di codice asincrone in una *callback* (una *callback* è una funzione che contiene il codice da eseguire quando la richiesta asincrona viene completata).

```typescript
console.log("1")
richiestaAsincrona( /*la callback*/function(){
 console.log("2")
 console.log("3")
})
console.log("4")
```

Qui l’output sarà con molta probabilità 1,4,2,3. Dico con molta probabilità perché in realtà non abbiamo il controllo su **quando** effettivamente il codice asincrono verrà eseguito.

Ma se potessimo scrivere qualcosa del genere e essere sicuri sull’ordine di esecuzione?

```typescript
console.log("1")
```

```
await richiestaAsincrona2()
console.log("2")
console.log("3")
console.log("4")
```

Cioè in sostanza vogliamo trattare le richieste asincrone come se fossero sincrone e dire al sistema “aspetta che arrivi la risposta dalla funzione asincrona e solo dopo esegui e istruzioni che seguono”. Dietro le quinte il tutto avviene ancora in maniera asincrona, ma lo sviluppatore può scrivere il codice in maniera molto più semplice e leggibile. Inoltre si noti come il numero di righe di codice è molto ridotto e meno righe di codice significa meno potenziali bug.

Scopriamo allora come implementare tutto ciò.

Introduzione

Vedremo come funzionano le *Promise*, le richieste asincrone con l’uso delle API Fetch per richiedere informazioni dal Web, e come usare i concetti di *async/await* per semplificare il tutto. Vedremo infine come usare *async/await* in *Angular*.

Cosa sono le Promise

Abbiamo accennato nella sezione precedente che per eseguire del codice quando una funzione asincrona termina potremmo usare una callback. Supponiamo di dover eseguire un’altra istruzione asincrona dentro la *callback*, e un’altra ancora. Presto ci ritroveremo con un codice inguardabile e ingestibile per gli annidamenti multipli.
In Javascript si è quindi pensato di semplificare tutto ciò con l’uso di una nuova tecnica: le *Promise*.

Un *promise* è un oggetto che controlla il flusso di una chiamata asincrona, ci dice quando è terminata la chiamata o se la chiamata è fallita.

Le *promise* sono oggetti della classe *Promise*, quindi possiamo crearne anche noi di nuove. Creare una *promise* è un pochino articolato, ma teniamo a mente che raramente creeremo delle *promise*: quasi sempre le useremo. Vediamo però come crearle per capirne il funzionamento.

```
lang:typescript
let p = new Promise(....)
```

Il costruttore di una *promise* prende in input una funzione con 2 parametri:

```
lang:typescript
let p = new Promise( (res, rej) => { .. })
```

Per convenzione questi parametri si chiamano *res* e *rej*, che stanno rispettivamente per *resolve* e *reject*.
Abbiamo detto che una *promise* deve notificare quando una esecuzione asincrona è terminata, oppure notificare che l'esecuzione della chiamata non è andata a buon fine. Il parametro *resolve* serve appunto per dire quando una *promise* è andata a buon fine, mentre il parametro *reject* serve per dire che una esecuzione asincrona NON è andata a buon fine.
Cerchiamo di capire meglio il funzionamento con un esempio.
Il metodo *setTimeout* di *Javascript* esegue una funzione dopo *x* millisecondi:

```
lang:typescript
setTimeout( ()=>{ console.log('ciao')  } , 1000)
```

Nell'esempio sopra stampiamo *ciao* dopo un secondo.
Proviamo a trasformare il *setTimeout* con l'uso delle *Promise* per ottenere questo risultato:

```
lang:typescript
attendi( 1000  ).then(()=>{ console.log('ciao')  })
```

Il codice appena visto è molto simile al precedente, non è che ci abbiamo guadagnato molto, ma vi anticipo che tra poco vedremo come questo si trasformerà in

```
lang:typescript
await attendi(1000)
console.log('ciao')
```

Bello eh? :-) Ma andiamo per passi.

Creiamo il metodo attendi:

```
lang:typescript
attendi(milliseconds){
    let p = new Promise((res, rej) => {
      setTimeout( res() , milliseconds)
   })
}
attendi(1000).then(...)
```

Quindi quello che c'è dentro il *then* viene richiamato quando nella *Promise* eseguiamo il parametro *res()*. Lo so, sembra di aver sparato ad una mosca con il cannone, ma vi assicuro che ne vale la pena, perché tra poco introdurremo il concetto di *async/await*, e *async/await* usa le *promise* per semplificare notevolmente la scrittura del codice asincrono.

Per comprendere ancora meglio le *promise*, facciamo ora un altro esempio con una funzione asincrona usata spesso e ormai disponibile su tutti i browser principali: il metodo *fetch* per ottenere dei dati da internet.

A tale scopo ho creato un endpoint server side che restituisce dei dati di esempio: se andiamo all'indirizzo https://api.devexp.io/utenti vediamo che viene stampato un *array* di oggetti *JSON*:

```json
lang:json
[
	{
		nome: "Sa",
		cognome: "Ro",
		eta: 33,
		color: "#6ac2ff"
	},
	{
		nome: "Ma",
		cognome: "Bi",
		eta: 31,
		color: "#ff7474"
	},
	{
		nome: "Fa",
		cognome: "Ma",
		eta: 17,
		color: "#ecff73"
	},
	{
		nome: "Be",
		cognome: "Ca",
		eta: 6,
		color: "#80ffb0"
	}
]
```

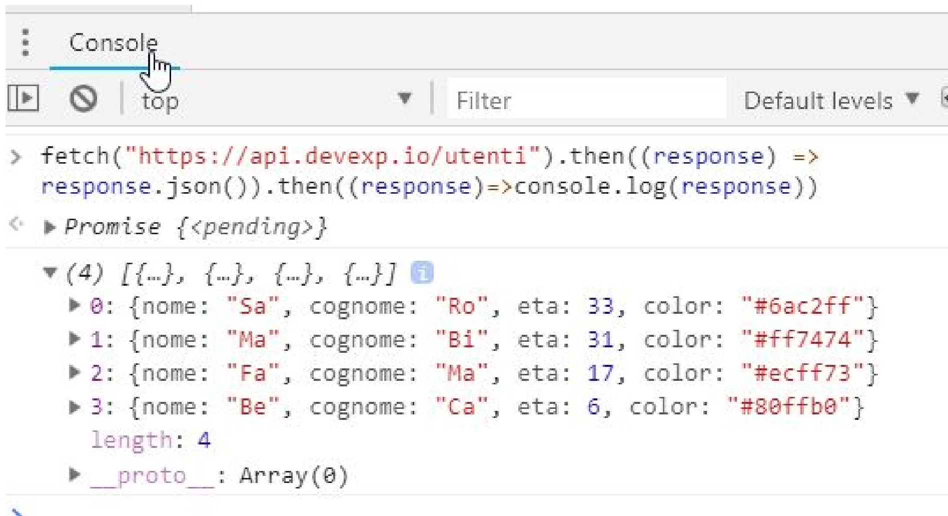
Vediamo quindi come ottenere tale array con la *API fetch*.

L'uso di tale *API* è molto semplice:

```typescript
lang:typescript
fetch("https://api.devexp.io/utenti")
```

```
.then((response) => response.json())
.then((response)=>console.log(response))
```

Il metodo *fetch(URL)* restituisce una *promise*. Come detto in precedenza ogni oggetto *promise* ha un metodo, il metodo *then*, che permette di eseguire del codice quando il browser riceve la risposta dal server. Tecnicamente si dice che la *promise* viene "risolta".
Nel caso della *API fetch*, la risposta dal server non è direttamente l'oggetto *JSON*, ma una risposta con tutte le informazioni come *header*, codici di stato, ecc. Per ottenere il *JSON* dobbiamo richiamare il metodo *json()* della *response*, che è a sua volta asincrono. Finalmente con un ulteriore *then* otteniamo il nostro JSON, che stampiamo nella console:



Rispetto alla singola *callback* non cambia molto, ma non avremo più il problema di *callback* annidate. Ora però andiamo al sodo e vediamo come usare una nuova sintassi per scrivere codice con *Promise* come se fosse codice sincrono.

Usare Async/Await con le Promise

Nella sezione precedente abbiamo visto come usare l'*API fetch*. E se potessimo scrivere qualcosa del genere?

```
lang:typescript
let response = await
fetch("https://api.devexp.io/utenti")
let json = await response.json()
console.log(json)
```

Bè, effettivamente quello che abbiamo appena scritto è codice sintatticamente valido e, sorpresa sorpresa, funziona esattamente come ci aspettiamo: la stampa della variabile *json* avviene correttamente.
Notiamo però che abbiamo dovuto usare una parola chiave nuova per ottenere tutto ciò: la parola chiave ***await***. Con *await* diciamo al sistema di attendere che la *Promise* restituita dal metodo *fetch* venga risolta prima di eseguire le istruzioni successive.
Per usare *await*, le istruzioni che usano await devono stare dentro una funzione dichiarata come *async*:

```
lang:typescript
async function stampaArray(){
   let response = await
fetch("https://api.devexp.io/utenti")
   let json = await response.json()


}
stampaArray()
```

Tutto qui. *Async await* è quindi un modo per abbreviare la sintassi delle *promise*. Qualcuno potrebbe dire che è superfluo, ma per esperienza vi assicuro che scrivere e leggere codice che usa *async/await* è una goduria. Ed è molto meno probabile creare un bug se il codice è facilmente leggibile.
Andiamo ora a conoscere meglio le varie potenzialità di questa sintassi.

Risultati, If e For

Abbiamo già visto con l'*API fetch* che se una *promise* restituisce un risultato, possiamo assegnare questo risultato ad una variabile come se il *fetch* fosse sincrono:

```typescript
lang:typescript
let response = await
fetch("https://api.devexp.io/utenti")
```

La *response* conterrà tutte le informazioni di risposta.

La cosa si fa interessante quando cominciamo ad usare un *if*. Supponiamo ad esempio di voler caricare dei dati se e solo se non li abbiamo già caricati:

```typescript
lang:typescript
if(!utenti) {
    let response = await
fetch("https://api.devexp.io/utenti")
    let json = await response.json()
    utenti = json
}
```

Quindi *await* funziona anche dentro un *if*. E dentro un *for*? Anche dentro un *for* è garantito il rispetto del flusso simil-sincrono.
Attenzione però perché spesso ciò non è quello che vogliamo: un *await* dentro un *for* fa chiamate asincrone solo quando la precedente chiamata è stata completata. Se dovessi fare il *for* di un array di 1000 elementi rischierei di fare mille chiamate asincrone. E' questo il risultato che vogliamo ottenre? Probabilmente è possibile in questi casi trovare un'alternativa più efficiente.

Gestione errori col try/catch

Abbiamo anticipato che una *promise* può anche andare in errore. In questi casi il codice per la *promise* sarebbe il seguente:

```
lang:typescript
fetch("https://api.devexp.io/utenti")
 .then((response) => response.json()).catch( error
=> console.log(error))
```

Come si traduce tutto ciò con *async/await*? Prima di rispondere faccio un'altra domanda: come gestiamo un evento eccezionale, un'eccezione, con i codici sincroni? La risposta è: con il *try/catch*. Ebbene con *async/await* possiamo ancora usare il *try catch* per gestire errori dentro chiamate asincrone:

```
lang:typescript
try {
   fetch("https://api.devexp.io/utenti")
   .then((response) => response.json())
} catch(e) {
  console.log(e)
}
```

Uso di Async/Await in Angular

Ora che conosciamo meglio come fare chiamate asincrone con l'uso di *async/await*, vediamo come usare il client *HTTP* di *Angular* per scrivere più facilmente il nostro codice.

In *Angular* per fare chiamate *HTTP* si usa il servizio *HttpClient*: per fare una get usiamo i le istruzioni:

```
lang:typescript
```

```
constructor(http: HttpClient) {

http.get("https://api.devexp.io/utenti").subscribe(
success => {
   console.log(success)
 }, error => {
   console.log(error)
 })
}
```

Vogliamo trasformare il servizio per scrivere le istruzioni precedenti come segue:

```
lang:typescript
let utenti = await
this.awaitHttp.get("https://api.devexp.io/utenti")
console.log(utenti)
```

Per ottenere questo risultato dobbiamo creare un nuovo servizio, che chiameremo *AwaitHttpClient*. Questo servizio usa internamente l'*HttpClient* di *Angular* e crea un metodo *get* alternativo

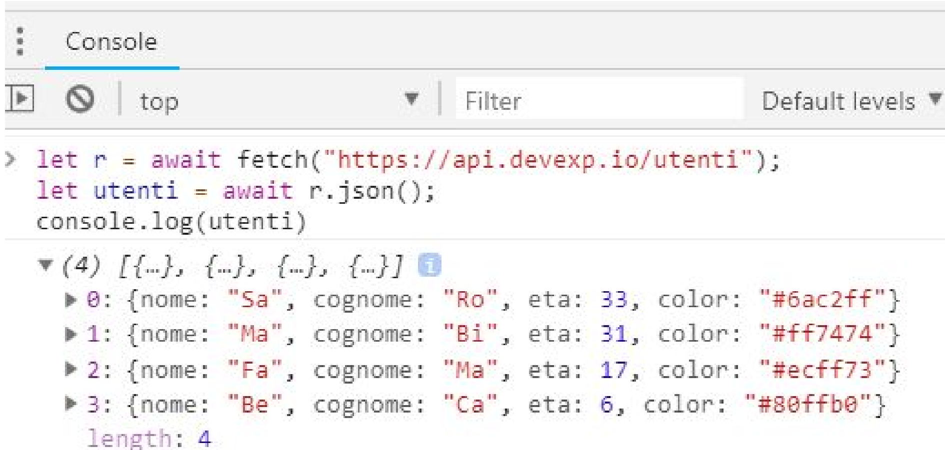
```
lang:typescript
class AwaitHttpClient {

 constructor(private http:HttpClient) { }

 get(url:string){
   let p = new Promise((res, rej)=> {
     this.http.get(url).subscribe(success =>
res(success), error => rej(error))
   })
   return p
 }
}
```

Il metodo *get* restituisce una *Promise* che viene risolta quando la risposta arriva dal server dentro il metodo *subscribe*.

Per tutti gli altri metodi, tra cui il *post*, il discorso è analogo. Nelle conclusioni vi rimando ad un esempio completo.

Debugging

La cosa interessante è che in Chrome possiamo anche debuggare il codice *async await* normalmente, proprio come se fosse codice sincrono.
Addirittura possiamo usare l'*await* nella *console* di *Google chrome*:



Cosa imparerai con l'esperienza

Con l'esperienza osserverai che l'uso di *async await* rende il codice molto elegante e pulito. Per quanto riguarda *Angular*, personalmente non sento la mancanza degli *Observer*, che ritengo utili solo in specifiche circostanze. Il vantaggio di avere un codice molto più snello è invece sicuramente molto maggiore rispetto all'avere uno strumento potente come gli *observer*, ma quasi mai utilizzato.

A livello di uso dell'*async/await*, è bene notare che quando non usiamo il *try/catch*, in caso di errore una eventuale variabile restituita dall'istruzione di *await* avrebbe valore *null*, quindi facciamo attenzione.

Un altro aspetto, velocemente introdotto con l'uso dei *for*, riguarda l'uso di *await* in cascata. Facciamo attenzione quando usiamo molti *await* in cascata, poiché le richieste asincrone saranno eseguita l'una alla fine dell'altra, e non in parallelo. Se non ci serve il risultato dell'*await* precedente sarebbe uno spreco di tempo non eseguire le richieste in parallelo.
Per eseguire le richieste in parallelo possiamo sempre non usare l'*await* e ritornare alle vecchie e care *Promise*. Nel caso volessimo invece non perdere tempo, eseguire le richieste in parallelo e solo dopo eseguire del codice possiamo combinare l'*await* con le *promise* con un meccanismo di questo tipo:

```
lang:typescript
let response1 = http.get("https://api.devexp.io/utenti");
let response2 = http.get("https://api.devexp.io/utenti")
await Promise.all([response1, response2])
// codice da eseguire dopo le richieste asincrone
```

Conclusioni

In questa guida abbiamo visto come funzionano le *Promise* e come sfruttare la nuova sintassi *async/await* disponibile su quasi tutti i browser (IE11 non è tra questi *:-)*). Possiamo usare tale tecnica anche con *TypeScript*, che tradurrà poi in *JavaScript* le nostre istruzioni.
Abbiamo anche visto come usare la potenza dell'*async/await* in *Angular*.

A questo indirizzo potete trovare un esempio di servizio *http* completo che usa *async await* al posto degli *observer*. Questo servizio carica anche un file *preferences.json* prima di eseguire le chiamate, perché in queste *preferences* è contenuto l'*URL* base dinamico a cui inviare tutte

le chiamate. Inoltre tale servizio aggiunge un *header* ad ogni chiamata per includere un *token* di autenticazione. Può essere utile come spunto per creare il vostro servizio *AwaitHttpClient* completo, ma vi sconsiglio di usarlo direttamente in quanto si basa sul servizio *Http*, ormai deprecato anche se ancora funzionante anche in Angular 6.

https://gist.github.com/salvatoreromeo/a4ba14c0815b051497239502bf7ce671

Grazie per la lettura e ti ricordo che se vuoi approfondire gli argomenti di *Angular* puoi trovare il libro completo su Amazon seguendo i link riportati di seguito.